ENEE 467: Robotics Project Laboratory
Manipulator Kinematics with the Robotics Toolbox for Python
Lab 2

Learning Objectives

The purpose of this lab is to provide a hands-on introduction to fundamental concepts in manipulator kinematics using the Robotics Toolbox for Python. After completing this lab, students should be able to:

- Model robots using URDF and DH parameters

- Map joint configurations to tool poses and vice versa via forward and inverse kinematics

- Map joint and end-effector trajectories to robot motion

- Simulate unconstrained robot motion by interpolating between joint configurations

- Command joint velocities based on desired tool velocities using the manipulator Jacobian and velocity kinematics

- Move a serial-chain robot (simulated using basic line plots) to trace a path in Cartesian space.

Manipulator kinematics refers to the mathematical study of how a robot arm's joint motions relate to the spatial position and orientation of its **end-effector** or **tool**. It is commonly categorized into two connected sub-problems, namely: forward kinematics (hereafter, FK) and inverse kinematics (hereafter, IK). While FK computes the tool pose from known joint variables, IK solves the more challenging problem of finding joint configurations that achieve a desired tool pose. Together, they form the backbone of higher-level motion planning and control, and are thus crucial for robotic manipulation. Therefore, this lab is designed to equip you with the tools and mathematical thinking to understand kinematic modeling, practice kinematics-based problem solving, and prepare for ROS 2 motion planning and control in later labs.

Related Reading

1. K. Lynch and F. Park, *Modern Robotics: Mechanics, Planning, and Control*, Cambridge University Press, 2nd ed., 2019 (§4, §5, and §6).

2. ROS URDF Tutorials. External Link.

Lab Instructions

Follow the lab procedure closely. Some code blocks have been populated for you and some you will have to fill in yourself. Pay close attention to the lab Questions (highlighted with a box). Answer these as you proceed through the lab and include your answers in your lab writeup. There are several lab Exercises at the end of the lab manual. Complete the exercises during the lab and include your solutions in your lab writeup.

List of Questions

List of Exercises

How to Read This Manual

Throughout the manual, boxes beginning with a **Question #** title contain questions to be answered as you work through the lab:

> **Question 0.** A question for you.

Boxes with a ✎ symbol (like the one below) are extra theoretical details provided for your enlightenment or just general information about the lab. We recommend that you read them during the lab, but they may be skipped during the lab and revisited at a later time (e.g., while working on the lab report):

> ✎ Some extra context and info.

Terminal commands and code blocks will be set in gray-colored boxes (usually for Python syntax)

```python
import math # some important comment you should totally read
```

while terminal outputs will be set in yellow-colored boxes

```
some terminal output
```

Files and directories will be set in magenta color with a teletype font, e.g., `~/some/dir/to/a/file.txt`, code variable names and in-text Python commands or snippets will be set in blue teletype font, e.g., `a = [0, 0, 0]`, while Python type hints (and occasionally XML tags) will be set in green teletype font, e.g., `save(path: str)` signifies that the argument to the `save` function is a string. Lastly, figure captions go below figures, table captions appear above tables, and cyan text highlights a link to an external online reference.

1 Lab Procedure

1.1 Prerequisites

This lab assumes some familiarity with the Bash shell at the level of file system navigation and basic read-write operations via the Command Line Interface. We also assume that you have done some prior Python programming, and optionally, have some exposure to numerical computation with the NumPy Python library, basic linear algebra, and rigid-body motion.

1.2 Getting this Lab's Code

To begin, in a new terminal session, navigate to the `~/Labs` directory and then clone the lab repo:

```
cd ~/Labs
git clone https://github.com/ENEE467-F2025/lab-2.git
cd lab-2/docker
```

Verify that the lab-2 image is built on your system by running the following command

```
docker image ls
```

Make sure that you see the `lab-2` image in the output (the tag, container size, and time information will differ from those on your machine):

```
lab-2-image    latest    88f2baf1996e    2 minutes ago    6.01GB
```

Now launch the Docker container with the following command

```
userid=$(id -u) groupid=$(id -g) docker compose -f lab-2-compose.yml run \
--rm lab-2-docker
```

You should be greeted with the following prompt indicating you are now inside the Docker container:

```
(lab-2) robot@docker-desktop:~$
```

Test your setup by executing a Python script within the `lab-2/src` directory in the terminal running the container:

```
cd ~/lab-2/src
python 2r_dh.py
```

The script should print a table showing the DH parameters of a simple 2R manipulator. You may now proceed with the rest of the lab procedure.

2 Robot Modeling & Description Formats

Generally, there are two standard and complementary ways of describing robots, namely the URDF (Unified Robot Description Format) and the Denavit–Hartenberg (DH) parameters. In this section, you'll work on retrieving kinematic information from both representations starting with the URDF.

2.1 URDF

The URDF is a robot description format based on the e**X**tensible **M**arkup **L**anguage or XML. It provides a complete description of a robot modeled by a collection of rigid bodies *(links)* connected by *joints* (see Figure 1; top-left) using nested XML tags (Figure 1; right):

i. **Link tags**: these contain meta-tags (tags containing tags) specifying visual info (`<visual></visual>`) like color and material, collision info (`<collision></collision>`) like geometry and link origin for physics simulation and collision checking, and inertial info (`<inertial></inertial>`) for robot dynamics and simulation

ii. **Joint tags** : these define the connection between two links and can be **revolute**, **prismatic**, **cylindrical**, or **spherical** (specified by the joint tag's **type** attribute). A joint's metadata contains info about its parent and child links, referenced by the **name** attributes of the parent (`<parent />`) and child (`<child />`) tags, its SE(3) pose (`<origin />`), its axis of rotation (for revolute joints) or translation (for prismatic joints), specified in the `<axis />` tag, and its effort, position, and velocity limits (defined in its <limit /> tag).
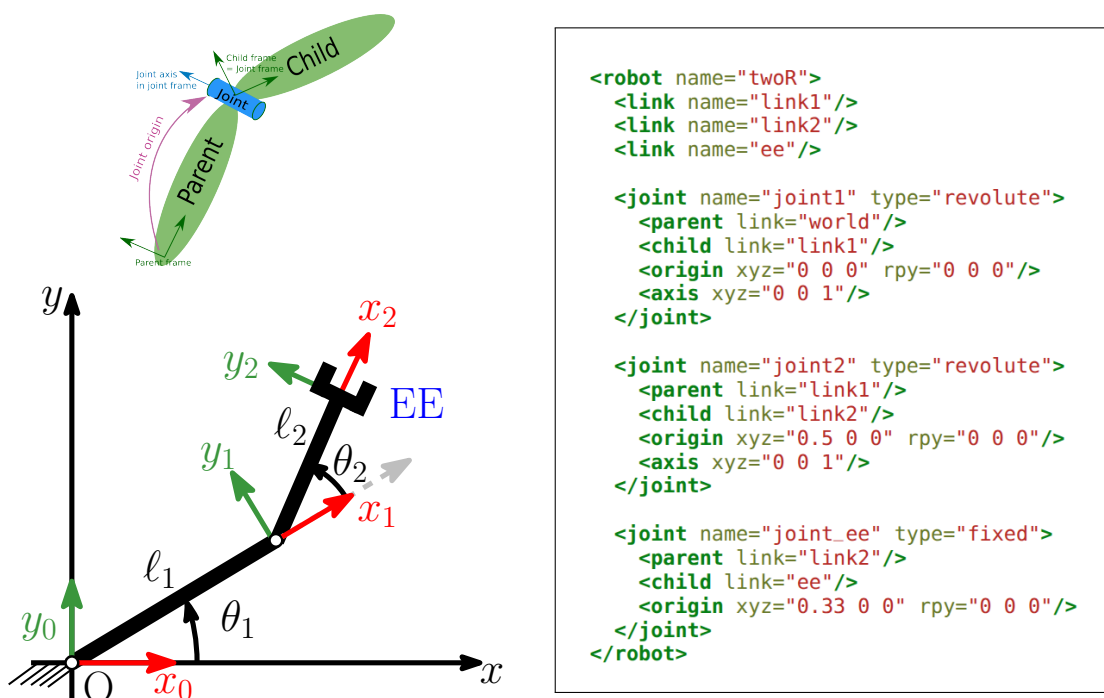

Figure 1: Illustrating how a joint (top left) connects links in the kinematic chain, with the kinematic diagram of a 2R planar manipulator (bottom left) compared against its URDF description (right). The $+z$ axis points out of the page.

