Robot Control Basics using DQ Robotics - Part 2

Table of Contents

Introduction

In the last lesson, I introduced you to the basics of kinematic modeling and kinematic control using a 1-DoF planar robot. The main point of that lesson was to teach you how to develop your robots from scratch, if needed.

Nonetheless, the DQ Robotics library has many of those functionalities built-in. In this lesson, you will learn how to model serial manipulators using the Denavit-Hartenberg parameters and how to calculate important Jacobians using DQ Robotics. You will also learn how to create a basic kinematic controller using DQ Robotics.

Notation

Keep these in mind (we will also use this notation when writting papers to conferences and journals):

- $\boldsymbol{h} \in \mathbb{H}$: a quaternion. (Bold-face, lowercase character)
- $\underline{\boldsymbol{h}} \in \mathcal{H}$: a dual quaternion. (Bold-face, underlined, lowercase character)

- $\boldsymbol{p}, \boldsymbol{t}, \cdots \in \mathbb{H}_p$: pure quaternions. They represent points, positions, and translations. They are quaternions for which $\mathrm{Re}(\boldsymbol{h}) = 0$.
- $\boldsymbol{r} \in \mathbb{S}^3$: unit quaternions. They represent orientations and rotations. They are quaternions for which $||\boldsymbol{h}|| = 1$.
- $\underline{\boldsymbol{x}} \in \underline{\mathcal{S}}$: unit dual quaternions. They represent poses and pose transformations. They are dual quaternions for which $||\boldsymbol{h}|| = 1$.
- $\underline{\boldsymbol{l}} \in \mathcal{H}_p \cap \underline{\mathcal{S}}$: a Plücker line.
- $\underline{\boldsymbol{\pi}} \in \{P(\underline{\boldsymbol{\pi}}) \in \mathbb{H}_p\} \cap \underline{\mathcal{S}}$: a plane.

Problem Definition

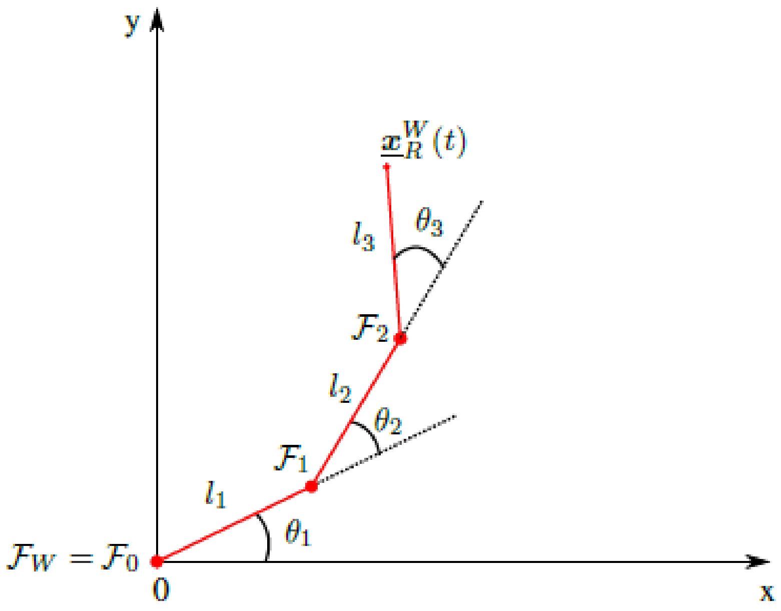


Fig. 1. Three degree-of-freedom planar robot.

1. Let the robot R be a 3-DoF planar robot, as drawn in Fig.1.
2. Let $\mathcal{F}_W$ be the world-reference frame.
3. Let $\underline{\boldsymbol{x}}_R^W(t) \triangleq \underline{\boldsymbol{x}}_R \in \underline{\mathcal{S}}$ represent the pose of the *end effector*.
4. Let R be composed of three rotational joints that rotates about their z-axis, composed in the joint-space vector $\boldsymbol{q}(t) \triangleq \boldsymbol{q} = [\theta_1 \;\; \theta_2 \;\; \theta_3]^T$ with $\theta_1(t) \triangleq \theta_1, \theta_2(t) \triangleq \theta_2, \theta_3(t) \triangleq \theta_3 \in \mathbb{R}$. The rotation of the reference frames of each joint coincide with the rotation of $\mathcal{F}_W$ when $\theta_1 = \theta_2 = \theta_3 = 0$. The length of the links are $l_1, l_2, l_3 \in \mathbb{R}^+ - \{0\}$.
5. Consider that we can freely control the joint vector $\boldsymbol{q}$.

