Exercise 2a: Dynamics of the ABB IRB 120

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

October 23, 2019

Abstract

In this exercise you will develop a tool which implements the equations of motion of an ABB robot arm. To this end, you will need to compute the mass matrix, the Coriolis and the centrifugal terms, and finally the gravity terms. A MATLAB visualization of the robot arm is provided, as well as scripts which initialize the kinematic and dynamic parameters of the arm. 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.

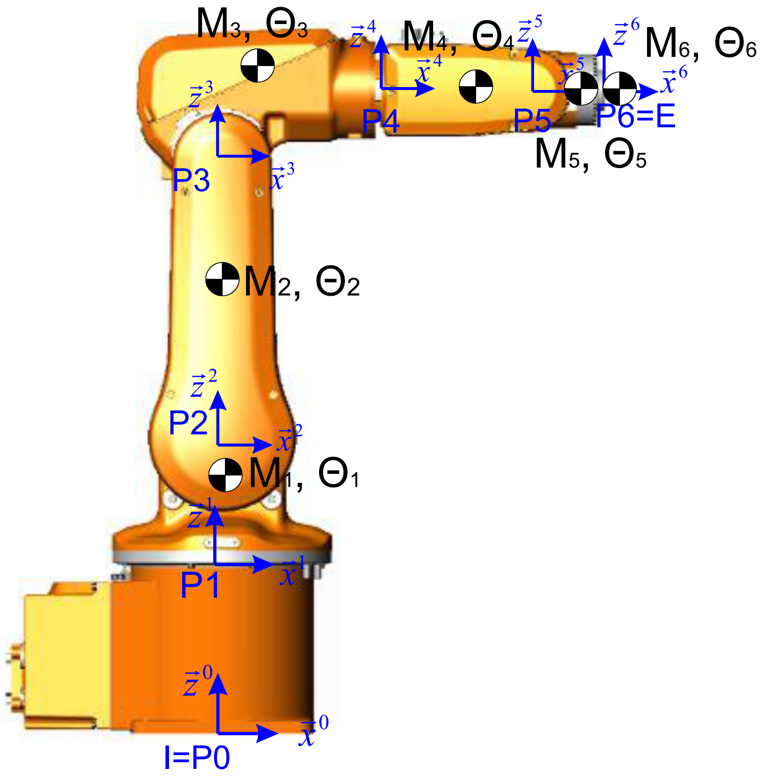

Figure 2: ABB IRB 120 with coordinate systems and joints

This exercise focuses on implementing the mass matrix $\mathbf{M}(\mathbf{q})$, the Coriolis and centrifugal terms $\mathbf{b}(\mathbf{q}, \dot{\mathbf{q}})$, and the gravity terms $\mathbf{g}(\mathbf{q})$. Before starting this exercise, take a look at the `generate_model.m` script to understand how the equations of motion are generated. Most of the necessary functions are already provided for you (for example `generate_kin.m` and `generate_jac.m`) since this was solved in previous exercises.

Exercise 1.1

Your task here is to fill in the missing code in the `generate_eom.m` script, which generates all the quantities which are used in the equations of motion, as well as the total mechanical energy of the system.

Hint: To generate time derivatives of the Jacobians, use the provided `dAdt.m` function. It essentially calculates $\frac{\mathrm{d}\,\mathbf{A}(\mathbf{q}(t))}{\mathrm{d}\,t}$ given $\mathbf{q}$ and $\dot{\mathbf{q}}$:

```
function [ dA ] = dAdt( A, q, dq )

dA = sym(zeros(size(A)));
for i=1:size(A,1)
    for j=1:size(A,2)
        dA(i,j) = jacobian(A(i,j),q)*dq;
    end
end
end
```

When you are done with the implementation, you should be able to execute the `generate_model.m` file (this may take a few minutes). This script will compute all the kinematic and dynamic quantities symbolically[1], and then save them as MATLAB `*.m` function files. These will be used later on to simulate the dynamics of the robot in *Simulink*.

`evaluate_problems.m` can be used to validate your generated equations of motions. Moreover, during simulation, the total energy (= Hamiltonian) can be used to validate your results (e.g., if no external forces are acting on the system, the total mechanical energy should remain constant over time).

```
% generate equations of motion
function eom = generate_eom(gen_cor, kin, params, jac)
% By calling:
%   eom = generate_eom(gen_cor, kin, dyn, jac)
% a struct 'eom' is returned that contains the matrices and vectors
% necessary to compute the equations of motion. These are additionally
% converted to matlab scripts.

%% Setup
phi = gen_cor.phi;
dphi = gen_cor.dphi;

T_Ik = kin.T_Ik;
R_Ik = kin.R_Ik;

k_I_s = params.k_I_s;
m = params.m;
I_g_acc = params.I_g_acc;
k_r_ks = params.k_r_ks;

I_Jp_s = jac.I_Jp_s;
I_Jr = jac.I_Jr;

eom.M = sym(zeros(6,6));
eom.g = sym(zeros(6,1));
eom.b = sym(zeros(6,1));
eom.hamiltonian = sym(zeros(1,1));



%% Compute mass matrix
fprintf('Computing mass matrix M... ');
M = sym(zeros(6,6));
for k = 1:length(phi)
    M = M + m{k}*I_Jp_s{k}'*I_Jp_s{k} ...
          + I_Jr{k}'*R_Ik{k}*k_I_s{k}*R_Ik{k}'*I_Jr{k};
end

% Use symmetry of B matrix to make computation time shorter
fprintf('simplifying... ');
for k = 1:length(phi)
    for h = k:length(phi)
```