Question 1. In the file `src/utils/urdf_builder.py` is a class for building and visualizing basic URDF files. The **add_link** and **add_joint** methods are for adding link and joint tags; they are reproduced here partially to provide more context:

```python
# method for constructing a link in the URDF description
def add_link(self, name: str, length:Union[float, None]=None, \
        radius:Union[float, None]=None, mass:float=1.0, geometry="box"):

# method for defining a joint between two links in the URDF description
def add_joint(self, name:str, parent:str, child:str, joint_type="revolute",
              origin=(0,0,0), rpy=(0,0,0), axis=(0,0,1)):
```

The **plot_robot(q)** method will produce a simple plot of the robot (using info defined in the built URDF) at the specified configuration **q** (which is a **list** of length equal to the number of joints, with each element in the list corresponding to a joint angle in radians). The **save(path: str)** method will save the built URDF file to the **string** file path argument specified.

(a). Using the appropriate methods from the aforementioned, modify the sections of the `src/build_2r_urdf.py` file marked with a `TODO` flag to build a 2R manipulator with the following specs:

   - name: **twoR_lab2**
   - Link 1: {name: **"twoR_link1"**, length: **0.4**, geometry: **"box"**}
   - Link 2: {name: **"twoR_link2"**, length: **0.25**, geometry: **"box"**}
   - End-effector (EE) link: name:**twoR_ee**
   - Three joints, comprising:
     * one (+z axis) revolute joint (named **twoR_joint1**) between a **world** frame at the origin (i.e., **(0, 0, 0)**) and the link named **twoR_link1**
     * one (+z axis) revolute joint (named **twoR_joint2**) between the links named **twoR_link1** and **twoR_link2**
     * a fixed joint (named **twoR_joint_ee**) between the link named **twoR_link2** and the end-effector.

(b). Save the built URDF file as `twoR.urdf` using the **save** method. Include the complete URDF (text) description in your final lab report.

(c). Plot the robot at the **[0, 0.7854]** configuration, and save the plot in your current working container directory using the save icon on Matplotlib, and include the plot in your final lab report.

2.2 Computing Homogeneous Transformation Matrices from a URDF File

Using information from the URDF file of a serial-chain robotic manipulator, it is straightforward to perform kinematic computations, such as computing homogeneous transformation matrices that transform the pose of a **source** link frame in the URDF to a **target** link frame, expressed in the
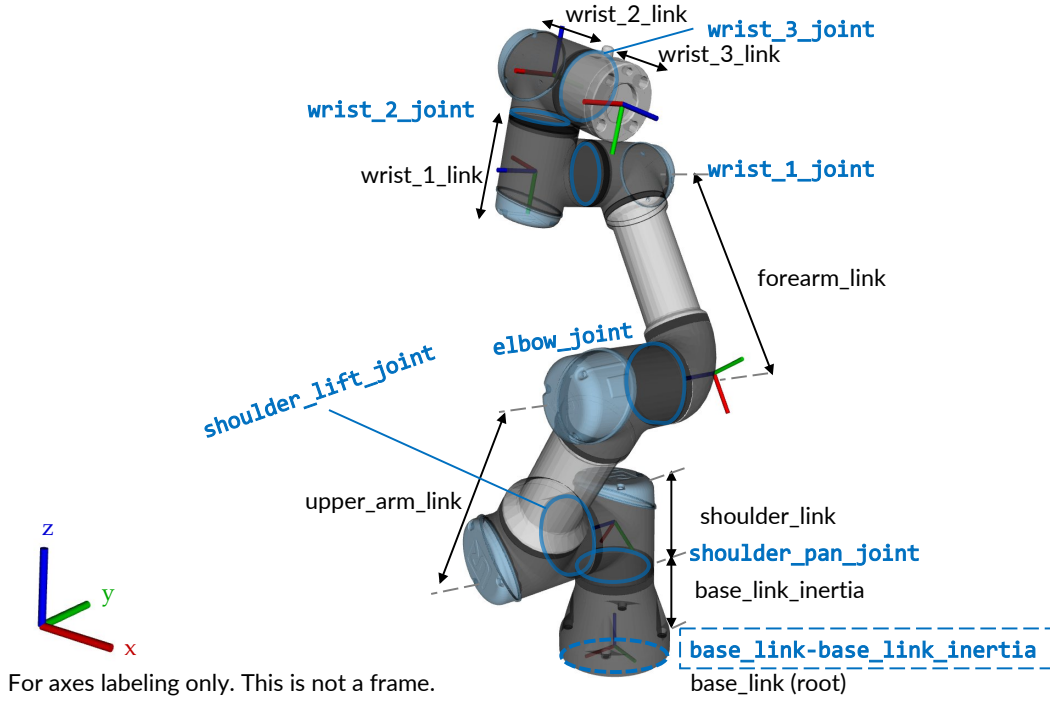
Figure 2: Link frames and joints of the UR3 robot. Joint names are set in blue color, while link names are set in black. The UR3's end-effector frame is typically taken to be the attached to the `tool0` link (not annotated), a fictitious link whose origin differs from that of the `wrist_3_link` frame only by a rotation, i.e., the translation component of the homogeneous transformation between both frames is the zero vector in $\mathbb{R}^3$. The `base_link-base_link_inertia` joint is fixed.

"world" or inertial frame. We do this by multiplying the individual transformation matrices from the source up to the specified target frame as you saw from Exercise 2 in Lab 1:

$$T_{\text{target}}^{\text{source}} = T_{\text{frame\_1}}^{\text{source}} \cdot T_{\text{frame\_2}}^{\text{frame\_1}} \cdots T_{\text{target}}^{\text{frame\_n}}, \tag{1}$$

where each individual transformation is an SE(3) matrix. Since XML files can be easily parsed into a tree-like data structure, we can compute the transformation between two link frames directly from the URDF by reading off and multiplying the **static** (origin) and the motion (i.e., **axis**) transforms from the metadata of the joints between link pairs from the source to the target frame. If the source and target frames coincide, then there is no joint between them and the transformation is the **identity** matrix. Question 2 outlines a task that should help solidify this concept.

> **Question 2.** A Python class for parsing URDF files and querying for the homogeneous transformation between link frames has been provided for you; we have also provided a URDF file for the UR3 at the file directory: `src/urdf/ur3.urdf`. The Python class provides a method `_compute_T(from_frame: str, to_frame: str, config: Robot.Configuration)` that returns a `spatialmath.SE3` object containing the homogeneous transformation between the provided link frames at a specific robot configuration (encapsulated by a `Robot.Configuration` constructor). The configuration constructor takes as argument the actuated joints of the robot and a list of joint angles to set (in radians). For example, to

retrieve the transformation matrix from the UR3's **base_link** frame to the **shoulder_link** frame at the **zero** configuration, you would use:

```
# create a Robot.Configuration object with all-zero joint angles
q_test = robot.Configuration(joints=robot.actuated_joints,
                             joint_values=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0] )
# Compute the transformation from base_link to shoulder_link
SE3_base_shoulder = robot._compute_T("base_link", "shoulder_link", \
                                     config=q_test)
# To access the underlying 4x4 matrix, use the .A attribute
T = SE3_base_shoulder.A
```

Given the aforementioned background, complete the following questions:

(a). Modify sections of the file `src/compute_urdf_T.py` marked with a `TODO` flag to compute the homogeneous transformation matrix (at the configuration with **joint_values = [1.5, -2.5, 1.5, -1.05, -1.55, 0.0]** between **each consecutive link** frame pair of the UR3 (see: Figure 2). The link names are stored in the variable **link_list**. Arrows (`->`) indicate the direction of tree/chain traversal:

```
base_link -> shoulder_link -> upper_arm_link -> forearm_link ->
wrist_1_link -> wrist_2_link -> wrist_3_link -> tool0
```

Provide the transformation matrix for each pair in your report.

(b). By inspecting the UR3 URDF file, express the homogeneous transformation from **base_link** to **base** using only the basic transformations R_x, R_y, R_z and T_x, T_y, T_z, and include a **one-sentence justification** of your answer in your report.

(c). Using the same `_compute_T` function in (a), compute the transformation matrix for the (`base_link`, `base`) link pair, and include the matrix in your report.

