Modern Robotics practical
EXERCISE 2

Geert Folkertsma, UT-RaM

2017

This final exercise derives a control law for the robot, using earlier results. You can test your code on a simulated version of the robot in Matlab and then, if it seems to work, connect to the real robot to see if you did well!

Refer to drawings and measurements of the robot in the first exercise.

1 End-effector motion

Since the robot can only be controlled kinematically (joint positions or velocity) we shall have to do inverse kinematics. The robot has only three DoF, so the setpoint will really be a point ((x, y, z) location) and not include orientation.

We can define the error vector e as follows:

$$e := p_{\text{sp}} - p_{\text{ee}}, \tag{1}$$

where p_{sp} is the given setpoint and p_{ee} is the current end-effector location, equal to p_3^0, in MATLAB accessible as `H30(1:3,4)`. A sure way to make the robot arm move towards the setpoint is to somehow make the velocity of the end-effector proportional to this error, as in (2). It is easy to verify that the error e will go to 0 with this end-effector motion; at least if p_{sp} doesn't move too quickly.

$$\dot{p}_{\text{ee}} := K_v \cdot e = K_v \cdot (p_{\text{sp}} - p_{\text{ee}}) \tag{2}$$

Now we must find a way to make the robot arm (or its end-effector) execute this velocity $\dot{p}_{\text{ee}}$.

2 Inverse kinematics?

Inverse differential kinematics give a map from a desired end-effector twist $T_e^{0,0}$ to the required joint velocities, $\dot{\boldsymbol{q}}$. Calculating the inverse of a Jacobian, as in (3), is in general problematic in case of singularities—and in our case, the Jacobian is always singular; it is not even square.

$$T_3^{0,0} = J(\boldsymbol{q})\dot{\boldsymbol{q}} \quad \overset{?}{\Rightarrow} \quad \dot{\boldsymbol{q}} = J^{-1}(\boldsymbol{q})T_3^{0,0} \tag{3}$$

With a non-square matrix, we use the Moore-Penrose pseudo-inverse (4), which calculates a $\dot{\boldsymbol{q}}$ that gives the least-squares optimal value of $T_e^{0,0}$.

$$J^\dagger := J^\top \left(J \cdot J^\top \right)^{-1} \tag{4}$$

However, our twist $T_3^{0,0}$ has only a relevant linear velocity v. Were we to set ω to 0, the pseudo-inverse would try to keep ω to 0, which would not necessarily be what we want. Should we set ω to something else, then?

3 Solution

If we describe the end-effector twist in a frame located at the end-effector, the v-part of the twist will indeed be the linear velocity $\dot{p}_{\text{ee}}$, since $r \wedge \omega$ will be zero ($r = 0$). We cannot simply use Ψ_3, because that frame rotates, and our reference position p_{sp} and velocity direction $\dot{p}_{\text{ee}}$ are given in Ψ_0.

The solution[1] is to define a fourth frame, located at the end-effector ($p_3^0 = p_4^0 = p_{\text{ee}}$), but with the same orientation as the base. Clearly, it can be described with:

$$H_4^0 := \begin{pmatrix} I_{3\times3} & p_3^0 \\ 0_{3\times1} & 1 \end{pmatrix}. \tag{5}$$

The end-effector twist described in this frame can be calculated by:

$$T_3^{4,0} = \text{Ad}_{H_0^4} T_3^{0,0} = \text{Ad}_{H_0^4} J(q)\dot{q}. \tag{6}$$

[1] Or at least, one possible solution.

(Note that this is the Adjoint of the *inverse* of H_4^0.) We can now define a new Jacobian that maps directly from $\dot{q}$ to $T_3^{4,0}$:

$$J' := \mathrm{Ad}_{H_0^4} J(q). \tag{7}$$

Again, we're only interested in the v-part, so when taking the pseudo-inverse, we don't want to "least-squares-fit" ω. For that, we can simply only look at the bottom half of this new Jacobian:

$$T_3^{4,0} = J'(q) \cdot \dot{q} \tag{8}$$

$$= \begin{pmatrix} \hat{T}_1' & \hat{T}_2' & \hat{T}_3' \end{pmatrix} \cdot \dot{q} \tag{9}$$

$$\begin{pmatrix} \omega_3^{4,0} \\ v_3^{4,0} \end{pmatrix} = \begin{pmatrix} \hat{\omega}_1' & \hat{\omega}_2' & \hat{\omega}_3' \\ \hat{v}_1' & \hat{v}_2' & \hat{v}_3' \end{pmatrix} \cdot \dot{q}, \tag{10}$$

where $\hat{T}'$, $\hat{\omega}_i'$ and $\hat{v}_i'$ are the columns (and half columns) of J' as defined in (7).

