Exercise 2b: Model-based control of the ABB IRB 120

Prof. Marco Hutter*
Teaching Assistants: Jan Carius, Joonho Lee, Takahiro Miki, and Koen Krämer

October 26, 2020

Abstract

In this exercise you will learn how to implement control algorithms focused on model-based control schemes. A MATLAB visualization of the robot arm is provided. You will implement controllers which require a motion reference in the joint-space as well as in the operational-space. Finally, you will learn how to implement a hybrid force and motion operational space controller. The partially implemented MATLAB scripts, as well as the visualizer, are provided.

Figure 1: The ABB IRW 120 robot arm.

*original contributors include Michael Blösch, Dario Bellicoso, and Samuel Bachmann

1 Introduction

The robot arm and the dynamic properties are shown in Figure 2. The kinematic and dynamic parameters are given and can be loaded using the provided MATLAB scripts. To initialize your workspace, run the `init_workspace.m` script. To start the visualizer, run the `loadviz.m` script.

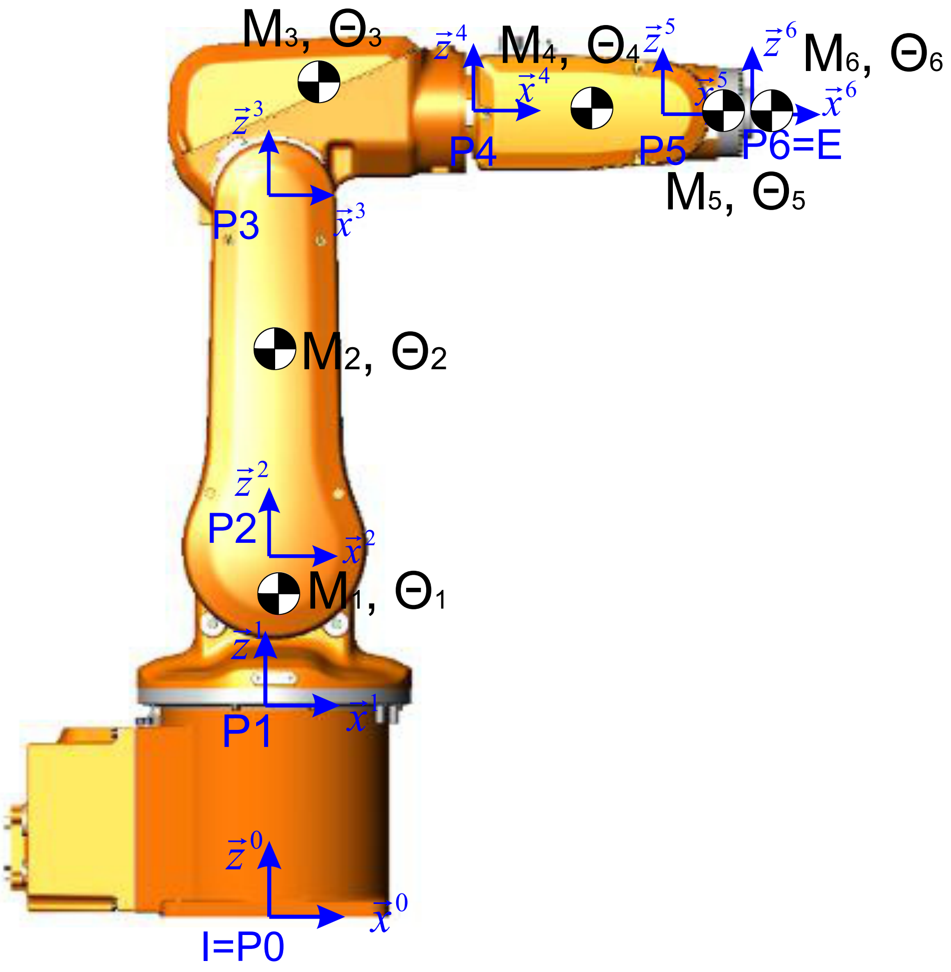

Figure 2: ABB IRB 120 with coordinate systems and joints

2 Model-based control

In this section you will write three controllers which use of the dynamic model of the arm to perform motion and force tracking tasks. The template files can be found in the `problems/` directory. Each controller comes with its own Simulink model, which is stored under `problems/simulink_models/`. To test each of your controllers, open the corresponding model and start the simulation.