🌿 In kinematics, the position and orientation of each link in a kinematic chain are determined **entirely** by the joint variables. This is the fundamental principle in forward kinematics where we calculate the pose of the chain's end-effector (and by extension, all **intermediate** link frames) based on the joint angles (for revolute joints) or joint displacements (for prismatic joints). We include the motion axis transform to incorporate the joint's **current** position or displacement, thus avoiding stale transforms.

2.3 The Denavit–Hartenberg (DH) Convention

The DH convention is a systematic way to assign coordinate frames to the links of a serial-chain robot and express FK as a sequence of transformations. Each link i is characterized by four parameters described on Table 1. Each joint i is associated with a joint variable q_i that corresponds to either the **joint angle** θ_i if the joint is **revolute**, or the **joint offset** d_i if the joint is prismatic. For revolute joints, the joint angle θ_i is variable and d_i is constant; conversely, d_i is variable while θ_i is constant for prismatic joints.

In the DH convention, each homogeneous transformation $A_i(q_i)$ corresponding to the joint variable q_i, is represented as a product of four basic transformations

$$A_i(q_i) \equiv A_i = R_z(\theta_i) \cdot T_z(d_i) \cdot T_x(a_i) \cdot R_x(\alpha_i), \qquad (2)$$

which works out to be the following matrix:

$$A_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (3)$$

Using (3), we can then write the transformation matrix that expresses the position and orientation of link j with respect to link i by simply multiplying the homogeneous transformations

$$T_j^i = \begin{cases} A_{i+1}A_{i+2}\cdots A_{j-1}A_j & \text{if } i < j \\ I_4 \text{ } (4\times 4 \text{ identity matrix}) & \text{if } i = j \\ \left(T_i^j\right)^{-1} & \text{if } j > i \end{cases}. \qquad (4)$$

To concretize things, let's construct the DH model for a 2R robot. Open the file `2r_dh.py` in the **src** folder. Using the Robotics Toolbox for Python (hereafter, **RTB4PY** or "**The Toolbox**"), the script builds a DH model of the 2R planar robot shown in Figure 1 by specifying its four DH parameters. The link lengths are set to 0.5 m and 0.33 m, consistent with the URDF specification, and no angular offsets are applied for either joint (i.e., `robot.q = q + [0, 0] = q`):

```python
import roboticstoolbox as rtb # we import the toolbox using a simpler alias

# costruct a 2R robot using standard DH params:
links = [rtb.RevoluteDH(a=0.5, alpha=0),      # joint 1
         rtb.RevoluteDH(a=0.33, alpha=0)]     # joint 2
twoR = rtb.DHRobot(links, name="2R")
```

Print a summary of the constructed model using **print(twoR)**, and verify that this yields:

```
DHRobot: 2R, 2 joints (RR), dynamics, standard DH parameters
+-----+----+------+-------+
| thj | dj |  aj  | alphaj|
+-----+----+------+-------+
| q1  | 0  | 0.5  | 0.0   |
| q2  | 0  | 0.33 | 0.0   |
+-----+----+------+-------+
```

Table 1: Denavit–Hartenberg (DH) Convention

Parameter	Description
a_i	Link length (distance along x_i from z_i to z_{i+1})
α_i	Link twist (angle about x_i from z_i to z_{i+1})
d_i	Link offset (distance along z_i from x_{i-1} to x_i)
θ_i	Joint angle (angle about z_i from x_{i-1} to x_i)

Instances of built-in DH models within the toolbox can be created using the `rtb.models.DH.<RobotName>()` constructor (an example of the UR3 is provided below):

```python
import roboticstoolbox as rtb
ur3 = rtb.models.DH.UR3() # load a DH UR3 model
```

> **Question 3.**
>
> (a). Using a **two-column** table, collect the UR3's joint names and corresponding origins (from the `src/urdf/ur3.urdf` file).
> (b). Compare the joint data in your table with the DH model from `rtb.models.DH.UR3()`. What kinematic parallels link the URDF and DH representations (see Equations (2) and (3)), and can both robot descriptions be reconciled into a common model? In your report, cite your UR3 joint data and list at least **two** specific correspondences.

2.4 Computing the Forward Kinematics Equations using the DH Parameters

Using the DH convention, we can compute the forward kinematics of a serial chain with n joints (and thus $n + 1$ links, including the base) by multiplying the transformations from the base frame (**frame 0**) to the n-th link:

$$T_n^0 = T_1^0 \cdot T_2^1 \cdot \cdots \cdot T_n^{n-1} = A_1 \cdot A_2 \cdot \cdots \cdot A_n. \tag{5}$$

Let's practice calculating FK for a simple 2R robot using the DH convention. Using the output of the `print(twoR)` command and Equations (3) and (5), we can easily see that the transformation matrix for the two-link manipulator is given by:

$$T_2^0 = T_1^0 \cdot T_2^1 = A_1 \cdot A_2. \tag{6}$$

Each element of the **links** array, which we used to build the **twoR** model in the `2r_dh.py` file shown in the previous Python snippet, has the following properties:

```
link.a       link.alpha      link.d       link.theta
```

Using these properties, we can compute T_1^0 as shown below (with T_2^1 left as an exercise):

```python
import numpy as np

T_01 = np.array([
    [np.cos(links[0].theta),
        -np.sin(links[0].theta) * np.cos(links[0].alpha),
        np.sin(links[0].theta) * np.sin(links[0].alpha),
        links[0].a * np.cos(links[0].theta)],
    ]) # Truncated. See the 2r_dh.py file.
```

Question 4. Modify the file `2r_dh.py` to implement Equation (6) for the 2R planar robot in Figure 1 by computing the second (and last) transformation in the chain (i.e., T_2^1) and multiplying it to `T_01` in the code block above using the `@` operator (since `T_01` and `T_12` are NumPy arrays). Add the final matrix to your lab report. Verify that your result yields the same output as the simpler command: `twoR.fkine([links[0].theta, links[1].theta])`. As you will soon see, the `fkine` command is a simple method provided by the toolbox for FK calculations.

3 Forward Kinematics of Serial Chains

3.1 Computing FK from Homogeneous Transformation Matrices

Because robotic tasks are most naturally expressed in the tool frame but we can typically only measure the **position** of the **joints** via encoders and have no way of ascertaining the end-effector's position without external instrumentation, we need a mapping between the joint angles we can measure and the end-effector's pose. The FK equations (Equation (7)) provide this very important mapping, typically in terms of the manipulator's link lengths and the joint configuration (q):

$$f(q) = \begin{bmatrix} R(q) & p(q) \\ 0 & 1 \end{bmatrix} \in SE(3). \tag{7}$$

To compute $f(q)$ for a kinematic chain, we compose the homogeneous transformations between consecutive links (from the root link to the end-effector) as

$$f(q) = T_{\text{link1}}^{\text{root}} \cdot T_{\text{link2}}^{\text{link1}} \cdots T_{\text{ee\_link}}^{\text{linkN}}, \quad \text{where each } T \text{ term is an } SE(3) \text{ object.} \tag{8}$$

Work through Question 5 to practice computing a robot's FK from its URDF.

Question 5.

(a). Within the same file you modified in Question 2, modify the block highlighted with a `TODO` comment to find the **matrix product** of **all** transformation matrices in Question 2 (a) in ascending order of the frame pair's index within the kinematic chain, i.e., $T_{\text{shoulder\_link}}^{\text{base\_link}} \cdot T_{\text{upper\_arm\_link}}^{\text{shoulder\_link}} \cdots T_{\text{tool0}}^{\text{wrist\_3\_link}}$. Include the final matrix product in your report.

Note: To multiply two `spatialmath.SE3` objects, use the $*$ operator; if working instead with `spatialmath.SE3` objects converted to NumPy arrays (e.g., `spatialmath.SE3.A`), then you can use the `@` operator or the **numpy.matmul** function:

```python
import numpy as np
A = np.eye(4)
B = A
mat_prod = np.matmul(A, B) # works only if A and B are NumPy arrays
```

(b). Compare your answer in Question 5 (a) with the homogeneous transformation matrix between the link pair: (`base_link`, `tool0`) (use the same function in (a) to find this). In your report, provide the **modified code block** (the raw text or a screenshot is fine) as well as reasoning (in no more than **two** sentences) why you think the matrices are the

same or different (depending on your results).