[1] https://www.mathworks.com/help/symbolic/

```
        m_kh = simplify(M(k,h));
        if h == k
            M(k,h) = m_kh;
        else
            M(k,h) = m_kh;
            M(h,k) = m_kh;
        end
    end
end
fprintf('done!\n');


%% Compute gravity terms
fprintf('Computing gravity vector g... ');
enPot = sym(0);
for k=1:length(phi)
    enPot = enPot - m{k}*I_g_acc'*[eye(3) ...
        zeros(3,1)]*T_Ik{k}*[k_r_ks{k};1];
end
g = jacobian(enPot, phi)';
fprintf('simplifying... ');
g = simplify(g);
fprintf('done!\n');


%% Compute nonlinear terms vector
fprintf('Computing coriolis and centrifugal vector b and ...
    simplifying... ');
b = sym(zeros(6,1));
for k=1:6
    fprintf('b%i... ',k);
    dJp_s = simplify(dAdt(jac.I_Jp_s{k},gen_cor.phi, gen_cor.dphi));
    dJr_s = simplify(dAdt(jac.I_Jr{k},gen_cor.phi, gen_cor.dphi));
    omega_i = simplify(jac.I_Jr{k}*gen_cor.dphi);
    I_sk = simplify(R_Ik{k} * k_I_s{k} * R_Ik{k}');

    b = b + simplify(I_Jp_s{k}' * m{k} * dJp_s * dphi) + ...
            simplify(I_Jr{k}' * I_sk * dJr_s * dphi) + ...
            simplify(I_Jr{k}' * cross( omega_i , I_sk * omega_i));
end
fprintf('done!\n');


%% Compute energy
fprintf('Computing total energy... ');
enKin = 0.5*dphi'*M*dphi;
hamiltonian = enKin + enPot;
fprintf('simplifying... ');
hamiltonian = simplify(hamiltonian);
fprintf('done!\n');


%% Generate matlab functions

fname = mfilename;
fpath = mfilename('fullpath');
dpath = strrep(fpath, fname, '');
dpath = strcat(dpath,'../model/irb120/');

fprintf('Generating eom scripts... ');
fprintf('M... ');
matlabFunction(M, 'vars', {phi}, 'file', strcat(dpath,'/M_fun'));
fprintf('g... ');
matlabFunction(g, 'vars', {phi}, 'file', strcat(dpath,'/g_fun'));
fprintf('b... ');
matlabFunction(b, 'vars', {phi, dphi}, 'file', strcat(dpath,'/b_fun'));
fprintf('hamiltonian... ');
```

```
matlabFunction(hamiltonian, 'vars', {phi, dphi}, 'file', ...
    strcat(dpath,'/hamiltonian_fun'));
fprintf('done!\n');


%% Store the expressions
eom.M = M;
eom.g = g;
eom.b = b;
eom.hamiltonian = hamiltonian;
eom.enPot = enPot;
eom.enKin = enKin;

end
```

Exercise 1.2

Open the visualization (run `loadviz.m`) and inspect the model `abb_irb120.mdl`. Execute the simulation by pressing the play button in *Simulink* to see how the model behaves under gravity. Double-click on the plot element to see how the different quantities evolve. Does the total energy remain constant? You may adjust the inputs `tau` or `external_force`.

2 Software

There are several open-source software packages available today which do a very good job at implementing the kinematics and dynamics of generic fixed-base and floating-base systems. It is in general not a good idea to implement your own unless for very simple systems. Popular packages that are in use in our lab include

- *proNEu*[2] and *proNeu.v2*[3], are MATLAB tools which analytically derive the kinematics and dynamics based on projected Newton-Euler methods. The tools support both fixed-base and floating-base systems, as well as providing visualization tools.
- *RBDL*[4], a C++-based library that implements many rigid body algorithms and closely follows the conventions and notations introduced by Featherstone (Rigid body dynamics algorithms, Springer, 2014).
- *RobCoGen*[5] also implements many modern rigid body dynamics algorithms and generates code for a specific robot model.
- *Pinocchio*[6], an advanced C++ library for rigid body algorithms including derivative calculations

[2] `https://bitbucket.org/leggedrobotics/c_proneu`
[3] `https://bitbucket.org/leggedrobotics/proneu`
[4] `https://bitbucket.org/rbdl/rbdl`
[5] `https://robcogenteam.bitbucket.io/intro.html`
[6] `https://github.com/stack-of-tasks/pinocchio`