Robotics Back log

This aims to be chronological recompilation of the progress made with the Robotics arm.
By: **Jorge Pérez Ramos**
Contact: jorgepramoscontact@gmail.com

Purchase of equipment:

Before starting the project some equipment was needed.
Desktop computer in order to run Ubuntu and ROS
The desktop was purchase though IT Labs: Dell OptiPlex 7000

The robot communicates with the computer making use of a IP address. This means that a router was needed in order to make use of a private network. **Router of choice** --> Linksys Hydra Pro 6

Additionally some lesser materials were needed:

- 2x Ethernet cables (Router and PC connection)
- 1x Ethernet 10+m cable (Router robot connection)
- 1x USB hub (robot peripherals)
- 1x USB extension cord (USB hub PC connection)
- Webcam LifeCam Cinema

Note:
The choice of the webcam comes from the inexpensive nature of it and the already exitance of a 3d model holder to mounted on the robotics arm.

Equipment installation

Computer installation

The first step was to install the computer in the robotics lab over in the prototyping building.
We must note that this computer already came with a windows 10 license in which we are not interested, since this machine will be dedicated to Ubuntu development ROS.

Router installation and configuration

The router hardware installation was straight forward.
Problems arisen from the software configuration of the equipment.
First attempts were conducted using the standard procedure, which was guided by the Linksy

android app. Sadly it was not possible to complete the configuration through this route due to the app looping in the last step of the configuration, asking for rest of the router and commencing again the whole loop.
The solution came though contact of the technical service making use of the live chat option, which is the most recommendable for future problems --> Live chat Linkys
Though the indications of the Linksy technician we were able to skip the guided configuration jumping straight to the manual settings making use of second computer and a direct connection though ethernet cable.

Resulting credentials:

Network name:
Password: bicrobotics
Settings password: admin

Ubuntu installation

Prior considerations:
Some experimentation was conducted in a virtual machine about the different options given by ROS, which was:
ROS Noetic version | Ubuntu 20.4 | Recommend option by ROS
ROS Melodic version | Ubuntu 18.4 | Already implemented version for other robots in BIC

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys (**Recommended**)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)

Even though ROS noetic has been released for quite some time, some of the necessary resources like ROS-Industrial from the universal robotics repository or the `robotiq` repository were not up to date with noetic. The existence of an already made configuration by the other robotics team in BIC was the last factor which tipped the balance in favor of the more studied but older version of ROS melodic.

Installation of ROS melodic

Checking the official documentation for ROS melodic installation, the recommended supported version of was Ubuntu Bionic 18.04 downloaded.
In order to prepare it for installation the `.iso` file needs to be flash on an flash drive. For this Balena etcher was used.

Problem:
The installation process on the target computer encounter its first problem when the system was unable to detect the internal hard drive of the computer to install the Linux OS.

Solution:
After a lot of digging the problem was found to be the way the file system in the hard drive was initialize in the BIOS. The default SATA and SATA3 mode change to the modes IDE and AHCI. This workaround meant that the windows partition was not longer accessible since it needed from the SATA3 mode to initialize correctly. For this the option of making the ubuntu the only OS in the system was taken.

Problem:
After installing Ubuntu Bionic 18.04 in the machine the next problem became clear. The OS was unable of detecting any of the network adapters in the computer, hence not allowing internet access.

Solution:
After a lot of digging, it was found out that our equipment, the dell OptiPlex 7000, was only supported by ubuntu 20.4 as is pointed out by the ubuntu certified hardware: Dell OptiPlex 7000
This meant that in order to have a working and connected ROS machine we would need to install ubuntu Focal 20.04, this entailed jumping to the last released of ROS Noetic. The downside of this move was the absence of some components necessary for the development with the UR5e and the `robotiq` gripper.

The same flashing procedure was followed with Ubuntu Focal and Etcher.
Once installed, the computer was able to connect to the internet with no issue.
This marked the end of the general installation process.

Important Note: Ubuntu Focal 20.4 Must NOT be updated since it will start conflicts with the noetic ROS distribution.

ROS Installation

ROS Noetic -- in --> Ubuntu Focal (20.04)

Single-line installation

Allows installation of ROS with a single command:

```
wget -c --no-check-certificate
https://raw.githubusercontent.com/qboticslabs/ros_install_noetic/master/ros_insta
ll_noetic.sh && chmod +x ./ros_install_noetic.sh && ./ros_install_noetic.sh
```

Full process installation (traditional):

Traditional installation

Additional

Setting ROS environment (adding it to "environment variables")
Add ROS environment to .bashrc. ---> this will arrow the use of ROS commands

```
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Catkin workspaces

Ros makes use of catkin workspaces in order to mantain development modular.

Catkin workspace basics

Start a workspace
Create folder with src subfolder

```
mkdir -p name_project/src
```

Source ROS:

```
source /opt/ros/noetic/setup.bash
```

Once is created we can navigate inside of it to existing catkin make

```
catkin_make
```

Note: ALWAYS run catkin_make on workspace folder **NOT** on src folder

Package creation and use

Create package INSIDE **src** folder:

```
catkin_create_pkg myname_pkg rospy dependency1 ...
```

Or used existing:

```
rosdep update
rosdep install --rosdistro $ROS_DISTRO --ignore-src --from-paths src
```

Once we have created the dependencies we should build the changes with:

```
catkin_make
```

Simulation

Before connecting to the robot is important to set up a simulation environment, for this we will base our project on two pieces of software:

- Gazebo --> robot simulation
- RViz --> robot control.

For motion planning we will use the MoveIt frame work.
For simulation we would need Ros-industrial
Note: ROS- industrial which is necessary for the simulation of the UR-5e is not updated to noetic, the version which has been tested and works with noetic is the branch `melodic-devel-stagging`
For connecting to the UR5e we will need the appropriate UR-Driver

RViz Basics:

Installing RViz (if not already included with ROS)

```
sudo apt-get install ros-$ROS_DISTRO-rviz
```

Installing joints publisher
Publisher node which publishes states of joints (e.g. angles)

```
sudo apt-get install ros-$ROS_DISTRO-joint-state-publisher-gui
```

Install Universal robots ROS Industrial package
This will allow RViz to represent the robot arm in addition this package provides a lot of files and functionality needed in order to work with the different robots from Universal Robots.

With Git

```
~/catkin_ws/src$ git clone -b $ROS_DISTRO https://github.com/ros-
industrial/universal_robot.git
```

Note: Working branch for Noetic = `melodic-devel-staging`

Dependency installation

```
~/catkin_ws/src$ cd ..
rosdep update
rosdep install --rosdistro $ROS_DISTRO --ignore-src --from-paths src
```

Note: Might need to install python3-rosdep2

Missing MoveIt dependency

```
sudo apt install ros-noetic-moveit
```

Build and activate your workspace

```
catkin_make
source $HOME/catkin_ws/devel/setup.bash
```

Note: might have to adjust the second command to serve the file structure that we are personally running

Tutorial package

```
sudo apt-get install ros-noetic-urdf-tutorial
```

Note: if the package can not be located we will need to add:

```
echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" | sudo tee
/etc/apt/sources.list.d/ros-latest.list
```

Tutorial URDF
URDF source: URDF File
Paste the URDF file from the provided source to: `ur5_rviz.urdf`

Display the robot

```
roslaunch urdf_tutorial display.launch model:='<path-to-the-urdf-
file>/ur5_rviz.urdf'
```

Or by making use of the launch files provided by the universal robots description package

```
cd /carkin_ws/src/universal_robot_ur_description
roslaunch load_ur5.launch
roslaunch view_ur5.launch
```

Gazebo basics

Pre-steps

- ROS installation

Installing Gazebo (if not already included with ROS) and launching
Install Gazebo:

```
curl -sSL http://get.gazebosim.org | sh
```

Launch Gazebo

```
roslaunch gazebo_ros empty_world.launch
```

Gazebo uses URDF description files to simulate robots, this files are converted to SDF by gazebo. Added properties needed from Gazebo for working and simulating the robot correctly with URDF in comparison to RViz are:

- Weights (inertia)
- Collisions
- Limits and dynamics

Gazebo URDF file --> Modified URDF

Before continuing with the following steps we must create and source a work space

Install Universal robots ROS package and dependencies
If we don't have the `ur_description package` installed already we must follow these commands:

With Git

```
~/catkin_ws/src$ git clone -b $ROS_DISTRO-devel https://github.com/ros-industrial/universal_robot.git
```

Note: There can be conflicts on the branch since it can not be longer available
Use melodic branch: `melodic-devel-staging`

```
~/catkin_ws/src$ cd ..
rosdep update
rosdep install --rosdistro $ROS_DISTRO --ignore-src --from-paths src
```

Note: Might need to install python3-rosdep2

Build and activate your workspace

```
catkin_make
source $HOME/catkin_ws/devel/setup.bash
```

Launch empty world

```
roslaunch gazebo_ros empty_world.launch
```

Spawn robot

```
rosrun gazebo_ros spawn_model -file <path-to-your-gazebo-urdf/ur5_.urdf -urdf -x 0 -y 0 -z 0.1 -model ur5
```

Or making use of the launch files:

```
roslaunch ur_gazebo ur5.launch
```

Beginning of development

After getting familiar with the basics of ROS, it was time to tackle the main problem.

The first set of challenges which we would tackle included setting up all the necessary systems for the robot simulation. This would be even more challenging than usual due to the update to ROS Noetic from melodic.

List of key points:

- UR5e URDF and simulation
- `Robotiq` gripper simulation
- Coupling of robot arm, gripper and stand
- Hanging of the whole system upside own to match real set up

- Creation of MoveIt configuration package
- Correct simulation and control with RViz and Gazebo

URDF and part coupling

The main challenge in this stage came from the URDF of the `robotiq` gripper, since the last available version of this was compatible with kinetic and was extremely out-of-date.

After lots of research a repository was found containing a version compatible with ROS noetic
Repository: robotiq ros-noetic

Challenge:
Now it was necessary to update the existing URDF and XACRO files of the coupled gripper + arm + stand since they were aimed towards the ROS melodic distro.

Outcome:
Main problems presented during the resolution of this challenges were originated by the conflicts created amongst the different XACRO, with redundant link, duplicated names, or missing features. After lots of debugging the XACRO and URDF files I was able to compile a full builds making us of the `catkin_namke` command.

After this a prelaminar MoveIt configuration was created with the robot assembly on top of a stand.

Problems: MoveIt package compiles and launches RViz with out problem but the simulation over on gazebo run in to some hiccups due to what I believe it being overlapping between collisions messes.
Further investigation needed before moving to the hanging upside on step.

Coupling on RViz Motion planning and Gazebo simulation

Problem description: With the objective of being able to simulate the arm in Gazebo while controlled by RViz a controller from needs to be stablished in order to transfer the motions from the planner to the simulation.

- In the long process of trying to arrive to the solution the following attempts were made:
- Attempt to compare the configuration of the UR5e to the settings of the panda arm from the repository: Panda Simulation
- Attempt to change the controllers and reconstruct a second version of the MoveIt configuration package. Found the error in the URDF of ur5e2f where the namespace which links the URDF with the MoveIt controller was incorrect. This solved the erratic movement of the arm.
- Attempt the revisit the MoveIt Docs Which resulted on the attachment of the drawers to the world using a fixed link to prevent movement.
- Attempt of following The construct webinar Managing to successfully launch RViz and MoveIt with the `demo.launch` file.

This brings us to a new problem in which we need to create a launch file so gazebo starts publishing the necessary topics for our robot system (ur5e + Robotiq). This idea was abandoned

Attempt to create a third version of the MoveIt configuration package in which we implement the controllers described for the panda arm in the MoveIt documentation adapting it to our hardware. MoveIt documentation

Current errors on launch of `roslaunch ur5e_robotiq_moveit_config demo_gazebo.launch`
Errors on load:

