Exercise 1c: Inverse Kinematics of the ABB IRB 120

Marco Hutter, Michael Blösch, Dario Bellicoso, Samuel Bachmann

October 12, 2016

Figure 1: The ABB IRB 120 industrial 6-DoF Manipulator

Abstract

The aim of this exercise is to calculate the inverse kinematics of an ABB robot arm. To do this, you will have to implement a pseudo-inversion scheme for generic matrices. You will also implement a simple motion controller based on the kinematics of the system. A separate MATLAB script will be provided for the 3D visualization of the robot arm.

1 Introduction

The following exercise is based on an ABB IRB 120 depicted in Fig. 1. It is a 6-link robotic manipulator with a fixed base. During the exercise you will implement several different MATLAB functions, which, you should test carefully since the following tasks are often dependent on them. To help you with this, we have provided the script prototypes at `https://bitbucket.org/ethz-asl-lr/robotdynamics_exercise_1c` together with a visualizer of the manipulator.

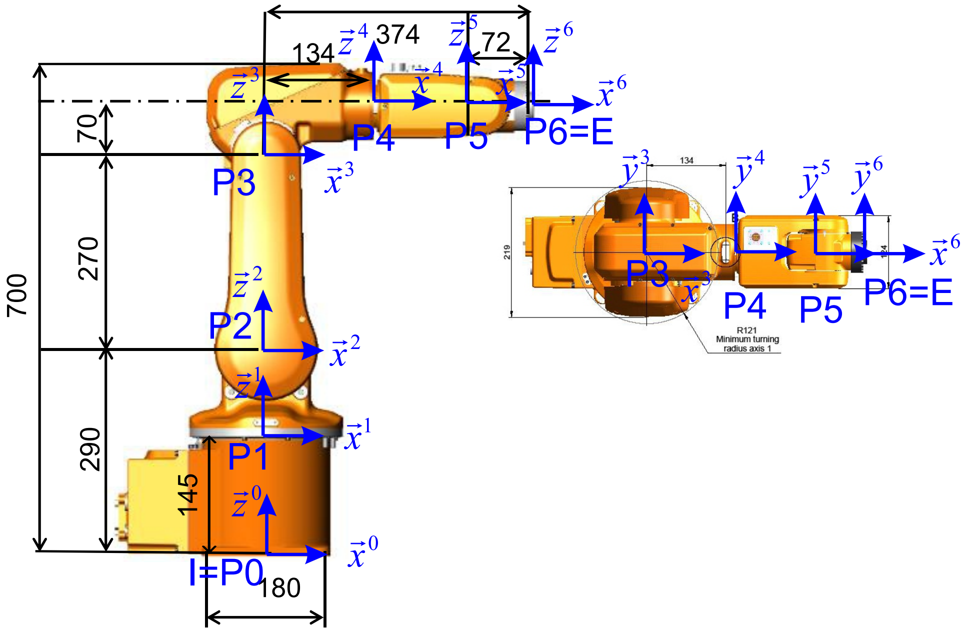

Figure 2: ABB IRB 120 with coordinate systems and joints

Throughout this document, we will employ I for denoting the inertial world coordinate system (which has the same pose as the coordinate system P0 in figure 2) and E for the coordinate system attached to the end-effector (which has the same pose as the coordinate system P6 in Fig. 2).

2 Matrix Pseudo-Inversion

Exercise 2.1

The *Moore-Penrose pseudo-inverse* is a generalization of the *matrix inversion* operation for non-square matrices. Let a non-square matrix A be defined in $\mathbb{R}^{m\times n}$. When $m > n$ and $rank(\mathrm{A}) = n$, it is possible to define the so-called *left pseudo-inverse* A_l^+ as

$$\mathrm{A}_l^+ := (\mathrm{A}^T\mathrm{A})^{-1}\mathrm{A}^T, \tag{1}$$

which yields $\mathrm{A}_l^+\mathrm{A} = \mathbb{I}_{n\times n}$. If instead it is $m < n$ and $rank(\mathrm{A}) = m$, then it is possible to define the *right pseudo-inverse* A_r^+ as

$$\mathrm{A}_r^+ := \mathrm{A}^T(\mathrm{A}\mathrm{A}^T)^{-1}, \tag{2}$$

which yields $\mathrm{A}\mathrm{A}_r^+ = \mathbb{I}_{m\times m}$. If one wants to handle singularities, then it is possible to define a *damped pseudo-inverse* as

$$\mathrm{A}_l^+ := (\mathrm{A}^T\mathrm{A} + \lambda^2\mathbf{I}_{n\times n})^{-1}\mathrm{A}^T, \tag{3}$$

and

$$\mathrm{A}_r^+ := \mathrm{A}^T(\mathrm{A}\mathrm{A}^T + \lambda^2\mathbf{I}_{m\times m})^{-1}, \tag{4}$$

