ME 467 Robotics Project 3

Boise State University

Name: Caleb Hottes
Date: 5/1/2025
Class: ME 467

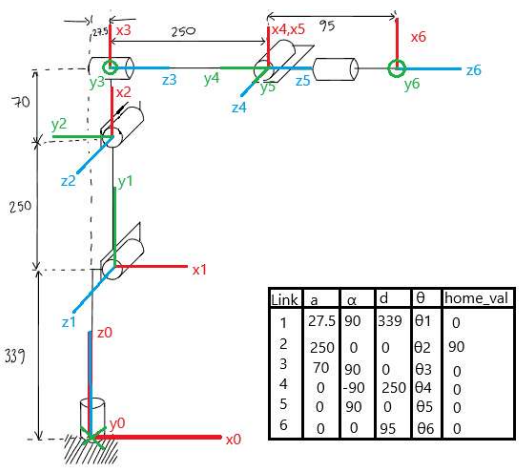
In this project, the forward and inverse position and velocity level kinematics of the NeXCOM 6-DoF miniBoT will be solved. The solutions will then be verified using MuJoCo simulations and other methods.

The initial focus is on position-level kinematics, specifically addressing both forward and inverse problems. Coordinate frames are assigned using the Denavit-Hartenberg (DH) convention and DH parameters are determined. This setup allows for a straightforward computations of the foreward kinematics. The inverse kinematics problem is approached using a closed-form geometric method, which is well-suited to this particular robot's DH parameter configuration. The implementation of this geometric inverse function accounts for potential singularities and employs a computational strategy to handle multiple possible solutions. The core challenge in inverse kinematics is to accurately determine the joint angles required to achieve a desired end-effector pose, given the complexities introduced by numerous edge cases.

Velocity-level kinematics are also addressed, involving the computation of the Jacobian matrix. The Jacobian relates joint angle rates to end-effector spatial twist, and its computation is crucial for determining the joint velocities required for desired end-effector motion.

Question 1-Position level kinematics

This question is about the foreward and inverse kinematics of the robot. Once coordinate frames are chose and Denavit-Hartenberg (DH) parameters are determined, it is trivial to compute the foreward kinematics. The coordinate frames were assigned and Denavit-Hartenberg parameters chosen as seen below.



Link	a	α	d	θ	home_val
1	27.5	90	339	θ1	0
2	250	0	0	θ2	90
3	70	90	0	θ3	0
4	0	-90	250	θ4	0
5	0	90	0	θ5	0
6	0	0	95	θ6	0

Figure 1: Diagram of coordinate frame placement and DH parameters

Figure 2: Real life image of the NeXCOM MiniBOT

Below some preliminary variables are created to drive the rest of the code.

In [9]:
```
import mujoco as mj, numpy as np, time
from numpy import deg2rad
from pathlib import Path
from spatialmath import SE3
from kinematics import DHKinematics, mini_bot_geometric_inverse
def get_pose(data, body_id):
    rot = data.xmat[body_id].reshape(3,3)
    pos = data.xpos[body_id] * 1000  # mm
    T = np.eye(4); T[:3,:3], T[:3,3] = rot, pos
    return SE3(T)
```

```
np.set_printoptions(suppress=True)

model = mj.MjModel.from_xml_path(str(Path("mujoco_files") / "robot_model.xml"))
data = mj.MjData(model)
end_effector_body_id = model.body(name="end-effector").id

# Define the Denavit-Hartenberg (DH) parameters for this robot arm.
dh_table = [[True, 27.5, np.pi/2, 339],
            [True, 250, 0, 0],
            [True, 70, np.pi/2, 0],
            [True, 0, -np.pi/2, 250],
            [True, 0, np.pi/2, 0],
            [True, 0, 0, 95]
            ]

home_angles = np.array([0, np.pi/2, 0, 0, 0, 0])
# Create the kinematics object with the home angles and the DH table.
mini_bot_kinematics = DHKinematics(home_angles, dh_table)
# Compute the transformation of a known position for use later.
home_pos = mini_bot_kinematics.foreward(home_angles)
```