Problems:

1. Obtain the (pose) foward kinematic model of the robot R using a set of DH parameters.
2. Obtain the pose Jacobian, rotation Jacobian, and translation Jacobian of R.
3. Using 1. and 2., design a closed-loop pose controller, rotation controller, and translation controller.

Modeling serial robots using Denavit-Hartenberg Parameters

In the last lesson, you modelled a 2-DoF planar robot. As the number of DoF and the complexity of the robots increase, modeling them requires a more general, systematic, and scalable strategy. In this lesson we will show how serial manipulators are modeled using the Denavit-Hartenberg (DH) parameters. This is the standard methodology used in DQ robotics for modeling serial robots.

Forward Kinematic Model using DH parameters

For robots in 3D space, obtaining the robot's pose transformation is the most generic form of FKM for the end effector. When using unit dual quaternions, retrieving the rotation, translation, etc from the pose is quite straighforward. So let us obtain the pose FKM of the robot R using the DH parameters.

Before going into detail about the DH parameters, let $\underline{\boldsymbol{x}}_0^W \in \underline{\mathcal{S}}$ be the reference frame at the base of the robot. For convenience, it can coincide with the reference frame of the first joint of the robot.

The first joint enacts a pose transformation from the reference frame of the first joint to the reference frame of the second joint given by

$$\underline{\boldsymbol{x}}_1^0(\theta_1) \triangleq \underline{\boldsymbol{x}}_1^0 \in \underline{\mathcal{S}}$$

that depends on the joint value of the first joint.

Given that the 3-DoF planar robot has three joints, the robot can be modeled with three consecutive transformations

$$\underline{\boldsymbol{x}}_R = \underline{\boldsymbol{x}}_1^0 \underline{\boldsymbol{x}}_2^1 \underline{\boldsymbol{x}}_3^2,$$

where $\underline{\boldsymbol{x}}_2^1(\theta_2) \triangleq \underline{\boldsymbol{x}}_2^1 \in \underline{\mathcal{S}}$ and $\underline{\boldsymbol{x}}_3^2(\theta_3) \triangleq \underline{\boldsymbol{x}}_3^2 \in \underline{\mathcal{S}}$. This sequence of transformation is a methodology that can be used for a serial manipulator with any number of joints.

The DH parameters provide a systematic way to calculate each individual joint transformation of any n-DoF serial manipulator. Each joint transformation, $\underline{\boldsymbol{x}}_i^{i-1}(\theta_i) \triangleq \underline{\boldsymbol{x}}_i^{i-1} \in \underline{\mathcal{S}}$, with $i = 1, 2, 3 \ldots n$ is composed of four intermediate transformations, as follows

$$\underline{\boldsymbol{x}}_i^{i-1} \triangleq \underline{\boldsymbol{x}}_{i'}^{i}(\theta_i)\underline{\boldsymbol{x}}_{i''}^{i'}(d_i)\underline{\boldsymbol{x}}_{i'''}^{i''}(a_i)\underline{\boldsymbol{x}}_{i}^{i'''}(\alpha_i),$$

where the DH parameters, for each joint, are $\theta_i, d_i, a_i, \alpha_i \in \mathbb{R}$. Each of those parameters is related to one transformation. The first is the rotation of θ_i about the z-axis of frame $\mathcal{F}_{i-1}$

$$\underline{\boldsymbol{x}}_{i'}^{i-1}(\theta_i) = \cos\left(\frac{\theta_i}{2}\right) + \hat{k}\sin\left(\frac{\theta_i}{2}\right),$$

the second is a translation of d_i about the z-axis of frame $\mathcal{F}_{i'}$,

$$\underline{\boldsymbol{x}}_{i''}^{i'}(d_i) = 1 + \varepsilon\frac{1}{2}\hat{k}d_i,$$

the third is the translation of a_i about the x-axis of frame $\mathcal{F}_{i''}$,

$$\underline{\boldsymbol{x}}_{i'''}^{i''}(a_i) = 1 + \varepsilon\frac{1}{2}\hat{\imath}a_i,$$

the fourth, and last, is the rotation of α_i about the x-axis of frame $\mathcal{F}_{i'''}$

$$\underline{\boldsymbol{x}}_{i}^{i'''}(\alpha_i) = \cos\left(\frac{\alpha_i}{2}\right) + \hat{\imath}\sin\left(\frac{\alpha_i}{2}\right).$$

Back to our example, the following table summarizes the DH parameters of the 3-DoF planar robot.

