LAB02 - Direct Kinematics

Marco Camurri

In this Lab, you will learn:

- how to visualize a robot model using the Unified Robot Description Format (URDF)
- how to compute the direct (or forward) kinematics of a 4-DoF serial manipulator and visualize it using the RViz software

To complete the Lab, you need the following open-source pieces of software:

- A Python interpreter (version 3.8)[1]
- Robot Operating System (ROS) Noetic[2]
- Pinocchio library, for Rigid Body Dynamics Algorithms[3]
- Visual Studio Code[4]

The goal of this assignment is to simulate the behavior of a 4-DoF anthropomorphic robot manipulator under zero-torque at the joints (i.e., free-falling under the effect of gravity, which is the only force acting on the robot).

First, we will learn how to describe a robot using the URDF file format, so it can be visualized with the RViz tool[5]. Second, we will develop the Python code to compute the forward and inverse kinematics functions of our robot manipulator. Finally, we will design a simple trajectory based on polynomials for our robot to follow.

Disclaimer

This lab is based on the exercises and lectures by Michele Focchi and Octavio Villarreal available here. The code used for these exercises is derivative of the Locosim framework, available here.

[1] https://docs.python.org/3.8
[2] https://www.ros.org/
[3] https://github.com/stack-of-tasks/pinocchio
[4] https://code.visualstudio.com/
[5] https://dl.acm.org/doi/abs/10.1007/s11235-015-0034-5

1 Preliminaries

This lab assumes you have a working setup with Ubuntu 20.04 and ROS Noetic.

Before continuing, make sure that this line appears at the end of your `.bashrc` file:

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

1.1 Install Visual Studio Code

Visual Studio Code (VSC) is a popular Integrated Development Environment (IDE), designed to be extensible and multi-language. When looking and editing complex Python files, we will use VSC instead of `nano` or `vim`.

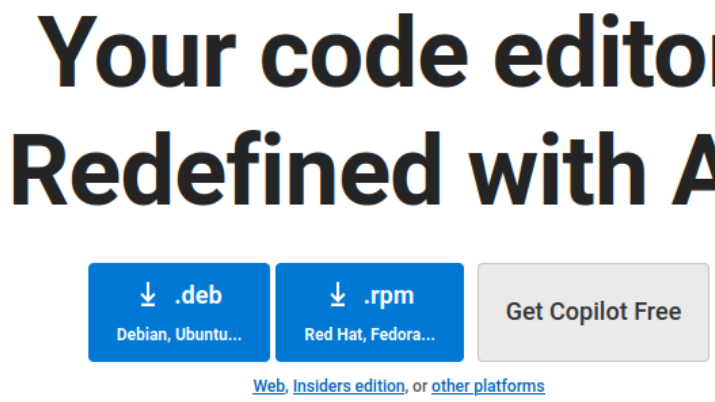
1.1.1 On Windows with WSL

If you are on Windows and using WSL, follow the instructions for WSL available at the following address: code.visualstudio.com/docs/remote/wsl

1.1.2 On Ubuntu

If you are using Ubuntu directly, you can download the `*.deb` file from code.visualstudio.com and install it manually with the `dpkg` command.

First, click on the `.deb` button and choose where you want to install it.



Then, open up a terminal where you downloaded the file and install it by running the following command. For example, if the file name is `code_1.98.2-1741788907_amd64.deb` and it is stored in `~/Downloads` you can do:

```
cd ~/Downloads
sudo dpkg -i code_1.98.2-1741788907_amd64.deb
```

1.1.3 On Mac

If you are using a Mac computer, you are likely to be using a virtual machine running Ubuntu already. Just follow the same instructions for Ubuntu from within the virtual machine itself.

1.2 Setting up Pinocchio

Let's now install the Pinocchio library, following the instructions from the official documentation available here:

1.2.1 Add robotpkg apt repository

1. Ensure you have some required installation dependencies

   ```
   sudo apt install -qqy lsb-release curl
   ```

2. Register the authentication certificate of robotpkg:

   ```
   sudo mkdir -p /etc/apt/keyrings
   curl
       http://robotpkg.openrobots.org/packages/debian/robotpkg.asc
       | sudo tee /etc/apt/keyrings/robotpkg.asc
   ```

3. Add robotpkg as source repository to apt:

   ```
   echo "deb [arch=amd64 signed-by=/etc/apt/keyrings/robotpkg.asc]
       http://robotpkg.openrobots.org/packages/debian/pub
       $(lsb_release -cs) robotpkg" | sudo tee
       /etc/apt/sources.list.d/robotpkg.list
   ```

4. You need to run at least once apt update to fetch the package descriptions:

   ```
   sudo apt update
   ```

1.2.2 Install Pinocchio

The installation of Pinocchio and its dependencies is made through the line:

```
sudo apt install -qqy robotpkg-py38-pinocchio
```

It will install all the systems and additional required dependences.