(a) Forward Kinematics Problem

The instructions say: "Solve the position-level forward kinematics problem. Use this solution to find the end-effector pose **ξ** given that the joint angles are:

$$\theta = [0^\circ \quad 90^\circ \quad 0^\circ \quad 0^\circ \quad -90^\circ \quad 0^\circ]$$

Your end-effector ξ should be given by a 6-vector, the first three components of which are the components of the translation vector from the base to the origin of the end-effector expressed in the base frame, and the last three of which are the EulerZYX angles of the end-effector frame with respect to the base frame."

The DH convention states the homogeneous transform from frame i to frame j is given by:

$$^iT_j = A_{i+1} \cdots A_j = \begin{pmatrix} ^iR_j & ^i\mathbf{o}_j \\ 0 & 1 \end{pmatrix}.$$

Again by the DH convention each intermediate transformation is given by:

$$A_i = \mathcal{R}_z(\theta_i)\,\mathcal{T}_z(d_i)\,\mathcal{T}_x(a_i)\,\mathcal{R}_x(\alpha_i)$$

The `DHKinematics` class implements this formula in its function `DHKinematics.foreward(joint_angles: np.ndarray, *args)`. Using this function we can compute, as asked, the pose of the end-effector at the given pose.

In [10]:
```
question_1_angles = np.array([0, deg2rad(90), 0, deg2rad(-90), 0])
print("Question 1a transformation from foreward kinematics:")
question_one_transformation = mini_bot_kinematics.foreward(question_1_angles)
print(question_one_transformation)
print(f"[v,Euler ZYX] format {mini_bot_kinematics.transformation_to_minimal_representation(question_one_transformation)}")
```

```
Question 1a transformation from foreward kinematics:
   1         0         0         277.5
   0        -1         0         0
   0         0        -1         564
   0         0         0         1

[v,Euler ZYX] format [277.5  -0.  564.  180.   -0.    0. ]
```

(b) Inverse Position Kinematics

The instructions specify solving the position-level inverse kinematics of the problem in closed-form. The expressions for the joint angles in terms of the given end-effector pose are to be provided. This solution is then used to find the joint angles for the end-effector pose given by $\xi = (\mathbf{R}, \mathbf{t})$, where

$$R = \begin{bmatrix} 0.7551 & 0.4013 & 0.5184 \\ 0.6084 & -0.7235 & -0.3262 \\ 0.2441 & 0.5617 & -0.7905 \end{bmatrix}, \quad t = \begin{bmatrix} 399.1255 \\ 171.01529 \\ 416.0308 \end{bmatrix}$$

The inverse kinematics are solved here through a closed-form geometric method, tailored to this specific robot's Denavit-Hartenberg parameters. While extensively tested, the geometric inverse function might encounter limitations at singularities. A fundamental challenge in geometric inverse kinematics is the presence of numerous edge cases, making a single universal formula impractical. Consequently, the determination of correct solutions across all inputs necessitates a robust strategy.

Two primary approaches exist for this problem:

- Employing logical conditions to select the appropriate formula.
- Evaluating all permutations of potential angles and selecting valid resulting configurations.

Given the potential complexity and error susceptibility of implementing intricate logical conditions, a computational approach is adopted, leveraging the processing power of a computer to explore multiple possibilities. Rather than a singular formula, several formulas are utilized to compute many joint angle combinations. Subsequently, each combination undergoes validation, and only those yielding the desired end-effector pose are retained as valid solutions.

The objective is to find the joint angles that produce a given end-effector pose $(\mathbf{R}, \mathbf{o})$, representing the desired rotation and translation. The process involves several stages:

1. Decoupling

The wrist center $\mathbf{o}_c$ is defined as the intersection point of the joint axes of the spherical wrist. Its position is solely dependent on the first three joint angles, $\theta_1, \theta_2, \theta_3$. This characteristic allows for the "decoupling" of the position and orientation inverse kinematics problems into two independent sub-problems.