In this first exercise, you are required to provide an implementation of (3) and (4) as a MATLAB function. The function place-holder to be completed is:

```
function pinvA = pseudoInverseMat(A, lambda)
  % Input: Any m-by-n matrix.
  % Output: An  n-by-m pseudo-inverse of the input according to the ...
      Moore-Penrose formula

  % Get the number of rows (m) and columns (n) of A
  [m,n] = size(A);

  ...

end
```

Solution 2.1

We can implement the two pseudo-inversions in one single script by checking the dimensions of matrix A and choosing the appropriate pseudo-inversion scheme. Note that we could use Matlab's *inv()* function to compute the inverse of $\mathrm{A}\mathrm{A}^T$ or $\mathrm{A}^T\mathrm{A}$. However, a more accurate method is to use the "\" and "/" operators, which produce more accurate results. For more information, please check the Matlab documentation.

```
function pinvA = pseudoInverseMat(A, lambda)
  % Input: Any m-by-n matrix.
  % Output: An  n-by-m pseudo-inverse of the input according to the ...
      Moore-Penrose formula

  % Get the number of rows (m) and columns (n) of A
  [m,n] = size(A);

  if (m>n)
      % Compute the left pseudoinverse.
      pinvA = (A'*A + lambda*lambda*eye(n,n))\A';
  elseif (m<=n)
      % Compute the right pseudoinverse.
      pinvA = A'/(A*A' + lambda*lambda*eye(m,m));
  end

end
```

3 Iterative Inverse Kinematics

Exercise 3.1

Consider a desired position ${}_{\mathcal{I}}\mathbf{r}_{IE}^* = \begin{bmatrix}0.5649 & 0 & 0.5509\end{bmatrix}^T$ and orientation $\mathbf{C}_{IE}^* = \mathbf{I}_{3\times 3}$. We wish to find the joint space configuration $\mathbf{q}$ which corresponds to the desired pose. This exercise focuses on the implementation of an iterative inverse kinematics algorithm, which can be summarized as follows:

1. $\mathbf{q} \leftarrow \mathbf{q}^0$ ▷ start configuration
2. while $\|\boldsymbol{\chi}_e^* - \boldsymbol{\chi}_e(\mathbf{q})\| > tol$ ▷ while the solution is not reached
3. $\mathbf{J}_{e0} \leftarrow \mathbf{J}_{e0}(\mathbf{q}) = \frac{\partial \boldsymbol{\chi}_e}{\partial q}(\mathbf{q})$ ▷ evaluate Jacobian for $\mathbf{q}$

4. $\mathbf{J}_{e0}^{+} \leftarrow (\mathbf{J}_{e0})^{+}$ ▷ update the pseudoinverse
5. $\Delta\boldsymbol{\chi}_e \leftarrow \boldsymbol{\chi}_e^* - \boldsymbol{\chi}_e(\mathbf{q})$ ▷ find the end-effector configuration error vector
6. $\mathbf{q} \leftarrow \mathbf{q} + \mathbf{J}_{e0}^{+}\Delta\boldsymbol{\chi}_e$ ▷ update the generalized coordinates

Note the use of the geometric Jacobian $\mathbf{J}_{e0}$, which was derived in the last exercise. You should implement the algorithm by defining the orientation error as the rotational vector extracted from the relative rotation between the desired orientation $\mathbf{C}_{IE}^*$ and the one based on the solution of the current iteration $\mathbf{C}_{IE}(\mathbf{q})$. The rotational vector is hence going to defined as

$$\Delta\varphi = {}_I\varphi_{EE*} = rotMatToPhi(\mathbf{C}_{IE}^*\mathbf{C}_{IE}^T(\mathbf{q})) \tag{5}$$