Now we take the bottom half of (10), which again defines a "Jacobian", mapping $\dot{q}$ to $v_3^{4,0}$:

$$v_3^{4,0} = \begin{pmatrix} \hat{v}_1' & \hat{v}_2' & \hat{v}_3' \end{pmatrix} \cdot \dot{q} \tag{11}$$

$$= J_v(q) \cdot q \qquad J_v(q) := \begin{pmatrix} \hat{v}_1' & \hat{v}_2' & \hat{v}_3' \end{pmatrix}. \tag{12}$$

By our choice of Ψ_4, $v_3^{4,0} = \dot{p}_{\text{ee}}$, which is the velocity we want to set according to (2). Therefore, finally, we can take a meaningful pseudo-inverse of our last Jacobian:

$$J_v^\dagger = J_v^\top \left(J_v \cdot J_v^\top \right)^{-1} \tag{13}$$

$$\dot{q}_{\text{set}} = J_v^\dagger \cdot \dot{p}_{\text{ee}} \tag{14}$$

If everything works out as it should, sending this $\dot{q}_{\text{set}}$ of (14) to the robot arm will result in it following the setpoint p_{sp}.

4 Matlab

The supplied MATLAB scripts `robot_control_skeleton.m` handle all things like loop timing, visualisation, simulation of a virtual robot, communication with the real robot, et cetera. You must implement the following functions:

`getHmatrices.m` The one that you made for Exercise 1.

`calculate_qd.m` A new function that, given the current joint positions $\boldsymbol{q}$ and the setpoint p_{set} (both 3-by-1 vectors), plus the robot parameters, calculates the proper joint velocities (a 3-by-1 vector).

In this latter function, you can implement the control law as derived above. You will probably need to call the `getHmatrices` and `getJacobian` functions to calculate J'.

Some notes on how to use the main script, `robot_control_skeleton`:

- At the top of the script, you can enable `realRobotOutput` when you are connected to the real robot. However, first try your scripts with the simulation only, leaving this setting on `false`.
- Sometimes, a laptop is too slow to show the robot animation and communicate with the real robot at the desired 50 Hz. If your control law worked fine in simulation, but the real robot moves very jerkily, try disabling the plot by setting `plotRobot` to `false`.
- You connect to the robot via USB. Inside is an Arduino; MATLAB will need to know the COM-port where it can reach the Arduino. This setting is called `com_port`.
- It may be wise to limit $\dot{p}_{\text{ee}}$ of (2) in your code to an absolute value of 10 cm s^{-1}, with some statement like

```
if norm(pdot)>10
  pdot = 10*(pdot/norm(pdot));
end
```

- The trajectory generated for the setpoint is a figure-8 in the xz-plane. The robot arm starts upright, slowly moves to the start of this trajectory, runs the loop for 10 s and at the end slowly moves straight up again.

Deliverables

In addition to the `getHmatrices` and `getJacobian` functions, you must also hand in your `calculate_qd` function. Please stick to the following function signature:

```
qd = calculate_qd(q, setpoint, L)
```

Where `qd` should provide the joint velocities. The `setpoint` should be the intended end effector position. The `L` argument is a vector of three elements representing the lengths of each link in the robot. The `q` argument is a vector of the current joint positions.

Put all three functions (and any other required functions) into a folder named after your student number. Zip this folder and hand it in.

The zipfile should contain the following:

```
UTs1234567\
UTs1234567\getHmatrices.m
UTs1234567\getJacobian.m
UTs1234567\calculate_qd.m
UTs1234567\any_other_helper_functions.m
```