```
[ERROR] [1686231028.882825483]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/shoulder_pan_joint
[ERROR] [1686231028.884727731]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/shoulder_lift_joint
[ERROR] [1686231028.885857646]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/elbow_joint
[ERROR] [1686231028.887213254]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/wrist_1_joint
[ERROR] [1686231028.888925697]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/wrist_2_joint
[ERROR] [1686231028.890162263]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/wrist_3_joint
```

```
[ERROR] [1686150744.231731, 1.213000]: Failed to load joint_state_controller
```

```
[ERROR] [1686150744.276138466, 1.258000000]: Could not find joint
'gripper_right_finger_joint' in 'hardware_interface::EffortJointInterface'.
[ERROR] [1686150744.276515126, 1.258000000]: Failed to initialize the controller
[ERROR] [1686150744.276584288, 1.258000000]: Initializing controller
'gripper_controller' failed
[ERROR] [1686150745.278468, 2.259000]: Failed to load gripper_controller
```

```
[ERROR] [1686150745.300073536, 2.281000000]: Could not find joint
'shoulder_pan_joint' in 'hardware_interface::EffortJointInterface'.
[ERROR] [1686150745.300264516, 2.281000000]: Failed to initialize the controller
[ERROR] [1686150745.300302885, 2.281000000]: Initializing controller
'ur5e_arm_controller' failed
```

```
[ERROR] [1686150746.302059, 3.280000]: Failed to load ur5e_arm_controller
```

Errors on execution:

```
[ERROR] [1686150828.179760150, 85.032000000]: Unable to identify any set of
controllers that can actuate the specified joints: [ elbow_joint
shoulder_lift_joint shoulder_pan_joint wrist_1_joint wrist_2_joint wrist_3_joint
]
[ERROR] [1686150828.179882636, 85.032000000]: Known controllers and their joints:
```

Next challenge comes in the form of flipping the system upside don in order to simulate the configuration the real system

Attempted solutions:

- Attempted to get controllers, in case they are missing:
  `sudo apt-get install ros-noetic-ros-control ros-noetic-ros-controllers`
- Attempted to install controllers from ROS Ur-Drivers Documentation
  Not related since the drivers and the trajectories controllers use in them are only related to the real robot connection and not the gazebo simulation which makes use of ROS Ur-industrial

---

Next steps:

- Creation of hanging robot models.
- Creation of catkin work space and launch files
- URDF assembly.
- Creation of hanging robot MoveIt configuration.

Creating Model of actual set-up

In order to continue the development of the system is necessary to recrate the set up of the robot arm.

Current set up

As appreciated in the image bellow the robot is set up on a hanging position in a metal frame which replicates the final dimensions of the Averna cabinet.

For this is necessary to update the model so the simulations closely resemble reality.

Creation of the metal frame

In order to represent the picture above, we need to create a 3d representation of it making use of the URDF format (Universal Robot Description Format).
In this URDF description we make use of the `.dae` format to represent visuals giving it complex structures while using simple geometry to represent collisions for fast computation.

Full model creation pipeline (Visual + collisions)

For the full creation of this 3D environments we will use a combination of 3D modeling software's are community plugins to translate our model in to URDF.

Step by step process

Video of the process --> Link

How to import SolidWorks model into Gazebo?

`Created: 19 June 2019`

1 Convert from solidWorks to URDF

- Use the SolidWorks to URDF Exporter plugin sw_urdf_exporter to convert your model into URDF format.
- Once installed, follow the Instructions
- The plugin will generate the following files and folders:

```
        ├── config
        ├── launch
        ├── manifest.xml
        ├── meshes
        │   └── mydesign.STL
        ├── textures
        └── urdf
        └── mydesign.urdf
```

- The 2 files that we will want are the .**STL** and .**URDF** files.

2 Convert the STL to .DAE

- Launch FreeCAD.
- Create new File > New.
- Import your STL file; File > Import, then browse and choose the file.
- Notice that the color information of the model is lost.
- Click on the model, then File > Export, save as a Collada (.DAE) type.

3 Color the model.

- Launch Blender.
- Delete the default cube.
- File > Import > Collada, then choose your .DAE file.
- The model is in m, but blender is using mm. So its too small to see.
- Click on the + sign on top right to open the Transform menu.
- Then in Dimensions, multiply all x,y, & z values by 1000.
- Now we color it.
- Press [Tab] to go into Edit mode.
- Then choose Face select (menu at bottom).
- At right panel, go to Material.
- Add new material, give it a name.
- Click under Diffuse, select the color you want.
- Right click on the surface you want to color, click on the new material, then click Assign.
- Keep doing this for all the surface you want to color.
- You can select multiple surface triangle at the same time with the Ctrl & Shift keys, you can also add other colors.
- File > Export > Collada.

4 Convert the .URDF to .SDF

- Launch a terminal
- Navigate to the model's .URDF folder
- In as terminal instance run:

```
gz sdf -p foo.urdf > foo.sdf
```

- Open the .SDF file
- Make sure it looks good. Change the name or the pose of the model if you want.
- Under collision and visual elements, you will see uri element. Now change the model file name from .STL to .DAE.
- Save the file as `model.sdf`

Note:
Some applications take .SDF v 1.5 only, not 1.6. Test yours and change accordingly.

Minimal model creation pipeline (collisions only)

This is the the process which was actually followed in order to generate the current URDF model. This process focused on the generation of the computational elements only disregarding the visual aspect.

Subdivision of the system

The system was subdivide in its working subparts:

- Metal frame/ Metal cabinet.
- Ur5e (Cobot).
- Robotiq gripper.

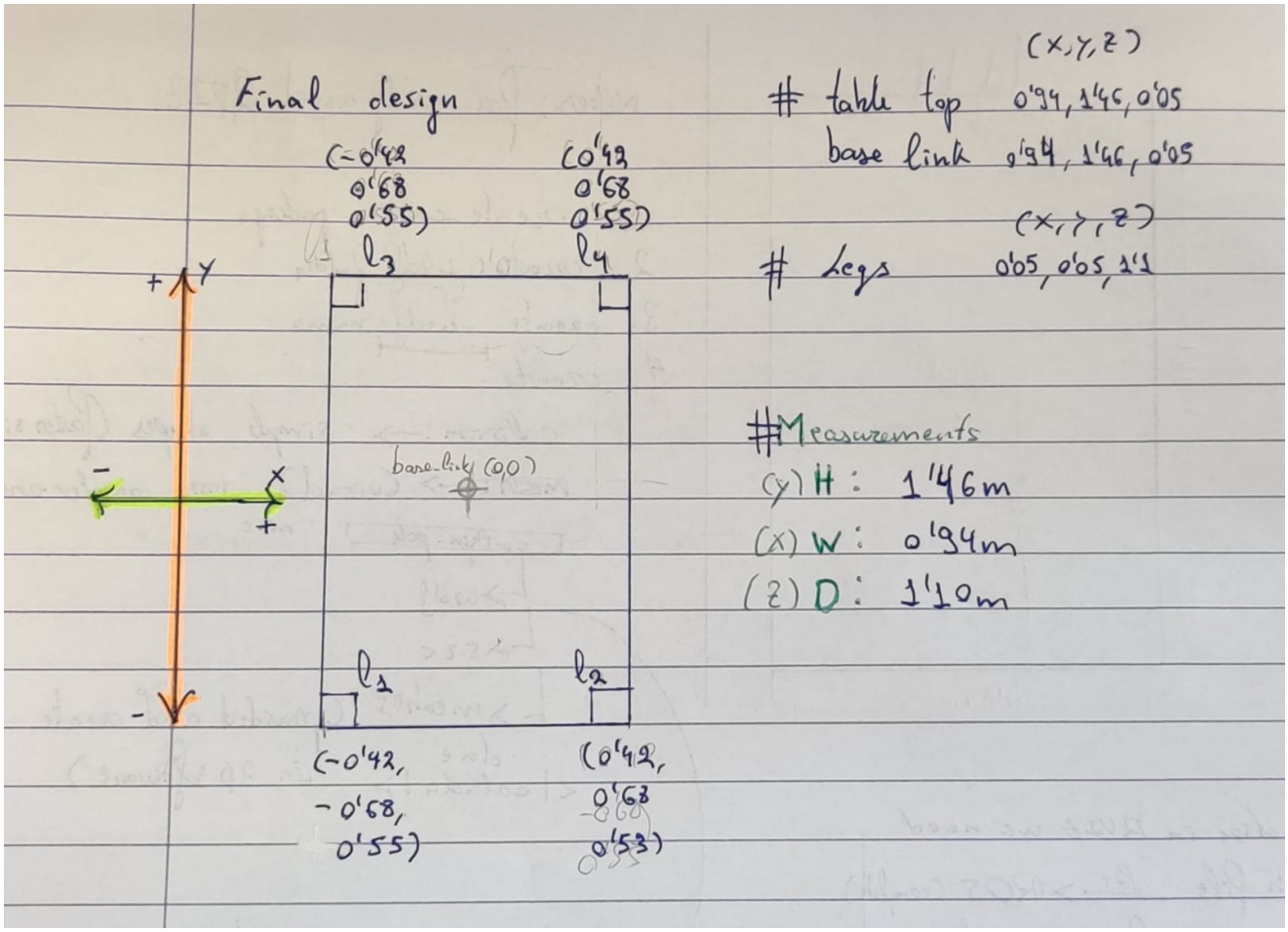
URDF models

The advantage of using off the shelf parts for our system is that we can really on the prebuild files from the manufacturer or the Robotic community. In our case:

- ROS Industrial - Universal Robots
- Robotiq gripper

With this two packages the only model we have left to build is the actual robot stand or steel frame. For this measurements were taken in order to represent the real dimensions of the hardware

piece.



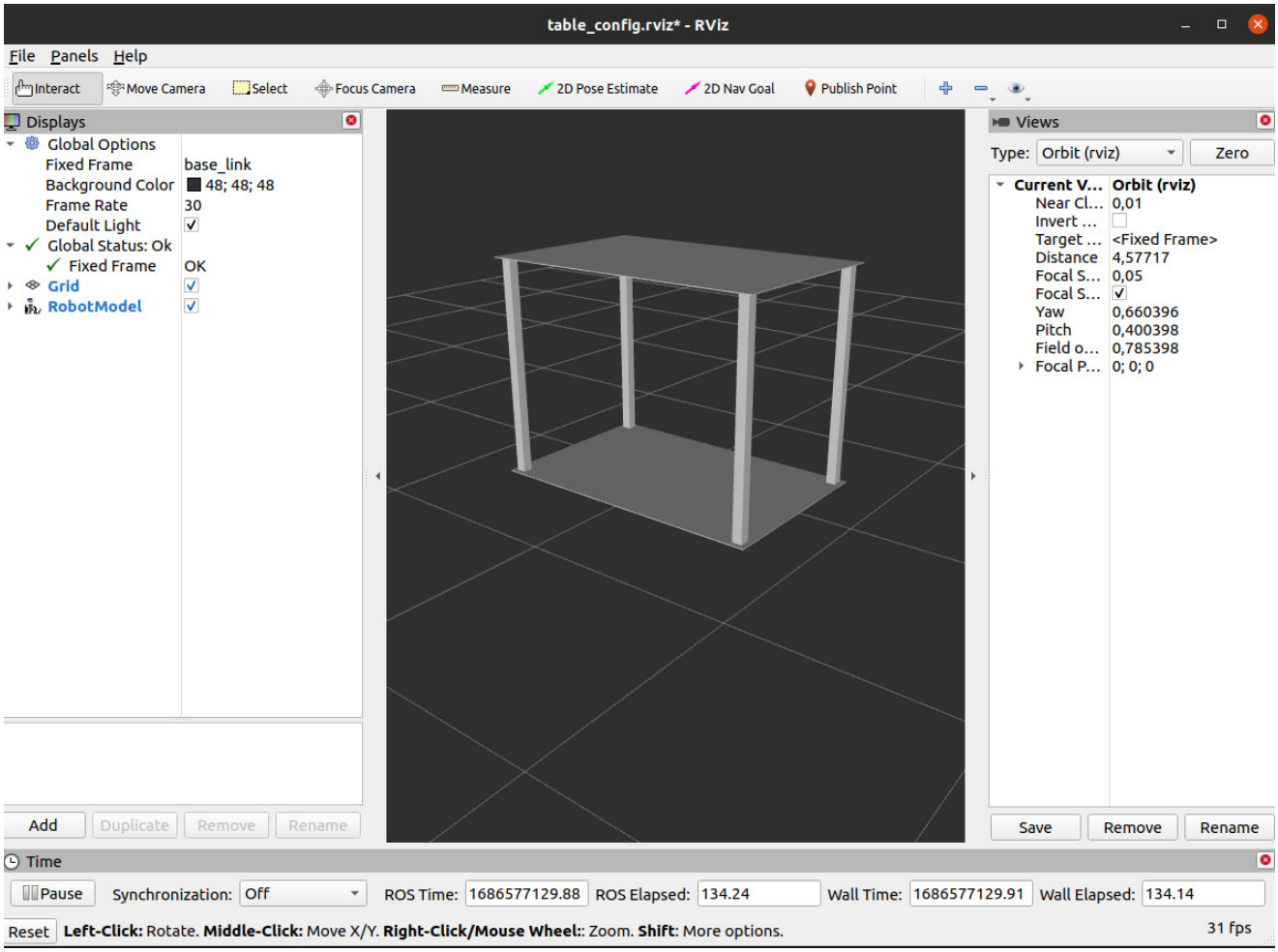
The resulting file: `frame.xacro`
In order to visualize our file we will use RViz. For this we will make use of a launch file in order to start the program and the necessary nodes.

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
<!--Find xacro file for the frame-->
  <arg name="urdf_file" default="$(find ur5e2f_hang)/urdf/frame.xacro"/>

  <param name="robot_description" command="$(find xacro)/xacro $(arg urdf_file)"
/>

  <!--Combine joint values-->
  <node name = "robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher"/>

  <!--Start RViz-->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
ur5e2f_hang)/config/table_config.rviz"/>

  <!--Publish joint Values-->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
    <param name="use_gui" value="True" />
```

```
  </node>
</launch>
```

In order to run this script we must execute:

```
roslaunch ur5e2f_hang spawn.launch
```



Note: XACRO is a macro language which allows us to easily concatenate XML format which is native to .URDF.

The next and final step to finalize our model is to "hang" our UR5e from the newly contructed frame. The following set up will be compossed:

- The frame will be fixed to the world.
- The robot will be hang from the frame.
- The gripper will be attached to the end of the arm model.

To recreate this set up we will need a new `.XACRO` file in order to concatenate our `URDF` instances.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro" name="ur5e_robotiq">

    <!--Import frame-->
    <xacro:include filename="$(find ur5e2f_hang)/urdf/frame.xacro"/>
```

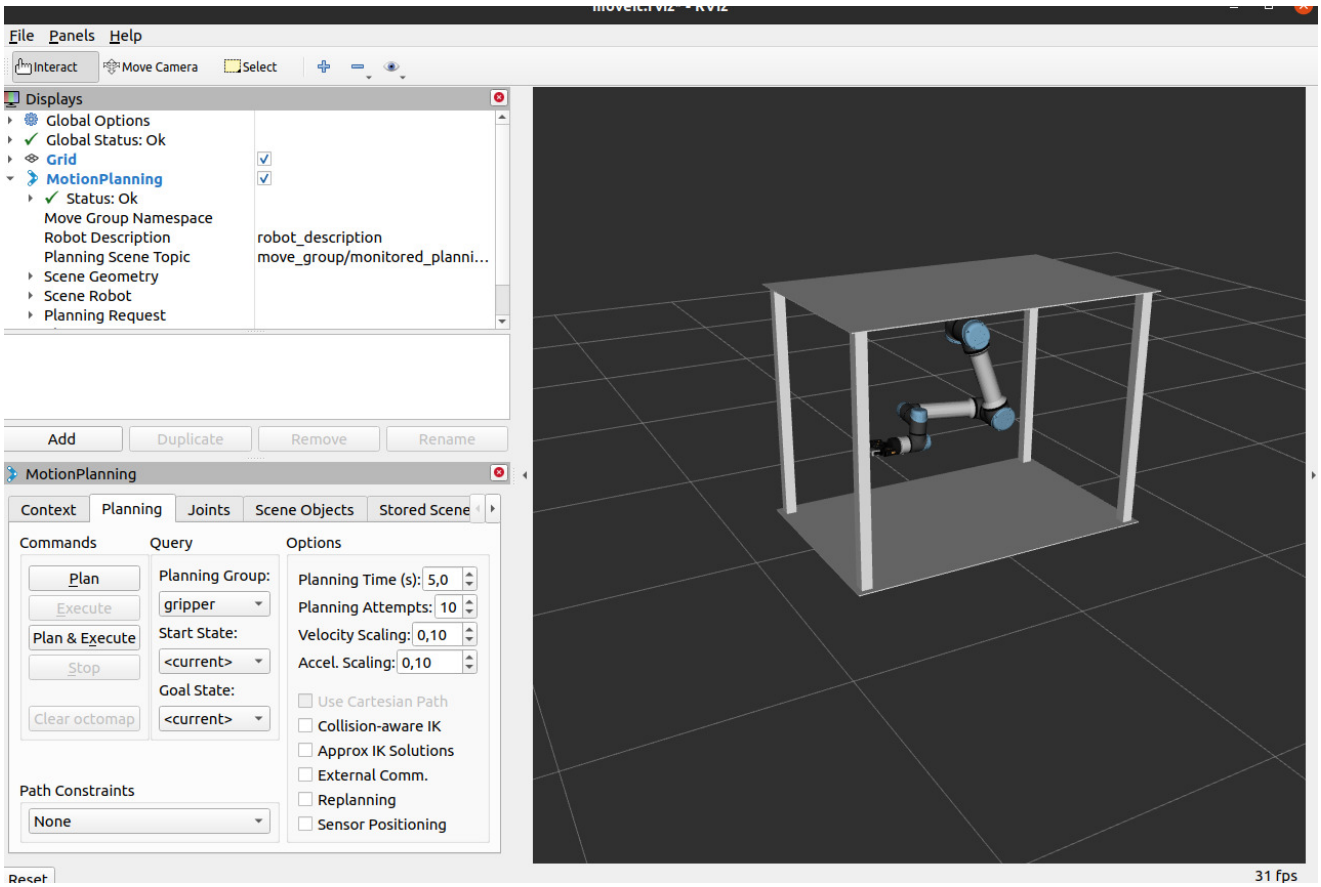
```
    <!--Import the ur5e robot-->
    <xacro:include filename="$(find ur_description)/urdf/ur5e.xacro"/>

    <!--Import the robotiq gripper-->
    <xacro:include filename="$(find
robotiq_2f_85_gripper_visualization)/urdf/robotiq_arg2f_85.xacro"/>


    <!--Join the UR5e with the Drawers Box-->
    <joint name="block_ur5_connection" type="fixed">
        <origin xyz="0 0 -0.005" rpy="0 ${radians(180)} 0"/>
        <parent link="top_link"/>
        <child link="base_link"/>
        <axis xyz="0.0 0.0 1.0"/>
    </joint>


    <!--Join ur5 and robotiq gripper-->
    <joint name="tool0_gripper_joint" type="fixed">
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 1.57"/>
        <parent link="tool0"/>
        <child link="gripper_base_link"/>
        <axis xyz="0.0 0.0 1.0"/>
    </joint>

    <!--Add an additional link just for planning that represents the tip of the
closed gripper-->
    <joint name="joint_gripper_tip" type="fixed">
        <origin xyz="0.0 0.0 0.18" rpy="0.0 0.0 0.0"/>
        <parent link="tool0"/>
        <child link="gripper_tip"/>
        <axis xyz="0.0 0.0 1.0"/>
    </joint>

    <!--Gripper tip-->
    <link name="gripper_tip">
        <inertial>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
            <mass value="0.01"/>
            <inertia ixx="0.001" ixy="0.001" ixz="0.001" iyy="0.001" iyz="0.001"
izz="0.001"/>
        </inertial>
        <visual>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
            <geometry>
                <sphere radius="0.01"/>
            </geometry>
            <material name="white">
                <color rgba="1.0 1.0 1.0 1.0"/>
                <texture filename=""/>
            </material>
        </visual>
```

```
        <collision>
            <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
            <geometry>
                <sphere radius="0.01"/>
            </geometry>
        </collision>
    </link>

<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
  </plugin>
</gazebo>

</robot>
```

Note: The UR5e model is mounted under the top plate, reference by setting the joint at the thickness of this top surface. We rotate the model `${radians(180)}` in order to have it hanging upside down.

In order to visualize it we will need to change our spawn file `.XACRO` file from `frame.xacro` to `ur5e2f.urdf.xacro` and run it as usual.

