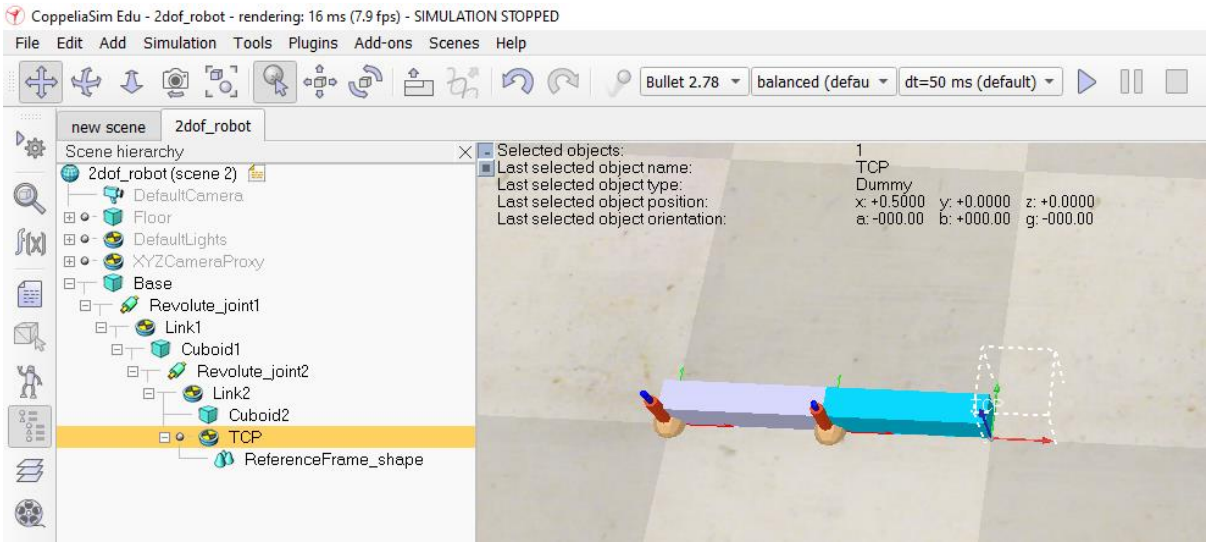
Practical Exercise 3: Inverse Kinematics

Objective:

In this practical exercise you should learn different possibilities to calculate the inverse kinematics of a robot: analytical solution, numerical solution calculating the jacobian by partial derivatives and numerical solution calculating the jacobian by numerical differentiation. For this purpose you implement the upcoming tasks in Python (here we provide a source code framework) or C++ (here, we **do not provide** a source code framework) and evaluate your results by visual inspection using CoppeliaSim.

Task 1:

Build a 2-DOF robot consisting of 2 rotational joints, rotating around the z-axis of the world coordinate system (WCS). You can use the snake robot and downgrade it to 2 dofs or you build from scratch. The following figure shows the 2-DOF robot in its zero position (all joints are zero).



Use the framework ***P2_Task1.py*** from exercise 2 as a starting point and calculate the inverse kinematics as a separate function with the following signature:

```
##############################################################################
# this needs to be implemented
def inverse_kinematics_2dof(x, y):
  # returning two angles, which generates the given TCP position in x and y
    ….
    ….
```

This function should provide an analytical solution (basic trigonometry). Your implementation should calculate two solutions (elbow up and down).

At start set all joints to 20 degrees to avoid singularities. Control your robot's TCP in cartesian space with your keyboard. You can use the following code snippet:

```
delta_pos = 0.002  # in m for each simulation step
delta_ori = 0.25 # in degree for each simulation step
if keyboard.is_pressed("u"):   # move up
    dx = np.array([0.0, delta_pos, 0.0])
if keyboard.is_pressed("n"):   # move down
    dx = np.array([0.0, -delta_pos, 0.0])
if keyboard.is_pressed("h"):    # move left
    dx = np.array([-delta_pos, 0.0, 0.0])
if keyboard.is_pressed("j"):    # move right
    dx = np.array([delta_pos, 0.0, 0.0])
if keyboard.is_pressed("k"):     # rotate TCP counterclockwise
    dx = np.array([0.0, 0.0, delta_ori*math.pi/180.0])
if keyboard.is_pressed("l"):     # rotate TCP clockwise
    dx = np.array([0.0, 0.0, -delta_ori*math.pi/180.0])
if keyboard.is_pressed("s"):    # stop movement
    dx = np.array([0.0, 0.0, 0.0])
```

