TU Berlin Robotics WS2018/19

Lab Assignment #2

Version: 5. Nov 2018, 12:00

Please upload your solution by **Sunday November 18, 2018 at 23:59**, using your ISIS account. Remember that this is a hard deadline, extensions are not possible! Uploading the solution once per group is sufficient.

Please follow these **guidelines** (we may deduct points, otherwise):

- Do not change method arguments and predefined names.
- Explain the steps you took, not only the result.
- Write all angles in radians, not in degrees. You may approximate values to 3 significant digits.
- `setJoints` assumes the values to be given in radians as well.
- Simplify your terms, i.e. use trigonometric identities [1] [2]
- In the documentation, use the following abbreviations for trigonometric terms: $s_i = \sin(q_i)$, $c_i = \cos(q_i)$, $s_{ij} = \sin(q_i + q_j)$, $c_{ij} = \cos(q_i + q_j)$, $s_{123} = \sin(q_1 + q_2 + q_3)$, $c_{123} = \cos(q_1 + q_2 + q_3)$. If you use any additional abbreviations, declare them clearly at the beginning of your assignment.

A Forward kinematics (20 points)

Deliverables for this part of the assignment: forwardkinematics.cpp and pdf-file

This assignment deals with the RRR planar manipulator shown in Figure 1. The joints of this manipulator correspond to the joints 2, 3, and 5 of a PUMA 560. You will implement the methods of the class ForwardKinematicsPuma2D in the provided file *forwardkinematics.cpp*.

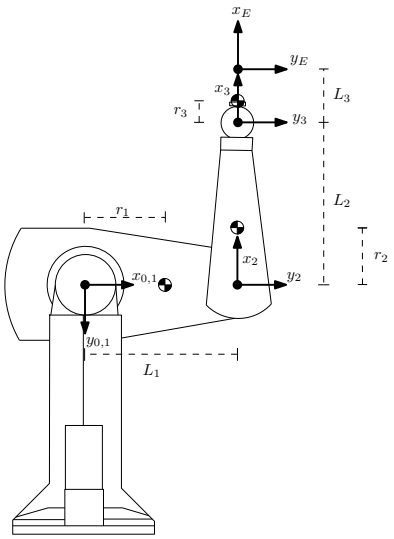

Figure 1: RRR Puma in zero configuration.

[1] http://en.wikipedia.org/wiki/Trigonometric_identity#Shifts_and_periodicity
[2] http://en.wikipedia.org/wiki/Trigonometric_identity#Angle_sum_and_difference_identities

1. Transformation between frames [5 Points]

   (a) Compute the forward kinematics ${}^{0}_{E}T(q)$ for the end effector and the homogenous ransformations between adjacent links (${}^{i+1}_{i}T(q)$) for this manipulator using the DH-parameter definitions from lab assignment 1:

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0	q_1
2	0	L_1	0	$q_2 - \frac{\pi}{2}$
3	0	L_2	0	q_3
$4(E)$	0	L_3	0	0

Table 1: DH-parameters

   Put the computation in the methods named ForwardKinematicsPuma2D::computeTx_x()! Test your functions with different angles to make sure they work! Derive the transformation ${}^{0}_{E}T(q)$ analytically and simplify as far as possible. [3 Points]

   Sanity check: set the joint angles to multiples of $\frac{\pi}{4}$ (e.g. $q = (0, 0, 0)^T$) . Verify that the resulting transformations make sense.

   (b) Comment your methods! Explain in the comments how you derived the equations and which DH-parameters were used. Use a clear language! [2 Points]

2. End effector position in Operational space [5 Points]
   Convert the function ${}^{0}_{E}T(q)$ that returns a homogenous transform matrix into a function returning a vector:

$$F(q) = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$$

   that contains the coordinates (position and orientation angle) of the end-effector in the base frame. Implement the corresponding function ForwardKinematicsPuma2D::computeF()! Explain how you derived the equations in a clear language in comments!
   Sanity check: set the joint angles to multiples of 45° (e.g. $q = (0, 45, 0)^T$). Verify that the resulting vector makes sense.

3. Compute the End Effector Jacobian [5 Points]
   Find the Jacobian $J(q) = \frac{\partial F(q)}{\partial q}$ for the end-effector. Implement it in ForwardKinematicsPuma2D::computeJ(). Explain how you derived the equations in a clear language in comments!
   Sanity check: set the joint angles to multiples of $\frac{\pi}{4}$ (e.g. $q = (0, 0, 0)^T$) . Verify that the resulting matrix makes sense.

