


Giraffe-Robot

 GitHub Repository

Matteo Grisenti

August 2025

1 Assignment

The project consists of designing a giraffe robot able to place a microphone in front of a person sitting inside a small conference room. The robot is attached to the ceiling of the room, located at its center. The room is $4m$ high, and the robot should be able to reach locations at $1m$ height in a $5 \times 12m$ area.

The robot must have 5 degrees of freedom: one spherical joint at the base (2 revolute joints with intersecting axes), one prismatic joint able to achieve a long extension, and two revolute joints to properly orient the microphone (not necessarily intersecting).

The goal is to position the microphone at any point in the $5 \times 12m$ conference room, with a specific pitch orientation of 30° with respect to the horizontal. This defines a 4D task.

2 Workplan

The project can be decomposed into the following steps:

- **URDF Design:** Design the URDF model of the robot, choosing the link dimensions and placing the frames.
- **Kinematics:** Compute the forward and differential kinematics of the end-effector.
- **Dynamics:** Use Pinocchio RNEA native function to create a simulator of the robot motion.
- **Polynomial Trajectory:** Plan a polynomial trajectory (in task space) to move from a hom-

ing configuration \mathbf{q}_{home} to a desired end-effector pose $(\mathbf{p}_{\text{des}}, \theta_{\text{des}})$.

- **Computed Torque Controller:** Write an inverse-dynamics control action in task space.
- **Gains Tune:** Tune the PD gains of the Cartesian controller (on the linearized system) to achieve a settling time of 7s without overshoot.
- **Secondary Task:** Exploit the null space to minimize the distance to a given configuration \mathbf{q}_0 of your choice.
- Simulate the robot to reach the location $\mathbf{p}_{\text{des}} = [1, 2, 1]^\top$ from the homing configuration $\mathbf{q}_{\text{home}} = [0, 0, 0, 0]^\top$.

3 URDF Design

The Giraffe-Robot has been modeled in the file `urdf/giraffe.urdf`, according to the project requirements.

- **Base Link:** Following the assignment instructions, the base link of the robot has been placed on the ceiling ($z = 4$), and positioned in the middle of the room ($x = 2.5, y = 6$). The base frame has been rotated by π so that the z -axis points downward, and by an additional $\pi/2$ around the z -axis so that the arm develops to the right.
- **Spherical Shoulder:** Implemented as two revolute joints (yaw and pitch) with intersecting axes. Each joint is represented by a cylindrical shape, oriented according to the axis of rotation.

- **Arm:** The arm has a length of 2.4 m. Since the room width is 5 m, the arm must not exceed 2.5 m, otherwise it would collide with the walls during rotation. The chosen length of 2.4 m leaves some margin to accommodate the following links.
- **Prismatic Joint:** Attached at the end of the arm, it provides extension along its axis. The extendable link has a maximum length of 4 m, which represents a trade-off: it must be long enough to reach distant points in the room, but not excessively long in order to avoid hitting the ceiling when the shoulder rotates.
- **Spherical Wrist:** Realized with two revolute joints (yaw and roll). Since the assignment specifies that these axes need not be intersecting, a wrist link of 1 m was introduced. This additional length allows the robot to better reach the desired heights.
- **Microphone:** At the end of the kinematic chain, a microphone of 15 cm length is mounted. The end-effector frame is placed at its tip, ensuring precise positioning and orientation in task space.

At the end of the report (see page X) several RViz screenshots are provided, showing the full robot structure and reference frames as defined in the URDF model.

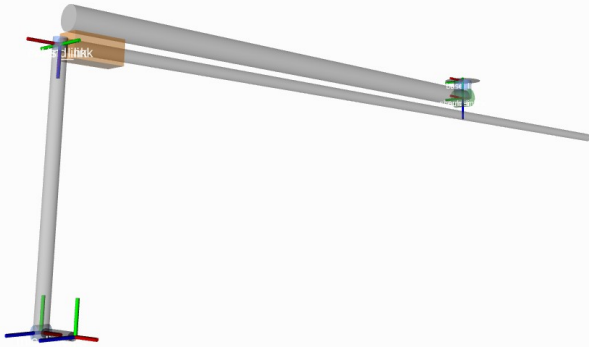


Figure 1: RViz Giraffe Robot

NB: The file `giraffe.urdf` has been created using Xacro. This is useful because it allows the use of variables, which are essential during the development of the robot structure. However, for use with the Pinocchio library, this version is not in the correct format. Therefore, it needs to be processed using the xacro command: to get the file `urdf/giraffe_processed.urdf` in the correct format for Pinocchio.