1.2.3 Configure environment variables

All the packages will be installed in the `/opt/openrobots` directory. To make use of installed libraries and programs, you must need to configure your `PATH`, `PKG_CONFIG_PATH`, `PYTHONPATH` and other similar environment variables to point inside this directory. For instance:

```
export PATH=/opt/openrobots/bin:$PATH
export
    PKG_CONFIG_PATH=/opt/openrobots/lib/pkgconfig:$PKG_CONFIG_PATH
export LD_LIBRARY_PATH=/opt/openrobots/lib:$LD_LIBRARY_PATH
export
    PYTHONPATH=/opt/openrobots/lib/python3.8/site-packages:$PYTHONPATH
    # Adapt your desired python version here
export CMAKE_PREFIX_PATH=/opt/openrobots:$CMAKE_PREFIX_PATH
```

You may directly add those lines to your `$HOME/.bashrc` for a persistent configuration.

1.3 Configuring the ROS workspace

Now that we have Pinocchio installed, let's create a new ROS workspace with `catkin`. First, we create some empty directories in your home folder:

```
cd ~
mkdir itr_ws
cd itr_ws
mkdir src
```

Now, ce can initialize the ROS workspace using the `catkin` command. If the command is not available, you can install it with `apt`:

```
sudo apt install python3-catkin-tools
```

assuming now we are inside the `itr_ws` directory, we can init, configure, and build the workspace:

```
catkin init
catkin config --cmake-args -DCMAKE_BUILD_TYPE=Release
catkin build
```

the output should now look something like this:

```
Profile:                     default
Extending:             [env] /opt/ros/noetic
Workspace:                   /home/mcamurri/itr_ws
----------------------------------------------------------------
Build Space:        [exists] /home/mcamurri/itr_ws/build
Devel Space:        [exists] /home/mcamurri/itr_ws/devel
```

```
Install Space:      [unused] /home/mcamurri/itr_ws/install
Log Space:         [missing] /home/mcamurri/itr_ws/logs
Source Space:       [exists] /home/mcamurri/itr_ws/src
DESTDIR:            [unused] None
------------------------------------------------------------
Devel Space Layout:          linked
Install Space Layout:        None
------------------------------------------------------------
Additional CMake Args:       -DCMAKE_BUILD_TYPE=Release
Additional Make Args:        None
Additional catkin Make Args: None
Internal Make Job Server:    True
Cache Job Environments:      False
------------------------------------------------------------
Buildlisted Packages:        None
Skiplisted Packages:         None
------------------------------------------------------------
Workspace configuration appears valid.

NOTE: Forcing CMake to run for each package.
------------------------------------------------------------
[build] No packages were found in the source space
    '/home/mcamurri/itr_ws/src'
[build] No packages to be built.
[build] Package table is up to date.
Starting  >>> catkin_tools_prebuild
Finished  <<< catkin_tools_prebuild                [ 1.9 seconds
    ]
[build] Summary: All 1 packages succeeded!
[build]   Ignored:   None.
[build]   Warnings:  None.
[build]   Abandoned: None.
[build]   Failed:    None.
[build] Runtime: 2.0 seconds total.
```

to tell our system that we should use this catkin workspace, we need to add this line at the bottom of our `.bashrc` file:

```
source $HOME/itr_ws/devel/setup.bash
```

once the file has been edited, close and reopen your terminal for this edit to take effect.

1.4 Install the labs package with `git`

This and other labs will be done interactively by executing Python code and ROS. Since the code gets frequently updated and improved, we will use `git` to

get and update the lab files. Don't worry, you don't need to a pro git user to do this, just follow the simple commands below.

If you don't have `git` installed, you can use `apt` as usual:

```
sudo apt install git
```

If it is the first time you get the files, you need to clone the repository inside your workspace:

```
cd ~/itr_ws/src
git clone https://github.com/idra-lab/intro_robotics_labs.git
```

if you cloned it already, just update the software:

```
git pull origin main
```

if the pull fails because you have edited some files, make a backup copy of the files and try this:

```
git stash
git pull origin main
git stash pop
```

if the above fails and you are not yet familiar with git, erase the folder and clone it again, and restore the files you backed up.