The wrist center's position is found by exploiting its constant relationship with respect to the end-effector frame:

$$\mathbf{o_c} = \mathbf{o} - d_6\mathbf{Rz}$$

where d_6 is a Denavit-Hartenberg parameter and $\mathbf{z} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$ is the z-axis of the end-effector frame.

2. Geometry

The subsequent task involves utilizing the robot's geometric structure to determine the first three joint angles, which allows for the construction of the rotation matrix ${}^0R_3$.

Derivation of θ_1

Observation of the robot's geometry in figure 2 reveals that the first joint angle θ_1 can be directly computed from the wrist center's x and y coordinates in the base frame:

$$\theta_1 = \operatorname{atan2}(y_c, x_c)$$

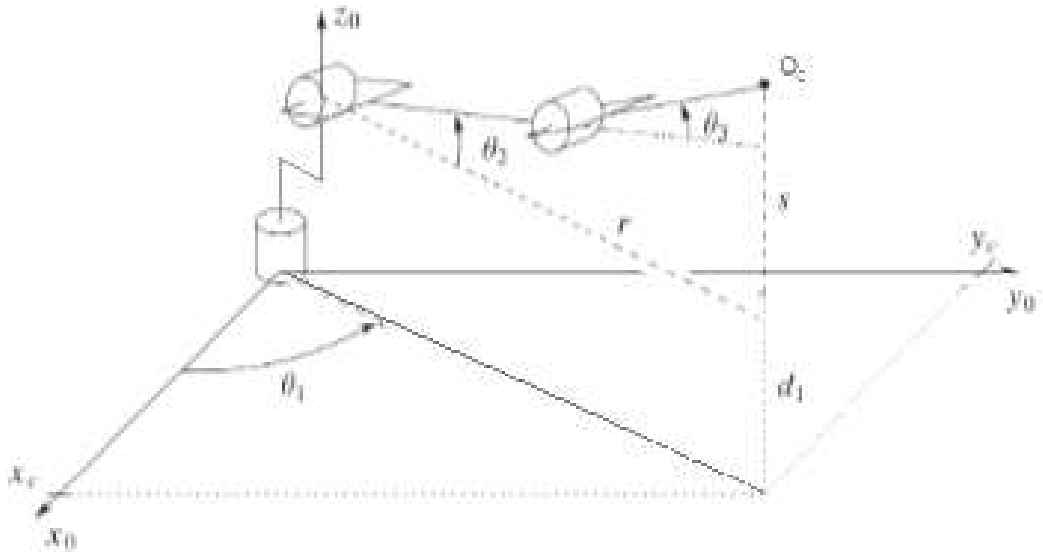


Figure 3: Isometric view of first three links

The articulated configuration of the robot prevents there being two solutions separated by 180° as there would be in a non-articulated robot.

Derivation of θ_3

Several parameters are first defined:

- d: The offset of the first link, $d_1 = 27.5$ mm.
- r: The projected distance from the second joint to the wrist center in the x_0y_0 plane.
- s: The height of the wrist center above the second joint.
- α: A constant angle, approximately 105.64°.
- β: Another constant angle, approximately 74.36°.

From the robot's geometric relations (fig. 2), the following equations are derived:

$$r = \sqrt{x_c^2 + y_c^2} - d$$

$$s = z_c - d_1$$

$$a_3 = \sqrt{a_3^2 + d_4^2} = \sqrt{70^2 + 250^2}$$

In the configuration shown in fig 3. the angle θ_3 is related to an intermediate angle C through $\theta_3 = C - \alpha$. By applying the law of cosines, $cos(C)$ is found:

$$cos(C) = \frac{1}{-2a_2a_3}(r^2 + s^2 - a_2^2 - a_3^2) := D$$

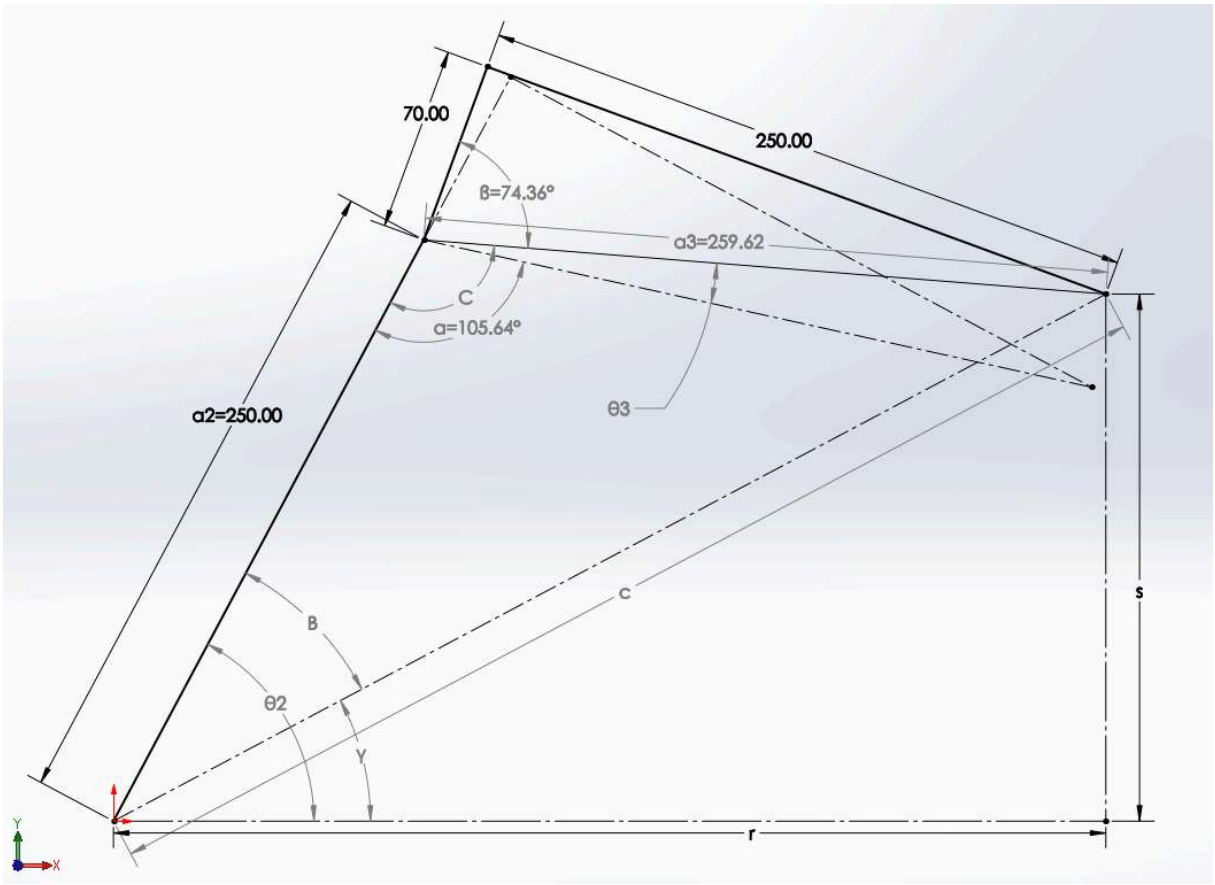


Figure 4: Projected view of links 2 and 3 from joint 2's frame.

Due to the non-injectivity of the cosine function, the atan2 function and the Pythagorean trigonometric identity are used to find C:

$$C = \text{atan2}(\pm\sqrt{1 - D}, D)$$

This yields two initial solutions for θ_3:

$$\theta_3 = \text{atan2}(\pm\sqrt{1 - D}, D) - \alpha$$

Further geometric analysis reveals additional relations for θ_3 valid for similar configurations not pictured, involving the constant angle β:

$$\theta_3 = \pi - C + \beta$$

$$\theta_3 = 2\pi - \alpha - C$$

The full expansions in terms of x_c and y_c are shown below.

