A Brief Introduction to the Jacobian for Robot Kinematics & Motion Control

Notes for the *Semester Project on Signals in Robot Systems*

Iñigo Iturrate

October 15, 2021

1 Introduction

As you may remember, *kinematics* is the branch of physics and classical mechanics that studies the motions of objects in the absence of external forces. Similarly, *robot kinematics* studies the motion of robots in the absence of forces[1].

An understanding of robot kinematics is a necessity if you are to control the motion of robots in order to perform a task. While *collaborative robots*, such as Universal Robots, have made it easier for end-users of to program robot motions without fully understanding the theory behind them, if you encounter issues with a robotic system or if you are to implement some more advanced motion control you will need at least to know the basic notions presented here. Note that this document is far from comprehensive and is merely meant to get you started. More advanced topics will be covered in some of your other courses.

Two problems in particular are at the core of robot kinematics: *direct kinematics* and *inverse kinematics*. Both of these problems are concerned with relating *configuration space* (also known as *joint space*) to *task space* (also referred to as *operational* or *Cartesian* space). The relationship between these two spaces is expressed through the Jacobian matrix, denoted $\boldsymbol{J}(\boldsymbol{q})$. These concepts will be covered in section 2.

Once we understand the relationship between joint and task space through the Jacobian, we can begin to design trajectories that will be used to control the robot's motion. We will look at this in section 3.

2 Differential Robot Kinematics

2.1 Joint or Configuration Space

For an n degree-of-freedom (n-DOF) robot, positions in joint space can be described by a vector $\boldsymbol{q} \in \mathbb{R}^n$:

$$\boldsymbol{q} = \begin{bmatrix} q_1 \\ \vdots \\ q_n \end{bmatrix}. \tag{1}$$

Similarly, joint-space velocities can be described as $\dot{\boldsymbol{q}} \in \mathbb{R}^n$:

$$\dot{\boldsymbol{q}} = \begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix}. \tag{2}$$

A common case for many industrial robot manipulators, including the Universal Robots UR5, is to have 6-DOF, such that $\boldsymbol{q}, \dot{\boldsymbol{q}} \in \mathbb{R}^6$.

[1] Whereas *robot dynamics* is the branch of robotics that studies the motion of robots subject to forces.

2.2 Task or Cartesian Space

For the rest of this document, we will assume that the robot operates in a 3D world[2], where poses of the end-effector in the task space can be described by a vector $\boldsymbol{x} \in \mathbb{R}^6$:

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{\phi} \end{bmatrix}, \tag{3}$$

where $\boldsymbol{p} \in \mathbb{R}^3$ is a vector describing the position $[p_x, p_y, p_z]$ of the end-effector and $\boldsymbol{\phi}_e \in \mathbb{R}^3$ is a vector describing its orientation $[\phi_x, \phi_y, \phi_z]$.

Similarly, velocities, $\dot{\boldsymbol{x}} \in \mathbb{R}^6$, can be written:

$$\dot{\boldsymbol{x}} = \boldsymbol{v} = \begin{bmatrix} \boldsymbol{v} \\ \boldsymbol{\omega} \end{bmatrix}, \tag{4}$$

where $\dot{\boldsymbol{x}} = \boldsymbol{v} \in \mathbb{R}^3$ is the translational velocity of the end-effector, and $\boldsymbol{\omega} \in \mathbb{R}^3$ is its angular velocity.

2.3 The Jacobian Matrix

Given the joint position and velocity vectors, $\boldsymbol{q}, \dot{\boldsymbol{q}} \in \mathbb{R}^n$, and task space velocity vector, $\boldsymbol{x}, \dot{\boldsymbol{x}} \in \mathbb{R}^6$, we would now like to find a mapping between them, such that given the movement of the robot's joints, we can compute how the robot end-effector will move in Cartesian space (and vice versa). This mapping is given by the Jacobian matrix.

