IE416: Robot Programming Lab-02

Group Syntax

Yash Tarpara - 202201422
Kaushik Prajapati - 202201472

Links:

1. **GitHub Repository:** Access the course lab files – [Click Here].
2. **Notebook for Lab Question :** Direct link to the `.ipynb` file – [Click Here].

Example 01:

Code:

rtb_example1.py > ...

```python
#-------- Example-1---------#
import roboticstoolbox as rtb  # type: ignore

# Define the robot model
robot = rtb.models.Panda()

# Print the robot model
print(robot)

# Visulise the robot
robot.plot(robot.qr, block=True)
```

Output:

Activities   Terminal   Feb 14 10:57

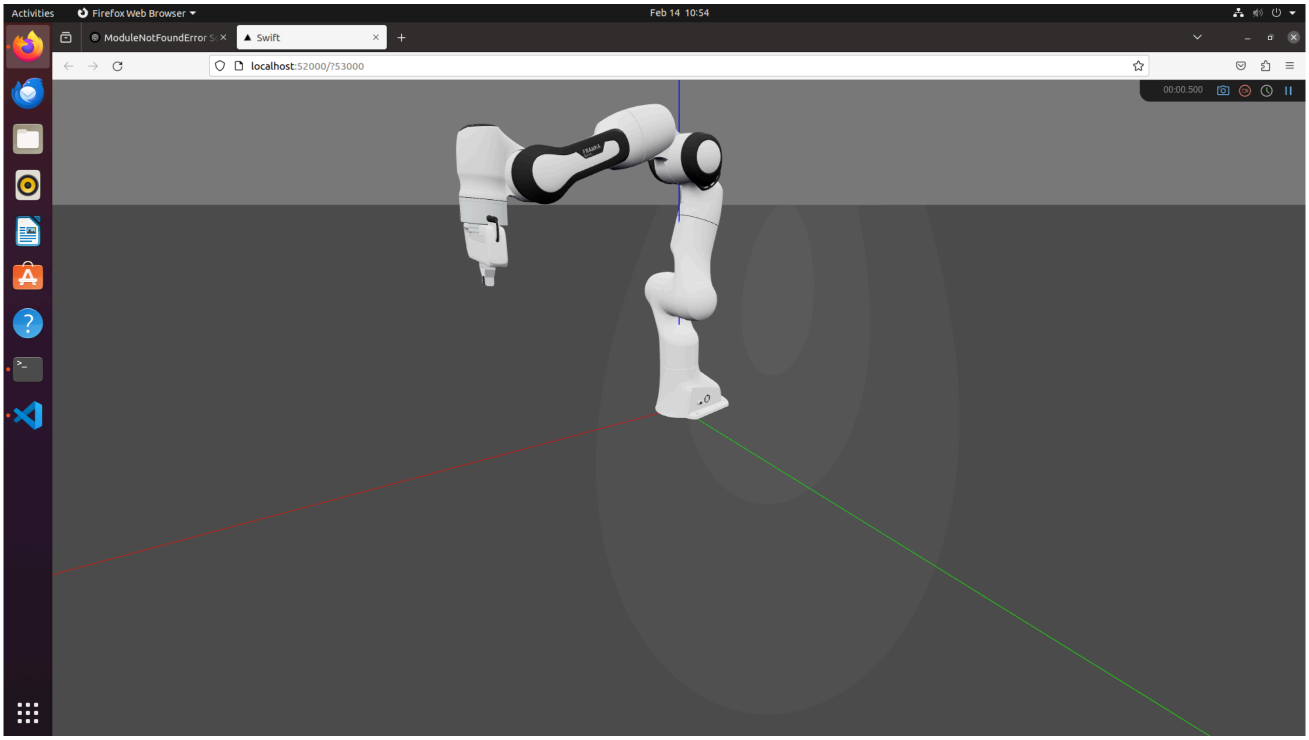
robot@ubuntu: ~

```
robot@ubuntu:~$ python3 /home/robot/Desktop/Kaushik/rtb_example1.py
ERobot: panda (by Franka Emika), 7 joints (RRRRRRR), 1 gripper, geometry, collision
```

link	link	joint	parent	ETS: parent to link
0	panda_link0		BASE	SE3()
1	panda_link1	0	panda_link0	SE3(0, 0, 0.333) ⊕ Rz(q0)
2	panda_link2	1	panda_link1	SE3(-90°, -0°, 0°) ⊕ Rz(q1)
3	panda_link3	2	panda_link2	SE3(0, -0.316, 0; 90°, -0°, 0°) ⊕ Rz(q2)
4	panda_link4	3	panda_link3	SE3(0.0825, 0, 0; 90°, -0°, 0°) ⊕ Rz(q3)
5	panda_link5	4	panda_link4	SE3(-0.0825, 0.384, 0; -90°, -0°, 0°) ⊕ Rz(q4)
6	panda_link6	5	panda_link5	SE3(90°, -0°, 0°) ⊕ Rz(q5)
7	panda_link7	6	panda_link6	SE3(0.088, 0, 0; 90°, -0°, 0°) ⊕ Rz(q6)
8	@panda_link8		panda_link7	SE3(0, 0, 0.107)

name	q0	q1	q2	q3	q4	q5	q6
qr	0°	-17.2°	0°	-126°	0°	115°	45°
qz	0°	0°	0°	0°	0°	0°	0°

Robotic Arm:



Code Explanation:

import roboticstoolbox as rtb

- Imports the **Robotics Toolbox**, which provides tools for robot modeling, kinematics, and visualization.

robot = rtb.models.Panda()

- Creates a **Panda** robotic arm model using the predefined class in **Robotics Toolbox**.
- The Panda robot has **7 degrees of freedom (DOF)**.

print(robot)

- Prints the **robot's model description**, including:
  - The number of joints.
  - Joint limits.
  - Link transformations.

robot.plot(robot.qr, block=True)

- **robot.qr**: Represents a **ready position** or a default joint configuration.
- **robot.plot(q, block=True)**:
  - Displays the **robot visualization** in a 3D environment.
  - block=True: Ensures the plot stays open until manually closed.