$$\theta_3 = \text{atan2}(\pm\sqrt{1 - \frac{1}{-2a_2a_3}((\sqrt{x_c^2 + y_c^2} - d)^2 + (z_c - d)^2 - a_2^2 - a_3^2)}, \frac{1}{-2a_2a_3}((\sqrt{x_c^2 + y_c^2} - d)^2 + (z_c - d)^2 - a_2^2 - a_3^2)) - \alpha$$

$$\theta_3 = \pi - \text{atan2}(\pm\sqrt{1 - \frac{1}{-2a_2a_3}((\sqrt{x_c^2 + y_c^2} - d)^2 + (z_c - d)^2 - a_2^2 - a_3^2)}, \frac{1}{-2a_2a_3}((\sqrt{x_c^2 + y_c^2} - d)^2 + (z_c - d)^2 - a_2^2 - a_3^2)) + \beta$$

$$\theta_3 = 2\pi - \alpha - \text{atan2}(\pm\sqrt{1 - \frac{1}{-2a_2a_3}((\sqrt{x_c^2 + y_c^2} - d)^2 + (z_c - d)^2 - a_2^2 - a_3^2)}, \frac{1}{-2a_2a_3}((\sqrt{x_c^2 + y_c^2} - d)^2 + (z_c - d)^2 - a_2^2 - a_3^2))$$

These relations, combined with the $\pm$ ambiguity in the atan2 function, lead to a total of six potential solutions for θ_3.

Derivation of θ_2

From the robot's geometry, the second joint angle θ_2 is related to another intermediate angle B and a constant angle γ: $\theta_2 = B + \gamma$. Similar to the derivation of θ_3, the law of cosines and the atan2 function are employed to find B:

$$c = \sqrt{r^2 + s^2}$$

$$cos(B) = \frac{1}{-2a_2c}(a_3^2 - c^2 - a_2^2) := E$$

$$B = \text{atan2}(\pm\sqrt{1-E}, E)$$

Thus, θ_2 is given by:

$$\theta_2 = \text{atan2}(\pm\sqrt{1-E}, E) + \gamma$$

$$\theta_2 = \text{atan2}(\pm\sqrt{1 - \frac{1}{-2a_2c}(a_3^2 - (\sqrt{r^2+s^2})^2 - a_2^2)}, \frac{1}{-2a_2c}(a_3^2 - (\sqrt{r^2+s^2})^2 - a_2^2)) + \gamma$$

This yields two potential solutions for θ_2.

3. Position Solution Filtering

Combining the derived solutions yields one θ_1, two θ_2's, and six θ_3's, resulting in 12 possible combinations for the first three joint angles. To identify the valid solutions, the forward kinematics are computed for each of these combinations, with the remaining three joint angles set to zero. A combination is considered a valid position-level solution if the resulting wrist center matches the wrist center calculated from the desired end-effector pose.

4. Orientation

The final three joint angles, $\theta_4, \theta_5, \theta_6$, are determined by extracting the ZYZ Euler angles from the rotation matrix ${}^3R_6$, which represents the orientation of the end-effector frame $\{6\}$ with respect to the third frame $\{3\}$. This matrix is obtained from the known desired rotation matrix R and the rotation matrix ${}^0R_3$ obtained from the first three joint angles:

$${}^3R_6 = {}^0R_3^{-1}R$$

The Euler angles $[\phi, \theta, \psi]$ are then extracted from ${}^3R_6$ using `spatialmath.base.transforms3d.r2x`. The function works as follows:

- **First angle ϕ (non–singular case)**

  From $R = R_z(\phi)\,R_y(\theta)\,R_z(\psi)$ we have

  $$R_{0,2} = \cos\phi\,\sin\theta, \quad R_{1,2} = \sin\phi\,\sin\theta.$$

  Hence

  $$\phi = \text{atan2}\big(R_{1,2},\, R_{0,2}\big),$$

- **Second angle θ**

  Still in the non–singular case, note

  $$R_{2,2} = \cos\theta, \quad \cos\phi\,R_{0,2} + \sin\phi\,R_{1,2} = \sin\theta.$$

