Institute for Cognitive Systems
Technische Universität München
M.Sc. Simon Armleder

Modeling and control of legged robots
Tutorial 3: Control

1 Introduction

In this tutorial we explore the basics of robotics control. In particular we will take a look the problem of tracking trajectories in joint and Cartesian space.

In the lecture have seen that *Feedback cancellation* can help us to compensate the non-linearities inside the robot's dynamic model

$$\mathbf{M}(\mathbf{q})\,\ddot{\mathbf{q}} + \mathbf{h}(\mathbf{q}, \dot{\mathbf{q}}) = \boldsymbol{\tau} \tag{1}$$

by transforming the control input with

$$\boldsymbol{\tau} = \mathbf{M}(\mathbf{q})\,\mathbf{y} + \mathbf{h}(\mathbf{q}, \dot{\mathbf{q}}) \tag{2}$$

Here, $\mathbf{y}$ is the new virtual control input of the linearized system. If the compensation is perfect (state and model are known) the closed loop dynamics of Eq. (1) and Eq. (2) simply becomes

$$\ddot{\mathbf{q}} = \mathbf{y} \tag{3}$$

Methods from linear control theory (e.g. pole placement) can be used to embed desired behaviors into the robot. E.g. We have seen that the control law

$$\boldsymbol{\tau} = \mathbf{M}(\mathbf{q})\left[\ddot{\mathbf{q}}^r - \mathbf{K}_d\left(\dot{\mathbf{q}} - \dot{\mathbf{q}}^r\right) - \mathbf{K}_p\left(\mathbf{q} - \mathbf{q}^r\right)\right] + \mathbf{h}(\mathbf{q}, \dot{\mathbf{q}}) \tag{4}$$

builds on this idea to achieve accurate tracking in joint space.
A similar feedback law exists for Cartesian space control. The first and second order derivative of the robot's forward kinematics $\boldsymbol{f}_{\text{fk}}(\mathbf{q})$ give the differential mappings between the two spaces:

$$\dot{\boldsymbol{X}} = \mathbf{J}\dot{\mathbf{q}} \tag{5}$$

$$\ddot{\boldsymbol{X}} = \mathbf{J}\ddot{\mathbf{q}} + \dot{\mathbf{J}}\dot{\mathbf{q}} \tag{6}$$

The tracking law then becomes

$$\boldsymbol{\tau} = \mathbf{M}\mathbf{J}^{\#}\left(\ddot{\boldsymbol{X}}^d - \dot{\mathbf{J}}\mathbf{q}\right) + \mathbf{h} \tag{7}$$

$$\ddot{\boldsymbol{X}}^d = \ddot{\boldsymbol{X}}^r - \mathbf{K}_d\left(\dot{\boldsymbol{X}} - \dot{\mathbf{X}}^r\right) - \mathbf{K}_p\left(\boldsymbol{X} \ominus \boldsymbol{X}^r\right) \tag{8}$$

where $\ominus$ denotes the difference operator of the specific Lie-Group (e.g. S^3 in case of Quaternions, $SE(3)$ in case of homogeneous Transformations, etc.). It expressed the difference as a vector in the flat euclidean tangential space $\mathcal{T}_{\mathcal{M}}$ and can be used as feedback signal. The operation is called the Logarithmic Map.

Institute for Cognitive Systems
Technische Universität München
M.Sc. Simon Armleder

2 Homework

You should build on the solution of the previous tutorial. The goal is to apply the two control methods from above to the humanoid robot `talos` and learn something about another important concept in robot control called *Nullspace Projection*.

a Exercise 1: Create a Robot Class

- Write a new class that derives from `Robot` (see file `simulator.robot`) and loads the `Talos` urdf

  ```
  wrapper = pin.RobotWrapper.BuildFromURDF(...)
  ```

  Save the `pin.RobotWrapper` object, as it gives you access to many robot specific functions E.g. the dynamic model `wrapper.model` or the data structure `wrapper.data`.

- Since we didn't talk about balancing so far, set the robot to be `fixed based`. For this: don't add a `pin.JointModelFreeFlyer()` joint when loading the urdf and set `useFixedBase=True` when calling the base constructor. You can check if it works by printing the size of `wrapper.model.nq`.

- Add a `JointState` publisher to the class and a method called `publish`. This should output the joint state to ROS.

- Finally, overload the `update` function of the baseclass and add

  ```
  wrapper.forwardKinematics(...)
  ```

  to update the kinematics of pinocchio's model in every iteration.

- Test your code, by running the robot model and check if you can visualize the robot in RViz. Load the robot with all joint in their zero position.