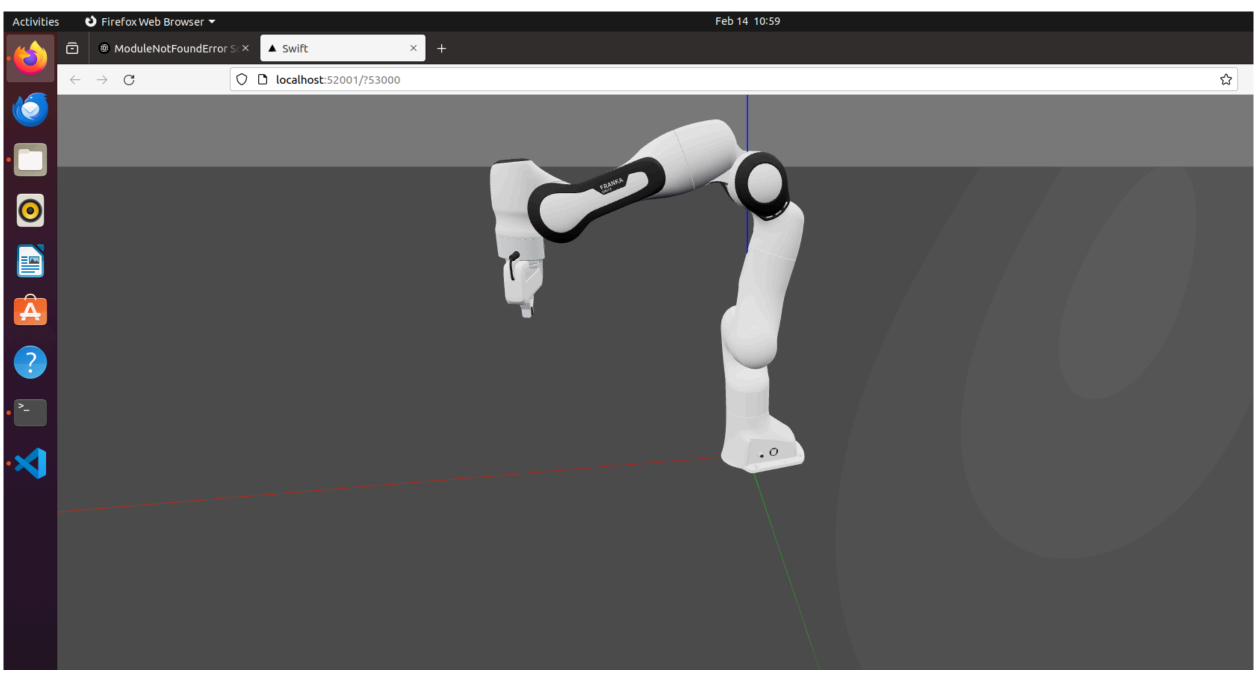
Example 02:

Code:

rtb_example2.py > ...

```python
#-------- Example-2---------#
# Display Robot with Swift
import roboticstoolbox as rtb # type: ignore
import swift # type: ignore

# Create swift instance
env =swift.Swift()
env.launch(realtime=True)

# Define the robot model
robot = rtb.models.Panda()
robot.q = robot.qr

robot.qd = [0.1, 0, 0, 0, 0, 0, 0.1]

# Add robot to swift
env.add(robot)

for _ in range(100):
    env.step(0.05)

# Stop the browser tab from closing
env.hold()
```

Robotic Arm:



Code Explanation:

roboticstoolbox: Provides the robotic models and kinematics tools.

swift: A web-based **3D visualization** library for robots.

swift.Swift(): Creates an instance of the Swift **visualization environment**.
env.launch(realtime=True):

- Starts the **Swift simulation** in real-time.
- Opens a **browser tab** to display the 3D environment.

rtb.models.Panda(): Loads the **Panda** robot model.

robot.q = robot.qr: Sets the **robot's joint positions (q)** to a default **ready position**.

robot.qd: Defines the **joint velocities** of the robot.
- The velocities [0.1, 0, 0, 0, 0, 0, 0.1] mean: The **first and last joints** move at **0.1 rad/s**, while others remain stationary.

env.add(robot): Adds the **robot model** to the **Swift simulation** for rendering.

for _ in range(100):
env.step(0.05)
- Runs a loop for 100 steps.
- env.step(0.05): Advances the simulation by **0.05 seconds** each step.

env.hold()

- Prevents the simulation from closing automatically.
- Keeps the Swift browser window open until manually closed.

Example 03:

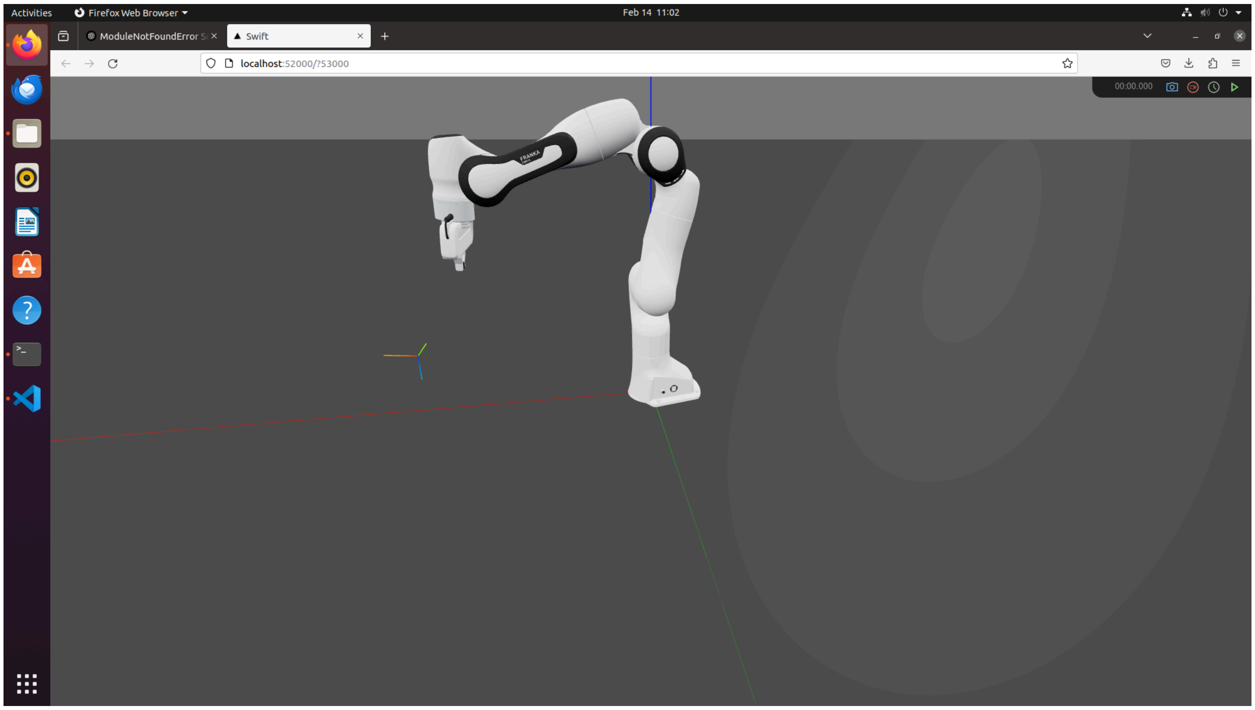
Code:

rtb_example3.py > ...