4. Understanding the Jacobian Matrix and Pose Singularities [5 Points]
   Each column vector of the Jacobian matrix indicates the translational and rotational velocity in operational space when the related joint moves. The translational part of the end effector velocity can be illustrated as in this figure:

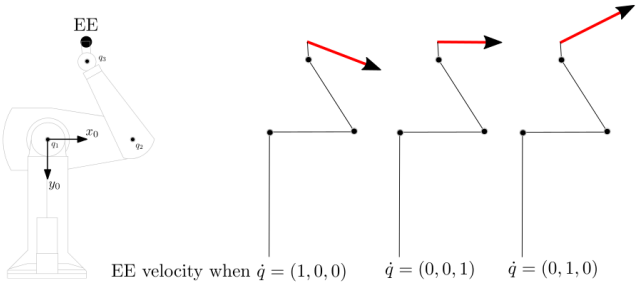

Figure 2: Illustration of translational velocities effected by each joint

(a) Sketch end effector position and the column vectors of the Jacobian for the following configurations: [2 Points]

i. $q_1 = (0, 0, 0)$
ii. $q_2 = (\frac{\pi}{2}, -\frac{\pi}{2}, 0)$
iii. $q_3 = (0, \frac{\pi}{2}, 0.01)$

Sanity check: For rotational joints, the column vector of the Jacobian is always perpendicular to the joint axis, and its magnitude is proportional to the distance of the joint axis

(b) Using your illustrations, determine for each given configurations whether it is close to a singularity, is in a singularity, or is fully controllable in all directions (and orientations). Explain how this can be concluded from your illustrations of the column vectors.[3 Points]

B Trajectory Generation in Joint Space (40 points)

Deliverables for this part of the assignment: control.cpp and pdf-file

1. `[10 Points]` THEORETICAL: GENERATION OF SMOOTH TRAJECTORIES WITH POLYNOMIAL SPLINES

   We want to move the robot from zero configuration $q_a = (0, 0, 0)^T$ to the joint configuration $q_c = \left(-\frac{\pi}{2}, \frac{\pi}{4}, 0\right)^T$within 5 seconds. After half the time, the manipulator should pass this intermediate point:

   $$q_b = (-\frac{\pi}{4}, \frac{\pi}{2}, 0)^T$$

   The end effector velocity at this via point is not defined. We apply a simple heuristic to choose joint velocities (which can also be found in the Craig textbook): If direction of velocity changes for a joint, its desired velocity at the via point is set to 0. Otherwise we set it to the average velocity of the two adjacent segments.

   (a) Compute the cubic spline parameters that describe the complete trajectory in joint space. `[6 Points]`
   Assume that your robot controller takes four scalar values and executes the spline as: $q(t) = a_1 + a_2(t - t_{start}) + a_3 (t - t_{start})^2 + a_4 (t - t_{start})^3$ with t_{start}being the starting time
   The parameters should be written with **2 decimal precision** into the pfd-file!
   Sanity check: plot your resulting splines and the joint configurations you used to compute them!

   (b) Create a diagram of the joint angles with respect to time. `[4 Points]`

2. `[30 Points]` IMPLEMENTATION: TRAJECTORIES IN JOINT SPACE:
   Deliverables for this part of the assignment: control.cpp, graphs in the pdf.

   Implement a trajectory generator using cubic splines! When activated, the controller should compute cubic spline parameters to get from the current joint configuration to the desired joint configuration. The robot should then follow the splines using a PD-controller in joint space (3-DOF mode).

   (a) TRAJECTORY GENERATION: The function `initNjtrackControl()` is called when you select njtrack and press Start. This function should compute the parameters of the cubic spline: a_0, a_1, a_2, a_3, t_0, t_f. for each joint.
   The trajectory should satisfy the following constraints:
   $|\dot{q}_i| \leq \dot{q}_{max_i}$(`gv.dqmax`) and
   $|\ddot{q}_i| \leq \ddot{q}_{max_i}$(`gv.ddqmax`).
   Use these constraints to determine the duration t_f of the trajectory. Use the data structure `CubicSpline` as shown below to define a global spline:

```
struct CubicSpline {
        double t0, tf;
        PrVector a0, a1, a2, a3;
};
CubicSpline spline;
```

```
//Compute total trajectory length
double computeTf(GlobalVariables& gv)
{
        //Your code here!!
        return 0.0;
}
```

Hints: Use the global variable `gv.curTime` to define and follow the trajectory. Define a global variable in `control.cpp` to make the spline parameters accessible to your control function.
Place the computation of t_f in a separate function `computeTf()`. Call `computeTf()` from `initNjtrackControl()` when computing the spline parameter `[15 Points]`

(b) CONTROL: Implement a joint-space PD-controller with gravity compensation in `njtrackControl()`. Your controller should follow the cubic spline trajectory until t_f and afterwards switch to float control. `[5 Points]`
Note that here we will both track the desired position (P) and its derivative (D), the desired velocity!

$$\tau = -k_p\,(q - q_d) - k_v\,(\dot{q} - \dot{q}_d) + G(q)$$

For G use the gravity torque vector `gv.G` computed by the simulator.
Suggestion: Here you can also try out the gravity torque vector you computed in assignment 1!