Deriving the Jacobian

Deriving the Jacobian matrix falls outside our current scope, and will be covered by other courses.
Suffice it to say that one can compute the Jacobian from the DH parameters of a robot, as these will allow us to establish the contributions of each link to the motion of the next link. Once the contributions of each link are computed, we can combine them to derive the overall motion of the end-effector.

Formally, the Jacobian is defined as a matrix of partial derivatives in the form[3]:

$$\boldsymbol{J}(\boldsymbol{q}) = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{q}} = \begin{bmatrix} \frac{\partial p_x}{\partial q_1} & \frac{\partial p_x}{\partial q_2} & \cdots \\ \frac{\partial p_y}{\partial q_1} & \frac{\partial p_y}{\partial q_2} & \cdots \\ \vdots & \ddots & \vdots \end{bmatrix} \tag{5}$$

As we have n-DOF in joint space and 6-DOF in task space, the Jacobian matrix will have dimension $6 \times n$. Again, if we consider a 6-DOF manipulator, the matrix will have dimension 6×6, meaning it will be a square matrix. This is important for reasons that will be discussed later. Notice also that we purposely wrote $\boldsymbol{J}(\boldsymbol{q})$ and not just $\boldsymbol{J}$, as the Jacobian matrix is a function the configuration that the robot is in – and will therefore change as the robot moves.

We can now play around with the Jacobian definition by inserting time, t, and come to the following:

$$\boldsymbol{J}(\boldsymbol{q}) = \frac{\partial \boldsymbol{x}}{\partial t}\frac{\partial t}{\partial \boldsymbol{q}} \longrightarrow \frac{\partial \boldsymbol{x}}{\partial t} = \boldsymbol{J}(\boldsymbol{q})\frac{\partial \boldsymbol{q}}{\partial t}, \tag{6}$$

or, in other words:

$$\dot{\boldsymbol{x}} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}}. \tag{7}$$

Equation (7) is what is called the *forward* (or *direct*) *kinematics equation*. It directly maps from joint-space velocities to Cartesian space end-effector velocities.

[2] You might be wondering why it could be that a robot would operate in a non-3D world; after all, our world is three-dimensional. However, there may be cases where it might be convenient to describe the world in, e.g. 2D, for example, if we are working with a robot that due to its structure can only move in a plane, or if we have a task where we are constrained to a plane.

[3] Note that there exist two forms of Jacobian: *analytical* and *geometric*, depending on how the orientation is represented. This has some important consequences that are relevant at a higher level than will be discussed here.

We now have a way of describing what the motion of the end-effector will be assuming that we know the motion of the joints. But what if we want to do the inverse? Well, we can – in principle – state the following:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}^{-1}(\boldsymbol{q})\dot{\boldsymbol{x}}. \tag{8}$$

This is what is known as the *inverse kinematics equation*.

Remember how we mentioned that the fact that, for a 6-DOF robot, the Jacobian is square had some important consequences? This is because only square matrices can have an inverse. If we consider, for example, a 7-DOF robot, the Jacobian will be a 6×7 matrix and will therefore not have an inverse[4].

2.4 Singularities

Recall that not all square matrices are invertible. Those that are not are called singular matrices (and their determinant will be 0). In robotics, when applying inverse kinematics (IK) using eq. (8), the Jacobian matrix can sometimes become singular, and this will present a practical problem for robot control. One of the ways this can manifest itself as extremely high accelerations of certain robot joints when approaching or moving close to a singularity using IK. This is because the Jacobian matrix becomes ill-conditioned, meaning that, numerically, a small change in the input will cause a very large change in the output, i.e. a small change in the task space velocity, $\dot{\boldsymbol{x}}$ produces a very large change in the joint space velocity $\dot{\boldsymbol{q}}$.

Singularities of a UR Robot

The *singular configurations* of a robot are related to its structure, and it is therefore hard to make any generalizations about them. However, if we consider specifically a Universal Robots UR5 6-DOF, we can find the following singularities[a]:

- **Wrist singularity**: This occurs when $q_5 = 0$, $q_5 = \pm 180°$, or $q_5 = \pm 360°$, resulting in q_4 and q_6 being parallel.
- **Shoulder singularity**: This occurs when the axes of q_5 and q_6 are co-planar with those of q_1 and q_2. When the robot approaches this singularity, q_1 will experience very sudden acceleration.
- **Elbow singularity**: This is the most straightforward singularity, as it occurs when the arm is fully stretched, i.e. when $q_3 = 0$ and q_2, q_3, and q_3 lie on a plane.