4 Kinematics

In the file `scripts/kinematics/kinematics.py`, the functions `directKinematics()` and `differentKinematics()` have been defined:

- `directKinematics()` takes as input the joint configuration, computes the homogeneous transformation matrix for each link, and then, using the chain rule, derives the global homogeneous transformation for each link.
- `differentKinematics()` takes as input the joint configuration and, for each link, computes the position vector and the rotation axis vector (\mathbf{z}). Based on these, it derives the contribution of each joint to the end-effector velocity.

Both functions have been tested in the file `scripts/kinematics/test_custom_kinematics.py` by comparing the results with those obtained from the Pinocchio library. Both functions achieve errors that are approximately zero.

5 Dynamics

In the file `scripts/dynamics/dynamics.py`, the function `forwardDynamics()` has been defined. It takes as input the joint configuration and velocity, the joint forces (τ), and the Pinocchio model and data of the robot.

The function uses Pinocchio's RNEA function to compute the gravity term (g), the Coriolis term (C), and the inertia matrix (M). It then solves the equation

$$\ddot{q} = M^{-1}(\tau - (C + g)).$$

To test the function, a test suite has been created in `scripts/dynamics/test.py`. Starting from arbitrary values of \mathbf{q} , $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ the torque τ is computed using Pinocchio's RNEA function. Then, \mathbf{q} , $\dot{\mathbf{q}}$, and τ are given as input to the simulation function `forwardDynamics()`, which returns the simulated acceleration $\ddot{q}_{\text{computed}}$. By comparing \ddot{q} and $\ddot{q}_{\text{computed}}$, the error is found to be approximately zero.

6 Polynomial Trajectory

The polynomial trajectory generator is implemented in `scripts/polynomial_trajectory/polynomial_trajectory.py` through the function `task_domain_polynomial_trajectory()`.

Since the requirement was to generate a trajectory in the task domain, the input is a desired pose ($\mathbf{p}_{\text{des}}, \theta_{\text{des}}$). This must first be converted into a desired joint configuration \mathbf{q}_{des} through inverse kinematics.

For this purpose, a numerical inverse kinematics function has been implemented and tested in `scripts/kinematics/kinematics.py` and `scripts/kinematics/test_custom_kinematics.py`.

Given the initial configuration \mathbf{q}_0 and the desired configuration \mathbf{q}_{des} , the generator computes the coefficients of a fifth-order polynomial with a default trajectory time of 5s, discretized into 10 steps. These coefficients are then used to iteratively update \mathbf{q} , $\dot{\mathbf{q}}$, and $\ddot{\mathbf{q}}$ at each step.

Instead of publishing the results directly on a ROS topic, an offline implementation was chosen. The trajectory data of each step is stored in a JSON file: `trajectory_data.json`.

This file is later read during the tests in `scripts/polynomial_trajectory/test.py`. To publish the joint configurations, the original `joint_state` ROS node was split into two nodes:

- `joint_state_publisher_gui`: contains a buffer with one joints configurations (\mathbf{q} , $\dot{\mathbf{q}}$, and $\ddot{\mathbf{q}}$),

which is published on the topic `joint_state` and read by RViz. These values can be updated by publishing new joint configurations on the topic `discrete_joint_states`, which this node subscribes to.

- `polynomial_trajectory_node`: reads the data from `trajectory_data.json` and publishes the joint configurations of each step on the topic `discrete_joint_states`.

During testing, a wait time was defined between consecutive steps in order to visualize and clearly distinguish the trajectory in RViz.

Finally, the error between the desired task pose (position and pitch) and the reached one was computed, and it can be considered approximately zero.