(c) TRAJECTORY TO A SPECIFIC POINT: Use your methods from (a) and (b) to execute a cubic spline trajectory from the current configuration to the desired joint configuration $q_d = (-0.45, 0.65, -0.17)^T$ in `proj1Control()`. All joints have to reach their goal position at the same time. Make graphs for this trajectory showing τ, q_i, q_d over time while moving from the zero configuration to the desired configuration. `[10 Points]`

C Operational Space Control(40 points)

Deliverables for this part of the assignment: control.cpp and pdf-file
Direct computation of desired joint angles for a given end effector position is difficult. Instead, we will use a closed-loop controller and forward kinematics to circumvent the explicit computation of inverse kinematics.
Note: Here, the vectors x and x_d denote the complete coordinates in operational space, not just the x-coordinate value.

1. `[10 Points]` Create a desired trajectory in operational space for the controller to follow: Implement a generator in `proj2Control()` that computes motion on a circle starting at

   $$x(t_{start}) = (0.45m, 0.80m)$$

   , with the circle center at $x_{center} = (0.45m, 0.6m)$ and a resulting radius of $r = 0.2m$. The robot's end effector should move on that circle with an angular velocity of $\dot{\beta} = \frac{2\pi}{5s}$ ($\dot{\beta}$ is positive, i.e. right-screw/clockwise when viewing into positive z direction). End effector orientation should stay 0 (upright) during the motion.

2. `[5 Points]` Implement an operational space controller in proj2Control(). Use the transposed Jacobian (`gv.Jtransposed`) of the simulator for this. The control law for tracking a position in operational space is:

   $$F = -kp\,(x - x_d) - kv\,(\dot{x} - \dot{x}_d)$$

3. `[10 Points]` Adjust kp and kv so that the trajectory is tracked well and the joint torques don't exceed the limits. Make graphs showing τ, x, x_d, θ, and the position error $e = x - x_d$ over time t of one circle. How does the graph for the error $x - x_d$ change for different values of kp? Make another graph showing x, x_d in the 2D plane. Use a plotting tool of your choice.
   Hints: `gv.x` contains three values: the current x-position, the y-position and the orientation α of the end-effector. Note that the y-axis in the simulator is orientated in the opposite direction as the one in figure1. You should first move the robot close to $x(t_{start})$ by executing `proj1Control`)

4. `[15 Points]` Add another trajectory generator in `proj3Control()` that does the same as `proj2Control()` but stops after 3 full circles. This time, use the Parabolic Blends method for creating the trajectory in order to avoid discrete jumps in velocities when starting and stopping, i.e. limit the angular acceleration along the circle to:

   $$|\ddot{\beta}| \leq \frac{2\pi}{25s^2}$$

Hint: You need to compute the blend time $t_b$during which velocity is increased linearly to its intended value. It can be obtained by solving $\dot{\beta}(t_b) = \int_0^{t_b} \ddot{\beta}(t)dt = t_b \cdot \ddot{\beta}$.

Make a graph showing the angle β and angular velocity $\dot{\beta}$ over time for the whole trajectory.

5. Get up to 10 Bonus points for investigating (and documenting) additional topics with your operational space controller! E.g.:
   (a) Explore different trajectories: move back and forth on a line, draw a polygon, draw letters. Which motion patterns are easy to follow, which are hard to follow?
   (b) What happens to your control if you artificially add errors and noise to your commanded joint torques? Is the behavior the same across the whole workspace?
   (c) Try to extend your position control to run in 3D space and with 6-DOF mode (omit orientation control, this is much more difficult)

Deliverables

Your solution must contain the following files:

- Your **commented code**: `control.cpp, forwardkinematics.cpp`
  Do not add, modify or upload any other source code files.
- The 3-DOF mode **gains**: `gains_1.txt (from bin/sim/)`
  - The file will only be created when you click on "Store gains" in the GUI.
  - Important: You must **not** hard-code the gains inside `control.cpp`; always use the appropriate global variables (`gv.kp` etc.) such that the gains can be stored in the file `gains_1.txt`.
- One **pdf-file** containing:
  - The solutions of the **Calculation** parts
  - Answers and explanations for the **Implementation** parts
  - A **table** listing for all **implementation tasks** which team member(s) implemented them (see explanation below).

Explanation and template for implementation table

- Every group memeber needs to be able to **answer 'high level' questions about *ALL* tasks** of the assignment.
  - We will check that during the presentations with general questions. Everyone needs to be able to answer these.
- For **implementation**, please **specify which team member** worked on which (sub-)task.
  - We will check that during the presentations with implementation specific questions.
  - You can split up implementation of (sub-)tasks. But over all, every one needs to contribute equally to the implementation. (Writing the report does *not* count as "contributing to the implementation".)
- Please use the following template in your submission:

Student Name	A1	A2	A3	(A4)	(B1)	B2	C1	C2	C3	C4	C5
Albert Albono	x	x	x								x
Betty Barlow						x	x	x			
Charlie Crockett						x	x	x			
Daisy Dolittle								x	x	x	x

Note

Do not limit the torques in your control code and disable the torque limiting in the simulator (Settings tab →"torque limits"). Tune your controllers to stay within the torque limits:

Joint:	1	2	3	4	5	6
τ_{max}:	$97,6Nm$	$156,4Nm$	$89,4Nm$	$24,2Nm$	$20,1Nm$	$21,2Nm$