For more information about the `pin.RobotWrapper` check here.

b Exercise 2: Create a Joint space Controller

- Create a new class `JointSpaceController` with constructor arguments like `(robot, Kp, Kd)`

- Create an `update` function that takes the reference `q_r, q_r_dot, q_r_ddot` and computes the controller in Eq. (4). It should return the new torque command τ

- modifying the main function of your script. Instantiate the controller, call its update function. Feed the torque to

  ```
  robot.setActuatedJointTorques(tau)
  ```

- Test the controller by moving the robot's arms from their starting configuration to a homing configuration (See previous tutorial). To create a joint spline and its first and second order derivatives you can use any other python library (e.g. scipy or ndcurves). Set the spline duration to 5 seconds.

To compute the required quantities of the dynamic model check out pinocchio's functions here. Note the Inertia Matrix can be computed with the function `crba(...)`

Institute for Cognitive Systems
Technische Universität München
M.Sc. Simon Armleder

c Exercise 3: Create a Cartesian space Controller

Now that we can visualize and move the robot to a nice homing position lets try to control one of its hands.

- Create a new class `CartesianSpaceController` with constructor arguments like `(robot, joint_name, Kp, Kd)` Here `joint_name` refers to a joint frame name in the kinematic chain that we want to control.
- Create update function that takes the reference `X_r, X_dot_r, X_ddot_r`. Here `X_r` is of type `pin.SE(3)`, pinocchio's type to express a position and rotation in space. `X_dot_r, X_ddot_r` are velocity and acceleration in $\mathbb{R}^6$.
- Now implement the control law in Eq. (7). As a starting point take a look at this example here, which show how to do inverse kinematics control. We are interested in inverse dynamics control which is a little bit more complicated:
  1. First, get the joint id from the joint name via

     ```
     id = robot.model().getJointId(joint_name)
     ```

  2. Compute the Joint Jacobian as show in the linked example.
  3. Get the current Cartesian pose and velocity of your controlled frame with

     ```
     X = data.oMi[id] # holds a list of all transformations
     X_dot = J@q_dot
     ```

  4. Compute the Cartesian desired acc $\ddot{\mathbf{X}}^d$ using the logarithmic Map: `pin.log()`
  5. Compute the term $\dot{\mathbf{J}}\dot{\mathbf{q}}$. In pinocchio this available through the function: `pin.getClassicalAcceleration(...)`
  6. Map $\ddot{\mathbf{X}}^d$ to $\ddot{\mathbf{q}}^d$, you can use the same method as in the linked example. They compute $\mathbf{J}^{\#} = \mathbf{J}^{\mathsf{T}} \left(\mathbf{J}\mathbf{J}^{\mathsf{T}} + \lambda \mathbf{I} \right)^{-1}$, which is known as the damped Jacobian.
  7. Finally pull this result through the dynamic model and return the control torques.
- Test your implementation by modifying your main function. First execute the joint spline as before. When it finishes switch to Cartesian space control.
  1. We will control the right hand: `joint_name="arm_right_7_joint"`
  2. The moment you switch from joint to Cartesian control, save the initial hand transformation in a variable `X_goal`
  3. Now set the targets for the controller to be like:

     ```
     X_r = self.X_goal
     X_dot_r = np.zeros(6)
     X_ddot_r = np.zeros(6)
     ```

  If everything works the robot should hold the initial pose with its right hand. If there is an error in the implementation it will probability diverge.

d Exercise 4: Set an Interactive Target for the right hand

If the cartesian control is working the next step is to update its target and get the hand moving. For this we will make use of ROS's Interactive Markers. The following steps be implemented in a new python script:

- First checkout the `make6DofMarker` example in above's link. It shows how to create a marker and add controls for each axis of its 6 motion directions (Note: there might be better python examples in the internet).
- Spawn a interactive marker in the `base_link` frame. Execute your script and verify that you can add and see the marker in RVIZ.
- If the marker visualization works, spawn the marker at the robots hand instead of the `base_link` origin. For this create a Tf listner that retrieves the `arm_right_7_joint` pos and orientation. Then create the `InteractiveMarker int_marker` at the retrieved position.
- Now, register a callback function to the marker when you add it to the server with `server.insert(...)`. Inside this callback publish the marker pose as a ROS topic (`geometry_msgs.msg.PoseStamped`)
- In the previous control script add a ros subscriber and subscribe to that topic. Set the new Cartesian goal pose in the callback. Hint: `X_goal = pin.XYZQUATToSE3(...)`
- Test, the two script, by launching: The pybullet simulation, RVIZ, and the interactive marker. In Rviz visualize the marker and start moving it around. Is the robot following it? You can improve the behavior by tuning the gains.

