

AMORO Lab Part 2: Simulation of a five-bar mechanism

Damien SIX
damien.six@ls2n.fr

Released: August 20, 2021



Figure 1: The DexTAR robot (Mecademic)

1 Objective

The main objective of the present lab is to compute the geometric, kinematic and dynamic models of a five-bar mechanism and to compare them with the results obtained with GAZEBO. Then, a controller will be designed to track a trajectory in simulation.

The kinematic architecture of the five-bar mechanism is shown in Fig.2. For the mechanism of the Gazebo mock-up, the geometric parameters are:

- Bar length (all bars are equal length): $l = 0.09$ m
- Distance between the two active joints: $d = 0.118$ m

and the base dynamic parameters are:

- ZZ_{1R} the grouped inertia on the first link of the left arm.
 $ZZ_{1R}=0.002$ kg.m²

- ZZ_{2R} , the grouped inertia on the first link of the right arm.
 $ZZ_{2R}=0.002 \text{ kg.m}^2$
- m_R the grouped mass on the end-effector. $m_R = 0.5 \text{ kg}$

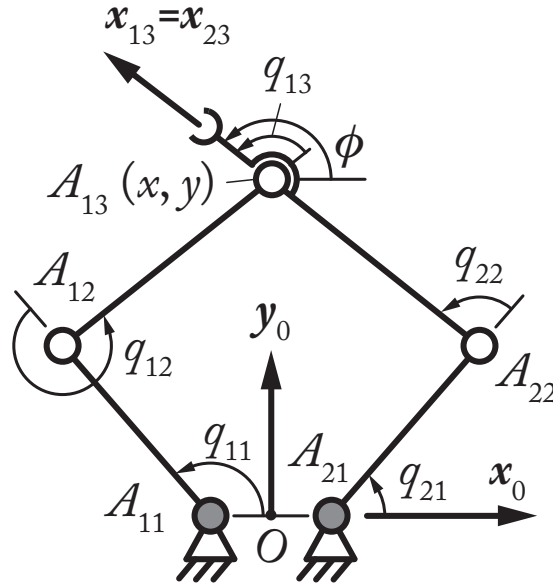


Figure 2: Kinematic architecture of the five-bar mechanism. Joints located at A11 and A21 are actuated.

2 Help for the LAB

Some computations required for this lab are given in the course and recalled in the appendix *Computation of the five-bar mechanism models*. This help is provided for the Five-Bar. For the Biglide mechanism (Part 3 of the Lab), a similar document is a **deliverable** from the lab. To avoid, loosing time for Part 3, it is **mandatory** to prepare this document **BEFORE** getting to Part 3 during the lab. As a consequence, this should be done **AT HOME** and not during the lab sessions.

3 Kinematic models of the five-bar mechanism

In the first part of this lab, we will simulate the kinematic behavior of the five-bar mechanism and compare it with GAZEBO output. To run the lab, execute the following steps:

1. Run Gazebo using the command `ros2 launch lab_amoro gazebo.launch.py`
2. Insert the appropriate model by selecting the *insert* tab then the *FiveBar* model.

You can now go in the *student_scripts* folder and complete the models in the file *five_bar_models.py*. You should not modify the input-output of the functions but just put the appropriate computation. To check that the models are properly computed, you must run the *model_test.py* script for the direct models and *inverse_model_test.py* for the inverse ones. To run the python scripts, just go in the *student_scripts* using the terminal and use the command

```
$ python3 your_script.py
```

All the following models must be validated

- The Direct Geometric Model
- The Direct Geometric Model for passive joints
- The Inverse Geometric Model
- The First Order Kinematic Model
- The First Order Kinematic Model for passive joints
- The Inverse First Order Kinematic Model
- The Second Order Kinematic Model
- The Second Order Kinematic Model for passive joints
- The Inverse Second Order Kinematic Model

Python math and numpy libraries:

To code the computation of your models. It is recommended to use the *math* and *numpy* libraries. *math* will contain the trigonometric functions you need and *numpy* allows the manipulation of arrays and matrices. *numpy* is usually imported under the alias *np*. The following methods will be handy.

- `v = np.array([0.0, 1.0])` to create a 2D vector **v**
- `M = np.array([[0.0, 1.0], [2.0, 3.0]])` to create a 2D matrix **M**
- `u.dot(v)` for the dot product of two vectors
- `M.dot(v)` for the multiplication for the product **Mv**
- `np.matmul(M, N)` for the matrix multiplication **MN**
- `np.linalg.inv(M)` for the matrix inversion
- `M.transpose()` for the matrix transpose

4 Computed torque control

4.1 Trajectory generation

Recall: A polynomial trajectory is defined with initials conditions and final conditions (eventually intermediate). The number of initial and final conditions will determine the order of the polynomial used for the trajectory (number of conditions -1).