2.1 Joint space control

Exercise 2.1

In this exercise you will implement a controller which compensates for the gravitational terms. Additionally, the controller should track a desired joint-space configuration and provide damping which is proportional to the measured joint velocities. Run `loadviz.m` to load the visualizer and run the provided Simulink block scheme `abb_pd_g.mdl` to test your controller. What behavior would you expect for various initial conditions?

Hint: For M, b, and g, use the provided solutions in `solutions/mfiles`.

```matlab
function [ tau ] = control_pd_g( q_des, q, q_dot )
% CONTROL_PD_G Joint space PD controller with gravity compensation.
%
% q_des --> a vector R^n of desired joint angles.
% q --> a vector R^n of measured joint angles.
% q_dot --> a vector in R^n of measured joint velocities

% Gains
% Here the controller response is mainly inertia dependent
% so the gains have to be tuned joint-wise
kp = 10.0;
kd = 2.0;
kpMat = kp * diag([5000 3000 5 1 0.5 0.01]);
kdMat = kd * diag([5000 3000 5 1 0.5 0.01]);

% The control action has a gravity compensation term, as well as a PD
% feedback action which depends on the current state and the desired
% configuration.
tau = zeros(6,1); % TODO

end
```

2.2 Inverse dynamics control

Exercise 2.2

In this exercise you will implement a controller which uses an operational-space inverse dynamics algorithm, i.e. a controller which compensates the entire dynamics and tracks a desired motion in the operational-space.

Run `loadviz.m` to load the visualizer and use the provided Simulink model stored in `abb_inv_dyn.mdl` to verify your solution. To simplify the way the desired orientation is defined, the Simulink block provides a way to define a set of Euler Angles XYZ, which will be converted to a rotation matrix in the control law script file.

```matlab
function [ tau ] = control_inv_dyn(I_r_IE_des, eul_IE_des, q, q_dot)
% CONTROL_INV_DYN Operational-space inverse dynamics controller ...
    with a PD
% stabilizing feedback term.
%
% I_r_IE_des --> a vector in R^3 which describes the desired ...
    position of the
%   end-effector w.r.t. the inertial frame expressed in the ...
    inertial frame.
% eul_IE_des --> a set of Euler Angles XYZ which describe the desired
%   end-effector orientation w.r.t. the inertial frame.
% q --> a vector in R^n of measured joint angles
% q_dot --> a vector in R^n of measured joint velocities

% Set the joint-space control gains.
kp = 10.0;
```

```
kd = 6.0;
kpMat = kp * diag([1.0 1.0 1.0 1.0 1.0 1.0]);
kdMat = kd * diag([1.0 1.0 1.0 1.0 1.0 1.0]);

% Find jacobians, positions and orientation based on the current
% measurements.
I_J_e = I_Je_fun_solution(q);
I_dJ_e = I_dJe_fun_solution(q, q_dot);
T_IE = T_IE_fun_solution(q);
I_r_Ie = T_IE(1:3, 4);
C_IE = T_IE(1:3, 1:3);

% Define error orientation using the rotational vector ...
    parameterization.
C_IE_des = eulAngXyzToRotMat(eul_IE_des);
C_err = C_IE_des*C_IE';
orientation_error = rotMatToRotVec_solution(C_err);

% Define the pose error.
chi_err = [I_r_IE_des - I_r_Ie;
           orientation_error];

% PD law, the orientation feedback is a torque around error ...
    rotation axis
% proportional to the error angle.
tau = zeros(6, 1); % TODO

end
```

