Exercise 1c: Inverse Kinematics of the ABB IRB 120

Prof. Marco Hutter*
Teaching Assistants: Vassilios Tsounis, Jan Carius, Ruben Grandia†

October 10, 2017

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.

Figure 1: The ABB IRW 120 robot arm.

*original contributors include Michael Blösch, Dario Bellicoso, and Samuel Bachmann
†tsounisv@ethz.ch, jan.carius@mavt.ethz.ch, ruben.grandia@mavt.ethz.ch

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 `http://www.rsl.ethz.ch/education-students/lectures/robotdynamics.html` together with a visualizer of the manipulator.

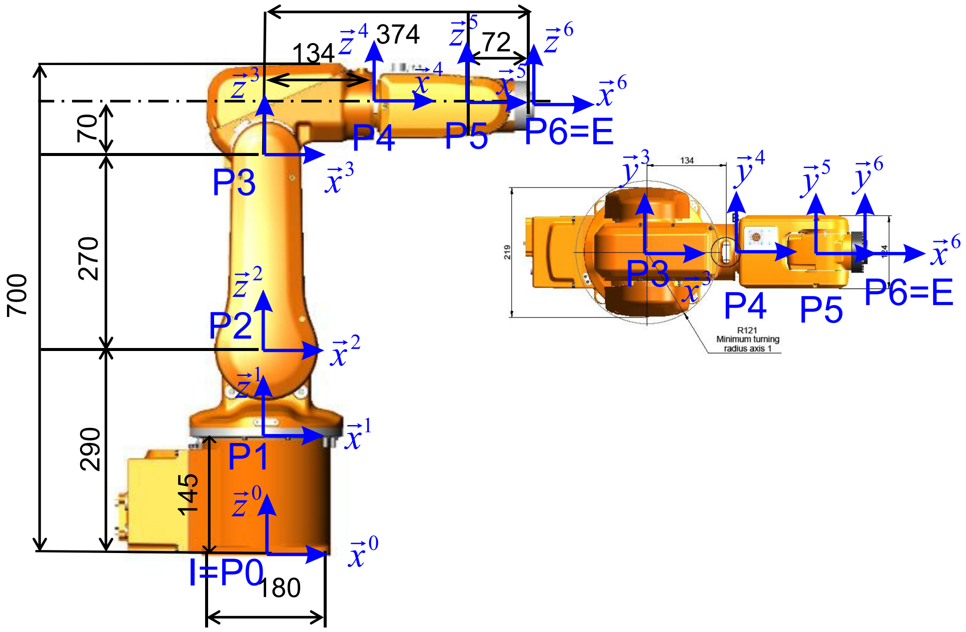

Figure 2: ABB IRB 120 with coordinate systems and joints

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

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

$$A_r^+ := A^T(AA^T + \lambda^2 \mathbf{I}_{m\times m})^{-1}. \tag{4}$$

Note that for square and invertible matrices, the pseudo-inverse is equivalent to the usual matrix inverse.

Exercise 2.1

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:

Listing 1: `pseudoInverseMat.m`

```
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);

% TODO: complete the computation of the pseudo-inverse.
% Hint: How should we account for both left and right ...
    pseudo-inverse forms?
pinvA = zeros(n, m);
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 AA^T or A^TA. However, a more accurate method is to use the "\" and "/" operators. For more information, please check the Matlab documentation.

Listing 2: `pseudoInverseMat.m`

```
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);

% Compute the pseudo inverse for both left and right cases
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

Consider a desired position ${}_{\mathcal{I}}\mathbf{r}^*_{IE} = \begin{bmatrix} 0.5649 & 0 & 0.5509 \end{bmatrix}^T$ and orientation $\mathbf{C}^*_{IE} = \mathbf{I}_{3\times 3}$ which shall be jointly called pose $\boldsymbol{\chi}^*_e$. 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 $\|\boldsymbol{\chi}_e^* \boxminus \boldsymbol{\chi}_e(\mathbf{q})\| > tol$ $\quad\triangleright$ while the solution is not reached
3. $\mathbf{J}_{e0} \leftarrow \mathbf{J}_{e0}(\mathbf{q})$ $\quad\triangleright$ evaluate Jacobian for current $\mathbf{q}$
4. $\mathbf{J}_{e0}^{+} \leftarrow (\mathbf{J}_{e0})^{+}$ $\quad\triangleright$ update the pseudoinverse
5. $\Delta\boldsymbol{\chi}_e \leftarrow \boldsymbol{\chi}_e^* \boxminus \boldsymbol{\chi}_e(\mathbf{q})$ $\quad\triangleright$ find the end-effector configuration error vector
6. $\mathbf{q} \leftarrow \mathbf{q} + \alpha\mathbf{J}_{e0}^{+}\Delta\boldsymbol{\chi}_e$ $\quad\triangleright$ update the generalized coordinates (step size α)