e Exercise 5: Improving the behavior

If everything is alright the right hand should follow the marker. However you may notice that the rest of the robots body is doing very strange movement as well.
The problem is that we have a redundant system with $n_{\text{dof, robot}} = 32 > m_{\text{dof, task}} = 6$. We need to give some task to the unused degrees of freedom. This can be done with the following modification to the control torque:

$$\boldsymbol{\tau} = \boldsymbol{\tau}_{\text{cart}} + \mathbf{N}\boldsymbol{\tau}_{\text{posture}} \tag{9}$$

Here $\boldsymbol{\tau}_{\text{cart}}$ is the Cartesian tracking torque from before (computed from Eq. (7)). The matrix $\mathbf{N} \in \mathbb{R}^{n_q \times n_q}$ is called a projector and allows us to add a second task to your robot. It is computed by:

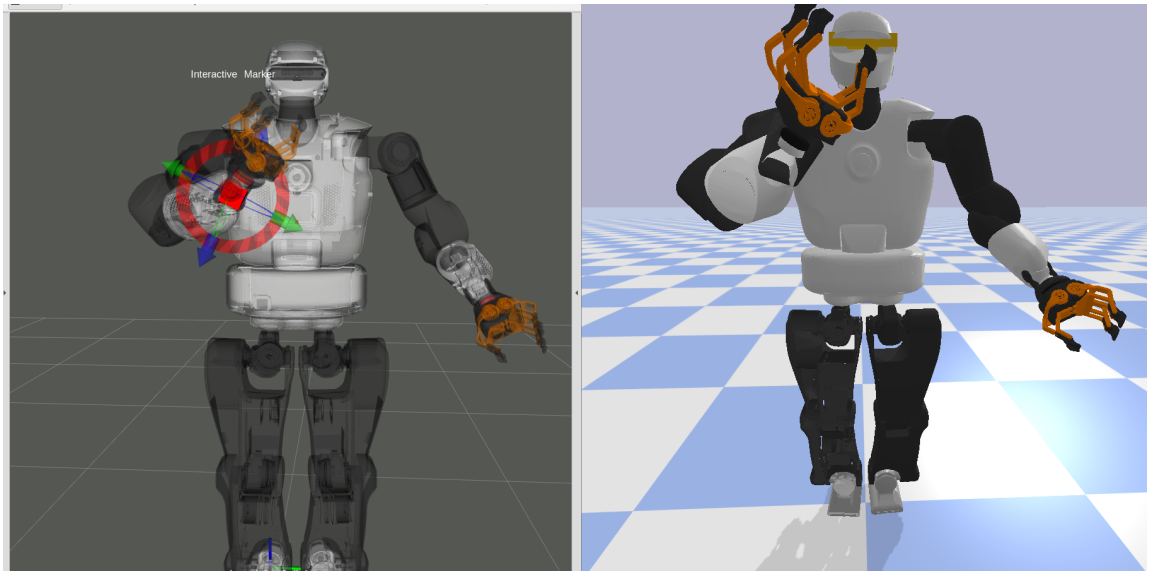
$$\mathbf{N} = \mathbf{I} - \mathbf{J}^{\mathsf{T}} \left(\mathbf{J}^{\#}\right)^{\mathsf{T}} \tag{10}$$

where $\mathbf{J}$ is the Jacobian of the robots hand and $\mathbf{I}$ the identity matrix. Finally, $\mathbf{N}\boldsymbol{\tau}_{\text{posture}}$ is the second task that makes the robot hold its joint space posture:

$$\boldsymbol{\tau}_{\text{posture}} = -\mathbf{K}_p \left(\mathbf{q} - \mathbf{q}^r\right) - \mathbf{K}_d \dot{\mathbf{q}} \tag{11}$$

That will penalize the robot when moving away from its original posture $\mathbf{q}^r$. You can reuse the Joint Controller from before to do the computations (just set zero acceleration and velocity targets).

- Modify the control script to include these improvements during Cartesian space control.
- Test the new control law and verify that the robot stops moving its left arm and legs.



3 Deliver

Deadline for the first **tutorial package (T1, T2, T3)** is on Tuesday the **28.05.23 at 23:59 PM**.

- Implement the tutorials in different script
- Add a readme.md on how to launch the individual tutorials
- Send a zip file of your workspace **src folder** to the lecture email address (**no build/devel folder**)
- Use the naming convention: `<last_name>_deliverable_1.zip`