To do this, you should implement a function which extracts a rotational vector from a rotation matrix.

```matlab
function phi = rotMatToRotVec(C)
% Input: a rotation matrix C
% Output: the rotational vector which describes the rotation C
end

function q = inverseKinematics(I_r_IE_des, C_IE_des, q_0, tol)
  % Input: desired end-effector position, desired end-effector ...
      orientation (rotation matrix),
  %        initial guess for joint angles, threshold for the ...
      stopping-criterion
  % Output: joint angles which match desired end-effector position ...
      and orientation
end
```

Solution 3.1

```matlab
function phi = rotMatToRotVec(C)
% Input: a rotation matrix C
% Output: the rotational vector which describes the rotation C
th = acos(0.5*(C(1,1)+C(2,2)+C(3,3)-1));

if (abs(th)<eps)
    n = zeros(3,1);
else
    n = 1/(2*sin(th))*[C(3,2) - C(2,3);
                       C(1,3) - C(3,1);
                       C(2,1) - C(1,2)];
end

phi = th*n;

end



function q = inverseKinematics(I_r_IE_des, C_IE_des, q_0, tol)
  % Input: desired end-effector position, desired end-effector ...
      orientation (rotation matrix),
  %        initial guess for joint angles, threshold for the ...
      stopping-criterion
  % Output: joint angles which match desired end-effector position ...
      and orientation

  %% Setup
  it = 0;
```

```
  I_r_IE_des = I_r_IE_des(:);
  q_0 = q_0(:);

  % Set the maximum number of iterations.
  max_it = 1000;

  % Initialize the solution with the initial guess.
  q = q_0;

  % Damping factor.
  lambda = 0.001;

  % Initialize error.
  C_IE = jointToRotMat(q);
  I_r_IE = jointToPosition(q);
  C_err = C_IE_des*C_IE';
  dph = rotMatToRotVec(C_err);
  dr = I_r_IE_des - I_r_IE;
  dxe = [dr; dph];


  %% Iterative inverse kinematics

  % Iterate until terminating condition.
  while (norm(dxe)>tol && it < max_it)

    J = [jointToPosJac(q);
         jointToRotJac(q)];

    dq = pseudoInverseMat(J, lambda)*[dr;
                                       dph];

    % Update law.
    q = q + 0.5*dq;

    % Update error
    C_IE = jointToRotMat(q);
    C_err = C_IE_des*C_IE';
    dph = rotMatToRotVec(C_err);

    dr = I_r_IE_des - jointToPosition(q);
    dxe = [dr; dph];

    it = it+1;
  end

  fprintf('Inverse kinematics terminated after %d iterations.\n',it);
  fprintf('Position error: %e.\n',norm(dr));
  fprintf('Attitude error: %e.\n',norm(dph));

end
```

4 Kinematic Motion Control

The final section in this series will demonstrate the use of the iterative inverse kinematics method to implement a basic end-effector pose controller for the ABB manipulator. The controller will act only on a kinematic level, i.e. it will produce end-effector velocities as a function of the current and desired end-effector pose. This will result in a motion control scheme which should track a series of points defining a trajectory in the task-space of the robot. For all of this to work we will additionally need the following functional modules:

1. A trajectory generator, which will produce an 3-by-N array, containing N points in Cartesian space defining a discretized path that the end-effector should track.

2. A kinematics-level simulator, which will integrate over each time-step, the resulting velocities generated by the kinematic motion controller of the previous exercise. This integration, at each iteration, should generate an updated configuration of the robot which is then provided to the visualization for rendering.

To save time during the exercise session, we have provided functions to implement most of the grunt work regarding the aforementioned points. The first function provided generates a *line* trajectory defined between two points for a given path duration and time step-size. The function provided is:

```
function r_traj = generateLineTrajectory(r_start, r_end, t_total, ∆_t)
  % Inputs:
  %      r_start : start position
  %      r_end   :  end position
  %      t_total : total time duration (in sec)
  %      ∆_t  : time discretization step−size (in ms)
  % Output: 3xN matrix mapping local rotational velocities to ...
      quaternion derivatives

  N = floor(tf / ts);
  x_traj = linspace(r_start(1),r_end(1),N);
  y_traj = linspace(r_start(2),r_end(2),N);
  z_traj = linspace(r_start(3),r_end(3),N);
  r_traj = [x_traj; y_traj; z_traj].';

end
```