```
roslaunch ur5e2f_hang spawn.launch
```



Possible error and solution:

```
ERROR: cannot launch node of type [robot_state_publisher/state_publisher]: Cannot
locate node of type [state_publisher] in package [robot_state_publisher]. Make
sure file exists in package path and permission is set to executable (chmod +x)
```

Solution: `state_publisher` was a deprecated alias for the node named `robot_state_publisher`

```
<node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" />
```

Connecting the UR5e to a ROS machine

For this we will need to prepare the two sides to be able to connect to each other. The two sides being: The UR5e arm and the ROS machine.

ROS machine

We need to follow a couple of pre-steps in order to have our version of ROS ready to work with our robot.

Prerequisites:

- ROS installation
- ROS Workspace
- MoveIt config for UR5e and gripper

Additionally we will need to clone the repository containing the drivers needed for our UR5e; ROS driver

Cloning process

Once created our basic work space we should progress as follow:

```
# clone the driver
$ git clone https://github.com/UniversalRobots/Universal_Robots_ROS_Driver.git
src/Universal_Robots_ROS_Driver

# clone the description. Currently, it is necessary to use the melodic-devel
branch.
$ git clone -b melodic-devel https://github.com/ros-
industrial/universal_robot.git src/universal_robot

# install dependencies
$ sudo apt update -qq
```

```
$ rosdep update
$ rosdep install --from-paths src --ignore-src -y

# build the workspace
$ catkin_make

# activate the workspace (ie: source it)
$ source devel/setup.bash
```

Once we have everything cloned and built we can start with the configuration process on the robot side.

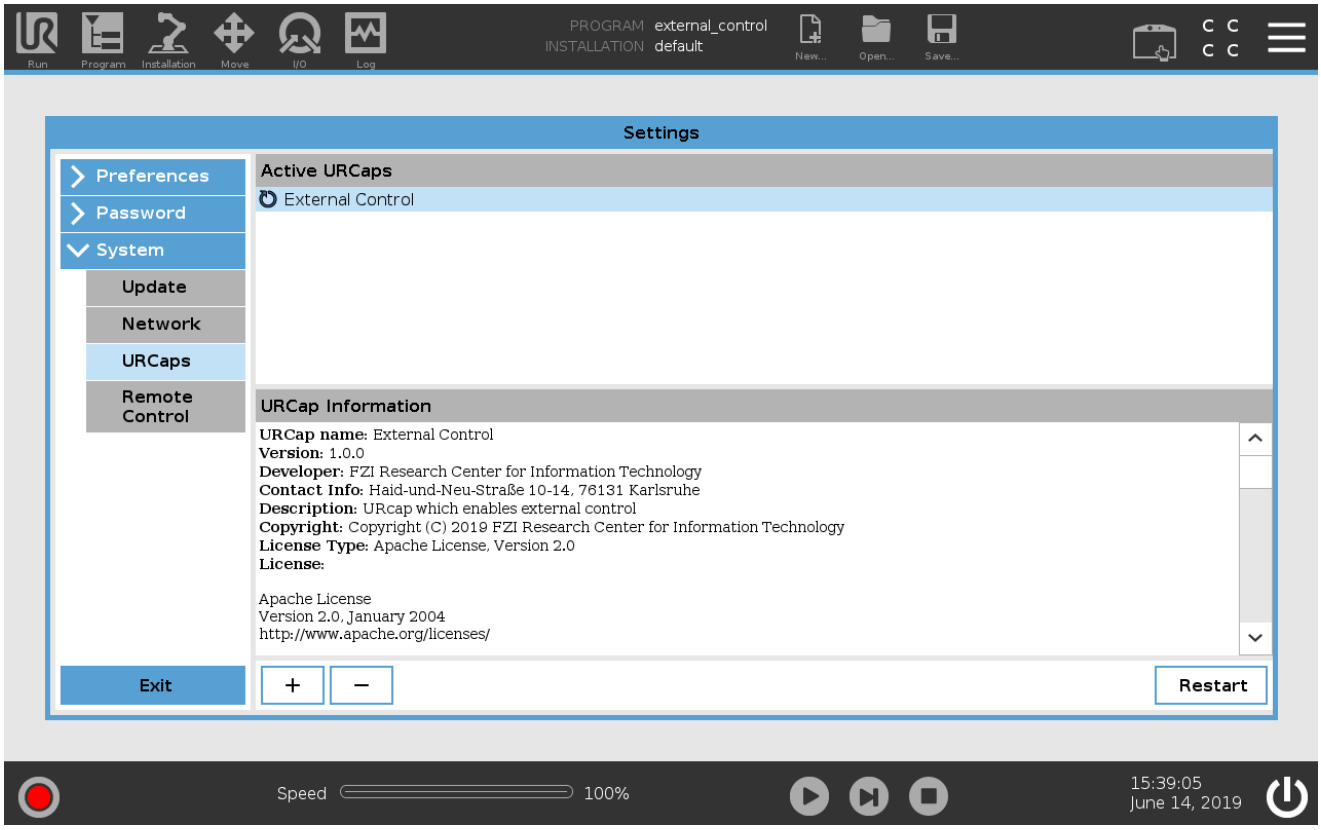
UR5e robotic arm

Official documentation: ROS driver documentation

After powering on the robot we must install the **externalcontrol-x.x.x.urcap**.
For this we will need a thumb drive and the `.urcap` software external control.

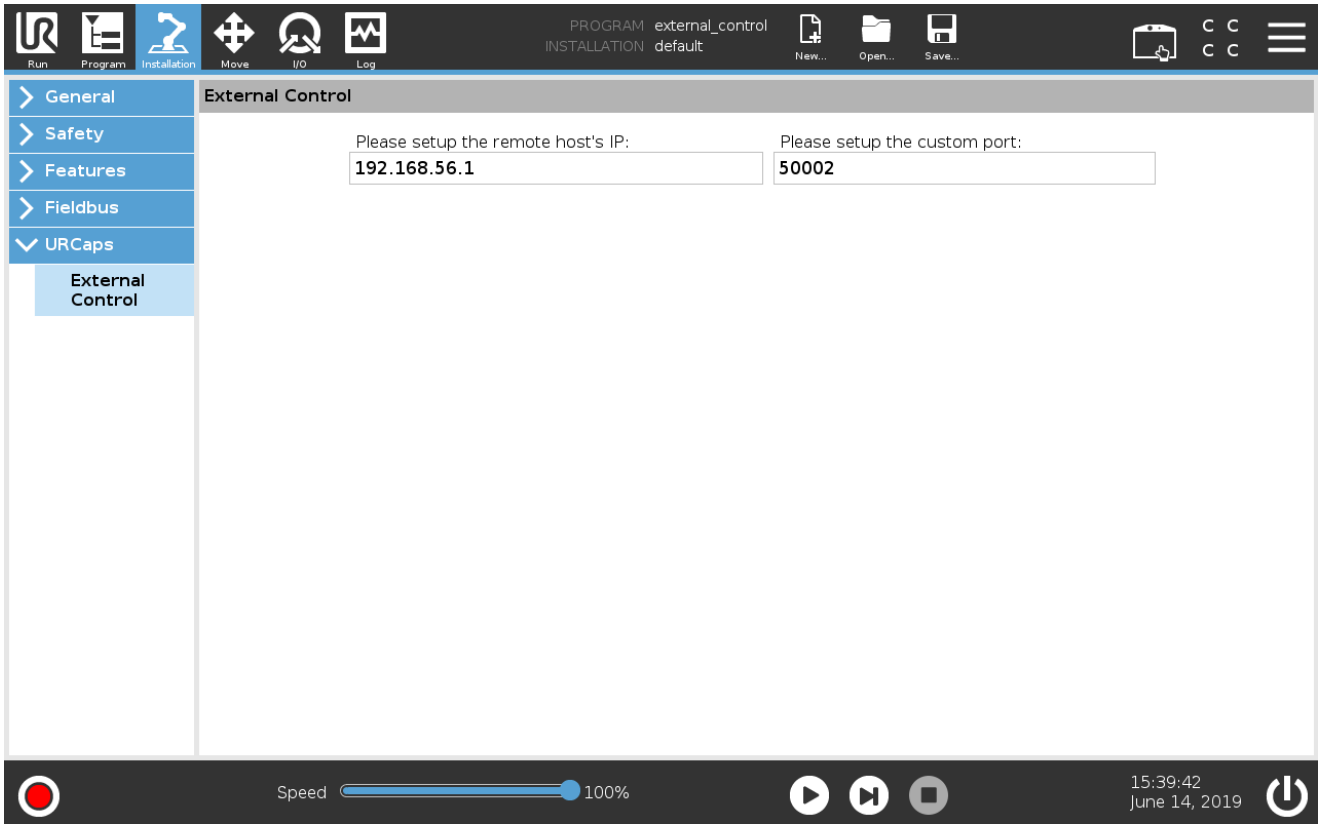
Once the `.urcap` file is transfer to our flash drive, we can connect it to the robot terminal though the screen controller.

