

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
- Write Python code to move a serial-chain robot 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, “*The Universal Robot Description Format*” in Modern Robotics: Mechanics, Planning, and Control, Cambridge University Press, 2nd ed., 2019, pp. 150-156.
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

Question 1.	4
Question 2.	6
Question 3.	8
Question 4.	9
Question 5.	9
Question 6.	10
Question 7.	12
Question 8.	14
Question 9.	15
Question 10.	16
Question 11.	18


List of Exercises


Exercise 1. Tracking a Cartesian Square with IK	19
Exercise 2. Tracking with Orientation Constraints	20

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 or implementation details provided for your enlightenment. 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)

```
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 commands will be set in bold 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 the symbol  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 size and time information may differ):

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

Now launch the Docker container with the following commands

```
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:~$
```

Verify that you have all the files necessary for this lab by entering the lab-2 folder and issuing an `ls` command from within the container, which should print the following output to shell:

```
cd lab-2 && ls
```

```
README.md  docker config src
```

We will be working within the `src` directory above using the Bash shell. Be mindful of your working directory, and ensure that you run the Python scripts from the appropriate folder.

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 eXtensible Markup Language 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):

- **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
- **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).

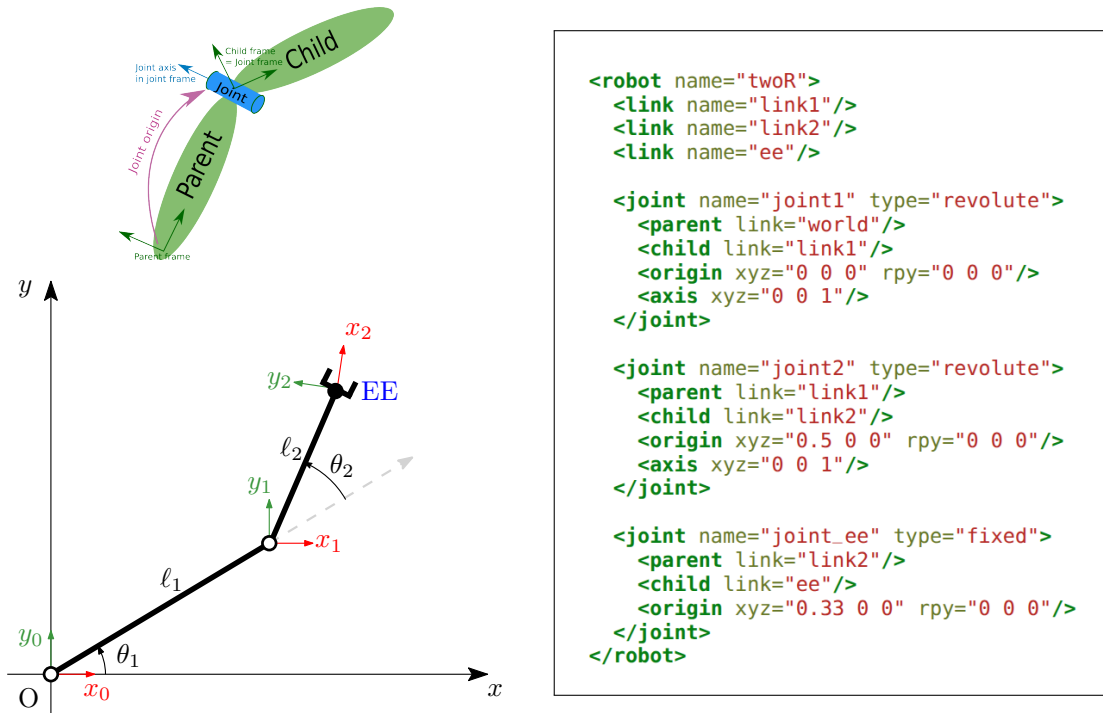


Figure 1: Illustrating how a joint (top left) connects links in the kinematic chain, with the kinematic diagram of a 2R planar manipulator (bottom left) compared against its URDF description (right). The +z axis points out of the page.

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, and the `plot_robot(q)` method will visualize 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 list, modify **lines 8 to 29** of the `src/build_2r_urdf.py` file 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"`}
- 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. Be aware that the file will be saved on the Docker container in your current working directory on the host lab machine, and any files in the mounted container drive that are not saved will be deleted after each lab. Copy the contents of the URDF file however you see fit (e.g., by saving the file itself to a pen drive or copying the contents to a live Google Docs document containing your team's report draft). 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 (ask your TA if you need help identifying this). Copy the saved plot from the Docker container to a pen drive, cloud storage, or a live report draft. 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 robot 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 “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:

$$T_{\text{target}}^{\text{source}} = T_{\text{frame_1}}^{\text{source}} \cdot T_{\text{frame_2}}^{\text{frame_1}} \dots T_{\text{target}}^{\text{frame_n}}, \quad (1)$$

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 concretize 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_shoulders = robot._compute_T("base_link", "shoulder_link", \
                                       config=q_test)
# o access the underlying 4x4 matrix, use the .A attribute
T = SE3_base_shoulders.A
```