If for some reason `git` doesn't work for you, you can also download a zip file with the code from `github.com`. Just navigate to github.com/idra-lab/intro_robotics_labs and click on the green button, selecting the `Download ZIP` option:

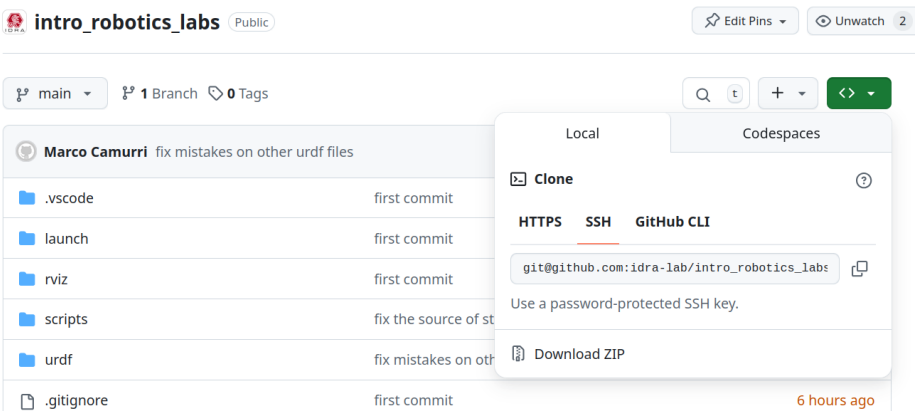


Figure 1: alt text

Note that any new version of the code will require to download it again.

Finally, we can build the lab's package:

```
cd ~/itr_ws
catkin build
```

1.5 Robot URDF description file and visualization

In this part of the lab we visualize a simple 4-DoF manipulator based on the real UR5 robot.

To visualize the robot correctly, you will need the official repository for the UR5, made available by ROS:

```
sudo apt install ros-noetic-ur-description
```

1.5.1 Basic geometry description

Navigate the folder containing the robot descriptions of our manipulator by typing the following command on the terminal:

```
cd ~/itr_ws/src/intro_robotics_labs/urdf
```

Let's start by inspecting a file that describe a robot made of regular geometries (a box and a cylinder). Type on the terminal:

```
nano simple_geometry.urdf
```

As mentioned, a robot is described by a `*.urdf` file, which is an XML-based format.

For large robot models, an XML macro language (and command) called `xacro` (XML Macro) can be used to simplify the creation of a URDF file. The macros provided by `xacro` can be used to parametrize the sub-components of a robot. For instance, with `xacro` you can define a macro describing a single leg of a quadruped robot and re-use it, instead of creating four copies of nearly the same code.

In this lab, we will not use `xacro` and we will limit ourselves to inspect the example files in the folder.

Inspect the `simple_geometry.urdf` file and try to understand what each line describes:

- How are the links described?
- Which parameters need to be described per link?
- How are joints defined? [6].

[6] For an in-depth explanation on how ROS works and check this useful book: Mastering ROS for Robotics Programming, Lentin Joseph, pag. 61.

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="link_name">
    <inertial> ... </inertial>
    <visual> ... </visual>
    <collision> ... </collision>
</link>
```

The `<inertial>` tag defines the mass, the location of the center of mass and the inertia tensor of the link (about the CoM). These parameters are usually obtained by CAD.

The `<visual>` tag describes how the link of the robot graphically appears in simulation. This description can make use of simple shapes or mesh files, such as `*.stl` or `*.dae`. The visual appearance of a link is not used by the simulator to compute the interactions with the environment, such as contacts.

The `<collision>` tag describes the geometry of the link that physically interacts with the simulated world via contacts. Since contacts are computationally expensive, the geometry of a collision is normally a simplified version of the one used for the `<visual>` tag and defined in a way that it "surrounds" it (e.g., as convex hull).

> **WARNING**
> If you don't define a geometry for the collision of your robot, it will penetrate everything when simulating it

> **TIP**
> When defining the collision for each link of the robot, you might want to avoid self-collisions when the robot moves. There are specific options in the simulator to avoid that.

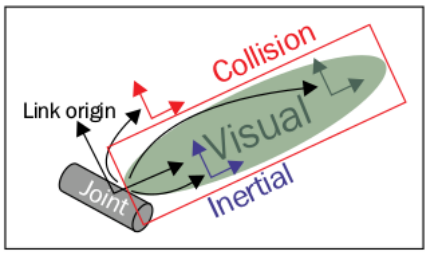


Figure 2: Visualization of a URDF link

The `<joint>` tag represents a robot joint. In addition to the `name` it also has a mandatory argument `type` which should specify the type of joint (one of `fixed`, `revolute`, `prismatic`, `floating`, `continuous`, `planar`).

A joint is always defined as a connection between two links, `parent` and `child`, which should be defined before the joint element connecting them.

The syntax is as follows:

```
<joint name="joint_name" type="joint_type">
  <parent link="link1"/>
  <child link="link2"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <axis xyz="0 0 1"/>
  <limit effort="5" />
</joint>
```

In addition to the names of the two links, we can specify the kinematics of the robot by setting where the child link is respect to the parent with the `<origin>` tag representing a *rigid transform* as two triplets for the position and Euler angles (in radians).

In general, the procedure to define a robot is as follows:

1. from a link frame i with a rigid transform we find the frame of the next joint $(i+1)$.
2. If the joint variable is *zero*, the joint $i+1$ frame is coincident with the link frame $i+1$ that the joint is moving.
3. If the joint moves, then the link $i+1$ frame and the joint $i+1$ frame no longer coincide and are linked by an additional transformation (e.g. a pure elementary rotation about the joint axis in the case of a revolute joint [7]) due to the joint motion.

Note that the joint axis is defined with respect to the parent link (e.g. i) not with respect to the world frame!

We can also set the limits of the joint movement and its velocity and the effort (i.e. force or torque). The following is an illustration of a joint and its link:

In this picture the frame supporting the *child* link is coincident with the joint frame so it means that the joint variable is 0. The position / orientation of the joint frame is expressed via a rigid transform (with the tag origin) with respect to the *parent* link frame.

1.5.2 Using an stl mesh file to describe a link.

We can use `*.stl` files to specify the geometry of the link instead of using basic geometries such as a cilinder or a block. The `.stl` format is a common extension used in CAD software such as SolidWorks to store rigid parts of a mechanical design. However, when using `.stl` files, one needs to be careful to not use an excessive number of polygons to describe the part [8], since this might cause the simulation to slow down significantly.

[7] In the case of a revolute joint the origin of the joint and link frame always coincide.
[8] People use Meshlab to reduce mesh complexity from a detailed CAD

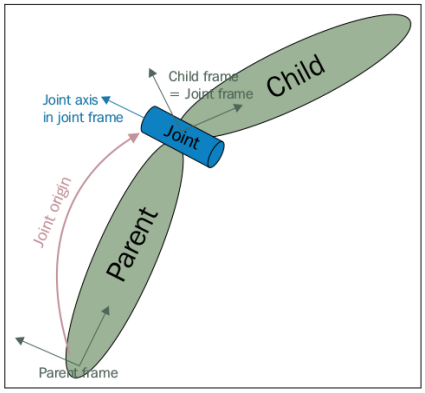


Figure 3: Visualization of a URDF joint.

The file contains to links associated to a mesh stl file. Inspect the by typing in the terminal (you can either open a new terminal or kill the process in the previous one by hitting)

```
nano ur4_2links.urdf
```

You will notice that the file contains more details and parameters per link and joint. What can they represent? Where are these values obtained from? Are they referred to a specific reference frame? Inspect each line of the file and try to understand the function of each of the tags. Check how the rigid (homogeneous) transform is set for the `shoulder_pan_joint` with respect to the supporting `base_link`. Check also how the dynamic parameters (mass, Center of Mass and inertia tensor) are defined for the link, with respect to the link frame.

1.5.3 Visualizing a model described by a URDF file in `RViz`

The basic unit of development in ROS is the node. A node is nothing else than a process that performs computations. It can be created in multiple languages and interfaces such as Python and C++. Nodes communicate through messages by either using a publisher-subscriber (synchronous)[9] or a request-reply (asynchronous)[10] scheme.

In this lab session. we will use the former one dedicated function to publish the state of the joints during runtime and make them available to the visualizer node running RViz. One can run each node separately using the terminal command [11], however, a more convenient way is to setup a launch file and run multiple nodes with only with terminal command (namely, [12]). Let us inspect the launch

[9] http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python)
[10] http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(python)
[11] http://wiki.ros.org/rosbash#rosrun
[12] http://wiki.ros.org/roslaunch

file to visualize our examples of URDF files.

In a terminal type:

```
cd ~/itr_ws/src/intro_robotics_labs/launch
nano visualize.launch
```

Once the file has been inspected, run the following command in the terminal to load the URDF of the robot:

```
roslaunch intro_robotics_labs visualize.launch
    robot_name:=simple_geometry
```

ou can play around with the value of the only joint `shoulder_pan_joint` present in the model, using the slider input in the GUI. You can also visualize the link frames, adding the TF plugin in RViz. To stop the process, press `CTRL + C` in the terminal where you ran the launch file.

Now, let's visualize a one-joint two-link manipulator by typing the following command:

```
roslaunch intro_robotics_labs visualize.launch
    robot_name:=ur4_2links
```

Again, you will visualize the described model and you can modify the value of the only joint to make the shoulder link move. Note that until now we were only *visualizing* a predefined joint motion, note that we would need to add a *trasmission* to the joint if we want to be able to control it in a simulator (i.e. sending torques to it).

To finish this exercise, let's visualize the 4-DoF robot manipulator that we will use in the next two parts of this lab session and visualize its joint motions. Stop the process in the terminal and then run the following command:

```
roslaunch intro_robotics_labs visualize.launch robot_name:=ur4
```

Note that we could have not specified the robot name since is the default value of the argument . This will launch RVIZ and it will allow us to move around all four joints of our manipulator.

1.6 ROS topics

So far we have only mentioned that a node can publish a message, but where is this message *published to*?

Messages are published to *topics*, and this makes them available to other nodes that can *subscribe* to a specific topic. In our case, our launch file `visualize.launch` runs a node called `joint_state_publisher_gui` which is in charge, as its name suggest, of publish the joint values on to the topic `/joint_states`. With the publisher, it also launches a gui that allows us to

manually set the `joint_states` topic. Pay attention that usually this will come directly from the robot or the simulator.

Now, let's check the content of the topic. Run again (in case you have stopped it) the command from the previous step and in a separate terminal run the following command:

```
rostopic echo /joint_states
```

This will give a series of information about the topic, including the name and value for each of the joints of our manipulator. Change the values using the GUI and verify that the values of the joints are changing. In essence, the `joint_state_publiser` node is publishing the value of the joints on to the `joint_states` topic and the node running RViz subscribes to this topic to display it in the visualizer.

You can check the communication tree running the command `rqt_graph`. The `robot_state_publisher` node instead computes the homogeneous transforms TF (i.e. the direct kinematics) for all the links, making RViz able to properly visualize the robot.

2 Direct kinematics

Once the model visualization is finished, we now proceed to derive and analyze the robot kinematics. In this part, we will manually compute the direct (or forward) kinematics computations and verify their correctness using the built-in Pinocchio functions.

We will perform these computations using Python and Visual Studio Code (VSC).

Open VSC and select the `intro_robotics_labs` inside the `itr_ws` workspace.

To verify that everything is working properly, you can run the `L2_joint_space_control.py` tutorial from a terminal:

```
rosrun intro_robotics_labs L2_joint_space_control.py
```

You should see the manipulator going up and down. This file was just for testing and we will use that in the next lab session.

For this part of the assignment, we will use the following files instead:

- `L1_1_kinematics.py` (Main file for this and later exercises)
- `L1_conf.py` (Initializations of variables and simulaion parameters)
- `utils/kin_dyn_utils.py` (Kinematics and dynamics functions)

Inspect these files in detail. In this assignment you will try to write your own kinematics functions of the file

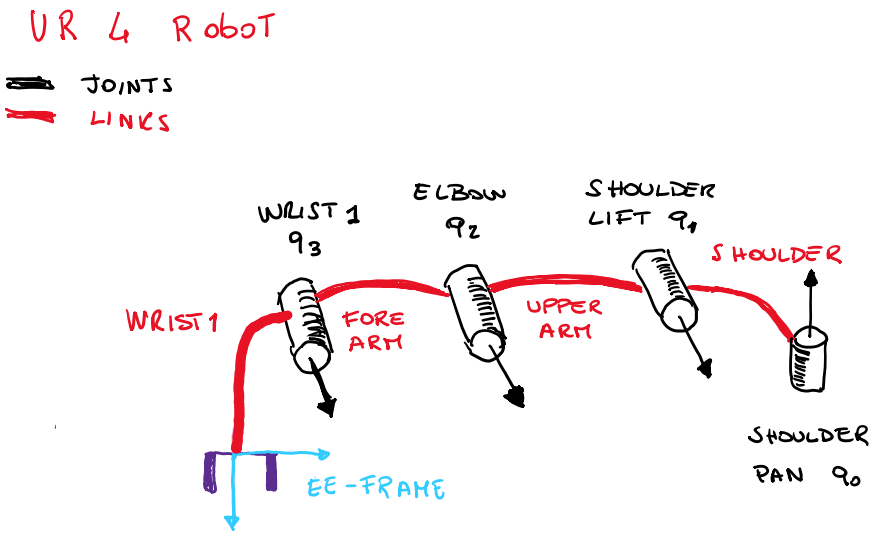


Figure 4: Sketch of the UR4 robot kinematics

2.1 Direct kinematics (lecture E1).

The first step is to obtain the homogeneous transformation matrix from the world frame to the end-effector ${}_{\mathbf{W}}\mathbf{T}_e$. To do so, you will need to obtain the homogeneous transformation matrices from one link to the other (i.e., the local transformations ${}_{\mathbf{W}}\mathbf{T}_1$, ${}_{\mathbf{1}}\mathbf{T}_2$, ${}_{\mathbf{2}}\mathbf{T}_3$, ${}_{\mathbf{3}}\mathbf{T}_4$ and ${}_{\mathbf{4}}\mathbf{T}_e$) and perform the omposition of matrices. Modify the function `directKinematics` inside `kin_dyn_utils.py` to perform this computation.

> **Hint:** you can use the previously shown visualization tool to understand the position and orientation of the different frames (remember to set joint positions to zero). You can alternative refer to Fig. 3 for a sketch of the kinematics and to Fig. 4 to check the locations of the link frames.

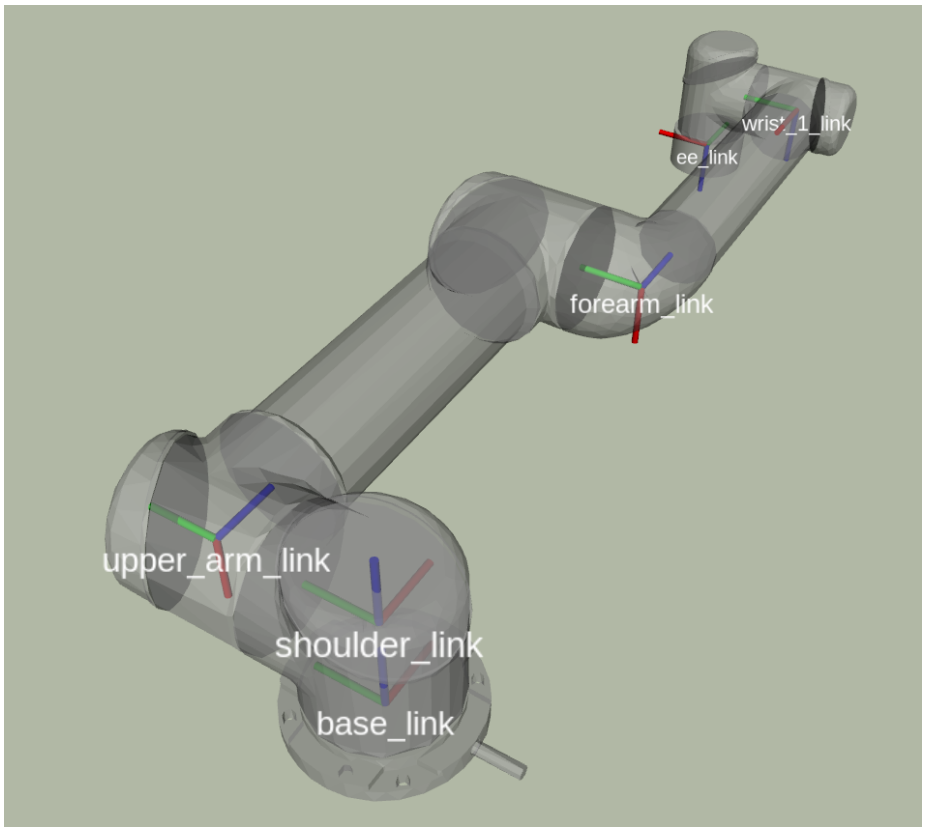


Figure 5: UR4 Robot: link frames locations

Call the function inside the package `kin_dyn_utils.py` from the main in the file. To verify the correctness of your direct kinematics function, compare the outputs with the ones from the built-in functions from Pinocchio for the

position vector `robot.framePlacement(q, frame_ee).translation` and the rotation matrix (`robot.framePlacement(q, frame_ee).rotation` (as seen in the file `L1_1_kinematics.py`) using different values for the positions of the joints (changing `q0` in `L1_conf.py`).