On the welcome screen click on the hamburger menu in the top-right corner and select *Settings* to enter the robot's setup. There select *System* and then *URCaps* to enter the URCaps installation screen.



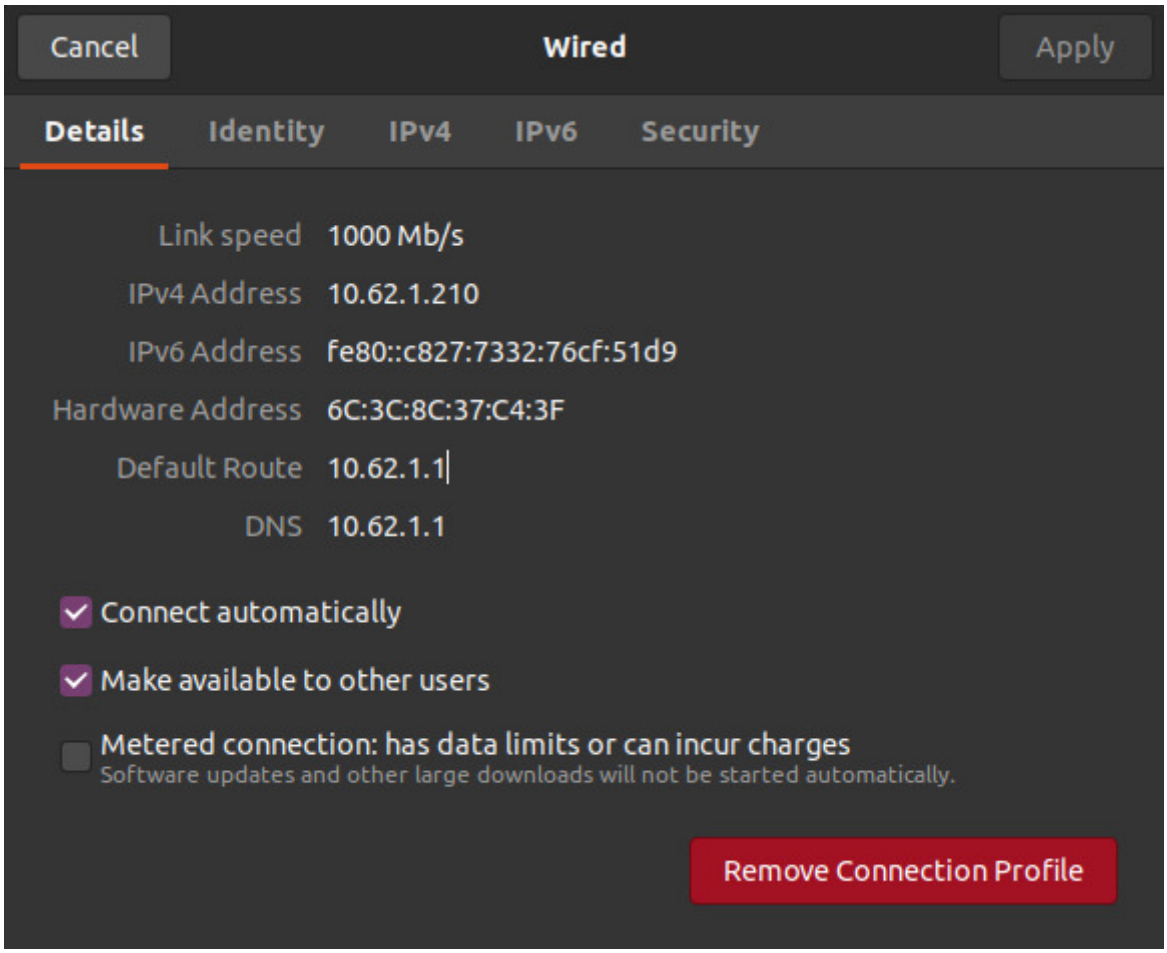
After the reboot you should find the **External Control** URCaps inside the *Installation* section. For this select *Program Robot* on the welcome screen, select the *Installation* tab and select **External**

Control from the list.



Here you'll have to setup the IP address of the external PC which will be running the ROS driver. Note that the robot and the external PC have to be in the same network, ideally in a direct connection with each other to minimize network disturbances. The custom port should be left untouched for now.

For this go to network settings on the ROS pc and access the wired configuration (if connected through ethernet). We can find here our IP (IPv4).

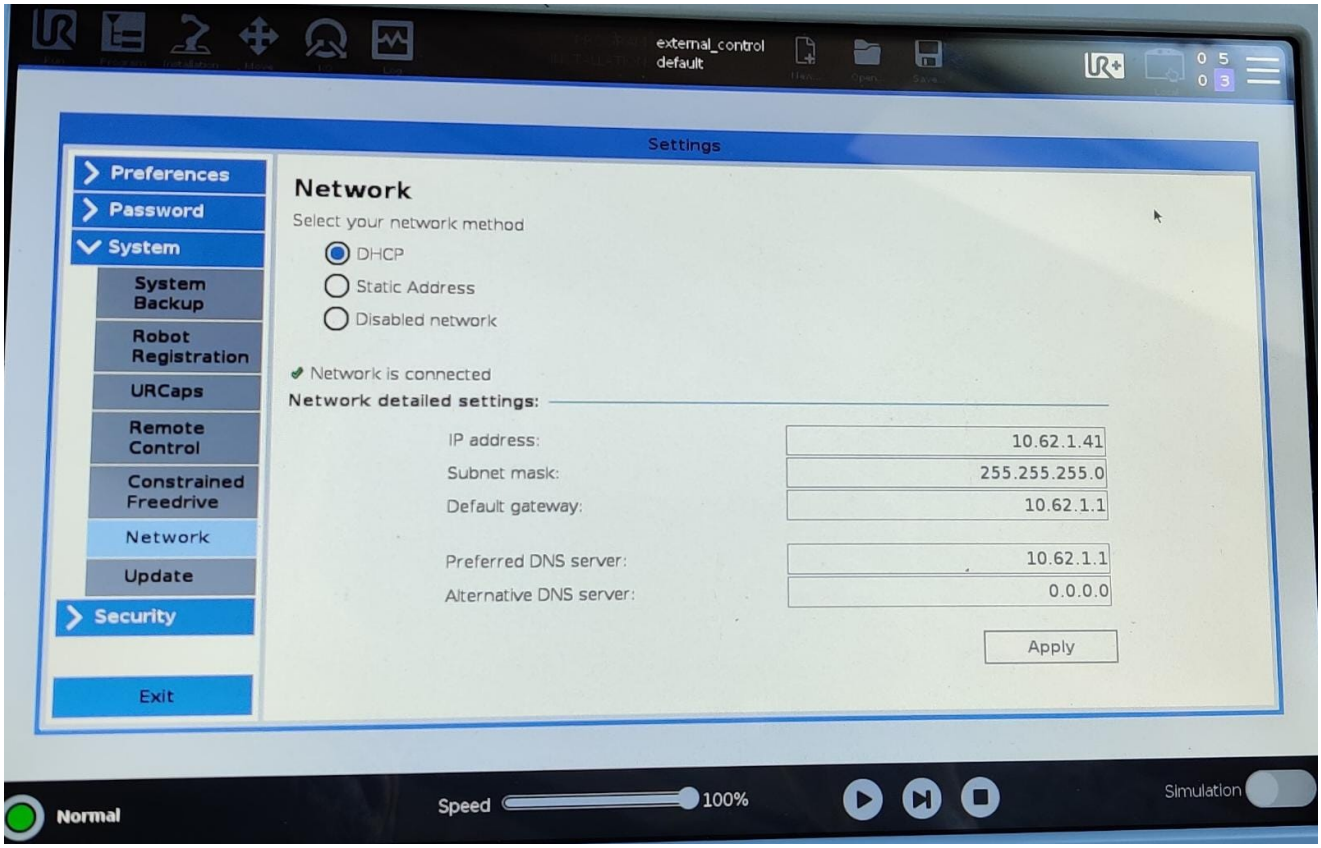


Last step we need to take before moving forward with the process, is to get the robots IP and connected it to the network if we haven't done it already.
The robot must be connected directly to a common router for both the pc ant itself.

Note: The ethernet port is located inside the control box of the robot.

Once the ethernet cable is connected we can navigate over the configuration section on the pad and select Network. Here we will select DHCP and let the system stablish the connection. Once this is done we can take note down of the IP address assigned to our robot since it will be use in

the next steps.



To use the new `URCaps`, create a new program and insert the **External Control** program node into the program tree. Before running the program we need to complete some last steps over at the ROS machine.

ROS machine final steps

To guarantee the good actuation of the robot, we would need to download the calibration file for our model. For this we should run the following snippet:

```
roslaunch ur_calibration calibration_correction.launch robot_ip:=[IP]
target_filename:="[target_path]/my_robot_calibration.yaml"
```

Once we have downloaded our calibration file in our desired path, we move forward to execute the following launch file which allows us to stablish connection with our UR5e passing generated calibration file as an argument.