The second script we provide, is that of the **motion_control_visualization.m**, which is essentially a modified version of the **test_visualization.m** script. Simply running new script will begin the motion control simulation, and if unmodified, one can observe a motion almost identical to that of the test visualization. Although you are not required to modify the **motion_control_visualization.m** script: we recommend to briefly read through it and understand what it does:

```
% Motor control visualization script

% Close all figures
close all; clear;

% Load the visualization
loadVisualization;

% Set the sampling time (in seconds)
ts = 0.05;

% Configure a new trajectory − use defaults if undefined
if ¬exist('r_start','var')
    r_start = [1 1 1].';
end
if ¬exist('r_end','var')
    r_end = [0.5 0.5 0.5].';
end
if ¬exist('tf','var')
    tf = 15.0;
end
if ¬exist('q_0','var')
    q_0 = zeros(6,1);
end

% Generate a new desired trajectory
r_traj = generateLineTrajectory(r_start, r_end, tf, ts);
```

```matlab

% Initialize the vector of generalized coordinates
q = q_0;

% Set the number of time steps
kf = tf/ts; % Number of iterations as a function of the duration ...
    and the sampling time

% Notify that the visualization loop is starting
disp('Starting visualization loop.');

% Run a visualization loop
for k=1:kf
    try
        % Start a timer
        startLoop = tic;
        % Set the updated vector of generalized coordinates.
        q = kinematicMotionControl(tf,ts,k,q,r_traj);
        % Set the generalized coordinates to the robot visualizer class
        abbRobot.setJointPositions(q);
        % Update the visualization figure
        drawnow;
        % If enough time is left, wait to try to keep the update ...
            frequency
        % stable
        residualWaitTime = ts - toc(startLoop);
        if (residualWaitTime > 0)
            pause(residualWaitTime);
        end
    catch
        disp('Exiting the visualization loop.');
        break;
    end
end

% Notify the user that the script has ended.
disp('Visualization loop has ended.');
```

Exercise 4.1

The final exercise to combine the tools in the previous exercise, to implement a kinematic controller to track a single 3D line trajectory. The function to generate the line has been provided, however, you are required to specify the following four parameters *prior* to executing the `motion_control_visualization.m` script:

1. `r_start`: The start point of the trajectory.
2. `r_end`: The end point of the trajectory.
3. `tf`: The total duration of the trajectory.
4. `q_0`: The initial configuration of the robot.

Note however that the `r_start` and `r_end` are not specified in any particular frame and thus it is up to you to experiment with either defining them in the end-effector or inertial reference frames. In this exercise, you are required to place your entire sequence of computations in the `kinematicMotionControl.m` file:

```matlab
function q_new = kinematicMotionControl(tf,ts,k,q_current,r_traj)
  % Inputs:
  %  tf          : total simulation time.
  %  ts          : simulation time-step.
```

```
    %  k           : current iteration.
    %  q_current   : current configuration of the robot
    % r_traj       : desired Cartesian trajectory
    % Output: joint-space state of the robot to send to the ...
        visualization.

    % Total number of iterations
    Nf = tf/ts;

    % Step 1. - Sample trajectory configuration
    omega = 0.25;
    time = k*ts;
    Dq_max = 0.5;

    % Step 2. - Compute the updated joint velocities - this would be ...
        used for
    % a velocity controllable robot
    Dq = 2*pi*omega*Dq_max*cos(2*pi*omega*time) * ones(6,1);

    % Step 3. - Time integration step - this is would be used for a ...
        position
    % controllable robot
    q_new = q_current + Dq*ts;

end
```

Solution 4.1

The final implementation can be solved as follows:

```
function pinvA = pseudoInverseMat(A, lambda)
    % Inputs:
    %  tf          : total simulation time.
    %  ts          : simulation time-step.
    %  k           : current iteration.
    %  q_current   : current configuration of the robot
    % Output: joint-space state of the robot to send to the ...
        visualization.

    % Total number of iterations
    Nf = tf/ts;
    K_p = 1.0;

    % Step 1. - Compute the next point in the Cartesian trajectory ...
        of the
    % end-effector
    r_new = r_traj(k,:).';

    % Step 2. - Update step
    r_current = jointToPosition(q_current);
    J_current = jointToPosJac(q_current);
    w_new = K_p*(r_new - r_current);
    Dq = J_current\w_new;

    % Step 3. - Time integration step - this is would be used for a ...
        position
    % controllable robot
    q_new = q_current + Dq*ts;

end
```