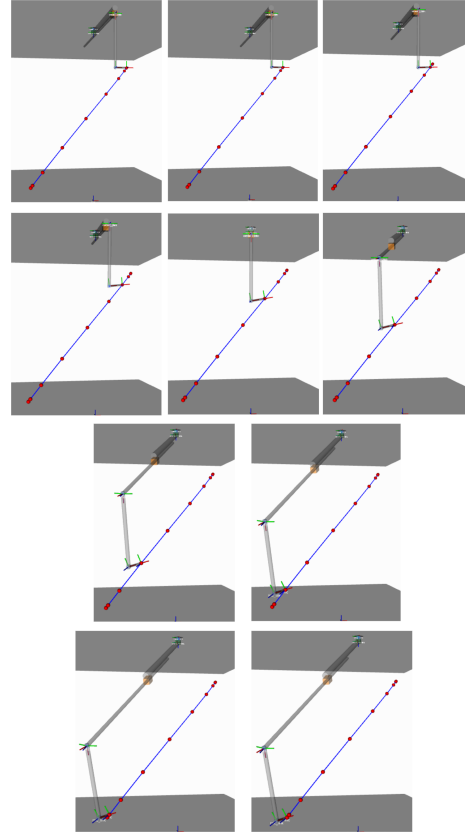


Figure 2: RViz Giraffe Robot

7 Computed Torque Controller 8 Gains Tuning

The task-space inverse dynamics controller is implemented in `scripts/computed_torque_control/computed_torque_control.py` through the function `task_space_torque_control()`. The controller is designed for a 4D task space composed of the Cartesian position $[x, y, z]$ and the pitch angle of the microphone.

The function takes as input the current joint configuration \mathbf{q} , joint velocities $\dot{\mathbf{q}}$, the task feedback $(\mathbf{x}, \dot{\mathbf{x}})$, and the desired task reference $(\mathbf{x}_{\text{des}}, \dot{\mathbf{x}}_{\text{des}}, \ddot{\mathbf{x}}_{\text{des}})$. The proportional and derivative gains K_p, K_d can be specified, otherwise default diagonal matrices are used.

First, the analytic Jacobian of the end-effector frame is computed from the Pinocchio model. From this Jacobian, a reduced task Jacobian J_x is built by selecting the translational part and the Euler rate corresponding to the pitch. The position and velocity errors are then combined with the desired acceleration to generate the commanded task-space acceleration in a computed-torque fashion:

$$\ddot{\mathbf{x}}_{\text{cmd}} = \ddot{\mathbf{x}}_{\text{des}} + K_d (\dot{\mathbf{x}}_{\text{des}} - \dot{\mathbf{x}}) + K_p (\mathbf{x}_{\text{des}} - \mathbf{x}).$$

The dynamics are computed using the inertia matrix M and bias term h obtained from Pinocchio. The task-space inertia matrix $A = J_x M^{-1} J_x^\top$ is then built and regularized if needed to avoid numerical singularities. The corresponding task-space mass matrix $\Lambda = A^{-1}$ is used to compute the desired task force

$$\mathbf{F}_x = \Lambda \ddot{\mathbf{x}}_{\text{cmd}} + \mu,$$

where μ is the task-space bias compensating for Coriolis, gravity, and the effect of $\dot{J}\dot{\mathbf{q}}$, estimated using Pinocchio's function `getFrameJacobianTimeVariation`.

Finally, the task force is mapped to joint torques

$$\boldsymbol{\tau} = J_x^\top \mathbf{F}_x,$$

which can be optionally clipped to respect joint torque limits.

This controller allows the robot to track the desired Cartesian trajectory with a prescribed pitch orientation, achieving stable regulation in the 4D task space.

The tuning of the PD gains was carried out empirically by testing different values. Starting from relatively large gains, it was observed that the system reached the desired task pose too quickly, with aggressive transients. Therefore, the gains were progressively reduced until the motion successfully spanned the required 7s settling time.

The final gains used in the implementation are:

$$K_p = \text{diag}([3.5, 3.5, 3.5, 8.0])$$

$$K_d = \text{diag}(2 \cdot [1.2, 1.2, 1.2, 0.25] \odot \sqrt{\text{diag}(K_p)})$$

However, during the tuning attempts it was not possible to identify a set of gains that completely avoided overshoot in all four task dimensions simultaneously, while at the same time maintaining satisfactory steady-state accuracy in each dimension.

9 Secondary Task

The controller has been further extended with the integration of a secondary task in the null space of the main task. Specifically, we aim to minimize the distance with respect to a given joint configuration q_0 , by generating a joint-space torque contribution

$$\tau_q = K_q (q_0 - q) - D_q \dot{q},$$

where K_q and D_q are the proportional and derivative gains in the joint space, respectively.

To ensure that this secondary action does not interfere with the task-space control, the torque is projected through the null-space projector

$$N = I - J_x^\top \Lambda J_x M^{-1},$$

The final torque command is therefore expressed as

$$\boldsymbol{\tau} = \boldsymbol{\tau}_{\text{task}} + N \boldsymbol{\tau}_q.$$

In our implementation, the desired posture q_0 was chosen as the initial configuration of the motion. In this way, the controller tends to achieve the task while minimizing unnecessary robot motion.

The results show that the primary task is successfully achieved even in the presence of this secondary control. Moreover, the tracking error is further reduced, probably due to the suppression of redundant free motions of the robot that could otherwise generate noise during task execution.

RVIZ GIRAFFE ROBOT SCREENS