TABLE I
DH PARAMETERS OF THE 3-DOF PLANAR ROBOT.

	θ	d	a	α
1	θ_1	0	l_1	0
2	θ_2	0	l_2	0
3	θ_3	0	l_3	0

DQ Robotics Example

Let us create a class representing the 3-DoF planar robot using DH parameters. The good news is that most of the hard work is handled by DQ Robotics, using the following class

```
help DQ_SerialManipulatorDH
```

```
    Concrete class that extends the DQ_SerialManipulator using the
    Denavit-Hartenberg parameters (DH)

    Usage: robot = DQ_SerialManipulatorDH(A)
    - 'A' is a 4 x n matrix containing the Denavit-Hartenberg parameters
      (n is the number of links)
       A = [theta1 ... thetan;
                d1  ...   dn;
                a1  ...   an;
            alpha1 ... alphan;
             type1  ... typen]
    where type is the actuation type, either DQ_SerialManipulatorDH.JOINT_ROTATIONAL
    or DQ_SerialManipulatorDH.JOINT_PRISMATIC
    - The only accepted convention in this subclass is the 'standard' DH
    convention.

    If the joint is of type JOINT_ROTATIONAL, then the first row of A will
    have the joint offsets. If the joint is of type JOINT_PRISMATIC, then the
    second row of A will have the joints offsets.

    DQ_SerialManipulatorDH Methods (Concrete):
          get_dim_configuration_space - Return the dimension of the configuration space.
          fkm - Compute the forward kinematics while taking into account base and end-effector's rigid transformations
          plot - Plots the serial manipulator.
          pose_jacobian - Compute the pose Jacobian while taking into account base's and end-effector's rigid transfor
          pose_jacobian_derivative - Compute the time derivative of the pose Jacobian.
          raw_fkm - Compute the FKM without taking into account base's and end-effector's rigid transformations.
          raw_pose_jacobian - Compute the pose Jacobian without taking into account base's and end-effector's rigid tr
          set_effector - Set an arbitrary end-effector rigid transformation with respect to the last frame in the kine
    See also DQ_SerialManipulator.
    See also DQ_SerialManipulator.

      Documentation for DQ_SerialManipulatorDH
```

Let us represent our robot in the following way, for $l_1 = l_2 = l_3 = 1$.

```
classdef ThreeDofPlanarRobotDH
    %ThreeDofPlanarRobot regarding all methods related to the 3-DoF planar robot

    methods (Static)
        function ret = kinematics()
            %kinematics returns the kinematics of the ThreeDofPlanarRobot as DQ_SerialManipulatorDH
            DH_theta=  [0, 0, 0];
            DH_d =     [0, 0, 0];
            DH_a =     [1, 1, 1];
            DH_alpha = [0, 0, 0];
            DH_type = repmat(DQ_SerialManipulatorDH.JOINT_ROTATIONAL,1,3);
            DH_matrix = [DH_theta;
                DH_d;
                DH_a;
                DH_alpha;
                DH_type];

            ret = DQ_SerialManipulatorDH(DH_matrix,'standard');
            ret.name = "3 DoF Planar Robot";
        end
    end
end
```