Note that we are using the geometric Jacobian $\mathbf{J}_{e0}$, which was derived in the last exercise. The boxminus ($\boxminus$) operator is a generalized difference operator that allows "substraction" of poses. The orientation difference is thereby defined 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 $\mathbf{C}_{IE}(\mathbf{q})$, i.e.,

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

Exercise 3.1

Your task is to implement the iterative inverse kinematics algorithm by completing the following two Matlab functions. Use `rotMatToRotVec` as a helper function to calculate the pose error.

Listing 3: `rotMatToRotVec.m`

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

% Compute the rotional vector
phi = zeros(3,1);
end
```

Listing 4: `inverseKinematics.m`

```
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

% 0. Setup
it = 0;
max_it = 100;        % Set the maximum number of iterations.
lambda = 0.001;      % Damping factor
alpha = 0.5;         % Update rate

close all;
loadviz;

% 1. start configuration
q = q_0;

% 2. Iterate until terminating condition.
while (it==0 || (norm(dxe)>tol && it < max_it))
    % 3. evaluate Jacobian for current q
    I_J = ;

    % 4. Update the psuedo inverse
```

```
    I_J_pinv = ;

    % 5. Find the end-effector configuration error vector
    % position error
    dr = ;
    % rotation error
    dph = ;
    % pose error
    dxe = ;

    % 6. Update the generalized coordinates
    q = ;

    % Update robot
    abbRobot.setJointPositions(q);
    drawnow;
    pause(0.1);

    it = it+1;
end

% Get final error (as for 5.)
% position error
dr = ;
% rotation error
dph = ;

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

Solution 3.1

The final implementation can be solved as follows:

Listing 5: `rotMatToRotVec.m`

```
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
```

Listing 6: `inverseKinematics.m`

```
function [ q ] = inverseKinematics_solution(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

% 0. Setup
it = 0;
```

```
max_it = 100;        % Set the maximum number of iterations.
lambda = 0.001;      % Damping factor.
alpha = 0.5;         % Update rate

close all;
loadviz;

% 1. start configuration
q = q_0;

% 2. Iterate until terminating condition.
while (it==0 || (norm(dxe)>tol && it < max_it))
    % 3. evaluate Jacobian for current q
    I_J = [jointToPosJac_solution(q); ...
           jointToRotJac_solution(q)];

    % 4. Update the psuedo inverse
    I_J_pinv = pseudoInverseMat_solution(I_J, lambda);

    % 5. Find the end-effector configuration error vector
    % position error
    I_r_IE = jointToPosition_solution(q);
    dr = I_r_IE_des - I_r_IE;
    % rotation error
    C_IE = jointToRotMat_solution(q);
    C_err = C_IE_des*C_IE';
    dph = rotMatToRotVec_solution(C_err);
    % 6D error
    dxe = [dr; dph];

    % 6. Update the generalized coordinates
    q = q + alpha*I_J_pinv*dxe;

    % Update robot
    abbRobot.setJointPositions(q);
    drawnow;
    pause(0.1);

    it = it+1;
end

% Get final error (as for 5.)
% position error
I_r_IE = jointToPosition_solution(q);
dr = I_r_IE_des - I_r_IE;
% rotation error
C_IE = jointToRotMat_solution(q);
C_err = C_IE_des*C_IE';
dph = rotMatToRotVec_solution(C_err);

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 problem set 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. Execute and inspect the `motion_control_visualization.m` function. It will start the motion control simulation using inputs from your motion controller (which will be implemented in `kinematicMotionControl.m`). The animation and corresponding plots will visualize the performance of your controller. The function `generateLineTrajectory.m` generates a straight-line trajectory defined between two points for a given path duration and time step size.

Listing 7: `motion_control_visualization.m`

```
function [] = motion_control_visualization()
% Motor control visualization script