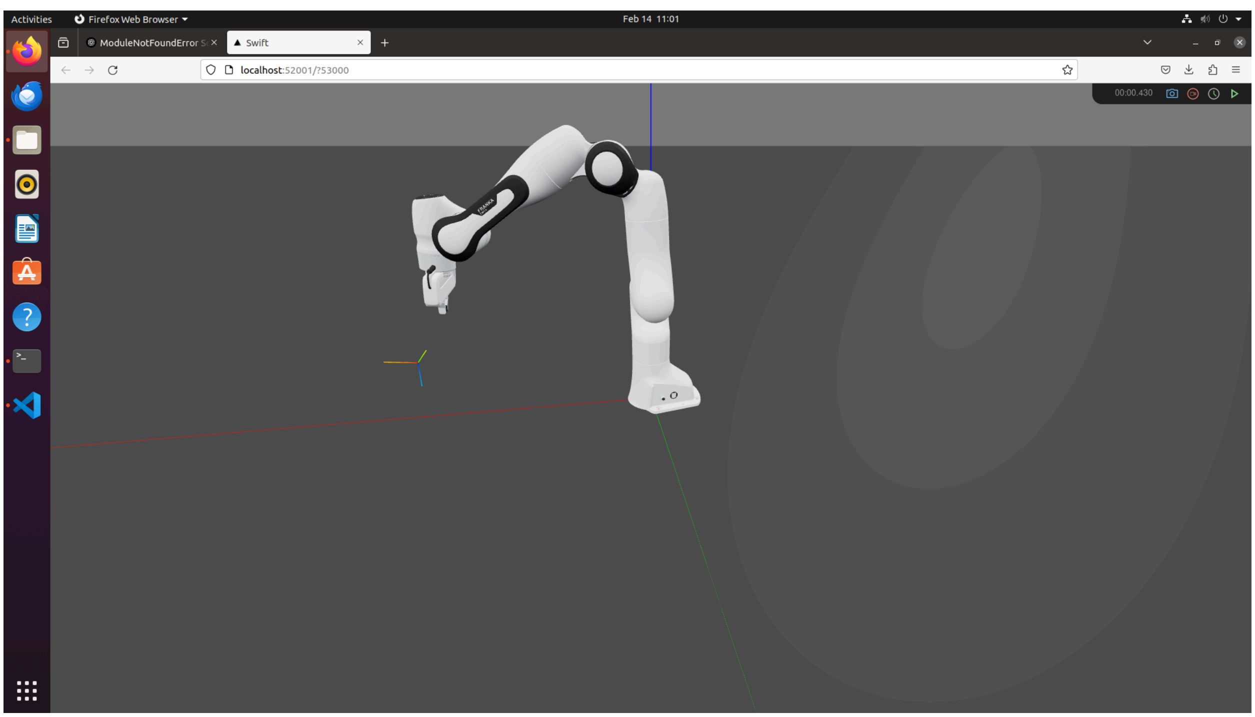
```python
#-------- Example-3---------#
# Set goal position

import roboticstoolbox as rtb # type: ignore
import swift # type: ignore
import numpy as np
import spatialmath as sm # type: ignore
import spatialgeometry as sg # type: ignore

# Create swift instance
env =swift.Swift()
env.launch(realtime=True)

# Define the robot model
robot = rtb.models.Panda()
robot.q = robot.qr

# Add robot to swift
env.add(robot)

# Set goal position
goal = robot.fkine(robot.q) * sm.SE3.Tx(0.2) * sm.SE3.Ty(0.2) * sm.SE3.Tz(0.35)
axes = sg.Axes(length=0.1, base=goal)
env.add(axes)

# Arrived at a destination flag
arrived = False

# Time step
dt = 0.01

while not arrived:
    # v is a 6 vector representing the spatial error
    v, arrived = rtb.p_servo(robot.fkine(robot.q), goal, gain=1, threshold=0.01)
    J = robot.jacobe(robot.q)
    robot.qd = np.linalg.pinv(J) @ v

    # Step the environment
    env.step(dt)

# Stop the browser tab from closing
env.hold()
```

Output:

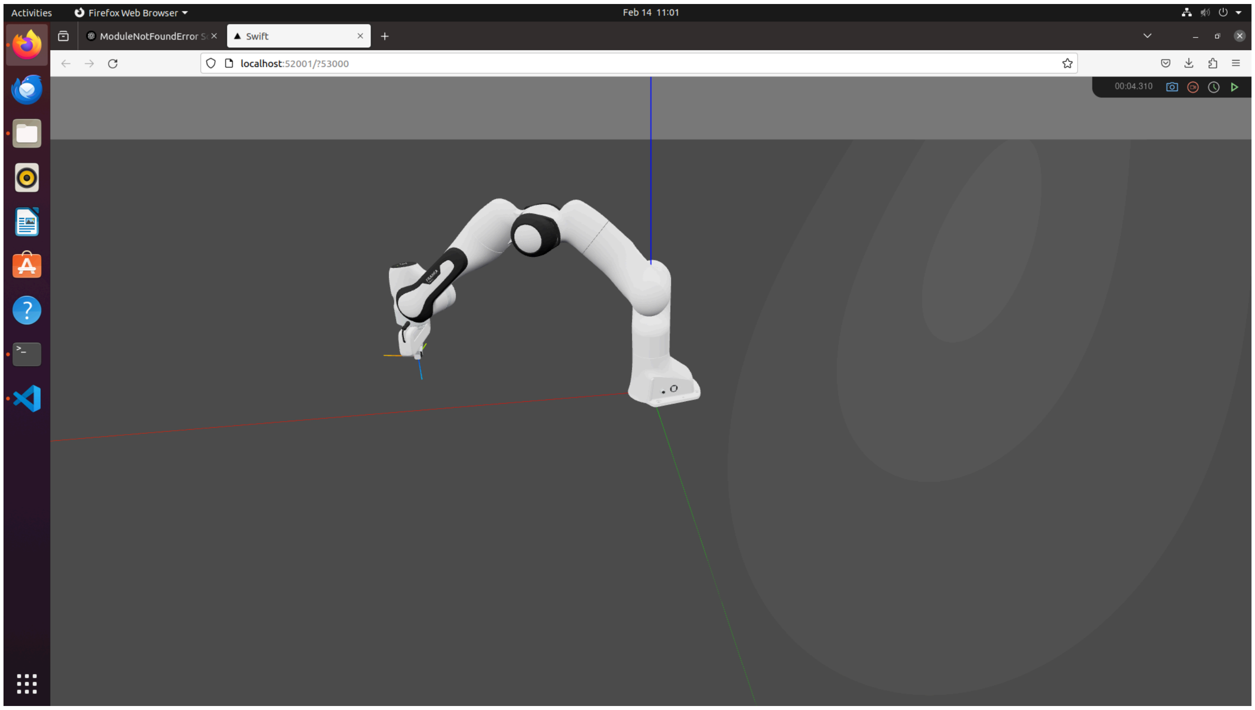
Start location:



Mid location:



End location:



Explanation:

The SpatialMath module provides SE(3) transformations, which define a 3D position and orientation. The script calculates a goal position for the robot using forward kinematics (`robot.fkine(robot.q)`) and then applies translations of 0.2m along X, 0.2m along Y, and 0.35m along Z using `sm.SE3.Tx(0.2) * sm.SE3.Ty(0.2) * sm.SE3.Tz(0.35)`.

To visualize the goal, coordinate axes are created at the goal position using `sg.Axes(length=0.1, base=goal)`, which are then added to the environment (`env.add(axes)`). A flag, `arrived = False`, is used to track whether the robot has reached the goal, and `dt = 0.01` defines the time step for simulation updates.

To move the robot towards the goal, `rtb.p_servo(robot.fkine(robot.q), goal, gain=1, threshold=0.01)` calculates the required velocity (`v`). The gain parameter (1) determines how aggressively the robot adjusts its movement, while the threshold (0.01m or 1 cm) defines the stopping condition. The function returns `arrived=True` when the robot is close enough to the goal.

The robot's Jacobian matrix is computed using `robot.jacobe(robot.q)`, which describes how joint velocities influence the end-effector motion. The pseudoinverse of the Jacobian is calculated using `np.linalg.pinv(J)`, and multiplying this with `v` determines the joint velocities (`qd`). Finally, the simulation advances by `dt=0.01` seconds using `env.step(dt)`, updating the robot's motion.

Example 04:

Code:

rtb_example4.py > ...

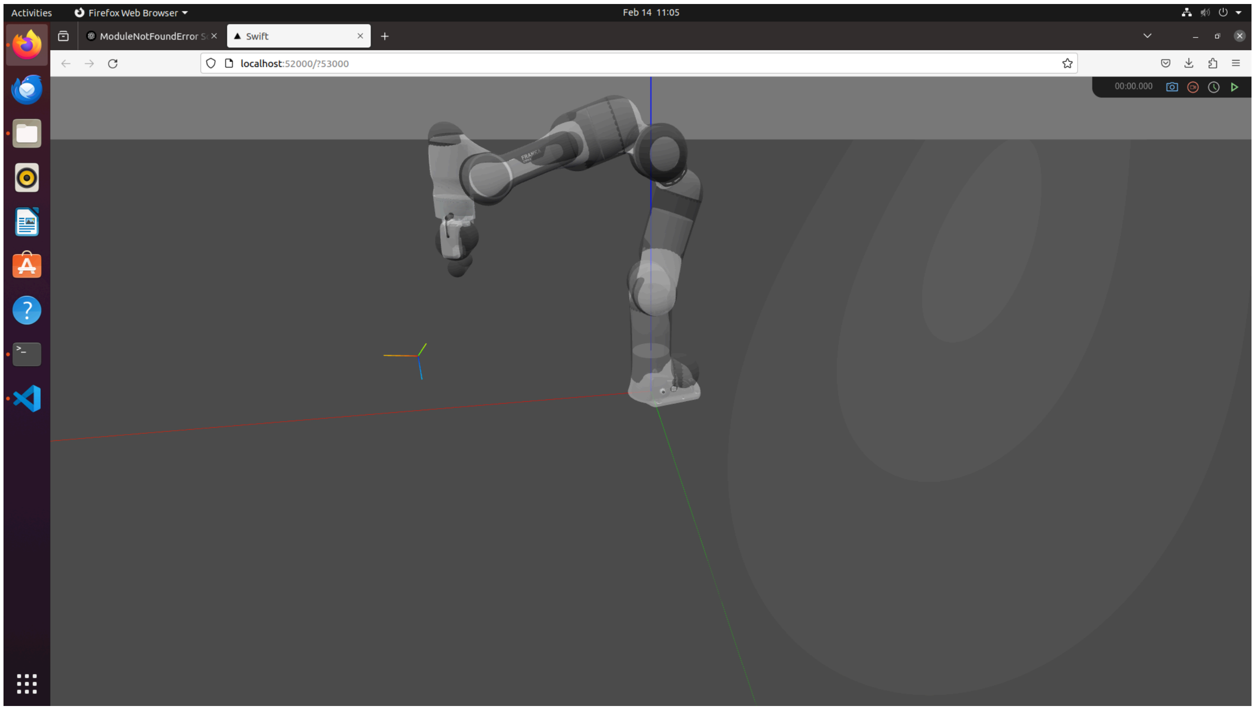
```python
#-------- Example-4---------#
# Collotion geometry

import roboticstoolbox as rtb # type: ignore
import swift # type: ignore
import numpy as np
import spatialmath as sm # type: ignore
import spatialgeometry as sg # type: ignore

# Create swift instance
env =swift.Swift()
env.launch(realtime=True)

# Define the robot model
robot = rtb.models.Panda()
robot.q = robot.qr

# Add robot to swift
env.add(robot, robot_alpha=0.5, collision_alpha=0.5)

# Set goal position
goal = robot.fkine(robot.q) * sm.SE3.Tx(0.2) * sm.SE3.Ty(0.2) * sm.SE3.Tz(0.35)
axes = sg.Axes(length=0.1, base=goal)
env.add(axes)

# Arrived at a destination flag
arrived = False
```


```python
# Arrived at a destination flag
arrived = False
```