Complete the following questions using the above method:

- Modify **line 28** and **lines 41 to 52** of the file `src/compute_urdf_T.py` 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 in the following kinematic chain for the UR3 (stored in the variable `link_list` in the file). 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 (you can use Python's built-in `print` function to print your answers to the terminal and copy the printed results to your report).

- By merely inspecting the provided UR3 URDF file, deduce the homogeneous transformation matrix between the `flange` and `tool0` link frames. The joint between these frames is defined between lines **234** to **239** of the referenced URDF file. Include the matrix along with **one** sentence providing the reasoning for your answer in your report.
- Using the same `_compute_T` function in [Question 2](#) (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. In the DH convention, each link i is characterized by four parameters described on Table 1. The joint angle θ_i is variable and d_i is constant for revolute joints; conversely, d_i is variable while θ_i is constant for prismatic joints.

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)

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`):

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

# construct 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")
```

We can print a summary of the constructed model using `print(twoR)`. 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   |
+-----+-----+-----+-----+
```

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):

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

Question 3.

- Using a **two-column** table, collect the UR3's joint names and corresponding origins (from the `src/urdf/ur3.urdf` file). Hint: For faster, error-free URDF parsing, you may use the `joints` attribute of the `Robot` class provided by the URDF parser in [Question 2](#). The `Robot().joints` attribute is a list of `Joint` objects with a string `name` attribute and a `T` attribute of type `spatialmath.SE3`.
- Compare the joint data from your table with the DH model specified by `rtb.models.DH.UR3()`. What kinematic parallels do you notice between both descriptions? Hint: To link DH to URDF, observe that each DH row tells you how to set the `<origin xyz>` and `<origin rpy>` for that joint in the URDF. Revolute joints correspond to θ_j , translations along z and x correspond to d_j and a_j , respectively, and rotations about x , i.e., α_j are built into the joint frame, i.e., the joint's `<origin>` tag.

2.4 Computing Transformation Matrices using the DH Parameters

Using the DH parameters, we can also construct the homogeneous transformation between consecutive link frames ($i-1$ and i) for a robot modeled as a kinematic chain:

$$T_i^{i-1} = \begin{bmatrix} \cos \theta_i & -\sin \theta_i \cos \alpha_i & \sin \theta_i \sin \alpha_i & a_i \cos \theta_i \\ \sin \theta_i & \cos \theta_i \cos \alpha_i & -\cos \theta_i \sin \alpha_i & a_i \sin \theta_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2)$$

Chaining these transforms yields the manipulator's FK (frame 0 is the base or world frame):

$$T_n^0 = T_1^0 \cdot T_2^1 \cdot \dots \cdot T_n^{n-1}. \quad (3)$$

Let's practice calculating the DH transformation matrix using Python code. Each element of the `links` array used to form the `twoR` model above has the following properties:

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

Using these properties, we can easily compute the transformation matrix in [Equation 2](#) as follows

```
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)],
1) ... # Truncated. See the 2r-dh.py file.

```

Question 4. Modify the file `2r-dh.py` to implement Equation 3 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. 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])`.

3. Forward Kinematics of Serial Chains

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 4) provide this very important mapping, typically in terms of the manipulator's link lengths and the joint configuration (q):

$$T(q) = \begin{bmatrix} R(q) & p(q) \\ 0 & 1 \end{bmatrix} \in SE(3). \quad (4)$$

To compute $T(q)$, we multiply the transformation matrices for every link pair in the chain from the root link (the link with no parent) to the end-effector link:

$$T(q) = T_{\text{link1}}^{\text{root}} \cdot T_{\text{link2}}^{\text{link1}} \cdots T_{\text{ee_link}}^{\text{linkN}} \quad (5)$$

Question 5.

- (a). Within the same file you modified in Question 2, modify the block highlighted with a **# MODIFY HERE for Question 5(a)** comment to find the matrix product of all transformation matrices in Question 2 (a) in ascending order of the 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.