(c). Modify the variable `q` in the file `q5c.py` to `[1.5, -2.5, 1.5, -1.05, -1.55, 0.0]`. Then change the `end` argument of the `robot.fkine` method to `'tool0'`. Run the modified script. The script will print an SE(3) matrix to the terminal. Read off the translation component of the matrix, and provide the translation vector in your report. Compare the printed SE(3) matrix and your result in (b). Verify that they are the same.

As you've seen from the questions up to this point, for a two or three link manipulator, it is a simple matter to compute FK by hand by straightforward matrix multiplication; anything more complex, and the calculations quickly become tedious. And because FK changes with the joint configuration, we don't want to keep multiplying matrices for each configuration anytime we want to do FK. Fortunately, we can easily compute FK for any serial chain using its kinematic model (contained in the URDF spec or specified as DH parameters) using a handy method provided by RTB4PY. We go over an example and question in the next section.

3.2   Computing FK for a Kinematic Chain with the Toolbox

Suppose we load a standard UR3 model and specify an array of joint angles (we will use variables namespaced by **q** for Python objects related to joint configurations). We can compute the resulting end-effector pose (as a transformation matrix) using the method **fkine** from the **UR3** class instance. Inspect the `src/ur3_fk.py` file for example Python code (shown below) that implements FK:

```python
import roboticstoolbox as rtb
import numpy as np # library containing important kernels for matrix math

robot = rtb.models.DH.UR3() # Load a UR3 model
q = np.array([0, -np.pi/4, np.pi/4, 0, np.pi/3, 0]) # joint configuration
T = robot.fkine(q) # Compute FK
```

Using the `fkine` method, the script computes a 4×4 homogeneous transformation matrix **T**, which encodes our robot's end-effector position and orientation in task space (for the UR3, this is the **tool0** frame). Issue a **print(T)** statement on a new line at the end of the script and run it. You should see the following matrix output in the terminal

```
 0.5        0        -0.866    -0.4565
-0.866      0        -0.5      -0.1533
 0          1         0         0.2388
 0          0         0         1
```

In the homogeneous transformation matrix (**T**), as you learned in class, the first three rows of the last column correspond to the end-effector's position, while the rotation matrix (the 3×3 matrix block in the first three columns of **T**) defines the end-effector's $SO(3)$ orientation. We can plot the robot at this configuration to confirm visually that the FK calculations are correct. After the **print(T)** line in the `ur3_fk.py` script you just modified, add the following new line

```python
robot.plot(q, block=True, jointaxes=True, shadow=True, name=True)
```

You should see a Matplotlib window showing a simple line plot of the robot.

12

> **Question 6.** By modifying the `ur3_fk.py` script, try computing FK at a configuration near the robot's upper limit (accessible using `robot.qlim[1]`, where `robot.qlim` is a 2×6 NumPy array). What is the **position** of the robot's end-effector? Plot the robot at the upper limit configuration using the `robot.plot` command. Notice how some links appear folded over one another, creating an undesirable configuration. Save the Matplotlib figure and add it to your report, along with the array representing the robot's end-effector position.

Great, you just computed FK! But does that mean the robot can actually reach the configuration **q**? FK only maps joint angles to an end-effector pose; it says nothing about the quality of that configuration. In fact, the math may work out even if the joint angles place the robot at or near a **singularity**, where the end effector loses motion in one or more directions.

4 Kinematic Singularity & The Manipulator Jacobian

4.1 The Manipulator Jacobian

To move from the FK realm of tool **positions** and **orientations** to **velocities**, we need a new object — the **Jacobian matrix** or **Jacobian** for short. The Jacobian is an important object in kinematics you'll see repeatedly. Mathematically, the (**space**) Jacobian J relates the velocity of the end-effector in the **base (space) frame** ($\dot{x}$) to joint-space velocities ($\dot{q}$), and depends on the configuration, q:

$$\dot{x} = J(q)\dot{q}, \quad \text{with} \quad J = \begin{bmatrix} J_v(q) \\ J_\omega(q) \end{bmatrix}. \tag{9}$$

J is a block matrix with 6 rows and n columns (corresponding to the number of joints) and is typically decomposed into **linear** (top 3 rows) and **angular** (bottom 3 rows) velocity components. For example, the Jacobian corresponding to the UR3 at the configuration $[0, -\frac{\pi}{2}, 0, -\frac{\pi}{2}, 0, 0]^\top$ is

```
[[0.19 -0.54  -0.3  -0.09   0.08   0.] #  }                 lin. vel. along base x
 [0.    0.    0.    0.    0.    0.] #  | J_v              lin. vel. along base y
 [0.    0.    0.    0.    0.    0.] #  }                 lin. vel. along base z
 [0.    0.    0.    0.    0.    0.] #  }                 ang. vel. about base x
 [ 0.   -1.   -1.   -1.    0.   -1. ] #  | J_omega        ang. vel. about base y
 [ 1.    0.    0.    0.    1.    0. ]]#  }                 ang. vel. about base z
```

which makes sense, since just by inspecting the first row, we see that motion in all joints except the last joint yields instantaneous velocity along the base x axis, and the last column is all zeros (except for rotation about the y axis) since the last joint can only **rotate** the tool frame but not **translate** it. Since the FK equations can be applied to any frame pair in a kinematic chain, we have all the components we need to compute J, so let's write a simple Jacobian calculator to concretize things.

💡 It helps to picture the Jacobian as a **summary** of **joint velocity contributions**:

Revolute joints contribute a column $\in \mathbb{R}^6$:   **Prismatic** joints contribute:

$$J_i = \begin{bmatrix} z_{i-1} \times (p_{ee} - p_{i-1}) \\ z_{i-1} \end{bmatrix} \quad \begin{matrix} {\color{blue}J_{v_i}} \\ {\color{red}J_{\omega_i}} \end{matrix} \qquad (10) \qquad J_i = \begin{bmatrix} z_{i-1} \\ 0 \end{bmatrix} \quad \begin{matrix} {\color{blue}J_{v_i}} \\ {\color{red}J_{\omega_i}} \end{matrix} \qquad (11)$$

where p_{ee} is the end-effector position (translation of T_n^0), p_{i-1} is the position of link $i-1$ (translation of T_{i-1}^0), z_{i-1} is the unit vector along joint i's motion axis ($0 \in \mathbb{R}^3$ for prismatic joints), and $\times$ denotes the cross product. Joint i connects link $i-1$ (child) to link i (parent), with the base as link 0; the origin of joint i in the base frame is the end of its child link, or the origin if at the root. With each J_i, the full Jacobian is $J = [J_1 \mid J_2 \mid \ldots \mid J_n]$, where $\mid$ denotes column-wise concatenation.

4.2 Writing a Simple Jacobian Calculator

As you can probably tell from Equations (10) and (11), computing $J(q)$ amounts to simply stacking columns into matrices, one for each joint of the kinematic chain. We can perform this "stacking" operation for each Jacobian component, as you'll explore in the next question.

Question 7. Open the file `compute_urdf_jac.py`. The file contains a partial implementation of a simple Jacobian calculator function (`compute_jacobian`) that computes the linear and angular velocity Jacobian components using forward kinematics kernels from a custom URDF parser class we have provided. The script populates a NumPy array of zeros with the computed values, and returns the final matrix as the Jacobian.

(a). By modifying the `TODO`-highlighted blocks, compute the Jacobian by implementing Equation (10) for the **revolute joint** (you can use NumPy's `cross` method to compute the cross product). For example, to compute the Jacobian for the case of a prismatic joint (i.e., Equation (11)), you would set:

```
Jv = z_i_prev
Jw = np.zeros(3)
```

for each joint in the kinematic chain. In your report, provide the resulting Jacobian, **as well as** (text or image) code snippets highlighting the **modified** lines that produced your result.

(b). After the `Simple URDF Jacobian Calculator:` print statement within the `compute_urdf_jac.py` file, add the following lines:

```
robot_rtb = rtb.models.UR3()
print("--------------------------")
print(f"Toolbox Jacobian Result: \n{np.round(robot_rtb.jacob0(q_arr,\
                                end='tool0'), 4)}")
```

> **Note**: Ensure the print statements are properly indented after pasting. Run the code and compare the matrix printed after **Toolbox Jacobian** with your result from (a), verifying agreement to three decimal places. Include the final `compute_urdf_jac.py` file with your report.