```
roslaunch ur_robot_driver ur5e_bringup.launch robot_ip:=[IP]kinematics_config:="
[target_path]/my_robot_calibration.yaml"
```

This will start a process which will wait until we run our external control program over our control panel. Once we run it, the terminal will promt us with a success message.

Note: We must not close this terminal, since of it depends the connection between the two parts (Robot and ROS machine).

Protective stops

If during the operation of the robot we trigger a protective stop, we will need to follow this steps to resume our work:

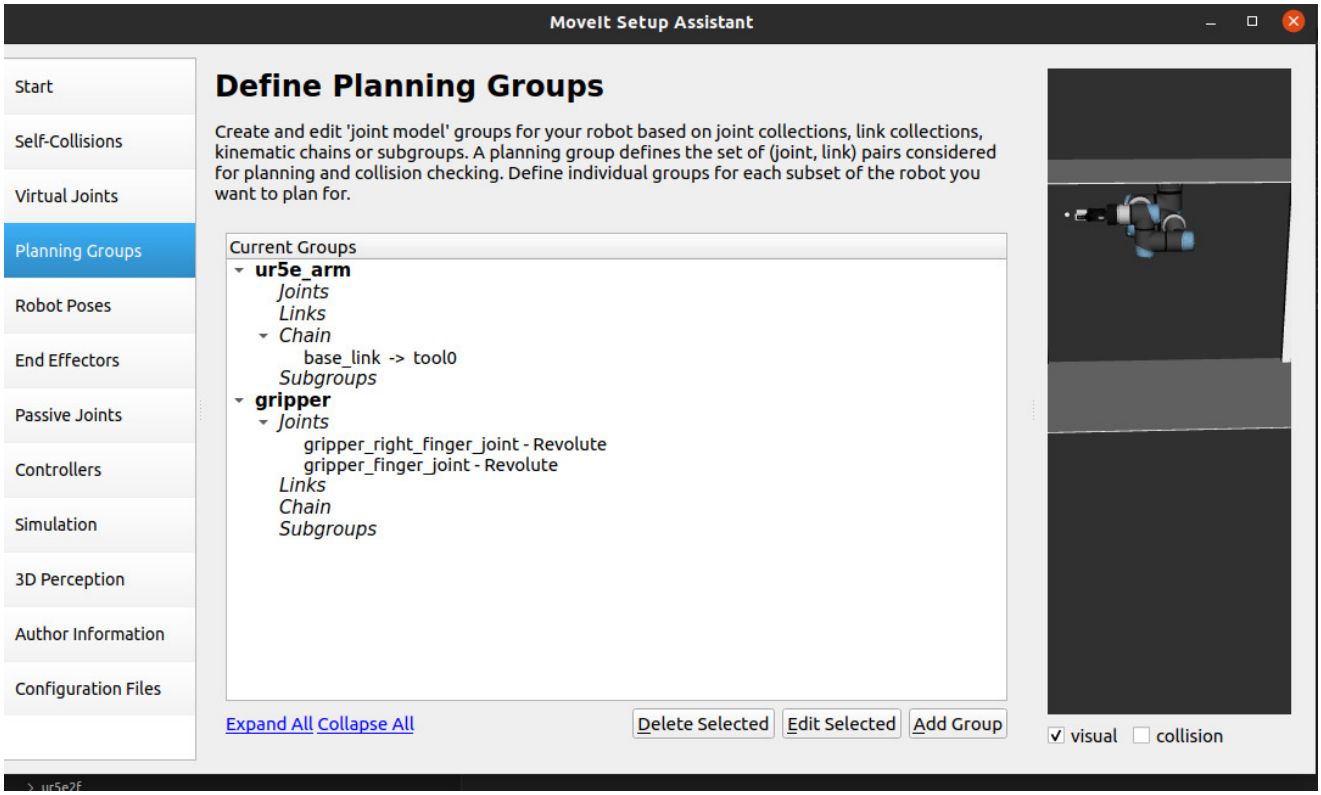
- Solve the collision by moving the robot arm in free drive into a safe position.
- Re run the `ur5e_bringup.launch` command over our ROS machine.
- Re start the external control program over our robot control panel.

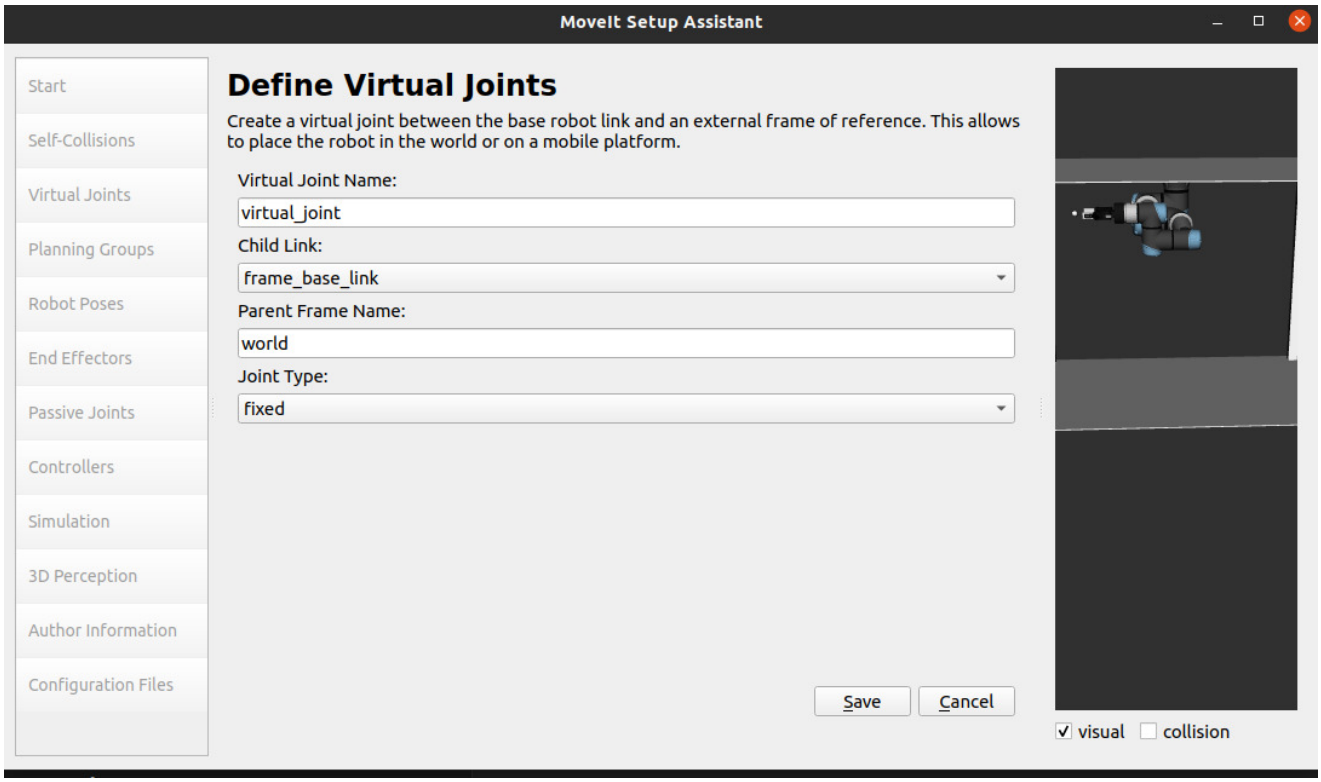
The last step is to run RViz with the real ROS controllers.

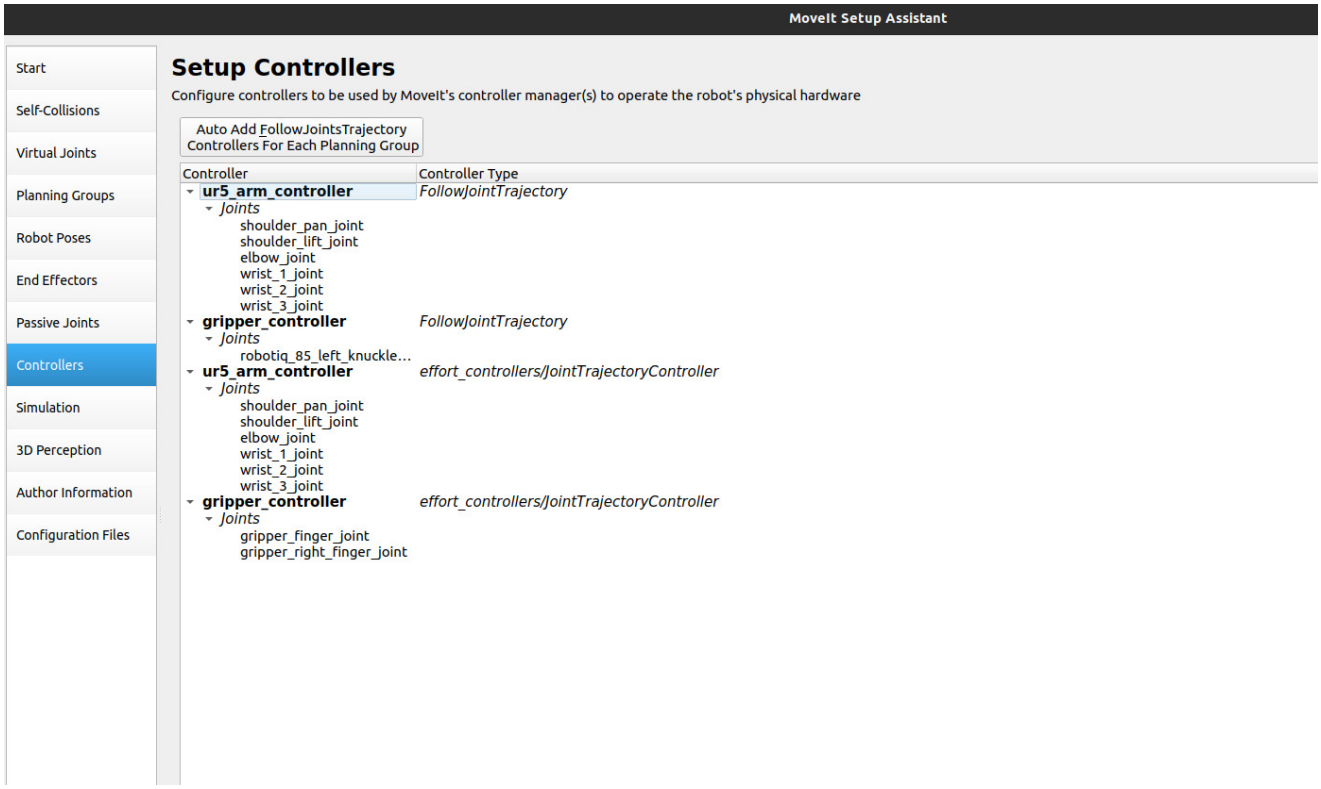
```
roslaunch ur5e_robotiq_hang_moveit_config demo.launch
moveit_controller_manager:=ros_control
```

Creation of MoveIt package for the new configuration

Since we are changing the way our robot is mounted, we would also need to reconfigure our MoveIt configuration package accordingly.
We would only focus on the principal changes to the configuration.







Solving configuration problems

- **Gripper Missing joint waning**
  When load:

```
[WARN] [1687184570.525863486]: The complete state of the robot is not yet known.
Missing gripper_finger_joint
```

When execute:

```
[ERROR] [1687274065.705779156]: Unable to identify any set of controllers that
can actuate the specified joints: [ gripper_finger_joint ]
[ERROR] [1687274065.705861342]: Known controllers and their joints:
controller '/pos_joint_traj_controller' controls joints:
  elbow_joint
  shoulder_lift_joint
  shoulder_pan_joint
  wrist_1_joint
  wrist_2_joint
  wrist_3_joint
```

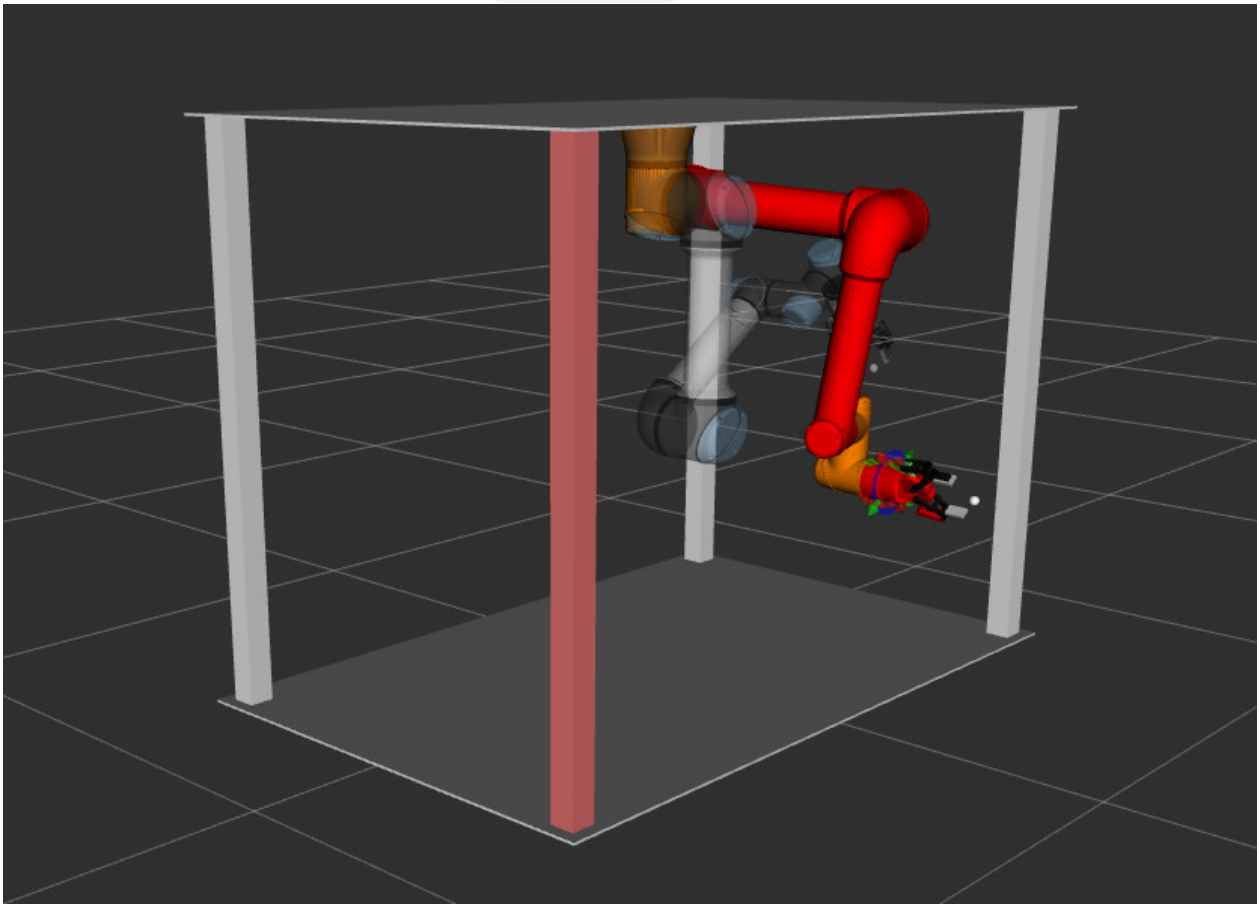
- **Displaced frame URDF**
  The current orientation of the robot does not represent the actual disposition of the arm
  For this a 90 rotation has been applied to the base of the robot in its connection with the frame
  File: `ur5e2f.urdf.xacro`

```
    <!--Join the UR5e with the frame Box-->
    <joint name="block_ur5_connection" type="fixed">
        <origin xyz="0 0 -0.005" rpy="0 ${radians(180)} ${radians(90)}"/>
        <parent link="top_link"/>
        <child link="base_link"/>
        <axis xyz="0.0 0.0 1.0"/>
    </joint>
```

We cam appreciate the two rotations:

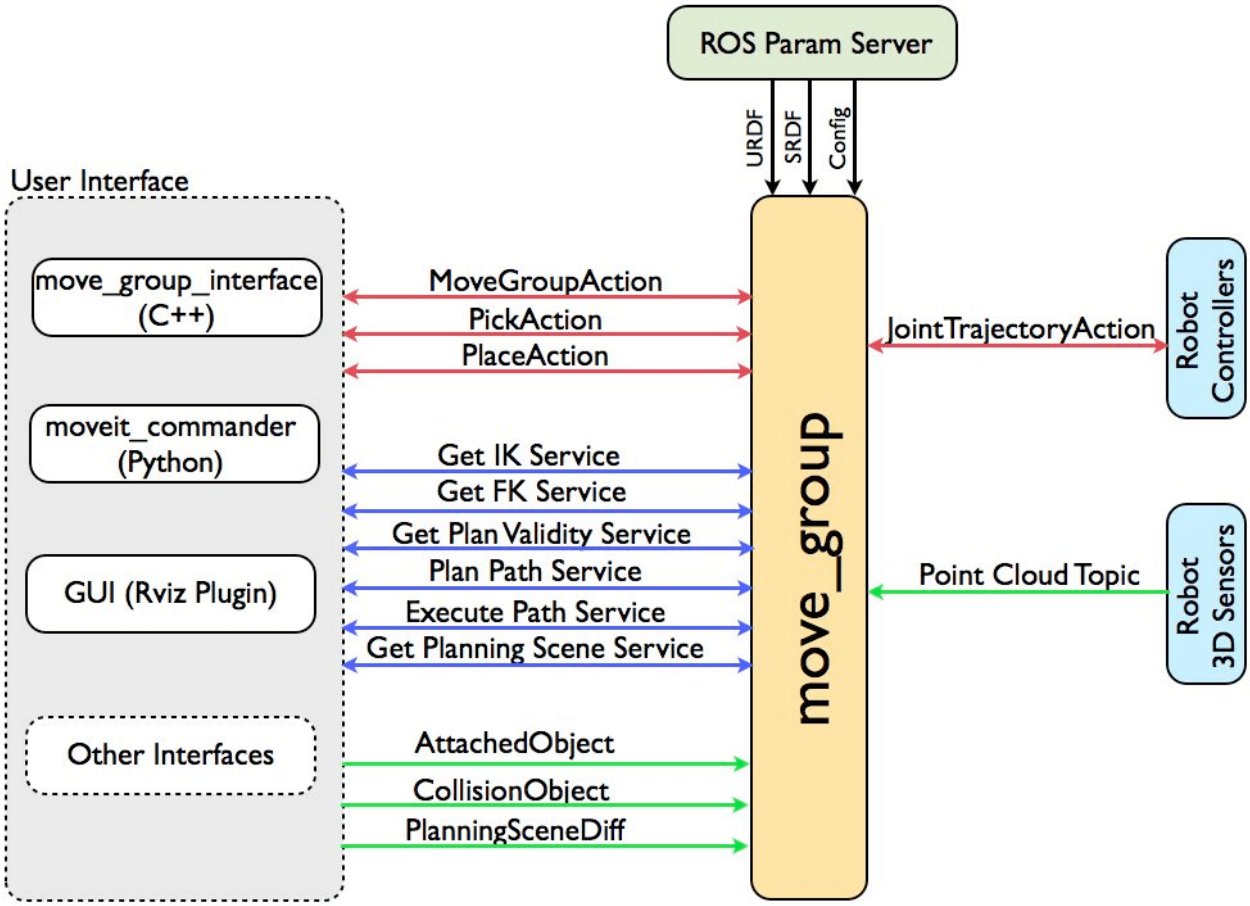
- 180º To mount the UR5e under the top plate.
- 90º offset to match the orientation of the real robot.
- **Virtual boundaries**
  In order to maintain the execution of the robot within the limits of the box it resides in, limits got created defining the perimeter of the operation space.

The changes were made on the file: `frame.xacro`.



Ur5e control using python

We will command our robot by making use of the ROS node created by MoveIt called `move_group`. We can interact with this node making use of multiple interfaces like RViz which we already know how to use. The python interface is just as another way to send commands to the `move_group` node



Preliminary steps

Requirements:

- Catkin workspace.
- MoveIt package and dependences.

From here we can start by creating our new package inside the `src` folder of our workspace with its dependencies.

```
catkin_create_pkg NAME rospy
```

Inside our newly created package `src` folder we will create our python files and modify its permission.

```
chmod +x file.py
```

From here we can start programming our python file as usual.
Its recommended to use **virtual environments** to contain the necessary dependencies. Here the introduction to pip Virtual environments.

Basics of a ROSPY program.

General dependencies

```
import sys
import rospy
import moveit_commander
import geometry_msgs
import moveit_msgs
```

Basic instillation

```
#Commander interface creation
moveit_commander.roscpp_initialize(sys.argv)
#Creation of node and its name
rospy.init_node('ur5e_current_pose', anonymous=True)

#Get robot and scene which the moveit_vcommander will control
robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()

#Set MoveIt arm group
group_name = "ur5e_arm"
move_group = moveit_commander.MoveGroupCommander(group_name)
```

Scene building

Scenes allows us to keep an accurate representation of the environment in which the robot operates. There is a multitude of ways to define set scenes either in a dynamic or static way, but we will cover the construction and update of set scenes through ROSPY scenes.

Dependences

```
import sys
import rospy
import moveit_commander
import geometry_msgs
import moveit_msgs
from moveit_msgs.msg import CollisionObject
from shape_msgs.msg import SolidPrimitive
from geometry_msgs.msg import Pose
```

Creation of the object: Creation of the newly introduce object to the scene

```
object = CollisionObject()
object.id = newObjectId
```

Assignation of reference frame: Assignation of reference frame which needs to already exist on the scene

```
object.header.frame_id = referenceFrameId
```

Definition of size and properties: given by (X,Y,Z) in metersc

```
solid.dimensions = dimensions
object.primitives = [solid]
```

Position of the object: In reference to the coordinate origin of the reference frame.