[a] To view these on a real robot, watch the following: https://www.youtube.com/watch?v=6Wmw4lUHlX8

3 Robot Trajectory Control & Design

Now that we have discussed both the forward and inverse kinematics equations, let us see how we can use them to make a robot follow a desired trajectory.

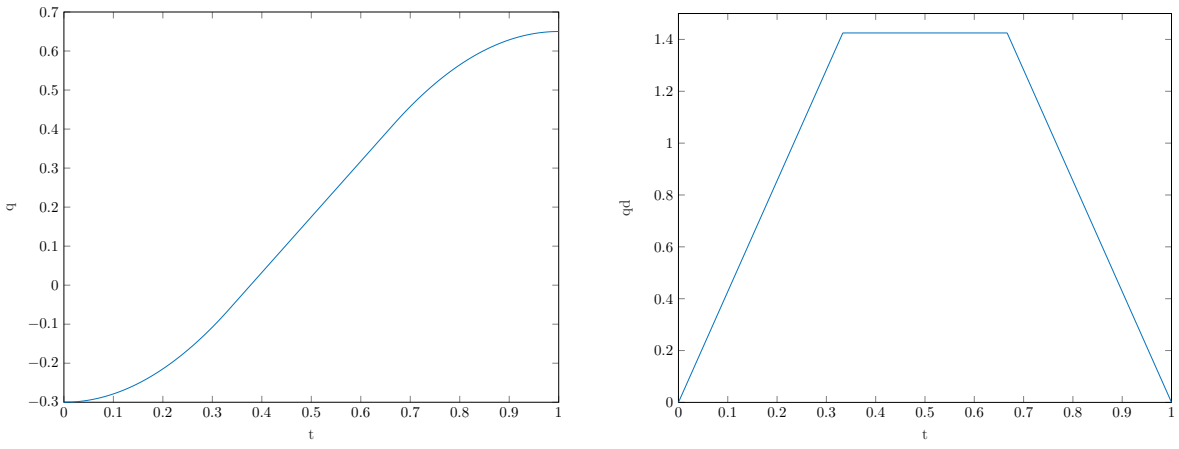
All robots have a specific cycle time and control frequency at which the internal motion controller sends targets to the joints[5]. From a user perspective, this might not be entirely apparent, as often the robots will be controlled *offline*, with the internal controller computing the timing required for the desired motion. If *online* control is used, more finely defined motions can be implemented.

3.1 Offline Control

Typically, most industrial robots are controlled using point-to-point motions in joint or Cartesian space. With Universal Robots, this corresponds to the *movej* and *movel* commands, respectively. However,

[4] In this case, other inverse kinematics formulations are used, such as the *(Moore-Penrose) pseudoinverse* or *Jacobian transpose*. Since, for now, you will only be working a 6-DOF manipulator, these more advanced cases will not be covered here.

[5] For CB3-Series UR robots, this is 8 ms/125 Hz, whereas for e-Series UR robots it is 2 ms/500 Hz.



(a) Position

(b) Velocity

Figure 1: Example of a trapezoidal velocity profile (fig. 1b) and the corresponding position (fig. 1a). This kind of trajectory profile is typically used in point-to-point motion.

these motions are restricted in that they will always start at standstill, accelerate to a constant speed, and decelerate again to standstill. That is, the speed at both the start and end of the trajectory will be zero. This is what is known as a trapezoidal velocity profile, as shown in fig. 1.

We call this type of control *offline control*, because from the user's point of view one simply specifies the parameters of a point-to-point move command, which could include velocity, acceleration and time duration in addition to a target position, and the internal robot controller figures out how to discretize this into control targets that comply with the cycle time of the robot.