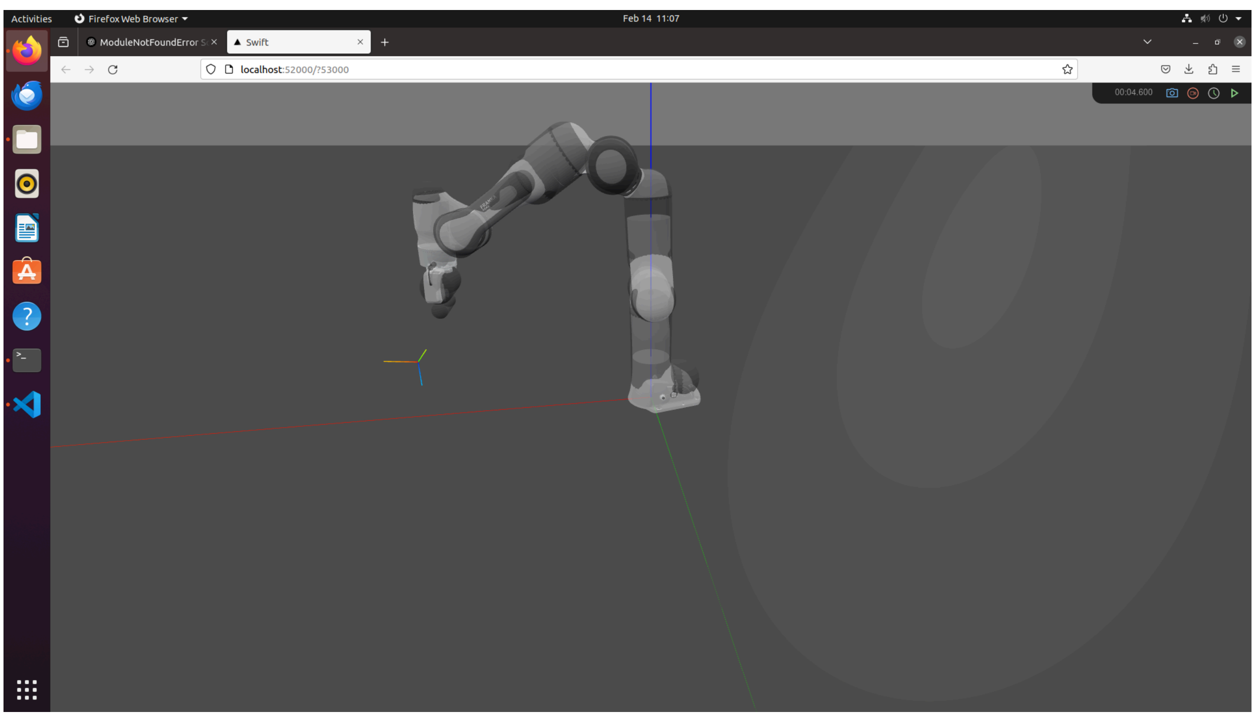

```python

# Time step
dt = 0.01

while not arrived:
    # v is a 6 vector representing the spatial error
    v, arrived = rtb.p_servo(robot.fkine(robot.q), goal, gain=0.1, threshold=0.01)
    J = robot.jacobe(robot.q)
    robot.qd = np.linalg.pinv(J) @ v

    # Step the environment
    env.step(dt)

# Stop the browser tab from closing
env.hold()
```

Output:

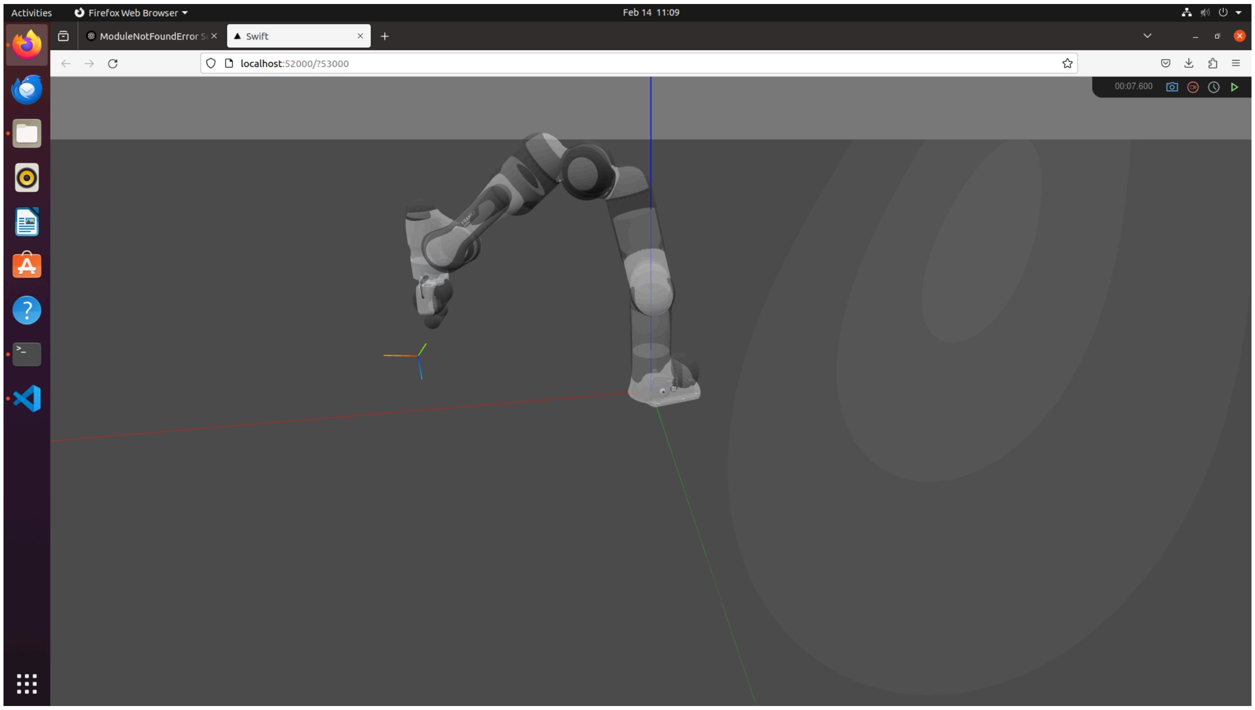
Start location:



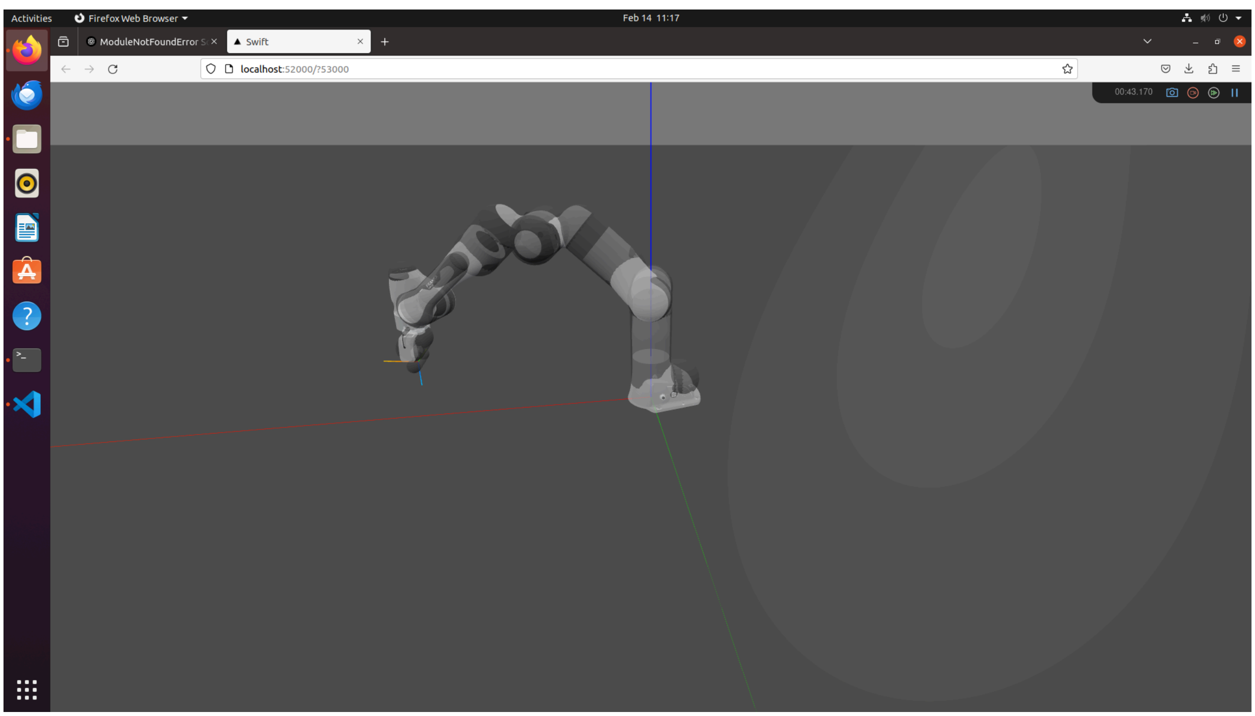
Mid location 1:



Mid location 2:



End location:



Explanation:

- **env.add(robot, robot_alpha=0.5, collision_alpha=0.5)**
  - Adds the robot to Swift's visualization.
  - robot_alpha=0.5: Makes the robot model semi-transparent.
  - collision_alpha=0.5: Enables a semi-transparent collision model to help visualize potential collisions.
- **goal = robot.fkine(robot.q) * sm.SE3.Tx(0.2) * sm.SE3.Ty(0.2) * sm.SE3.Tz(0.35)**
  - Computes the robot's current end-effector position using forward kinematics (fkine).
  - Moves the goal position by 0.2m in X, 0.2m in Y, and 0.35m in Z using SE(3) transformations.
- **v, arrived = rtb.p_servo(robot.fkine(robot.q), goal, gain=0.1, threshold=0.01)**
  - Computes velocity v needed to move towards the goal.
  - gain=0.1: Controls motion aggressiveness (lower than previous examples for smoother movement).
  - threshold=0.01: The robot stops when it is within 1 cm of the goal.
- **J = robot.jacobe(robot.q):** Computes the Jacobian matrix (J), which maps joint velocities to end-effector motion.
- **np.linalg.pinv(J):** Computes the pseudoinverse of the Jacobian (J).
- **@ v:** Multiplies the pseudoinverse with velocity vector (v) to compute joint velocities (qd).

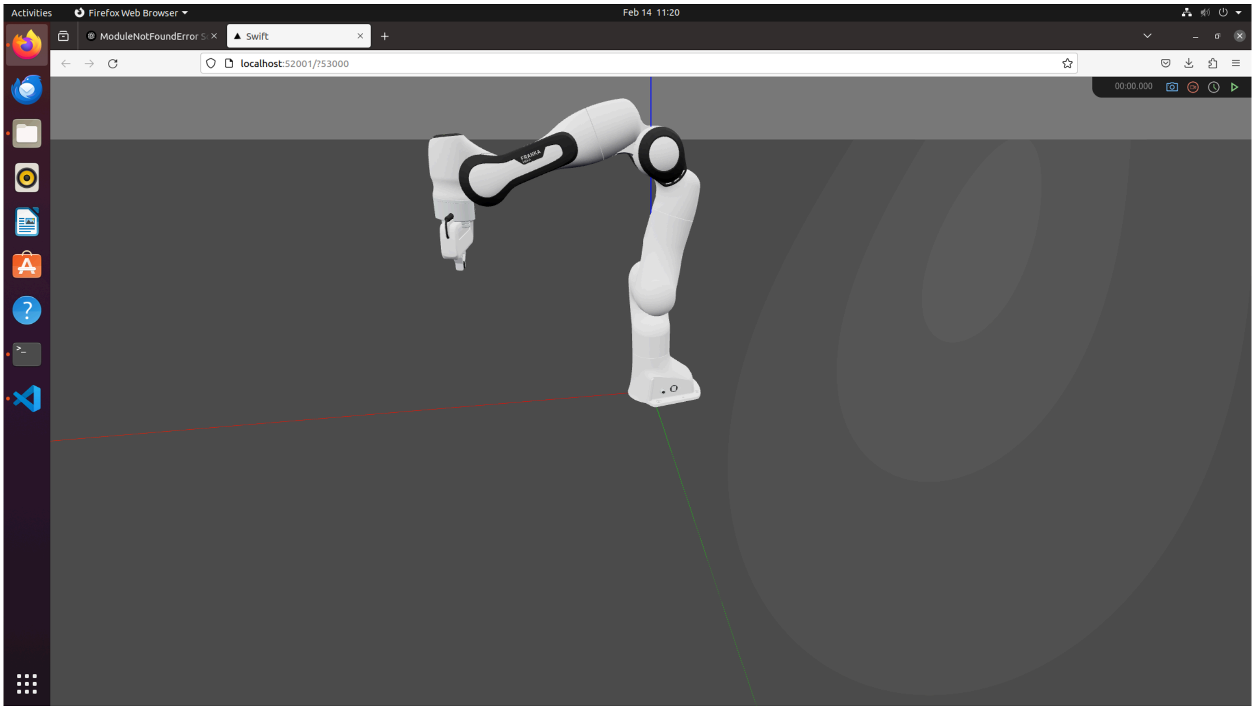
Example 05:
Code:

rtb_example5.py > ...