4.3 Computing the Jacobian using the Toolbox

In RTB4PY, we can directly compute the Jacobian using yet another one-liner, `robot.jacob0(q)`, a function you've already seen from Question 7 (b). The **robot.jacob0** method yields the robot's $6 \times n$ Jacobian at the joint configuration **q**, expressed in the base (**0**) frame. To use it, simply call the method from the corresponding `rtb.models.DH.<RobotClass>` or `rtb.models.<RobotClass>` robot model constructor instance. An example using the UR3 DH model follows:

```
robot = rtb.models.DH.UR3()
J = robot.jacob0(q) # assuming q is not NoneType and len(q) == 6
```

> **Question 8.** Using the toolbox, compute the Jacobian for the UR3 at the configuration:
>
> ```
> q = [-1.860, -2.990, 0.670, -2.000, 1.610, -0.007]
> ```
>
> What are the linear (J_v) and angular velocity (J_ω) Jacobians? Include these in your report.

> ✎ For a 6-DoF chain ($n = 6$), the Jacobian $J(q)$ is square and invertible if $\det(J) \neq 0$. For non-square cases, we use the **Moore–Penrose** pseudo-inverse $J^\dagger$ (e.g., `numpy.linalg.pinv(J)`). RTB4PY also provides related functions such as `robot.jacob0_dot` and `robot.jacobe`, though `jacob0` is sufficient here.

4.4 Identifying Singular Manipulator Configurations

Once the Jacobian is computed, determining whether the robot is at a singular configuration is simply a matter of computing the **Jacobian's inverse**. From linear algebra, we know that a non-invertible matrix is **rank-deficient**, meaning one or more rows or columns can be written as **non-zero** factors of another. So if **J(q)** is an $n \times n$ matrix and its rank (or equivalently, in most cases, the existence of its inverse) is less than **n**, then **q** must be singular. Thus, we can check singularity by computing the inverse of **robot.jacob0(q)**. To compute the inverse or rank of a matrix, NumPy provides handy functions:

```python
import numpy as np
J = robot.jacob0(q) # assuming q and robot = rtb.DH.UR() are not NoneType
J_inv = np.linalg.inv(J) # inverse
rank_J = np.linalg.matrix_rank(J)) # rank
```

Let's try an example. Open the file named `ur3_jac.py`, containing the following code (truncated):

```
q_regular = [-1.860, -2.990, 0.670, -2.000, 1.610, -0.007] # Non-singular q
q_at_lim = robot.qlim[0] # lower joint limit of robot, theta = -\pi
# Jacobians
J_reg = robot.jacob0(q=q_regular)
J_at_lim = robot.jacob0(q=q_at_lim)

# Matrix ranks
rank_J_reg = np.linalg.matrix_rank(J_reg)
```

Two manipulator configurations are included in the script: a non-singular configuration, **q_regular** and a singular configuration at the lower joint limit ($-\pi$ for all joints), **q_at_lim**. The script computes the Jacobians corresponding to each configuration and the associated matrix ranks, and then checks whether the inverse of the Jacobians exist using a nuanced approach involving the **condition number** and not the inverse directly (see: here). Run the script, and verify that the following output is printed to the terminal:

```
The rank of J_reg is 6, so q_reg is non-singular,
      and it is true that its inverse exists

The rank of J_at_lim is 5, so q_at_lim is singular,
     and it is not true that its inverse exists
```

Question 9. By modifying the `q_to_test` and `J_to_test` variables, along with attendant `print` statements in the `ur3_jac.py` file (all relevant code blocks are highlighted with a `TODO`), determine whether or not the following configurations are singular by comparing **both** the Jacobian inverse (if it exists) and the value returned by the provided `check_inv` function. Report your answer as **Singular** or **Non-singular** in each case (**Note**: you only need to report one answer per configuration. Use your best judgment to determine which method to trust: the de facto and intuitive inverse, or the lesser-known condition number approach). Each array is a value you should set `q_to_test` to:

(i). $\begin{bmatrix} -1.360 & -1.990 & 0.570 & -1.000 & 1.510 & -0.002 \end{bmatrix}^\top$

(ii). $\begin{bmatrix} 3.1415 & 3.1415 & 3.1415 & 3.1415 & 3.1415 & 3.1415 \end{bmatrix}^\top$

(iii). $\begin{bmatrix} -0.380 & -1.330 & 0.420 & -0.000 & 0.510 & -0.054 \end{bmatrix}^\top$

(iv). $\begin{bmatrix} 0 & 3.14159265 & 3.14159265 & 3.14159265 & 3.14159265 & 0 \end{bmatrix}^\top$

(v). $\begin{bmatrix} 1.65762426 & 3.14159265 & 3.14159265 & 3.14159265 & 3.14159265 & 1.65762426 \end{bmatrix}^\top$

5 From Single Joint Configurations to Trajectories

To move from configuration to motion, we must define a new object that specifies a *sequence* of configurations to step through over **time** to achieve the required motion. This object is the **joint trajectory**, typically denoted by $q(t)$, with t denoting the time variable.

5.1 Mapping Joint Trajectories to Robot Motion

While it might be straightforward to define a trajectory comprising a few joint configurations by hand, especially if the end-points are trivially close, defining a trajectory for say 50 steps manually is not only unreasonable but is also guaranteed to be fraught with errors. Luckily, we have a single-call method to simplify this process within the toolbox, namely the **jtraj** method.

The **jtraj** method takes three arguments: the start configuration, the end configuration, and the desired number of intermediate steps between both configurations, and produces a joint-space trajectory comprising as many configurations as the specified number of steps. Open the `src/ur3_jsp_motion.py` file which should contain the following lines of code (truncated):

```python
robot = rtb.models.DH.UR3() # Load a UR3 model

# Define the start and end configurations of the robot arm
q_start = [0.0, -np.pi/2, 0.0, -np.pi/2, 0.0, 0.0]
q_end = [-1.860, -2.990, 0.670, -2.000, 1.610, -0.007]

traj = rtb.jtraj(q_start, q_end, 50) # Interpolate trajectory (50 steps)
robot.plot(traj.q, block=True)       # Animate the motion
```

Run the script. A Matplotlib window should appear showing an animation where the robot moves from an initial upright configuration to a final configuration. In the file, the trajectory created by **traj** is a multidimensional NumPy array containing **50** joint configurations between **q_start** and **q_end**, inclusive. Confirm this by checking the trajectory's length. Comment out the **robot.plot()** call to disable the animation, and add the following two lines at the end of the Python file

```python
print(f"The trajectory's shape is: {np.shape(traj.q)}")
print(f"The trajectory contains {len(traj.q)} configurations \n \
      each of length {np.shape(traj.q)[1]}")
```

Run the script. It should print:

```
The trajectory's shape is: (50, 6)
The trajectory contains 50 configurations
        each of length 6
```

verifying that there are indeed 50 joint configurations in the trajectory, with each trajectory containing 6 joint angles, one for each joint of the UR3.

Question 10. Run `ur3_jsp_motion.py` file for 30, 60, 100, 120, and 150 steps (joint configurations). Log the approximate execution time (printed at the top-right corner of the Matplotlib window) for each run, i.e., the time (in seconds) it roughly takes for the robot to reach `q_end`. Include a plot of number of steps vs. execution time in your report.