  Thus

  $$\theta = \text{atan2}\big(\cos\phi\,R_{0,2} + \sin\phi\,R_{1,2},\; R_{2,2}\big).$$

- **Third angle ψ**

  From the upper-left 2×2 block of R:

  $$R_{0,0} = \cos\phi\cos\psi - \sin\phi\sin\psi, \quad R_{1,0} = \sin\phi\cos\psi + \cos\phi\sin\psi,$$

  one arrives at

  $$\psi = \text{atan2}\big(-\sin\phi\,R_{0,0} + \cos\phi\,R_{1,0},\; -\sin\phi\,R_{0,1} + \cos\phi\,R_{1,1}\big).$$

Conclusion

The described geometric inverse kinematics method is implemented in the function `minibotinversekinematics.mini_bot_geometric_inverse`. This function takes the desired end-effector pose as input and uses the robot's kinematic parameters to return the sets of joint angles that achieve the desired pose. Below the provided transformation matrix is used with this function to find the corresponding joint angles.

In [11]:
```
given_transformation = np.array([
    [.7551, .4013, .5184, 399.1255],
    [.6084, -.7235, -.3262, 171.01526],
    [.2441, .5617, -.7905, 416.0308],
    [0, 0, 0, 1]
])
start = time.time()
inverse_kinematics_solutions = mini_bot_geometric_inverse(given_transformation, mini_bot_kinematics)
stop_time = time.time()
for i, solution in enumerate(inverse_kinematics_solutions):
    deg_angles = np.round(np.degrees(solution), 3).tolist()
    is_correct = np.allclose(given_transformation, mini_bot_kinematics.foreward(solution).A, atol=1e-3)
```

```
    print(f"Solution {i + 1}: {deg_angles} is {'correct' if is_correct else 'incorrect'}")
print(f"Computed in {(stop_time - start) * 1000:.2f} ms")
```

```
Solution 1: [30.0, -15.998, 148.716, 145.037, 109.042, 156.916] is correct
Solution 2: [30.0, 60.0, -0.0, 134.998, 50.0, -157.501] is correct
Computed in 6.28 ms
```

Question 2 - Velocity Level Kinematics

The question states:

"Compute the kinematic Jacobian for this manipulator that relates the joint angle rates to the end-effector velocity in closed-form. This expression can be performed without choosing a frame, as discussed in class.

Implement your Jacobian matrix in Python and solve for the joint angle rates needed to produce an end-effector spatial twist of

$$\boldsymbol{\nu} = \begin{pmatrix} \boldsymbol{\omega} \\ \mathbf{v} \end{pmatrix}^T = \begin{pmatrix} 2 & -1 & 0.5 & 100 & -200 & -300 \end{pmatrix}^T$$

whenever the manipulator is at a configuration given in question 1b."

The question asks for the computation of the kinematic Jacobian for the manipulator, which relates the joint angle rates to the end-effector velocity in closed-form, without choosing a specific frame. Subsequently, the joint angle rates needed to produce a given end-effector spatial twist at a specified manipulator configuration are to be determined.

To address this, the kinematic Jacobian $\mathbf{J}$ for the manipulator, which relates the joint angle rates $\dot{\mathbf{q}} \in \mathbb{R}^n$ to the end-effector spatial twist $\boldsymbol{\eta} \in \mathbb{R}^6$, must be considered. This relationship is mathematically expressed as:

$$\boldsymbol{\eta} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}$$

where $\mathbf{q}$ is the vector of joint angles. The spatial twist $\boldsymbol{\eta}$ is a combination of the end-effector's linear velocity $\mathbf{v}_n \in \mathbb{R}^3$ and angular velocity $\boldsymbol{\omega}_n \in \mathbb{R}^3$, typically represented with respect to the base frame $\{0\}$:

$$\boldsymbol{\eta} = \begin{pmatrix} {}^0\mathbf{v}_n \\ {}^0\boldsymbol{\omega}_n \end{pmatrix}$$