% ============== Trajectory settings ======================
ts = 0.05;                          % Set the sampling time (in ...
    seconds)
r_start = [0.4 0.1 0.6].';          % 3x1 (m)
r_end = [-0.4 0.3 0.5].';           % 3x1 (m)
v_line = 0.4;                       % 1x1 (m/s)
q_0 = zeros(6,1);                   % 6x1 (rad)
use_solution = 1;                   % 0: user implementation, 1: ...
    solution
% =========================================================

% Load the visualization
f1 = figure(1); close(f1); loadviz;

% Initialize the vector of generalized coordinates
q = q_0;
abbRobot.setJointPositions(q);

% Generate a new desired trajectory
dr = r_end - r_start;
tf = norm(dr)/v_line; % Total trajectory time
N = floor(tf/ts);           % Number time steps
t = ts*1:N;
r_traj = generateLineTrajectory(r_start, r_end, N);
v_traj = repmat(v_line * (dr).'/norm(dr), N, 1); % Constant ...
    velocity reference
r_log = NaN*zeros(size(r_traj));
v_log = NaN*zeros(size(v_traj));

% Plot real trajectory
figure(2); clf; hold all
r_h = plot(t, r_log);
for i = 1:3
    plot(t, r_traj(:,i), '--', 'Color', get(r_h(i),'Color'));
end
title('End effector position in Inertial frame')
legend({'x','y','z','x_{ref}','y_{ref}','z_{ref}'})
```

```

figure(3); clf; hold all
v_h = plot(t, v_log);
for i = 1:3
    plot(t, v_traj(:,i), '--', 'Color', get(r_h(i),'Color'));
end
title('End effector linear velocity in Inertial frame')
legend({'x','y','z','x_{ref}','y_{ref}','z_{ref}'})

% Notify that the visualization loop is starting
disp('Starting visualization loop.');
pause(0.5);

% Run a visualization loop
for k = 1:N
    startLoop = tic; % start time counter

    % Get the velocity command
    switch use_solution
        case 0
            Dq = kinematicMotionControl(q, r_traj(k,:).', ...
                v_traj(k,:).');
        case 1
            Dq = kinematicMotionControl_solution(q, r_traj(k,:).', ...
                v_traj(k,:).');
    end

    % Time integration step to update visualization. This would ...
        also be used for a position controllable robot
    q = q + Dq*ts;

    % Set the generalized coordinates to the robot visualizer class
    abbRobot.setJointPositions(q);
    r_log(k,:) = jointToPosition_solution(q);
    v_log(k,:) = jointToPosJac_solution(q)*Dq;

    % Update the visualizations
    for i = 1:3
        set(r_h(i), 'Ydata', r_log(:,i));
        set(v_h(i), 'Ydata', v_log(:,i));
    end
    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
end

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


end

function [ r_traj ] = generateLineTrajectory(r_start, r_end, N)
% Inputs:
%       r_start : start position
%       r_end   :  end position
%       N       : number of timesteps
% Output: Nx3 matrix End-effector position reference
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
```

Exercise 4.1

The final exercise that combines the tools in the previous questions is to implement a kinematic controller that tracks a single 3D line trajectory. When implementing `kinematicMotionControl.m`, you can play around with the trajectory settings at the top of the file. Investigate what happens if you specify a pose that the robot cannot reach.

Listing 8: `kinematicMotionControl.m`

```
function [ Dq ] = kinematicMotionControl(q, r_des, v_des)
% Inputs:
%  k          : current iteration.
%  q          : current configuration of the robot
% r_traj      : desired Cartesian trajectory
% Output: joint-space velocity command of the robot.

% Compute the updated joint velocities. This would be used for a ...
    velocity controllable robot
% TODO:
Dq = 0.1*ones(6,1);
end
```

Solution 4.1

The final implementation can be solved as follows:

Listing 9: `kinematicMotionControl.m`

```
function [ Dq ] = kinematicMotionControl_solution(q, r_des, v_des)
% Inputs:
%  k          : current iteration.
%  q          : current configuration of the robot
% r_traj      : desired Cartesian trajectory
% Output: joint-space velocity command of the robot.

% TODO: User defined linear position gain
K_p = 5;

% TODO: User defined pseudo-inverse damping coefficient
lambda = 0.1;

% Compute the updated joint velocities. This would be used for a ...
    velocity controllable robot
r_current = jointToPosition_solution(q);
J_current = jointToPosJac_solution(q);
v_command = v_des + K_p*(r_des - r_current);
Dq = pseudoInverseMat_solution(J_current, lambda) * v_command;

end
```