Note that we use

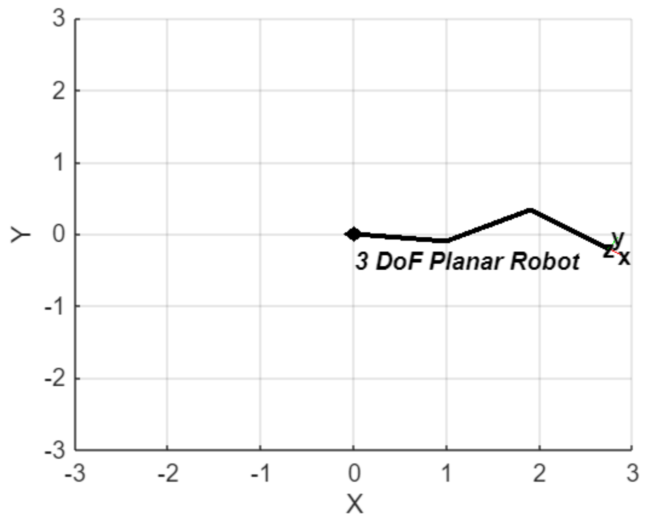
```
DQ_SerialManipulatorDH.JOINT_ROTATIONAL;
```

to define a rotational joint, so we do not explicilty define $\theta_1, \theta_2, \theta_3$ in our model.

To calculate the forward kinematics model and plot the robot model, we can simply call the methods already available in the class, as follows

```
clear all;
close all;

% Initial joint values [rad]
theta1 = -0.1;
theta2 = 0.55;
```

```
theta3 = -1.02;

% Joint vector
q = [theta1 theta2 theta3];

% Instantiate the robot kinematics
three_dof_planar_robot = ThreeDofPlanarRobotDH.kinematics();

% Plot the robot in the xy-plane
three_dof_planar_robot.plot(q);
```



```
% Get the fkm, based on theta
x_r = three_dof_planar_robot.fkm(q);
```

For more details about the methods, check the documentation of the class using

```
help DQ_SerialManipulatorDH.fkm
```

```
   fkm(q) calculates the forward kinematic model and
   returns the dual quaternion corresponding to the
   end-effector pose. This function takes into account the
   displacement due to the base's and effector's poses.

   'q' is the vector of joint variables
   'to_ith_link' defines up to which link the fkm will be
   calculated. If to_ith_link corresponds to the last link,
   the method DOES NOT take into account the transformation
   given by set_effector. If you want to take into account
   that transformation, use fkm(q) instead.
```

```
help DQ_SerialManipulatorDH.plot
```

```
 plot(robot,q,options) plots the robot of type DQ_kinematics.
 q is the vector of joint configurations
 options is an optional argument that has variable size and accept any
 number of the following pairs:

  'workspace', W          size of robot 3D workspace, where
```

```
                            W = [xmn, xmx ymn ymx zmn zmx]
    'cylinder', C           color for joint cylinders, C=[r g b]
    'scale', scale          annotation scale factor
    'base'|'nobase'         controls display of base plane
    'wrist'|'nowrist'       controls display of wrist
    'name'|'noname'         display the robot's name
    'xyz'|'noa'             wrist axis label
    'joints'|'nojoints'     controls display of joints

   The graphical robot object holds a copy of the robot object and
   the graphical element is tagged with the robot's name (.name property).

   1) Figure behavior:

   If no robot of this name is currently displayed then a robot will
   be drawn in the current figure.  If hold is enabled (hold on) then the
   robot will be added to the current figure.

   If the robot already exists then that graphical model will be found
   and moved.

   2) Multiple views of the same robot:

   If one or more plots of this robot already exist then these will all
   be moved according to the argument 'q'.  All robots in all windows with
   the same name will be moved.

   NOTE: Since each kinematic robot stores just one graphical handle,
   if we want to plot the same robot in different views, we must declare
   different robots with the same name. Otherwise, if just one robot is declared,
   but plotted in different windows/views, the kinematic robot will store
   the handle of the last view only. Therefore, only the last view will be
   updated

   3) Multiple robots in the same figure:

   Multiple robots (i.e., with different names) can be displayed in the same
   plot, by using "hold on" before calls to plot(robot).

   4) Graphical robot state:

   The configuration of the robot as displayed is stored in the DQ_kinematics
   object and can be accessed by the read only object property 'q'.

   5) Graphical annotations and options:

   The robot is displayed as a basic stick figure robot with annotations
   such as:
   - XYZ wrist axes and labels,
   - joint cylinders,
   which are controlled by options.

   The size of the annotations is determined using a simple heuristic from
   the workspace dimensions.  This dimension can be changed by setting the
   multiplicative scale factor using the 'scale' option.

 Help for DQ_SerialManipulatorDH/plot is inherited from superclass DQ_SerialManipulator
```

Differential Kinematics Model using DH parameters

Similar to how we calculated the the DKM in the last lesson, the DKM can be calculated for any set of DH parameters.