3.2 Online Control

Whereas offline control is simpler for the user, as calculating a full trajectory profile is left to the robot controller, it also limits the kind of trajectories that can be achieved. By using online control, we can first use standard math to design a trajectory that meets our constraints, and then send it to the robot using either position or velocity control.

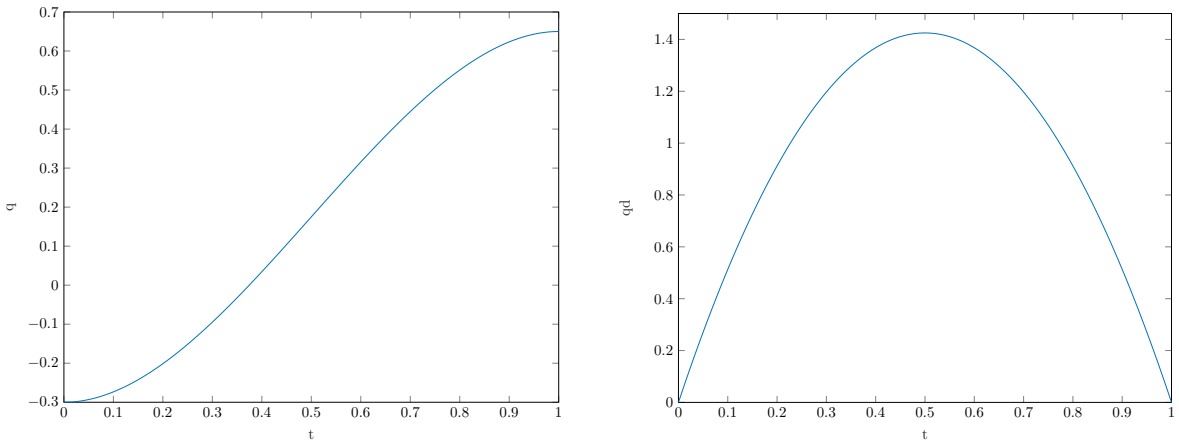
3.2.1 Designing Trajectories

Designing a robot trajectory is essentially a mathematical curve fitting problem. Given a series of constraints, one finds a mathematical function that complies with them.

Let us look at three examples:

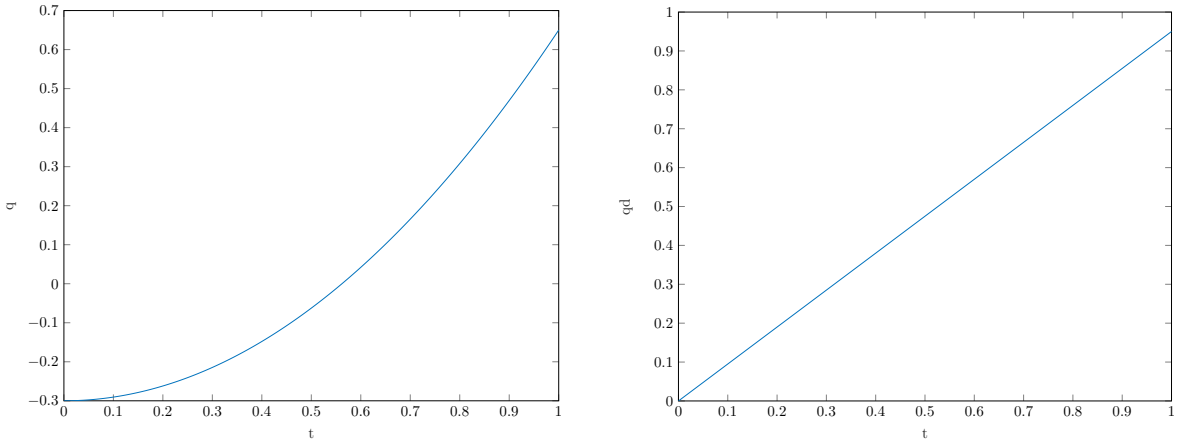
- **Trapezoidal velocity profile**: The trajectory shown in fig. 1 shows a trapezoidal velocity profile. The constraints here are: $q(t = 0) = q_0$, $q(t = T) = q_{\text{end}}$, $\dot{q}(t = 0) = 0$, $\dot{q}(t = t_{\text{end}}) = 0$, where T is the time at the end of the movement.
- **Cubic polynomial**: Figure 2 is the result of fitting a cubic polynomial with the constraints: $q(t = 0) = q_0$, $q(t = T) = q_{\text{end}}$, $\dot{q}(t = 0) = 0$, $\dot{q}(t = t_{\text{end}}) = 0$.
- **Constant acceleration/linear velocity ramp**: The trajectory in fig. 3 comes from the constraints: $q(t = 0) = q_0$, $q(t = T) = q_{\text{end}}$, $\ddot{q} = K$, where K is a constant.