To multiply two matrices, you can use the `numpy.matmul` function:

```

import numpy as np
a = np.eye(4)
b = a
mat_prod = np.matmul(a, b)

```

- (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).

Modify the code block marked with a **# MODIFY HERE for Question 5(b)**. In your report, provide the reasoning (in no more than **two** sentences) why you think the matrices are the same or different (depending on your results).

- (c). Change the robot's configuration (**line 28** of the `calculate_urdf_T.py` file) to `[0, -np.pi/4, np.pi/4, 0, np.pi/3, 0]`, and calculate the transformation for the (**base_link**, **tool0**) link pair again. Note down your answer in your report.

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.

3.1 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` for example Python code (shown below) that implements FK.

```
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
```

This produces 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

```
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.

Question 6. 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 of the robot's links appear folded on each other, causing the end-effector's position to look out of place. 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 for a robot arm! But how are we certain that the robot can reach the specific configuration stored in the variable `q`, if we command this configuration to the robot's controller? We can see here that successfully computing FK tells us nothing about the **quality** of a joint configuration, only that a certain joint configuration led to a certain end-effector pose. It is possible that while the math checks out, this set of joint angles themselves might together characterize or be near a **singular** configuration or **kinematic singularity**, which are configurations at which the manipulator's end effector loses the ability to move in one or more directions.

4. Kinematic Singularity & The Manipulator Jacobian

4.1 The Manipulator Jacobian

An object in the study of kinematics you'll see repeatedly is the **Jacobian matrix** or **Jacobian** for short. The Jacobian is an especially useful matrix because it provides both quantitative and qualitative information about the manipulator at a particular configuration. Mathematically, the Jacobian J relates the velocity of the end-effector (\dot{x}) to joint-space velocities (\dot{q}), and depends on the current configuration

$$\dot{x} = J(q)\dot{q}. \quad (6)$$

The Jacobian is a block matrix with 6 rows and n columns (corresponding to the number of joints):

$$J(q) = \begin{bmatrix} J_v(q) \\ J_\omega(q) \end{bmatrix}, \quad (7)$$

where $J_v(q)$ and $J_\omega(q)$ are the **linear velocity Jacobian** (top 3 rows) and **angular velocity Jacobian** (bottom 3 rows). For example, the Jacobian corresponding to the UR3 at the configuration $[0, -\frac{\pi}{2}, 0, -\frac{\pi}{2}, 0, 0]$ is

```
[ [ 0.19 -0.54 -0.3  -0.09  0.08  0. ] # }
  [-0.   -0.   -0.   -0.   0.   0. ] # | J_v
  [-0.   -0.   -0.   -0.   0.   0. ] # }
  [-0.   -0.   -0.   -0.  -0.  -0. ] # }
  [ 0.   -1.   -1.   -1.   0.  -1. ] # | J_omega
  [ 1.    0.    0.    0.    1.    0. ]] # }
```

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 end-effector frame but not **translate** it.

🍃 For a 6-DoF manipulator i.e., $\mathbf{n} = 6$, $J(\mathbf{q})$ is a square matrix, making computing quantities like the inverse and adjoint of J possible (assuming J has a **non-zero** determinant). For rectangular Jacobians for which we cannot compute an inverse, we use the Moore-Penrose pseudo-inverse, denoted as J^\dagger and computed via NumPy using `numpy.linalg.pinv(J)`.

4.2 Computing the Jacobian using the Toolbox

In RTB4PY, we can compute the Jacobian using yet another one-liner, `robot.jacob0(q)`, which yields the robot's $6 \times n$ Jacobian at joint configuration \mathbf{q} , expressed in the base ($\mathbf{0}$) frame. This function takes as argument the robot's configuration, as in

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

Other Jacobian-related kernels are also provided by the toolbox, including the derivative of the Jacobian (`robot.jacob0_dot`) and the end-effector Jacobian (`robot.jacobe`), but `jacob0` will be sufficient for our purposes.

Question 7. Compute the Jacobian for the UR3 at the following configuration:

```
q = [-1.860, -2.990, 0.670, -2.000, 1.610, -0.007]
```

What are the linear and angular velocity Jacobians?

4.3 Identifying Singular Configurations

Once computed, determining whether the robot is at a singular configuration is simply a matter of computing the **inverse of the Jacobian matrix**! 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 $\mathbf{J}(\mathbf{q})$ is an $n \times n$ matrix and its rank (or equivalently, the existence of its inverse) is less than \mathbf{n} , then \mathbf{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


```
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** (see: [here](#) ) and not the inverse directly. 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 not 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
```

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):

```
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 code. 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

```
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 8. Run `ur3_jsp_motion.py` file for 30, 60, 100, and 120 steps. 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 table 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 3](#) 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.