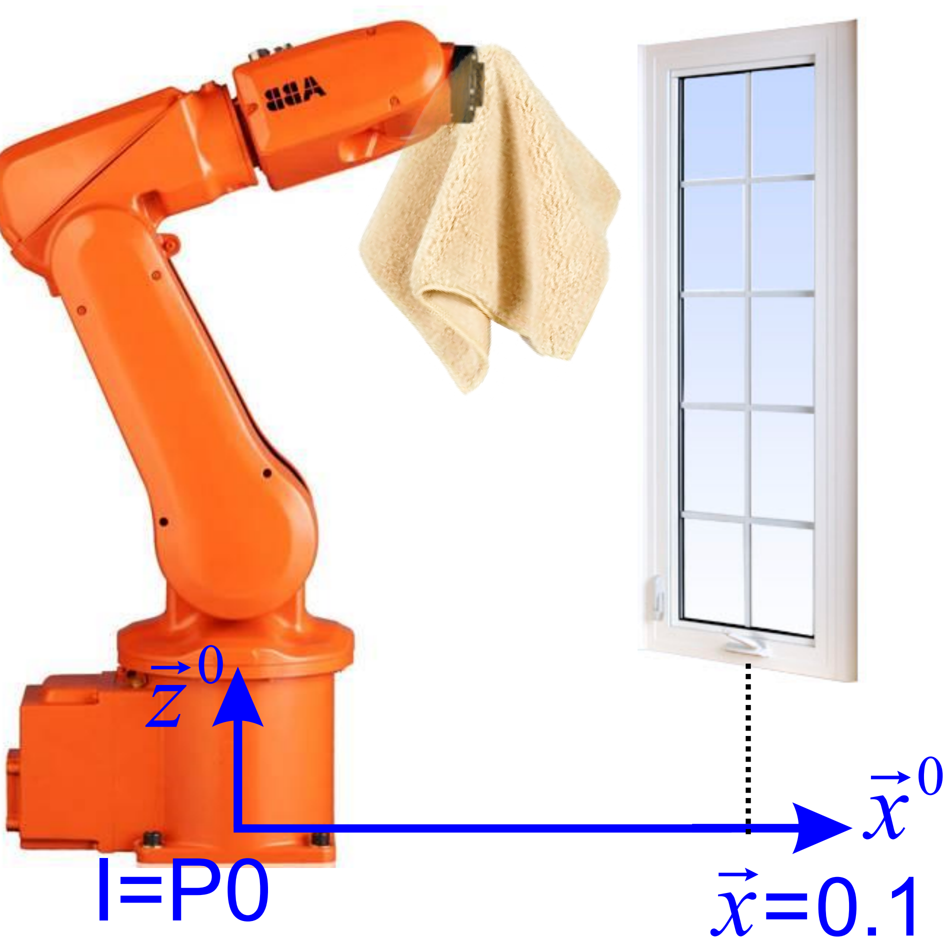

Figure 3: Robot arm cleaning a window

2.3 Hybrid force and motion control

Exercise 2.3

We now want to implement a controller which is able to control both motion and force in orthogonal directions by the use of appropriate selection matrices. As shown in Fig. 3, there is a window at $x = 0.1$ m. Your task is to write a controller that wipes the window. This controller applies a constant force on the wall in x-axis and follows a trajectory defined on $y - z$ plane. To do this, you should use the equations of motion projected to the operational-space. Use the provided Simulink model `abb_op_space_hybrid.mdl`, which also implements the reaction force exerted by the window on the end-effector.

```
function [ tau ] = control_op_space_hybrid( I_r_IE_des, eul_IE_des, ...
    q, dq, I_F_E_x )
% CONTROL_OP_SPACE_HYBRID Operational-space inverse dynamics controller
% with a PD stabilizing feedback term and a desired end-effector force.
%
% I_r_IE_des --> a vector in R^3 which describes the desired ...
    position of the
%   end-effector w.r.t. the inertial frame expressed in the ...
```

```
    inertial frame.
% eul_IE_des --> a set of Euler Angles XYZ which describe the desired
%   end-effector orientation w.r.t. the inertial frame.
% q --> a vector in R^n of measured joint positions
% q_dot --> a vector in R^n of measured joint velocities
% I_F_E_x --> a scalar value which describes a desired force in the x
%   direction

% Design the control gains
kp = 50.0;
kd = 14.0;
kpMat = kp * diag([1.0 1.0 1.0 1.0 1.0 1.0]);
kdMat = kd * diag([1.0 1.0 1.0 1.0 1.0 1.0]);

% Desired end-effector force
I_F_E = [I_F_E_x, 0.0, 0.0, 0.0, 0.0, 0.0]';

% Find jacobians, positions and orientation
I_Je = I_Je_fun_solution(q);
I_dJ_e = I_dJe_fun_solution(q, dq);
T_IE = T_IE_fun_solution(q);
I_r_IE = T_IE(1:3, 4);
C_IE = T_IE(1:3, 1:3);

% Define error orientation using the rotational vector ...
    parameterization.
C_IE_des = eulAngXyzToRotMat(eul_IE_des);
C_err = C_IE_des*C_IE';
orientation_error = rotMatToRotVec_solution(C_err);

% Define the pose error.
chi_err = [I_r_IE_des - I_r_IE;
           orientation_error];

% Project the joint-space dynamics to the operational space
% TODO
% lambda = ... ;
% mu = ... ;
% p =  ... ;

% Define the motion and force selection matrices.
% TODO
% Sm = ... ;
% Sf = ... ;

% Design a controller which implements the operational-space inverse
% dynamics and exerts a desired force.
tau = zeros(6,1); % TODO

end
```