Note that trajectories can be designed both in task space and joint space, and this should be decided on based on the problem at hand. Often, it will be easier to do so in joint space, because calculating integrals and derivatives in joint space is straightforward (you can use any of the methods you have already seen in your other courses), whereas doing so in task space is only straightforward for the translational part, but special care must be taken for the orientation. A discussion of this is well beyond our scope.



(a) Position (b) Velocity

Figure 2: Example of a cubic polynomial trajectory.



(a) Position (b) Velocity

Figure 3: Example of a constant acceleration trajectory with a linear ramp in velocity.

The Tossing Task

For the tossing task in your project, you will need to use basic physics to compute the desired motion of the ball in order to hit the target. From this, you will end up with some constraints on the position and velocity of the ball at the point that it is released from the robot gripper. This will most likely look like a pair $\boldsymbol{x}_{\text{release}}, \boldsymbol{v}_{\text{release}}$. It should be straightforward to see that you can, e.g. use eq. (8) to then derive a corresponding $\dot{\boldsymbol{q}}_{\text{release}}$. You could then, for example, use this to design a velocity trajectory in the joint space, and use online velocity control to send this to the robot.

3.2.2 Controlling the robot

Once we have computed a target robot trajectory that meets all of our constraints, we need to send this over to the robot. As mentioned, each robot will have a control frequency and cycle time. We will therefore need to discretize the trajectory accordingly. In our control code, we will create a loop that iterates through the discretized trajectory and sends a new target in each control cycle. It is crucial that this loop is time-synchronized to the robot's control frequency, or otherwise the computer will be sending targets over to the robot much faster than the robot can process them. This time-synchronized loop is the reason why this is called *online control*; this type of control also allows feedback and to

react to robot sensor inputs in real-time.

A choice when controlling the robot motion is whether to do so using position control or velocity control. This choice can sometimes be given by our trajectory. For example, the trajectories in figs. 1 and 3 are arguably much easier to define based on the velocity than based on the position. Keep in mind that it is always possible to go from a velocity trajectory to a position trajectory by integration or vice versa using differentiation.

Online Control of Universal Robots

To control the Universal Robots online, one can use the *servoj* command for joint-space position control, *speedj* for joint-space velocity control, or *speedl* for task-space velocity control. Documentation for URScript commands can be found here: https://www.universal-robots.com/download/manuals-cb-series/script/script-manual-cb-series-sw3154/.

Instead of using the teach pendant to program the robot, it is also possible to interface with it from a computer using a LAN connection and the following C++/Python library: https://gitlab.com/sdurobotics/ur_rtde
The following link also shows how to create a time-compensated loop to ensure that the targets sent to the robot comply with its cycle time: https://sdurobotics.gitlab.io/ur_rtde/examples/examples.html#speedj-example

References

[1] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*, ch. 4, pp. 161–189. London: Springer London, 2009.

[2] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*, ch. 3, pp. 105–160. London: Springer London, 2009.

[3] K. J. Waldron and J. Schmiedeler, "Kinematics," in *Springer Handbook of Robotics* (B. Siciliano and O. Khatib, eds.), pp. 11–36, Cham: Springer International Publishing, 2016.