5.2 Computing Singularity-Free Joint Trajectories

We can now put the Jacobian math from section 4 to some more use. Whereas the example in Section 4.4 determined if a **single** configuration was singular using the Jacobian, in practice, we'd like to plan **entire** robot arm motions in such a way that singular configurations are avoided. That is, we want to ensure that each configuration in the computed trajectory is non-singular. We have all the tools to do this. Open the file `ur3_jsp_jac.py`. You'll find that this file is somewhat similar to the `ur3_jsp_motion.py` script except that now instead of an initial and a final configuration, we have included an intermediate configuration by stacking trajectory segments, setting this middle configuration to a known singular configuration on purpose.

```python
q_start = [0.0, -np.pi/2, 0.0, -np.pi/2, 0.0, 0.0]
q_at_lim = robot.qlim[0] # singular intermediate config
q_end = [-1.860, -2.990, 0.670, -2.000, 1.610, -0.007]
```

The script implements two versions of the trajectory planner we saw in the `ur3_jsp_motion.py` example: a **naive** planner, where *all* configurations are included in the planned trajectory, and a **singularity-aware** planner that *filters* the trajectory from the naive planner by removing configurations near or at kinematic singularities using the method we saw in Section 4.4. The code block that implements this filtering routine is:

```python
def avoid_singular_q(traj):
    traj_non_sing = np.empty_like(traj[0,:])
    traj_sing = np.empty_like(traj[0,:])
    for q in traj:
        if check_inv(robot.jacob0(q)): # see code for definition
            traj_non_sing = np.vstack([traj_non_sing, q])
        else:
            traj_sing = np.vstack([traj_sing, q])
    return traj_non_sing, traj_sing
```

Run the code. You should see two animations play sequentially. In the first animation, the robot visits every configuration in the trajectory, including the singular ones, 'happily' folding on top of itself (yes, as you will notice from the animation and Question 11, including a singular configuration smack in the middle of configurations to visit creates even more singular configurations). In the second animation, however, the robot visits fewer configurations, and is actively steered away from reaching the singular configurations as the trajectory progresses. The **VERY_HIGH_NUM** variable in the `check_inv` function controls the filter's strength. At **VERY_HIGH_NUM = 1000**, for instance, 10 singular configurations are expunged from the final trajectory, leaving 40 valid ones; at 100, only 24 make it through.

Question 11. By modifying the `ur3_jsp_jac.py` file, test the performance of the singularity-avoiding planner with the variable **VERY_HIGH_NUM** on the TODO-highlighted line set to the following values: **1e3**, **1e4**, **1e5**, and **1e6**. Report the number of valid (non-singular) trajectories in each case. Hint: You should notice an upward trend.

---

6 Velocity Kinematics of Serial Chains

Velocity kinematics relate joint **velocities** ($\dot{q}$) to end-effector **twists** ($\dot{x}$), i.e., linear and angular velocities. From the FK equations in Equation (7), we can derive the velocity kinematics equations by straightforward differentiation:

$$\dot{x} := \begin{bmatrix} \dot{p} \\ \omega \end{bmatrix} = \overbrace{\begin{bmatrix} J_v(q) \\ J_\omega(q) \end{bmatrix}}^{\text{Jacobian}} \dot{q} \implies \dot{q} = \underbrace{J^{-1}}_{\text{Jacobian inverse}} \dot{x}, \tag{12}$$

where $J_v(q)$ and $J_\omega(q)$ are the linear and angular velocity blocks of the Jacobian we computed in section 4. Intuitively, while J_v is responsible for adapting $\dot{x}$'s linear velocity components, J_ω adapts its angular components. What this tells us is that to follow a curve at constant velocity in the end-effector frame, we have to vary the joint velocities in a nonlinear fashion (via J's inverse), since $\dot{q}(t) = J^\dagger(q(t))\dot{x}_d$ for constant $\dot{x}_d$, and the Jacobian pseudo-inverse (or inverse, if it exists) is nonlinear in general. Let's go over an example to make things tangible, followed by a question.

6.1 Velocity Kinematics with the Toolbox

Open the file named `ur3_vk.py`, which contains a minimal velocity kinematics example (truncated here to highlight the important bits):

```python
q_app = [-1.860, -2.990, 0.670, -2.000, 1.610, -0.007]
q = q_app                              # start near a ready pose
dt = 0.02                              # sample time
N = 200                                # num of steps over which to exec motion
v_des = np.array([0.05, 0, 0])         # 5cm/s in base X
traj = [q]
ee_pos = [robot.fkine(q).t]            # initial end-effector position

for _ in range(200):
    J = robot.jacob0(q)[:3, :]         # only need linear part of J to update v
    dq = np.linalg.pinv(J) @ v_des     # compute corresponding joint velocity
    q = q + dq * dt                    # update joint configuration
    traj.append(q)                     # keep track of q for animation
    ee_pos.append(robot.fkine(q).t)    # keep track of ee position for animation
```

In the script, we compute joint velocities by prescribing a constant end-effector velocity along the base frame X axis. The joint velocities are updated at a fixed time step `dt` to maintain this end-effector velocity. As a result, the end-effector traces a horizontal line in the ZX plane (see Figure 3). Specifically, with a constant tool velocity along the base X axis, v_d, the end-effector position evolves as

$$x_{t+1} = x_t + v_d\, dt,$$

which corresponds to the parametric form of a line, with the X coordinate changing over time while the Y and Z coordinates remain constant. However, since our script only commands linear velocity without considering the end-effector's angular velocity, there is some residual angular velocity error from the angular velocity Jacobian. We confirm this by computing both the linear velocity
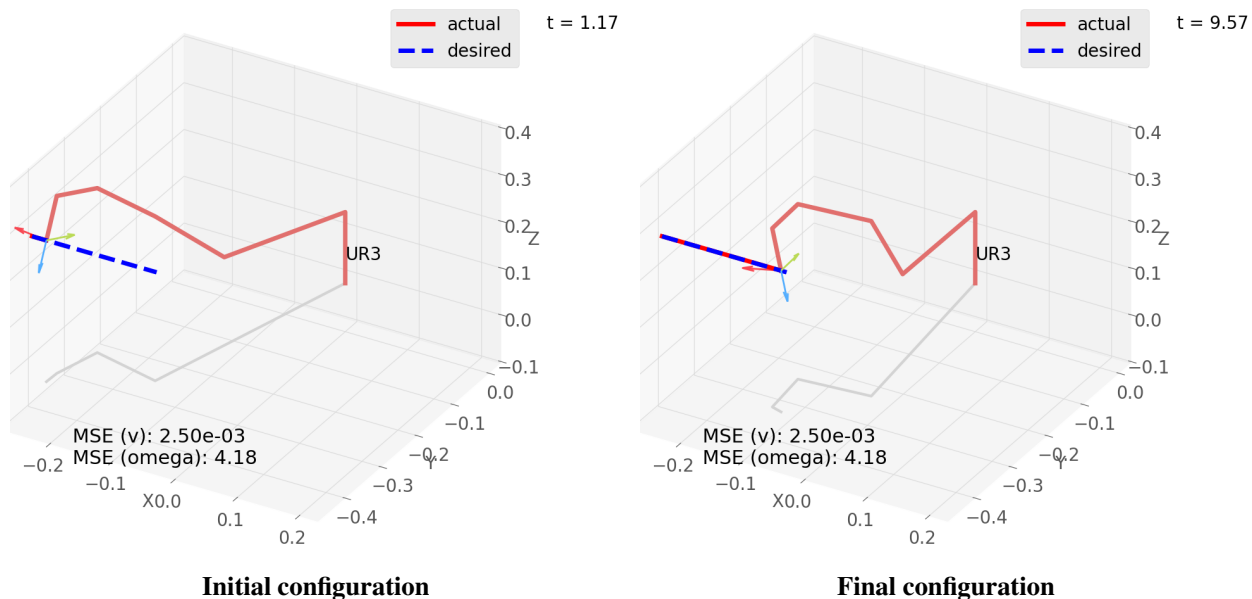
Figure 3: UR3 tool frame tracking a constant reference via velocity kinematics. In the final configuration, the tool maintains nearly constant linear velocity, but angular effects arise from $J_\omega(q)$.

and angular velocity errors using

$$e_v = \|J_v(q)\dot{q} - v_{\text{des}}\|, \qquad e_\omega = \|J_\omega(q)\dot{q} - \omega_{\text{des}}\|, \tag{13}$$

where v_{des} and ω_{des} are the desired linear and angular velocities, respectively. The calculations come out to be the negligible 0.0025 m/s for e_v and 4.18 rad/s for e_ω, a noticeable value.

Question 12. Modify the blocks marked with a `TODO` flag within the `src/ur3_vk.py` script to make the robot track the prescribed linear velocity (`v_des`) **and** a **zero desired angular velocity**, i.e., `np.array([0, 0, 0])`. To determine if your modification works, we have included text annotation helper functions that will print the mean-squared tracking error for the linear (e_v) and angular (e_ω) velocity components on the Matplotlib figure window (see Figure 3). Save the Matplotlib plot with these annotations and include them in your report (you do not need to annotate the figure yourself; the script will handle this).