Pose Jacobian

Using the FKM, we find

$$\mathrm{vec}_8\left(\dot{\underline{\boldsymbol{x}}}_R\right) = \frac{\partial\left(\mathrm{vec}_8\left(\underline{\boldsymbol{x}}_R\right)\right)}{\partial \boldsymbol{q}}\dot{\boldsymbol{q}},$$

where $\boldsymbol{J}_{\underline{\mathbf{x}}} \triangleq \frac{\partial\left(\mathrm{vec}_8\left(\underline{\boldsymbol{x}}_R\right)\right)}{\partial \boldsymbol{q}}$ is the pose Jacobian. We do not need to worry about the details of how to calculate this for now. The details of how to calculate the Jacobian for any serial manipulator are described from Page 38 of

ADORNO, B. V., Two-arm manipulation: from manipulators to enhanced human-robot collaboration [*Contribution à la manipulation à deux bras : des manipulateurs à la collaboration homme-robot*], Université Montpellier 2, Montpellier, France, 2011. (pdf) `(The pdf is not running)`

Rotation Jacobian

The goal of this section is to find the rotation Jacobian, $\boldsymbol{J}_r$, so that the following relation holds

$$\mathrm{vec}_4\left(\dot{\boldsymbol{r}}_R\right) = \boldsymbol{J}_r\dot{\boldsymbol{q}}.$$

The rotation Jacobian is useful to control the rotation of the end effector and can be used to calculate many other Jacobians.

We can conveniently find the rotation Jacobian using the pose Jacobian. To do so, remember that the robot's end-effector pose can be decomposed as follows

$$\underline{\boldsymbol{x}}_R = \boldsymbol{r}_R + \frac{1}{2}\varepsilon \boldsymbol{t}_R\boldsymbol{r}_R.$$

That means that the first-order time-derivative is

$$\mathrm{vec}_8\left(\dot{\underline{\boldsymbol{x}}}_R\right) = \mathrm{vec}_8\left(P\left(\dot{\underline{\boldsymbol{x}}}_R\right)\right) + \mathrm{vec}_8\left(D\left(\dot{\underline{\boldsymbol{x}}}_R\right)\right)$$

that can be re-written as

$$\boldsymbol{J}_{\underline{\mathbf{x}}}\dot{\boldsymbol{q}} = \begin{bmatrix}\boldsymbol{J}_{P(\underline{\mathbf{x}})}\\ 0\end{bmatrix}\dot{\boldsymbol{q}} + \begin{bmatrix}0\\ \boldsymbol{J}_{D(\underline{\mathbf{x}})}\end{bmatrix}\dot{\boldsymbol{q}}$$

which means that the pose Jacobian can be decomposed as

$$\boldsymbol{J}_{\underline{\mathbf{x}}}\dot{\boldsymbol{q}} = \begin{bmatrix}\boldsymbol{J}_{P(\underline{\mathbf{x}})}\\ \boldsymbol{J}_{D(\underline{\mathbf{x}})}\end{bmatrix}\dot{\boldsymbol{q}}.$$

Notice that

$$\boldsymbol{J}_{P(\underline{\mathbf{x}})}\dot{\boldsymbol{q}} = \mathrm{vec}_4\left(\dot{\boldsymbol{r}}_R\right)$$

$$\boldsymbol{J}_{P(\underline{\mathbf{x}})}\dot{\boldsymbol{q}} = \boldsymbol{J}_r\dot{\boldsymbol{q}}$$

which means that the rotation Jacobian is $\boldsymbol{J}_r = \boldsymbol{J}_{P(\underline{\mathbf{x}})}$. That is, the rotational Jacobian is composed of the first four rows of the pose Jacobian.

Translation Jacobian

The goal of this section is to find the translation Jacobian, $\boldsymbol{J}_t$, so that the following relation holds

$$\operatorname{vec}_4(\dot{\boldsymbol{t}}_R) = \boldsymbol{J}_t \dot{\boldsymbol{q}}$$

The translation Jacobian is useful to control the translation of the end effector and can be used to calculate many other Jacobians. We can conveniently find the translation Jacobian using the pose Jacobian and the end-effector's pose.