The Jacobian matrix $\mathbf{J}(\mathbf{q})$ is a $6 \times n$ matrix that is dependent on the robot's configuration $\mathbf{q}$. It can be partitioned into components representing linear and angular velocity contributions:

$$\mathbf{J}(\mathbf{q}) = \begin{pmatrix} \mathbf{J}_v(\mathbf{q}) \\ \mathbf{J}_\omega(\mathbf{q}) \end{pmatrix}_{6 \times n}$$

For a robot with n joints, the i-th column of the Jacobian, $\mathbf{J}_i(\mathbf{q})$, represents the contribution of the i-th joint's rate $\dot{q}_i$ to the end-effector twist. The form of this column depends on the nature of the i-th joint.

In the case of a revolute joint i with an axis of rotation defined by the unit vector $\mathbf{z}_i \in \mathbb{R}^3$ (expressed in the base frame) passing through a point with position vector ${}^0\mathbf{o}_i \in \mathbb{R}^3$, and given the end-effector's origin position ${}^0\mathbf{o}_n \in \mathbb{R}^3$, the i-th column of the Jacobian is:

$$\mathbf{J}_i(\mathbf{q}) = \begin{pmatrix} {}^0\mathbf{z}_i \times ({}^0\mathbf{o}_n - {}^0\mathbf{o}_i) \\ {}^0\mathbf{z}_i \end{pmatrix}$$

For a prismatic joint i with an axis of translation defined by the unit vector $\mathbf{z}_i \in \mathbb{R}^3$, the i-th column of the Jacobian is:

$$\mathbf{J}_i(\mathbf{q}) = \begin{pmatrix} {}^0\mathbf{z}_i \\ \mathbf{0}_{3\times 1} \end{pmatrix}$$

To determine the joint angle rates $\dot{\mathbf{q}}$ required to produce the desired end-effector spatial twist $\boldsymbol{\nu} = \begin{pmatrix} 100 & -200 & -300 & 2 & -1 & 0.5 \end{pmatrix}^T$ at a specific robot configuration $\mathbf{q}^*$ (obtained from Question 1b), the linear system $\boldsymbol{\eta} = \mathbf{J}(\mathbf{q}^*)\dot{\mathbf{q}}$ must be solved. Rearranging the desired twist to match the Jacobian's structure yields $\boldsymbol{\eta} = \begin{pmatrix} 100 \\ -200 \\ -300 \\ 2 \\ -1 \\ 0.5 \end{pmatrix}$.

If the Jacobian $\mathbf{J}(\mathbf{q}^*)$ is a square (6×6 for a 6-DOF robot) and non-singular matrix, the joint rates can be found by direct inversion:

$$\dot{\mathbf{q}} = \mathbf{J}(\mathbf{q}^*)^{-1}\boldsymbol{\eta}$$

However, in cases where $\mathbf{J}(\mathbf{q}^*)$ might be singular (rank-deficient), the Moore-Penrose pseudoinverse $\mathbf{J}(\mathbf{q}^*)^\dagger$ provides a more general solution. For a full-rank Jacobian, a common form is the right pseudoinverse:

$$\mathbf{J}(\mathbf{q}^*)^\dagger = \mathbf{J}(\mathbf{q}^*)^T(\mathbf{J}(\mathbf{q}^*)\mathbf{J}(\mathbf{q}^*)^T)^{-1}$$

The joint angle rates are then computed as:

$$\dot{\mathbf{q}} = \mathbf{J}(\mathbf{q}^*)^\dagger\boldsymbol{\eta} = \mathbf{J}(\mathbf{q}^*)^T(\mathbf{J}(\mathbf{q}^*)\mathbf{J}(\mathbf{q}^*)^T)^{-1}\boldsymbol{\eta}$$

The implementation of the `DHKinematics.jacobian` function in Python allows for the computation of $\mathbf{J}(\mathbf{q})$. The desired joint rates $\dot{\mathbf{q}}$ are obtained by multiplying the pseudoinverse with the desired spatial twist $\boldsymbol{\eta}$.