As an example, displacement in x with initial and final positions and velocities (4 conditions) requires a 3rd order polynomial function.

$$x = a_1 t^3 + a_2 t^2 + a_3 t + a_4$$

The initials and final conditions are expressed as

$$\begin{aligned} x(t_i) &= a_1 t_i^3 + a_2 t_i^2 + a_3 t_i + a_4 \\ \dot{x}(t_i) &= 3a_1 t_i^2 + 2a_2 t_i + a_3 \\ x(t_f) &= a_1 t_f^3 + a_2 t_f^2 + a_3 t_f + a_4 \\ \dot{x}(t_f) &= 3a_1 t_f^2 + 2a_2 t_f + a_3 \end{aligned}$$

which can be put in the form of a linear system

$$\mathbf{P}\mathbf{a} = \mathbf{c}$$

with $\mathbf{a} = (a_1, a_2, a_3, a_4)^T$ containing the coefficients of the polynomial function. Solving this linear system gives the polynomial coefficients for the trajectory.

A trajectory between two points $A(x_A, y_A)$ and $B(x_B, y_B)$, with null velocity and acceleration at initial and final position, in the time interval $[0, t_f]$ is given by a fifth order polynomial function. This specific case can be expressed as:

$$\begin{aligned} x(t) &= x_A + s(t)(x_B - x_A) \\ y(t) &= y_A + s(t)(y_B - y_A) \end{aligned}$$

with

$$s(t) = 10 \left(\frac{t}{t_f} \right)^3 - 15 \left(\frac{t}{t_f} \right)^4 + 6 \left(\frac{t}{t_f} \right)^5$$

To do: In the *control.py* file, write down a function to compute the trajectory in Cartesian space as a numpy array given some initial position, final position and duration. The function should output position, velocity and acceleration for each coordinate. The trajectory should be sampled with a 10 ms sampling time.

Write down a second function to compute the trajectory (position, velocity, acceleration) in joint space from Cartesian space using the inverse geometric and kinematic models.

4.2 Introduction to Computed Torque Control

The Computed Torque Control is based on the feedback linearization of the system through the inverse dynamic model. The dynamic model of a parallel robot can be written under the generic form

$$\boldsymbol{\tau} = \mathbf{M}\ddot{\mathbf{q}}_a + \mathbf{c}$$

\mathbf{M} being the inertia matrix, definite positive and \mathbf{c} the vectore of Coriolis and Centrifugal effects. Considering the input $\boldsymbol{\tau}$ this system is non-linear, an auxiliary control input is defined $\boldsymbol{\alpha}$

$$\boldsymbol{\alpha} = \mathbf{M}^{-1}(\boldsymbol{\tau} - \mathbf{c})$$

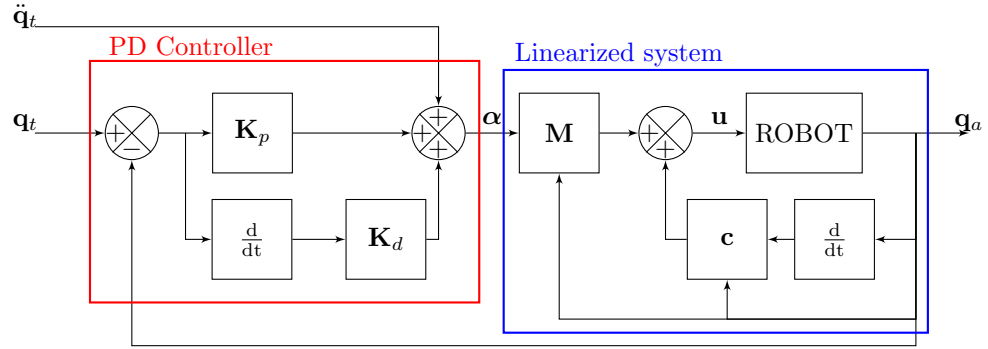


Figure 3: Computed Torque Control Scheme

If the dynamic model is accurate, this auxiliary input correspond to the robot acceleration.

$$\alpha = \ddot{q}_a$$

A PD control law can be applied on this auxiliary input

$$\alpha = \ddot{q}_t + K_d(\dot{q}_t - \dot{q}_a) + K_p(q_t - q_a)$$

With K_p et K_d definite positive (usually positive diagonal matrices). The torque input is deduced from this auxiliary control law

$$\begin{aligned} \tau &= M\alpha + c \\ &= M(\ddot{q}_t + K_d(\dot{q}_t - \dot{q}_a) + K_p(q_t - q_a)) + c \end{aligned}$$

Fig.3 shows the controller scheme. With $\tilde{q} = q_{ref} - q_a$, the closed-loop equation of the control system is

$$\ddot{\tilde{q}} + K_d\dot{\tilde{q}} + K_p\tilde{q} = 0$$

Ensuring the convergence of the error \tilde{q} toward 0.

4.3 Computed Torque Control simulation

- Using the Robot class input and output, code in *control.py* a computed torque control to track a desired trajectory.
- Define a trajectory between the initial position (0.09,0.06796322) and the position (0,0.1) in 2s.
- Run the simulation and check the trajectory tracking performed.
- Define a trajectory between the initial position (0.09,0.06796322) and the position (0.05, 0.0) in 4.0s. What happens? Explain why (hint, plot the joint effort you ask)