We start from the translation relation

$$\boldsymbol{t}_R = \operatorname{translation}(\underline{\boldsymbol{x}}_R) = 2D(\underline{\boldsymbol{x}}_R)P(\underline{\boldsymbol{x}}_R)^*$$

$$\dot{\boldsymbol{t}}_R = 2\left[D\left(\dot{\underline{\boldsymbol{x}}}_R\right)P(\underline{\boldsymbol{x}}_R)^* + D(\underline{\boldsymbol{x}}_R)P\left(\dot{\underline{\boldsymbol{x}}}_R\right)^*\right]$$

$$\operatorname{vec}_4(\dot{\boldsymbol{t}}_R) = 2\left[\overset{-}{\underset{4}{\mathbf{H}}}\left(P(\underline{\boldsymbol{x}}_R)^*\right)\boldsymbol{J}_{D(\underline{\mathbf{x}})} + \overset{+}{\underset{4}{\mathbf{H}}}\left(D(\underline{\boldsymbol{x}}_R)\right)\mathbf{C}_4\boldsymbol{J}_{P(\underline{\mathbf{x}})}\right]\dot{\boldsymbol{q}}$$

hence

$$\boldsymbol{J}_t = 2\left[\overset{-}{\underset{4}{\mathbf{H}}}\left(P(\underline{\boldsymbol{x}}_R)^*\right)\boldsymbol{J}_{D(\underline{\mathbf{x}})} + \overset{+}{\underset{4}{\mathbf{H}}}\left(D(\underline{\boldsymbol{x}}_R)\right)\mathbf{C}_4\boldsymbol{J}_{P(\underline{\mathbf{x}})}\right].$$

DQ Robotics Example

The pose Jacobian can be computed using the DQ_SerialManipulatorDH class as follows.

```
% Get the pose Jacobian
Jx = three_dof_planar_robot.pose_jacobian(q)
```

```
Jx = 8×3
    0.1406    0.1406    0.1406
         0         0         0
         0         0         0
    0.4798    0.4798    0.4798
         0         0         0
   -0.1433   -0.0514    0.2839
    0.6711    0.1796   -0.1913
         0         0         0
```

The rotation Jacobian and translation Jacobian can be calculated using methods of its DQ_Kinematics superclass. For instance, the rotation Jacobian can be obtained as

```
% Get the rotation Jacobian, based on the pose Jacobian
Jr = three_dof_planar_robot.rotation_jacobian(Jx)
```

```
Jr = 4×3
```

```
    0.1406    0.1406    0.1406
         0         0         0
         0         0         0
    0.4798    0.4798    0.4798
```

and the translation Jacobian can be obtained as

```
% Get the end-effector's pose
x = three_dof_planar_robot.fkm(q);
% Get the translation Jacobian
Jt = three_dof_planar_robot.translation_jacobian(Jx,x)
```

```
Jt = 4×3
         0         0         0
    0.2045    0.1047    0.5396
    2.7374    1.7423    0.8419
         0         0         0
```

Task-space position control

In the last lesson you were introduced to the basics of robot control using the inverse differential kinematics model.

Instead of building the controller from scratch, you can use controllers already available in DQ Robotics.

Pseudo-Inverse Controller

In the last lesson, we implemented a simple pseudo-inverse-based kinematic controller. Let us revisit this topic using DQ Robotics.

Let us start by cleaning up the workspace and write the control loop from scratch.

Preliminaries

Let us start with the initial conditions of the problem.

First, clean the workspace.

```
clear all;
close all;
include_namespace_dq
```

Define the sampling time and how many seconds of control we will simulate.

```
% Sampling time [s]
tau = 0.01;
% Simulation time [s]
final_time = 1;
```

Define the initial robot posture.

```
% Initial joint values [rad]
theta1 = -0.4;
theta2 = 1.71;
```

```
theta3 = 0.85;
% Arrange the joint values in a column vector
q_init = [theta1 theta2 theta3]';
```

Define the desired translation

```
% Desired translation components [m]
tx = 1.25;
ty = 1.25;
% Desired translation
td = tx*i_ + ty*j_;
```

then, the desired rotation.

```
% Desired rotation component [rad]
gamma = 0.49;
% Desired rotation
rd = cos(gamma/2.0) + k_*sin(gamma/2.0);
```

The desired pose will then be

```
% Desired pose
xd = rd + 0.5*E_*td*rd;
```

We then instantiate the robot kinematics, as follows.