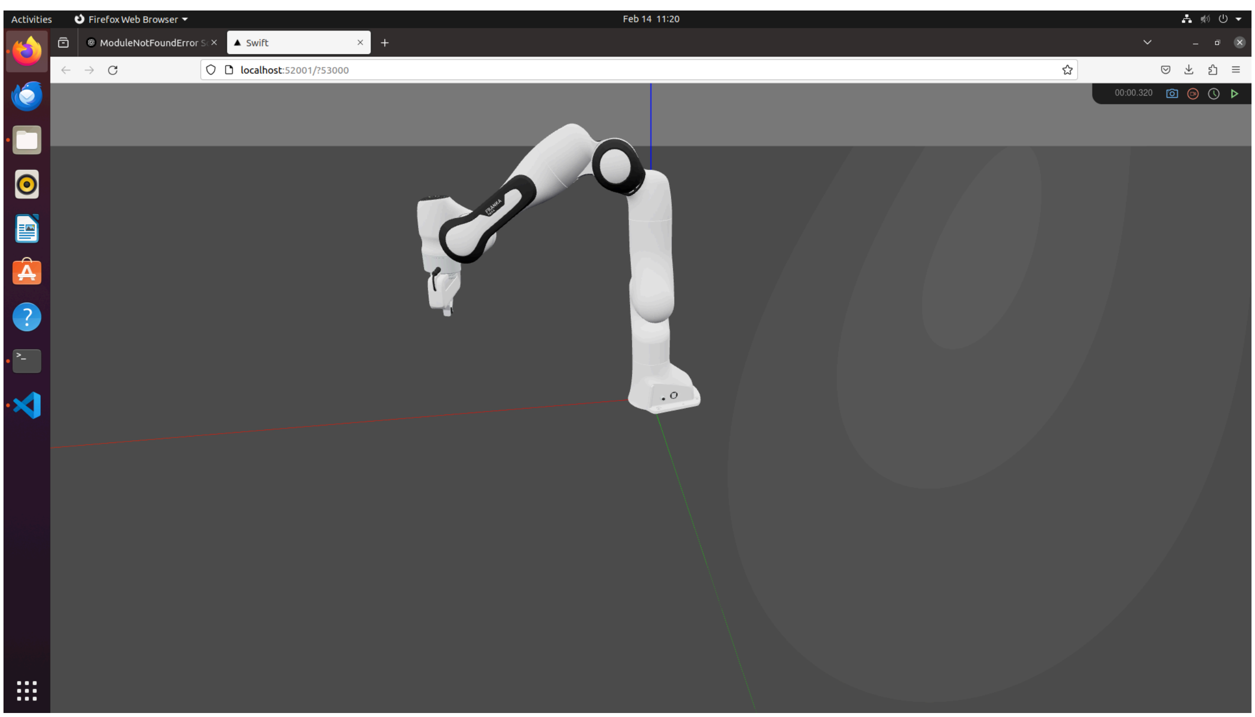
```python
env = swift.Swift()
env.launch(realtime=True)

panda = rtb.models.Panda()
panda.q = panda.qr

Tep = panda.fkine(panda.q) * sm.SE3.Trans(0.2, 0.2, 0.45)

arrived = False
env.add(panda)

dt = 0.005

while not arrived:

    v, arrived = rtb.p_servo(panda.fkine(panda.q), Tep, 1)
    panda.qd = np.linalg.pinv(panda.jacobe(panda.q)) @ v
    env.step(dt)

# Uncomment to stop the browser tab from closing
env.hold()
```

Output:

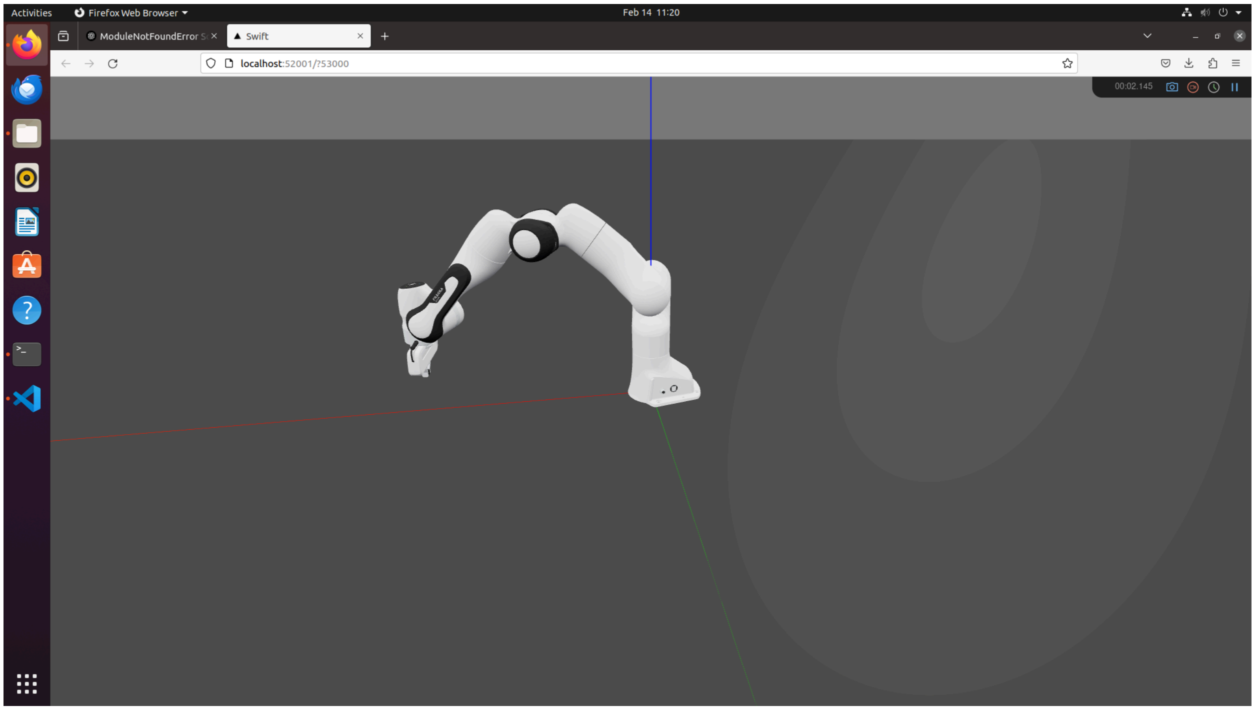
Start location:



Mid location:



End location:



Explanation:

- **panda = rtb.models.Panda()**
  - **panda.q = panda.qr**
  - Loads the Panda robotic arm model.
  - Sets the robot’s joint angles (q) to a default ready position (qr).
- **Tep = panda.fkine(panda.q) * sm.SE3.Trans(0.2, 0.2, 0.45)**
  - Computes the robot's current end-effector position (fkine(panda.q)).
  - Transforms the goal position by shifting it 0.2m in X, 0.2m in Y, and 0.45m in Z.
- **arrived = False** : Flag to check if the robot has reached the target position.
- **v, arrived = rtb.p_servo(panda.fkine(panda.q), Tep, 1)**
  - Computes the velocity v required to reach the goal using proportional servoing (p_servo).
  - Gain factor 1 controls the movement speed.
  - Updates **arrived** when the robot gets close to the goal.
- **panda.qd = np.linalg.pinv(panda.jacobe(panda.q)) @ v**
  - Computes joint velocities (qd) using the Jacobian pseudoinverse (np.linalg.pinv()).
- **env.step(dt)**: Advances the simulation by dt=0.005 seconds, updating the robot’s motion.

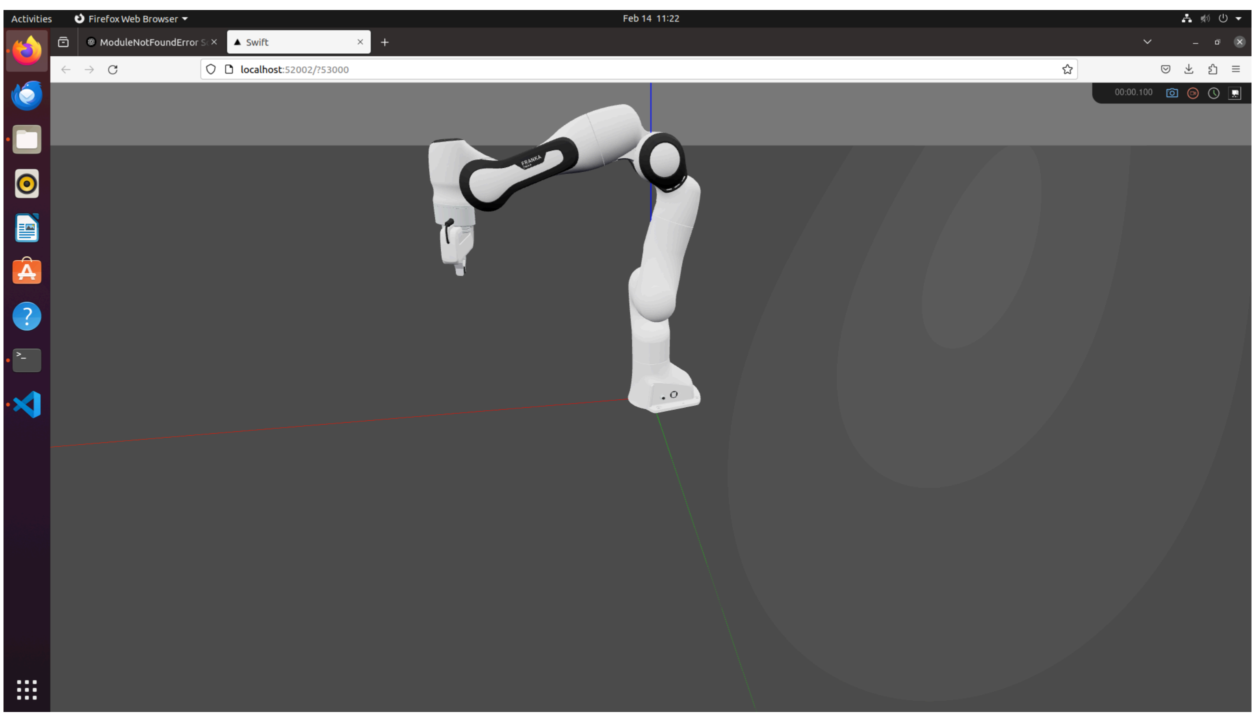
Example 06:

Code :

rtb_example6.py > ...

```python
# Step-1: Load a model of the Franka-Emika Panda robot defined by a URDF file
robot = rtb.models.Panda()
#print(robot)


# Step-2: Forward Kinematics
Te = robot.fkine(robot.qr)  # forward kinematics
#print(Te)

# Step-3: Inverse Kinematics
Tep = SE3.Trans(0.6, -0.3, 0.1) * SE3.OA([0, 1, 0], [0, 0, -1])
sol = robot.ik_LM(Tep)         # solve IK
#print(sol)

# Step-4: FK shows that desired end-effector pose was achieved
q_pickup = sol[0]
#print(robot.fkine(q_pickup))

# Step-5: Trajectory plot
qt = rtb.jtraj(robot.qr, q_pickup, 50)
#robot.plot(qt.q, backend='pyplot', movie='panda1.gif')

# Step-6: Plot the trajectory in the Swift simulator
robot.plot(qt.q)
```

Output:



Explanation:

- **Te = robot.fkine(robot.qr) # forward kinematics**
  - Computes the end-effector position (Te) when the robot is at its default ready position (qr).
- **Tep = SE3.Trans(0.6, -0.3, 0.1) * SE3.OA([0, 1, 0], [0, 0, -1])**
  - Defines the target pose (Tep) using SE(3) transformations:
  - SE3.Trans(0.6, -0.3, 0.1) → Moves the end-effector to (0.6m, -0.3m, 0.1m).
  - SE3.OA([0, 1, 0], [0, 0, -1]) → Sets the end-effector's orientation.
- **sol = robot.ik_LM(Tep) # solve IK**
  - Uses the Levenberg-Marquardt inverse kinematics solver (ik_LM) to find the required joint angles to reach Tep.
- **q_pickup = sol[0]**
  - Extracts the joint angles (q_pickup) from the IK solution.
- **robot.fkine(q_pickup)**
  - Checks if FK matches the target pose (Tep), ensuring IK was successful.
- **qt = rtb.jtraj(robot.qr, q_pickup, 50)**
  - Generates a smooth trajectory (qt) from the start position (qr) to the goal (q_pickup) in 50 steps.
- **robot.plot(qt.q, backend='pyplot', movie='panda1.gif')**
  - Plots the trajectory using Matplotlib (pyplot).
  - Saves the animation as panda1.gif (commented out in the provided code).
- **robot.plot(qt.q)**
  - Displays the robot's movement along the trajectory in Swift's 3D simulator.