```
q_start = [0.0, -np.pi/2, 0.0, -np.pi/2, 0.0, 0.0]
q_at_lim = robot.qlim[0] # 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 3](#). The code block that implements this filtering routine is:

```
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 9](#), 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 9. Test the performance of the singularity-avoiding planner with the variable **VERY_HIGH_NUM** on **line 10** set to the following values: **1e3**, **1e4**, **1e5**, and **1e6**. Report the number of valid 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 4](#), we can derive the velocity kinematics equations by straightforward differentiation:

$$\dot{x} := \begin{bmatrix} \dot{p} \\ \omega \end{bmatrix} = \overbrace{\begin{bmatrix} J_v(q) \\ J_\omega(q) \end{bmatrix}}^{\text{Jacobian}} \dot{q} \implies \dot{q} = \underbrace{J^{-1}}_{\text{Jacobian inverse}} \dot{x}, \quad (8)$$

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 pseudoinverse (or inverse, if it exists) is nonlinear in general. Let’s make this concrete with an example using the toolbox.

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):

```
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 pos for animation

```

In the script, we compute joint velocities by prescribing a constant end-effector velocity along the base frame X axis and updating the joint velocities at a fixed time step \mathbf{dt} to maintain the end-effector velocity. The result is that the end-effector follows a horizontal line in the ZX plane (see Figure 2), since at constant tool velocity along base X , \dot{x}_d , the position, $x_{t+1} = x_t + \dot{x}_d t$ describes the coordinates of the parametric form of a line equation with a varying X component and constant Y and Z axes components. However, since our script only commands linear velocity without paying

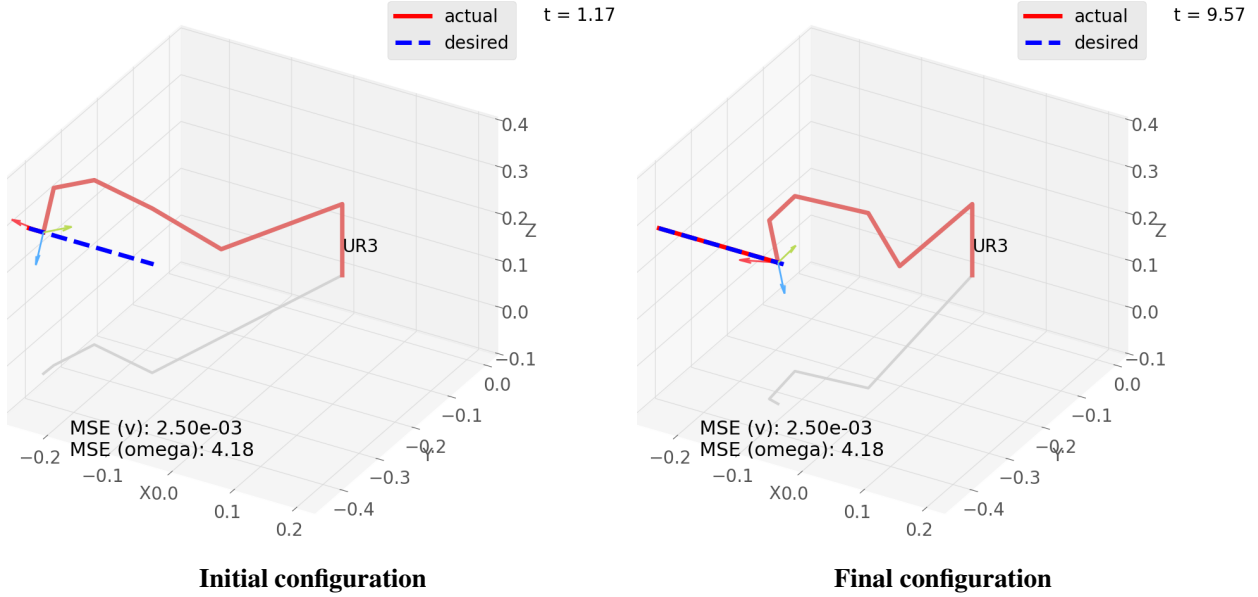


Figure 2: UR3 tool frame tracking a constant reference via velocity kinematics. In the final configuration, the tool maintains nearly constant linear velocity, but angular effects arise from $J_\omega(q)$.

attention to 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 and angular velocity errors using

$$e_v = \|J_v(q)\dot{q} - v_{\text{des}}\|, \quad e_\omega = \|J_\omega(q)\dot{q} - \omega_{\text{des}}\|, \quad (9)$$

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 10. Inspect the code in [src/ur3_vk.py](#). Modify **only** lines 38 and 39 (i.e., the \mathbf{J} and \mathbf{dq} computation logic) to make the robot track the prescribed linear velocity ($\mathbf{v_des}$) 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 2). 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). Hint: Your value for e_ω should be in the order of 10^{-6} .

7. Inverse Kinematics

7.1 IK, Explained

IK is simply the process of finding the joint configuration q that achieves a **desired** end-effector pose T_d . Unlike FK, which is mostly straightforward ($T = f(q)$), IK requires solving the nonlinear inverse mapping $q = f^{-1}(T_d)$, typically via numerical methods. Such methods use the pseudoinverse of the manipulator Jacobian J^\dagger (here's that object again) to update the joint configuration iteratively until convergence to a solution is achieved, as in

$$q_{k+1} = q_k + J^\dagger(q_k)(x_d - x(q_k)), \quad (10)$$

where $x(q_k)$ is the current end-effector pose. Intuitively, we can think of IK as essentially “nudging” the joints in the direction that reduces the tool pose error, converging to a solution over time.

🌿 In IK, convergence is achieved if $\|q_{k+1} - q_k\| \leq \epsilon$, for some finite k , where ϵ is a positive convergence tolerance. 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.

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.