---

7 Inverse Kinematics

7.1 Unpacking IK

IK is simply the process of finding the joint configuration q that achieves a **desired** end-effector pose $T_d \in SE(3)$. Unlike FK, which is mostly straightforward ($T = f(q)$), IK requires solving the nonlinear inverse mapping $q = f^{-1}(T_d)$ using one of two primary approaches: **analytical** methods, which are often difficult to derive, and **numerical** methods, which are more tractable but approximate. In Exercise 3, you will solidify your understanding of inverse kinematics by completing a basic numerical IK solver. For now, let's see what IK methods the toolbox provides.

7.2 Computing IK with the Toolbox

To compute IK via the toolbox, we can call one of several algorithm-specific methods from the same **robot.models.DH.UR3** class instance we've used in previous sections. The methods are summarized on Table 2. Open the file `ur3_ik.py` for an example that implements IK (a truncated version is included below). In the example, we want the end-effector to reach a desired pose expressed as a desired position or translation (using the method **sm.SE3(x, y, z)**) and a desired 90° counterclockwise rotation about the Z axis (using the method **sm.SE3.Rz(angle)**). We apply IK here to find the joint configuration(s) that achieve this pose.

```python
import roboticstoolbox as rtb
import numpy as np # library containing important kernels for matrix math
import spatialmath as sm # library containing important kernels for tf math

robot = rtb.models.DH.UR3() # Load a UR3 model; creates a UR3 class instance
T_goal = sm.SE3(0.5, 0.2, 0.3) * sm.SE3.Rz(np.pi/2) # Desired EE pose

# Compute IK
solution = robot.ikine_LM(T_goal)    # Levenberg-Marquardt solver
print(solution.q)                    # Joint angles (if solution found)
```

Run the code. You should see a 6D array of joint angles in radians, which corresponds to the solution found by the IK solver:

$$\begin{bmatrix} -2.97125889 & 0.0532965 & -0.43369849 & -1.19039433 & 1.57079633 & -0.17033376 \end{bmatrix}^\top.$$

Note that your solution might be different from the previous array, but the solver will return a solution regardless.

✎ Intuitively, we can think of IK as essentially "nudging" the joints in the direction that reduces the tool pose error $(x_d - f(q_k))$, converging to a solution over time. If no convergence occurs within the preset maximum number of iteration steps or a collision occurs, the procedure ends and reports failure. In addition, IK may yield multiple solutions or none at all, depending on the desired pose and joint limits, as numerical IK solvers often include joint limits as constraints. To ensure repeatability, it may thus be necessary to "freeze" or store desirable trajectories, especially for tasks involving repeated motions.

Table 2: Common numerical methods for IK. The string in each function's name identifies the numerical optimization IK solver associated with that function.

Function	Algorithm	External Reference
ikine_LM	**L**evenberg–**M**arquardt	Wikipedia
ikine_GN	**G**auss–**N**ewton	Wikipedia
ikine_NR	**N**ewton–**R**aphson	Wikipedia
ikine_QP	**Q**uadratic **P**rogramming	Wikipedia

> **Question 13.**
>
> (a). Run the `ur3_ik.py` script THREE times, freezing the solver to the **LM** and **GN** types for each triple run. Collect the different joint configuration solutions (rounded to 4 decimal places) each solver type yields each time in a table and include the table in your report.
>
> (b). Using the same IK script but only with the LM solver, find the joint configuration corresponding to the tool pose at the robot's lower and upper joint limits, i.e., the two elements of **robot.qlim**. Note: one IK solution is enough for each joint limit. Include the values (rounded to 4 d.p.) in your report.
>
> (c). For the two solutions you obtained in (b), compute the magnitude of the difference between the corresponding configuration at the joint limits and the IK solution. We provide an example for the lower limit below (assuming the IK solution corresponding to this limit is stored in the variable **sol_l**), but you should also compute the corresponding quantity for the upper limit as well:
>
> ```
> np.linalg.norm(robot.qlim[0] - sol_l.q)
> ```
>
> In your report, explain in **one** sentence why you think your answer is (or isn't) non-zero.

7.3 Mapping End-Effector Trajectories to Robot Motion via IK

You just crunched numbers to compute IK! Let's now explore a few things we can do with this new tool. We'll use a tracking example like in section 6, but with a circular trajectory. Open the file `ur3_ik_circle.py`. The relevant blocks are highlighted below.

```python
# Create a circle in task space (ZX plane; y constant)
r = RADIUS # m
t = np.linspace(0, 2*np.pi, NUM_PTS_TSP_CIRC) # controls smoothness; see code
x = r * np.cos(t)
y = ee_pose_des_t[1] * np.ones_like(x)
z = r * np.sin(t)
tsp_circle = np.vstack((x, y, z)).T

# Sample points on the task-space circle
indices = np.linspace(0, NUM_PTS_TSP_CIRC - 1, NUM_SAMP_PTS, dtype=int)
sampled_pts = tsp_circle[indices]

# Compute IK for each sampled point and add to trajectory
traj = np.empty((0, 6))
for sampled_pt in sampled_pts:
    T_goal = sm.SE3(np.array(sampled_pt)) * sm.SE3(sm.SO3((ee_pose_des_R)))
    q_sol = robot.ikine_LM(T_goal).q
    traj = np.vstack([traj, q_sol])
```

The main logic of the script is simple: we define a circle in task space. Since there are an infinite number of points on any geometric circle, we sample a few, making sure to cover the entire perimeter. We then compute IK for each sampled point and add the solution to our robot's trajectory. Try running the script. You should see an animation of the UR3 tracking a task-space circle using IK-computed joint configurations. However, while the task space trajectory appears

smooth, the joint space trajectory appears jerky and fragmented. We will revisit this jerky motion in a later lab. For now, try your hands on the next question to test your understanding.

> **Question 14.** Suppose the UR3 in Section 7.3 has been fitted with a painting tool and that we require the robot to trace the symbol for infinity (∞) in Cartesian space, on some surface in the ZX plane. By modifying the `ur3_ik_circle.py` file, compute the required set of joint configurations (using the LM solver) to achieve the task space infinity symbol, assuming the robot's controllers can only execute at most `(2 * NUM_SAMP_PTS)` configurations for any given task. Provide legible Matplotlib plots of the robot at its initial and final configurations (see Figure 4). Your figures should show the infinity symbol. **Note**: You may change the argument of the `plt.pause` function to **1** or higher, to slow down the animation.
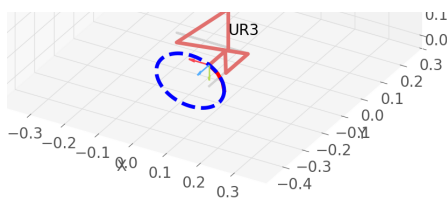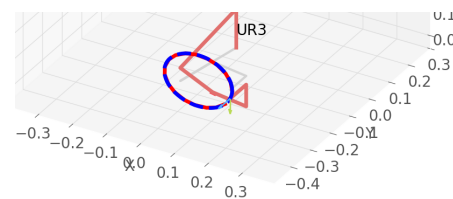
| Initial configuration | Final configuration |

Figure 4: UR3 tracking a task-space circle using numerical inverse kinematics.

8   Exercises

> **Instructions**: There are three exercises in total. Choose any two of the three exercises to complete and receive full credit for the lab. Exercises 1 and 2 are worth **15 points** each. Exercise 3 is worth **20 points** meaning if you complete Exercise 3 as one of your two choices you will earn 5 points of extra credit.

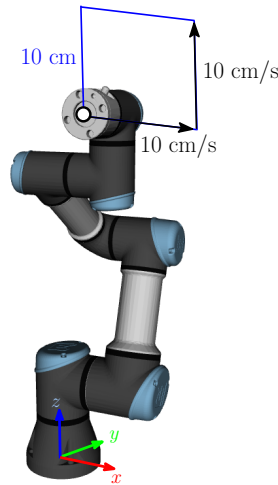# Exercise 1. Tracking a Cartesian Square with IK (15 pts)

Figure 5: UR3 and target Cartesian square path. The white dot marker on the robot's last link highlights a waypoint while arrows signify the motion direction.

Robots can move their end-effectors using different strategies. Velocity kinematics integrates desired end-effector velocities to produce smooth motion but does not guarantee **precise** positions. However, many industrial manipulation tasks, such as assembly, pick-and-place, or welding, require inverse kinematics (IK) to compute the joint configuration needed to reach **exact** end-effector poses.