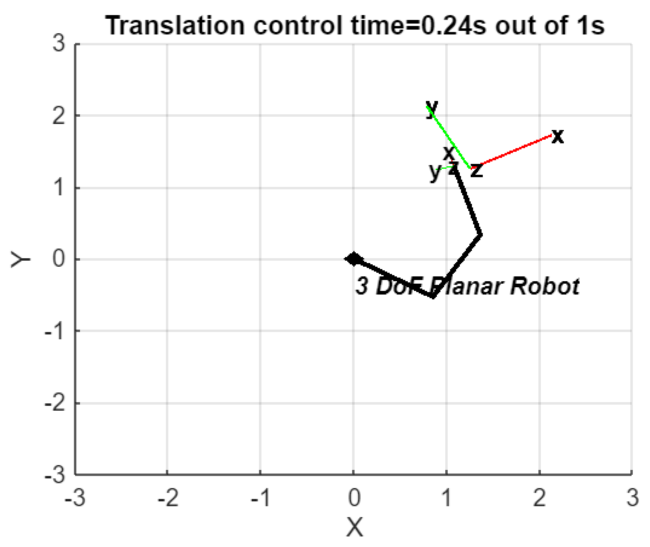
```
% Create robotranslation_controllert
three_dof_planar_robot = ThreeDofPlanarRobotDH.kinematics();
```

Translation Controller

The basic syntax to instantiate a translation controller is as follows.

The robot will align the end-effector translation with the desired translation. The rotation is not controlled.

```
% Instanteate the controller
translation_controller = DQ_PseudoinverseController(three_dof_planar_robot);
translation_controller.set_control_objective(ControlObjective.Translation);
translation_controller.set_gain(5.0);
translation_controller.set_damping(0); % Damping was not explained yet, set it as 0
to use pinv()

% Translation controller loop.
q = q_init;
for time=0:tau:final_time
    % Get the next control signal [rad/s]
    u = translation_controller.compute_setpoint_control_signal(q,vec4(td));

    % Move the robot
    q = q + u*tau;
    % Plot
    % Plot the robot
    three_dof_planar_robot.plot(q);
    title(['Translation control' ' time=' num2str(time) 's out of '
num2str(final_time) 's'])
    % Plot the desired pose
    hold on
    plot(xd);
    hold off
    % [For animations only]
    drawnow; % [For animations only] Ask MATLAB to draw the plot now
end
```



```
% Rerun controller
```

Rotation Controller

The basic syntax to instantiate a rotation controller is as follows.

The robot will align the end-effector rotation with the desired rotation. The translation is not controlled.

```
% Instantiate the controller
translation_controller = DQ_PseudoinverseController(three_dof_planar_robot);
translation_controller.set_control_objective(ControlObjective.Rotation);
translation_controller.set_gain(5.0);
translation_controller.set_damping(0); % Damping was not explained yet, set it as 0
to use pinv()

% Rotation controller loop.
q = q_init;
for time=0:tau:final_time
    % Get the next control signal [rad/s]
    u = translation_controller.compute_setpoint_control_signal(q,vec4(rd));
    % Move the robot
    q = q + u*tau;

    % Plot
    % Plot the robot
    three_dof_planar_robot.plot(q);
    title(['Rotation control' ' time=' num2str(time) 's out of '
num2str(final_time) 's'])
    % Plot the desired pose
    hold on
    plot(xd);
    hold off
    % [For animations only]
    drawnow; % [For animations only] Ask MATLAB to draw the plot now
end
% Rerun controller
```

Pose Controller

The basic syntax to instantiate a pose controller is as follows.