```
object_pose = Pose()
object_pose.position.x = pose[0]#X
object_pose.position.y = pose[1]#Y
object_pose.position.z = pose[2]#Z
object.primitive_poses = [object_pose]
```

Addition of object to scene

```
object.operation = object.ADD
scene.add_object(object)
```

Last attempt to understand Move It gripper control

Starting point

Warning of missing `gripper_finger_joint`
Error when executing movement:

```
[ERROR] [1688555983.379748096]: Unable to identify any set of controllers that
can actuate the specified joints: [ gripper_finger_joint ]

[ERROR] [1688555983.379879457]: Known controllers and their joints:
controller '/pos_joint_traj_controller' controls joints:
  elbow_joint
  shoulder_lift_joint
  shoulder_pan_joint
  wrist_1_joint
```

```
    wrist_2_joint
    wrist_3_joint
```

We see that is calling `pos_joint_traj_controller`
Search on the workspace yields result in the `Universal_Robots_ROS_Driver/ur_robot_driver/config/ur5e_controllers.yaml`
After checking the file we can see a variable define called: `robot_joints`
Variable definition on hardware interface:

```
ur_hardware_interface:
   joints: &robot_joints
     - shoulder_pan_joint
     - shoulder_lift_joint
     - elbow_joint
     - wrist_1_joint
     - wrist_2_joint
     - wrist_3_joint
```

Test: added `gripper_finger_joint` to the list of joints
results: After modification there is no change on the errors, which leaves us thinking ur5e_controllers.yaml is the wrong file.

Test 2: added `gripper_finger_joint` to list of joints in: `universal_robot/ur5e_moveit_config/config/ros_controlers.yaml`
result 2: no change.

Conclusion of tests: The error was not found, but it was isolated that it only occurs when running RViz with the real robot, which pints us to the `UR_Driver`.

New question

How is the UR driver called from the launched?
The UR driver is launch from other terminal using the file `ur5e_bringup.launch`, in the resulting chain:

`ur5e_bringup.launch`
`arg controller_config_file = ur5e_controllers.yaml`
--> `ur_common.launch` --> `ur_control.launch`

This chain load the controllers for the hardware of the `ur5e`.
Maybe there is something similar in the `robotiq package`.
Which gripper is ours? --> `robotiq_2f_85_gripper`

Expected pipeline
Robotic controller with must be installed only `**external-control** and **rs485** urcap`

```
roslaunch ur_robot_driver ur5e_bringup.launch robot_ip:=10.62.1.41
kinematics_config:="/home/bicrobotics/ROS-MoveIt-
UR5e/ur5_ws/src/my_robot_calibration.yaml" use_tool_communication:=true
tool_device_name:=/tmp/ttyUR
```

Start external control on robotic controller

```
rosrun robotiq_2f_gripper_control Robotiq2FGripperRtuNode.py /tmp/ttyUR
```

Error:

```
  File "/home/bicrobotics/ROS-MoveIt-
UR5e/ur5_ws/src/robotiq/robotiq_modbus_rtu/src/robotiq_modbus_rtu/comModbusRtu.py
", line 97, in getStatus
    output.append((response.getRegister(i) & 0xFF00) >> 8)
AttributeError: 'ModbusIOException' object has no attribute 'getRegister'
```

After researching the error extensively no solution was find in both ROS forums or Robotiq archive. Most solution pointing to the possible workaround of connecting the gripper directly to the ROS machine instead of the wrist connector: USB Gripper control

Other approaches

Approach 1:

Connection to the TCP port of the gripper using python.
Solution source
This solution tries to use the following code:

```
#Library importation

import socket

#Socket setings

HOST="10.1.0.50" #replace by the IP address of the UR robot

PORT=63352 #PORT used by robotiq gripper


#Socket communication

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:

    #open the socket

    s.connect((HOST, PORT))
```

```
    s.sendall(b'GET POS\n')

    data = s.recv(2**10)


#Print finger position
#Gripper finger position is between 0 (Full open) and 255 (Full close)

print('Gripper finger position is: ', data)

#################################################################
```

The IP used for connection is the one used in the `robot_bring_up` script.
The port is the default one: 54321. This connection has been tested with the `telent ip PORT`.

A form of return is produce by the script when we connect to the robot whit bout URCaps installed: `rs485`, `Robotiq_grippers`. the output in inconclusive.

Here below are some possible commands. Some commands can be use with both GET and SET:

```
**ACT:** Activation bit

0 - Gripper not activated

1 - Gripper activated

**GTO:** 1 if the gripper is set to move to requested position 0 if gripper is
set to stay at the same place

**PRE:** Position request eco. Should be same a the requested position if

the gripper successfully received the requested position.

**POS:** Current position of the gripper

**SPE:** Speed eco. Should be same as requested speed.

**FOR:** Force parameter of the gripper

**OBJ:** Object grippings status

0 - Fingers are inmotion towards requested position.No object detected.

1 - Fingers have stopped due to a contact while opening before requested
```

position.Object detected opening.

2 - Fingers have stopped due to a contact while closing before requested position.Object detected closing.

3 - Fingers are at requested position.No object detected or object has been loss / dropped.

STA: Gripper status, returns the current status & motion of theGripper fingers.

0 -Gripper is in reset (or automatic release)state. See Fault Status if Gripper is activated.

1 - Activation in progress.

2 - Not used.

3 - Activation is completed.

MOD: ...

FLT: Fault status returns general errormessages that are useful for troubleshooting. Fault LED (red) is present on theGripper chassis,

LED can be blue, red or both and be solid or blinking.

0 - No fault (LED is blue)

Priority faults (LED is blue)

5 - Action delayed, activation (reactivation)must be completed prior to performing the action.

7 - The activation bit must be set prior to action.

Minor faults (LED continuous red)

8 -Maximum operating temperature exceeded,wait for cool-down.

9 No communication during at least 1 second.

Major faults (LED blinking red/blue) - Reset is required (rising edge on activation bit rACT needed).

10 - Underminimum operating voltage.

```
11- Automatic release in progress.

12- Internal fault; contact support@robotiq.com.

13 - Activation fault, verify that no interference or other error occurred.

14-Overcurrent triggered.

15- Automatic release completed.

**MSC:** Gripper maximym current.

**COU:** Gripper current.

**NCY:** Number of cycles performed by the gripper

**DST:** Gripper driver state

0 - Gripper Driver State : RQ_STATE_INIT

1 - Gripper Driver State : RQ_STATE_LISTEN

2 - Gripper Driver State : Q_STATE_READ_INFO

3 - Gripper Driver State : RQ_STATE_ACTIVATION

Other - Gripper Driver State : RQ_STATE_RUN

**PCO:** Gripper connection state

0 - Gripper Connection State : No connection problem detected

Other - Gripper Connection State : Connection problem detected
```

Approach 2

Modbus-RTU Com
No connection since there is not `/dev/ttyUSB1` since this may be for the USB connection alternative.

Change to `/tmp/ttyURC` No communication achieved

Studied links:

- https://dof.robotiq.com/discussion/2291/send-and-get-data-from-ur-robot-and-control-robotiq-grippers-via-rtde-using-python-3-ur-rtde#latest
- https://dof.robotiq.com/discussion/1962/programming-options-ur16e-2f-85#latest
- https://dof.robotiq.com/discussion/92/controlling-the-robotiq-2-finger-gripper-with-modbus-commands-in-python?_ga=2.17431289.1133307955.1542044765-1547192274.1535646624
- https://dof.robotiq.com/discussion/1382/control-grippers-from-python-using-modbus-rtu
- https://wiki-ros-org.translate.goog/robotiq/Tutorials/Control%20of%20a%202-Finger%20Gripper%20using%20the%20Modbus%20RTU%20protocol%20%28ros%20kinetic%20and%20newer%20releases%29?_x_tr_sch=http&_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=sc

Solution !

After trial and error mixed with lots of reading, the connection and control to the gripper was achieved.
Steps to success:

- Touch control pad should:
  - Have External control URCaps installed
  - Have `rs485` URCaps installed
  - Uninstalled the gripper control URCaps.
  - In the installation tab --> Tools I/O --> `Controlled by: User`
- This should leave the gripper wrist light Red.
- Launch the full command to initiate the connection to the robot as usual:

```
roslaunch ur_robot_driver ur5e_bringup.launch robot_ip:=10.62.1.41
kinematics_config:="/home/bicrobotics/ROS-MoveIt-
UR5e/ur5_ws/src/my_robot_calibration.yaml" use_tool_communication:=true
tool_voltage:=24 tool_parity:=0 tool_baud_rate:=115200 tool_stop_bits:=1
tool_rx_idle_chars:=1.5 tool_tx_idle_chars:=3.5 tool_device_name:=/tmp/ttyURC
```

- This should create a directory at the `/tmp` folder called `ttyURC`.
- In another terminal, after sourcing the workspace and activating the virtual environment we should run the gripper `RtuNode`.

```
rosrun robotiq_2f_gripper_control Robotiq2FGripperRtuNode.py /tmp/ttyURC
```

- This command returns NO value but turn the gripper wrist light blue.
- Run external control at the robots touch pad controller.
- In another terminal, after sourcing the workspace and activating the virtual environment we run the grippe control test:

```
rosrun robotiq_2f_gripper_control Robotiq2FGripperSimpleController.py
```

- Which as an output should give us the different options to interact with the gripper.
- Recommended to first Reset and then Activate the gripper. (Some times it could take some time to run first instruction)

Additional comments on the fucntionoing
Using advance gripper control abstraction the gripper presentes a wraning

```
SyntaxWarning: The publisher should be created with an explicit keyword argument
'queue_size'. Please see
http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers for more
information.
  pub = rospy.Publisher('Robotiq2FGripperRobotOutput',
outputMsg.Robotiq2FGripper_robot_output)
```

This can be solve by addtiokgn a qie size to the `rospy.Publisher` in line 19 of `robotiq_2f_gripper_ctrl.py` in the robotiq pacakge

Additionally some time conection wthi cht advance control are not avaliable
for this after trying to connect first with `Robotiq2FGripperSimpleController.py` the conecytion then is possible with the advence control.
Solution:
Modification of `gripper_control.py` from `robotiq_2f_gripper_ctr`
Import line 1
lines 12 to 17

```
from robotiq_2f_gripper_control.msg import _Robotiq2FGripper_robot_output  as
_outputMsg

from robotiq_2f_gripper_control.msg import Robotiq2FGripper_robot_input as
inputMsg

class RobotiqCGripper(object):
    def __init__(self):
        self.cur_status = None
        self.status_sub = rospy.Subscriber('Robotiq2FGripperRobotInput',
inputMsg,
                                           self._status_cb)
        self.cmd_pub = rospy.Publisher('Robotiq2FGripperRobotOutput',
outputMsg,queue_size=20)

        command = _outputMsg.Robotiq2FGripper_robot_output();
        command.rACT = 1
        command.rGTO = 1
        command.rSP  = 255
        command.rFR  = 150
        self.cmd_pub.publish(command)
```