```
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: **[-2.97125889 0.0532965 -0.43369849 -1.19039433 1.57079633 -0.17033376]**. Note that your solution might be different from the previous array, but the solver will return a solution regardless.

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	Levenberg–Marquardt	Wikipedia ↗
ikine_GN	Gauss–Newton	Wikipedia ↗
ikine_NR	Newton–Raphson	Wikipedia ↗
ikine_QP	Quadratic Programming	Wikipedia ↗

Question 11.

- Run the `ur3_ik.py` script TWO times, freezing the solver to the **LM** and **GN** types for each double run. Collect the different solutions each solver type yields each time in a table and include the table in your report.
- 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 in your report.
- 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.

```

# 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 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.S03((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 the 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. Run the code. 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.

8. Exercises

Exercise 1. Tracking a Cartesian Square with IK

In the `src/exercise` folder, you should see two files: `ur3_draw_square_vk.py` and `exercise.py`. The former is a complete implementation that uses velocity kinematics to trace a Cartesian square by integrating joint velocities over time. It is provided as a reference example to guide you and help you benchmark your solution. In the latter, we have provided starter code to implement the same demo using only IK (no velocity kinematics). Your task is to complete `exercise.py` to re-implement the square demo by solving inverse kinematics (using the Levenberg–Marquardt solver) at each corner of the square.

To complete this task, you only need to modify two lines: **line 36**, which specifies a variable (**waypoints**) that should contain the corners of the square we want the UR3's tool frame to traverse, and **line 43**, which computes the IK solution for each tool pose in **waypoints**.

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. Tracking with Orientation Constraints

By modifying only the relevant lines within the **for** loop in `ur3_draw_square_vk.py` (lines 28-37), adapt the velocity update logic to enable the robot maintain a **zero** angular velocity (in other words, a fixed orientation) while tracking the Cartesian square.

Deliverables:

- A plot of the robot at the final waypoint
- The numeric **MSE(v)** and **MSE(omega)** values displayed on the Matplotlib animation window during execution
- The modified `ur3_draw_square_vk.py` file that produced the above results (see the file-naming convention in the next section ([Section 9](#))).

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 **Buzz Lightyear** in **Team 90** would submit the file `<filename>_90_Doe_Lightyear_Smith.py`. Submit your lab report as a .pdf (with the same naming convention as with Python files) along with any code to ELMS under the assignment for that week's lab. You have one week to complete the lab report and submit it on the day of your next lab session. See the files section on ELMS for a lab report template. Below is a rubric breaking down the grading for this week's lab assignment:

Component	Points
Lab Report and Formatting	15
Question 1.	7
Question 2.	7
Question 3.	5
Question 4.	5
Question 5.	6
Question 6.	5
Question 7.	3
Question 8.	6
Question 9.	5
Question 10.	7
Question 11.	9
Exercise 1.	10
Exercise 2.	10
Total	100