The robot will align the end-effector pose with the desired pose. If the rotation and translation cannot be achieved simulatenously, the robot will balance rotation and translation error, according to the controller definitions.

```
% Instantiate the controller
translation_controller = DQ_PseudoinverseController(three_dof_planar_robot);
translation_controller.set_control_objective(ControlObjective.Pose);
translation_controller.set_gain(5.0);
translation_controller.set_damping(0); % Damping was not explained yet, set it as 0
to use pinv()

% Translation controller loop.
q = q_init;
for time=0:tau:final_time
    % Get the next control signal [rad/s]
    u = translation_controller.compute_setpoint_control_signal(q,vec8(xd));

    % Move the robot
    q = q + u*tau;

    % Plot
    % Plot the robot
    three_dof_planar_robot.plot(q);
    title(['Pose control' ' time=' num2str(time) 's out of ' num2str(final_time)
's'])
    % Plot the desired pose
    hold on
    plot(xd);
    hold off
    % [For animations only]
    drawnow; % [For animations only] Ask MATLAB to draw the plot now
end
% Rerun controller
```

Homework

(1) Following the format of [ThreeDofPlanarRobotDH.m], create a class called [NDofPlanarRobotDH.m] as shown in the figure.

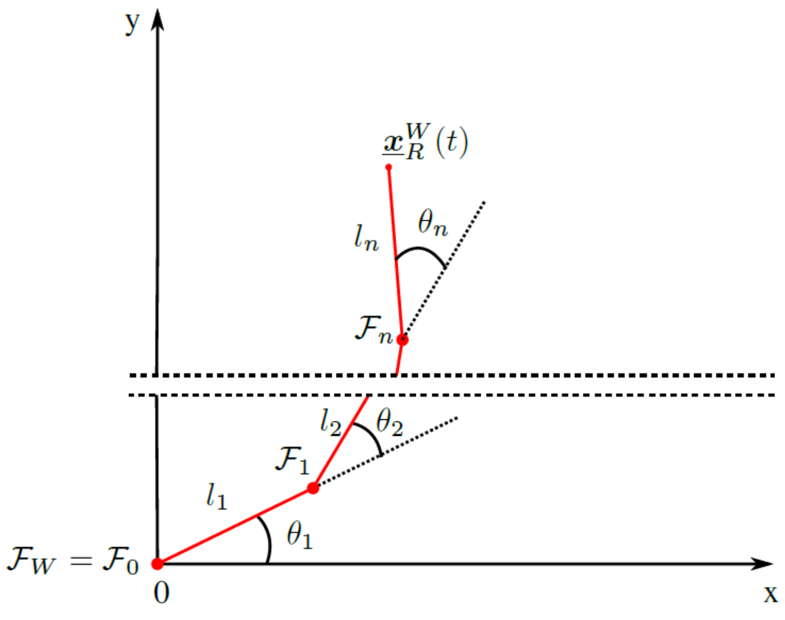


Fig. 2. n-degree-of-freedom planar robot.

The class must

1. Consider all lengths of the links as unitary. That is, $l_1 = l_2 = \cdots = l_n = 1$.
2. Have a "kinematics" method that takes the desired number of DoFs as input and returns the corresponding DQ_SerialManipulatorDH instance.

(2) Consider the desired translation $\boldsymbol{t}_d = 5\hat{\imath} + 2\hat{\jmath}$, desired rotation $\boldsymbol{r}_d = \cos\left(-\frac{\pi}{8}\right) + \hat{k}\sin\left(-\frac{\pi}{8}\right)$, and initial posture $\theta_i(0) = \frac{\pi}{8}$ for $i = 1...7$. Use the class you created in (1) to instantiate a 7-DoF planar robot.

1. create a script called [seven_dof_planar_robot_translation_control.m] that implements a task-space translation controller using a DQ_PseudoinverseController. Control the 7-DoF planar robot to $\boldsymbol{t}_d$.

2. create a script called [seven_dof_planar_robot_rotation_control.m] that implements a task-space rotation controller using a DQ_PseudoinverseController. Control the 7-DoF planar robot to $\boldsymbol{r}_d$.
3. create a script called [seven_dof_planar_robot_pose_control.m] that implements a task-space pose controller using a DQ_PseudoinverseController. Control the 7-DoF planar robot to $\underline{\boldsymbol{x}}_d = \boldsymbol{r}_d + \frac{1}{2}\varepsilon \boldsymbol{t}_d \boldsymbol{r}_d$.