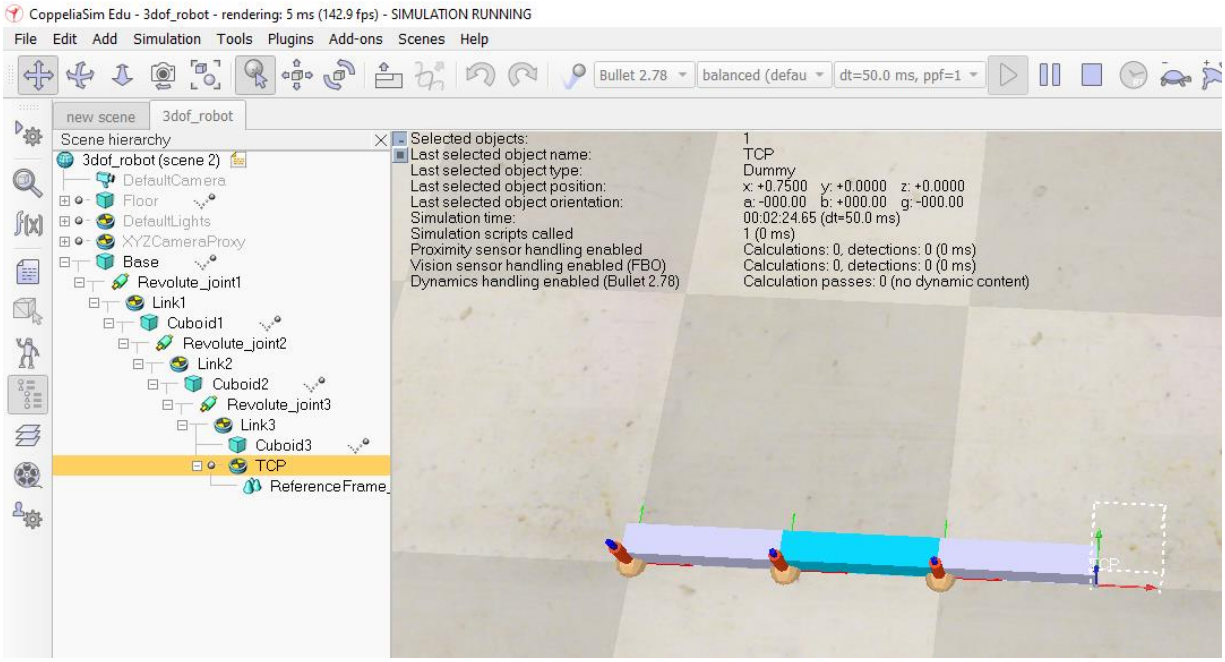
The key 'e' should toggle elbow up and down.

Proof of correctness: By inspecting correct motion behavior in CopelliaSim

Alternative (voluntary): Evaluate your result with a small GUI (adopted from exercise 1) to control the TCP in cartesian space.

Task 2:

Build a 3-DOF robot consisting of 3 rotational joints, rotating around the z-axis of the world coordinate system (WCS). You can use the snake robot and downgrade it to 3 dofs or you build from scratch. The following figure shows the 3-DOF robot in its zero position (all joints are zero).



Use the framework ***P2_Task2.py*** as a starting point and calculate the inverse kinematics as a separate function with the following signature:

```
##############################################################################
# this needs to be implemented

def inverse_jacobian_3dof(joint):
    # returning the inverse of the jacobian matrix depending on the 3 joint configurations
    ….
    ….
```

This function should provide a numerical solution calculating the jacobian matrix of the robot using partial derivatives. Use the ***np.linalg.pinv(m)*** function to compute the Moore-Penrose pseudo inverse of the jacobian matrix.

At start set all joints to 20 degrees to avoid singularities. Control your robot's TCP in cartesian space with your keyboard (see task 1).

Be aware, that this is a differential method based on linearization, so choose your values to steer the TCP carefully.

Proof of correctness: By inspecting correct motion behavior in CopelliaSim

Alternative (voluntary): Evaluate your result with a small GUI (adopted from exercise 1) to control the TCP in cartesian space.

Task 3:

Perform task 2 for your snake robot from practical exercise 2.

Alternatively, calculate the jacobian by numerical differentiation for the snakerobot. Calculate the inverse kinematics as a separate function with the following signature:

```
##########################################################################
# this needs to be implemented
def inverse_kinematics_numerical(snake_robot, joints):

# returning the inverse of the jacobian matrix depending on the snake's
joint configurations
    …..
```

Reuse your code of exercise 2 for the forward kinematic chain, which gives you the 4-by-4 homogenous transformation for a certain joint configuration. From this matrix derive the position (x and y) and the orientation (one angle) of the TCP and build the appropriate 3-dimensional column vectors for the jacobian. So, the jacobian matrix will be a 3x6 matrix.

At start set all joints to 0 degrees to avoid singularities. Control your robot's TCP in cartesian space with your keyboard (see task 1).

Alternative (voluntary): Evaluate your result with a small GUI (adopted from exercise 1) to control the TCP in cartesian space.

Proof of correctness: By inspecting correct motion behavior in CopelliaSim

Preparation:

Task1:

Calculate the inverse kinematic solution on paper.

Task2:

Calculate the jacobian on paper.

Task3:

Prepare, how to generate a numpy matrix of a certain dimension, how to multiply a matrix with a vector and how to invert a 3x6 matrix (in your case). Since your matrix is not quadratic you need to calculate the pseudo inverse.

Report:

The report should contain

- the task description itself (you can reuse this document, at the top your name(s) must be mentioned)
- a comprehensive description of the implemented algorithm
- Sketches and explaining figures
- source code with result output
- you deliver one zipped file. It contains the report as PDF, the .ttt scene files and the solution python files including all additional project files necessary, to run the program. Your program must start out of the box.
- You can provide one solution for one group consisting of max. 2 students.

Naming Conventions: as in exercise 2