In this exercise, you will complete the square-drawing task for the UR3 using only IK by modifying the script `exercise.py` in the **src/exercise** folder. To accomplish this, you need to:

1. Specify the **corner points of the square** (`waypoints`) in Cartesian space (see Figure 5)

2. Solve the **IK problem** for each corner using a Levenberg–Marquardt solver to compute joint configurations.

The relevant blocks within the code are highlighted with a `TODO` comment, and the deliverables for this exercise are enumerated below:

Deliverables:

- The 6×6 IK solution array that is printed to the terminal when your code runs

- The modified `exercise.py` file that produced the IK solution results (see the file-naming convention in the next section (section 9)).

Exercise 2. Velocity Kinematics with Orientation Constraints (15 pts)

When controlling a robot's end-effector with only linear velocity using velocity kinematics, any unaccounted angular motion can cause the tool to rotate unintentionally, which may distort the intended Cartesian trajectory. For tasks like tracing a square or performing precision operations, it may be necessary to maintain a **fixed orientation** by explicitly setting the angular velocity to zero. Currently, the velocity-kinematics square-tracking implementation provided in `ur3_draw_square_vk.py`, located in the **src/exercise** folder, only tracks the desired **linear** velocities of the tool

$$\dot{x}_{\text{tool}} = v_{\text{des}} := \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix},$$

by updating joint velocities using the Jacobian

$$\dot{q} = J(q)^{-1} \dot{x}_{\text{tool}},$$

where J^{-1} is the Jacobian inverse if it exists, or the pseudo-inverse otherwise, for the 6×6 Jacobian. To maintain a **desired angular velocity** while tracing the Cartesian square, we extend this to a full 6D twist:

$$\dot{x}_{\text{tool}}^{6D} = \begin{bmatrix} v_{\text{des}} \\ \omega_{\text{des}} \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix},$$

where $\omega_{\text{des}} = \begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix} \in \mathbb{R}^3$ is the desired angular velocity in radians-per-second. By modifying the `TODO`-highlighted code blocks within the `for` loop in `ur3_draw_square_vk.py`, **adapt the velocity update logic to keep the robot's angular velocity at zero** (in other words, to maintain a fixed orientation) while tracking the Cartesian square. To complete this exercise, you are expected to provide the following:

Deliverables:

- A plot of the robot at the final waypoint of the square (in your report)

- The **MSE(v)** and **MSE(omega)** values displayed on the Matplotlib animation window during execution. Provide these in your report. **Note**: Your grade for this exercise will largely depend on these MSE values, so try to make them as low as possible.

- The modified `ur3_draw_square_vk.py` file that produced the above results (submit as an attachment).

Exercise 3. Numerical Inverse Kinematics Solver (15 pts + 5 pts Extra Credit)

Numerical IK methods such as the Jacobian pseudo-inverse method and the Damped Least Squares algorithm use the **pseudo-inverse** of the manipulator Jacobian $J^\dagger$ (here's that object again) to update the joint configuration iteratively until **convergence to a solution** (i.e., to a **joint configuration**) is achieved, as in

$$q_{k+1} = q_k + \gamma_k J^\dagger(q_k)(x_d - f(q_k)), \tag{14}$$

where $f(q_k)$ is the end-effector pose at time step k realized from computing FK, and γ_k is a step size parameter that may be constant or vary with k. Convergence is achieved if

$$\|q_{k+1} - q_k\| \le \epsilon, \text{ for } k < \infty, \tag{15}$$

and where $\epsilon > 0$ is a convergence tolerance. **Note**: γ_k and ϵ are not the same quantity.

To implement our IK solver, we need a few components:

   i. a convergence tolerance, ϵ to implement Equation (15)

   ii. a Jacobian-computing kernel that outputs Jacobians given a joint configuration, making sure that the configurations are non-singular and within joint limits before computing anything

   iii. An FK map to compute the task-space error in Equation (14), i.e., $x_d - f(q_k)$

   iv. A step-size schedule for γ_k. For simplicity, we'll set γ_k to a constant ($\gamma \in (0, 1]$) for all k.

Fortunately, the necessary components have already been developed in earlier sections and questions; they only need to be integrated into a single function. Your task is to finish the partial implementation of the numerical inverse-kinematics solver located in `src/min_ik_solver.py`. The deliverables for this exercise are:

Deliverables:

  (a). In the file `min_ik_solver.py` is a partial implementation of a minimal IK solver (`compute_ik`) that takes as argument the desired $SE(3)$ pose (`x_d`), a positive convergence tolerance (`eps`), an initial configuration guess (`init_guess`), a value for γ (stored in the variable `gamma`), and a specification of the maximum number of iterations (`max_iters`) after which the routine reports failure if a solution is not found within the tolerance specified by `eps`. The function should output a set of joint angles in radians whose FK work out to be the desired $SE(3)$ pose argument. By modifying the `TODO`-highlighted blocks, **complete** the numerical IK routine by implementing Equation (15) and Equation (14), respectively. Your function should return three results upon convergence:

    – The final configuration, stored in the variable: `config_k`

    – The norm of the task-space error, stored in the variable: `error`

    – The number of iterations at convergence.

  Using the provided `main` function, test your IK function with the following desired $SE(3)$ Cartesian poses:

– **Test Case 1**: $T_1 = T_x(0.3)T_y(0.2)T_z(0.35) \, R_x(0)R_y(0)R_z(0)$

– **Test Case 2**: $T_2 = T_x(0.4)T_y(-0.1)T_z(0.3) \, R_x(0)R_y(\pi/4)R_z(0)$

– **Test Case 3**: $T_3 = T_x(0.2)T_y(0.1)T_z(0.4) \, R_x(\pi/2)R_y(0)R_z(\pi/2)$,

noting down the final configuration, task space error norm, and number of iterations at convergence (these metrics will appear in the terminal if your programming logic is correct). In your report, provide the joint configuration, number of iterations, and error for all three test cases above.

(b). Compare your solutions in (a) with the solutions returned by the following code block (add it to the `main` function in `min_ik_solver.py` after the `print(f"IK Solution ...")` line, and run the script):

```
# RTB4PY for verification
print(f"IK Solution (RTB): {np.round(ik_sol_rtb.q, 3)}, \
        num_iterations: {ik_sol_rtb.iterations}, \
          error: {np.round(ik_sol_rtb.residual, 5)}\n")
```

Explain in at most **two** sentences in your report why you think the results in (a) and (b) are different.

(c). By computing forward kinematics for the solution returned by your solver in (a), verify that the desired Cartesian poses are recovered for each test case. In your report, provide the resulting 4×4 homogeneous transformation matrix in each case, along with a text snippet of the code that produced your FK results.

(d). Your modified Python script (include this as an attachment with your Canvas submission).

9   Conclusion and Lab Report Instructions

After completing the lab, summarize the procedure and results into a well written lab report. Include your solutions to all of the question boxes in the lab report. Refer to the List of Questions to ensure you don't miss any. Include your full solution to the exercises in your lab report. If you wrote any Python programs during the exercises, include these files as a separate attachment, appending your team's number and last names (in alphabetical order) to the file, e.g., **Jane Doe**, **Alice Smith**, and **Richard Roe** in **Team 90** submitting their modified `main.py` would submit the file `main_Team90_Doe_Roe_Smith.py` and the report `Lab_<LabNumber>_Report_Team90_Doe_Roe_Smith.pdf`, where `<LabNumber>` is the number for that week's lab, e.g., **0**, **1**, **2**, etc. Submit your lab report along with any code to ELMS under the assignment for that week's lab. You must complete and submit your lab report by **Friday, 11:59 PM** of the week following the lab session. See the **Files** section on ELMS for a lab report template named **Lab_Report_Template.pdf**. Below is a rubric breaking down the grading for this week's lab assignment:

Component	Points	Component	Points
Lab Report and Formatting	11	Question 9.	4
Question 1.	6	Question 10.	2
Question 2.	6	Question 11.	5
Question 3.	4	Question 12.	5
Question 4.	4	Question 13.	5
Question 5.	4	Question 14.	4
Question 6.	3	Exercise 1.	15
Question 7.	5	Exercise 2.	15
Question 8.	2	Exercise 3.	15+5

Total	100