In [12]:
```
question_2_twist = np.array([100, -200, -300, 2, -1, .5])
rate_solutions = []
for i, solution in enumerate(inverse_kinematics_solutions):
    jacobian = mini_bot_kinematics.jacobian(joint_angles=solution, link=6)
    rate_solutions.append(np.linalg.pinv(jacobian) @ question_2_twist.tolist())
    print(f"Joint rates for solution {i + 1}: {np.round(rate_solutions[i], 3)}")
```

```
Joint rates for solution 1: [-0.815 -1.407  1.779  2.056 -1.299  0.792]
Joint rates for solution 2: [-0.815  0.425 -1.779 -0.46  -3.518  1.352]
```

Question 3-Mujoco verification

Here the mujoco model of this robot is used to verify the answers to the rest of the project.

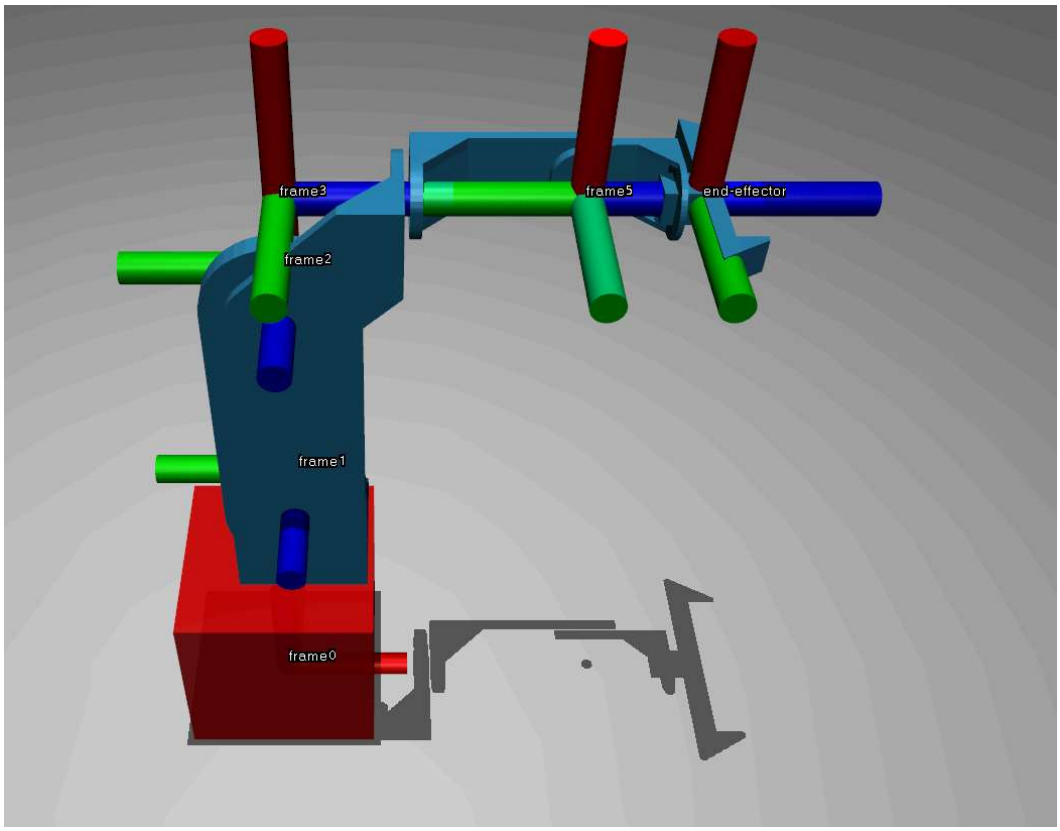


Figure 5: The mujoco model in the home position

Question 1a Verification

For the foreward kinematics we can simply set the model to the joint angles given and read off the transformation of the end-effector. Below the model is shown in that position.

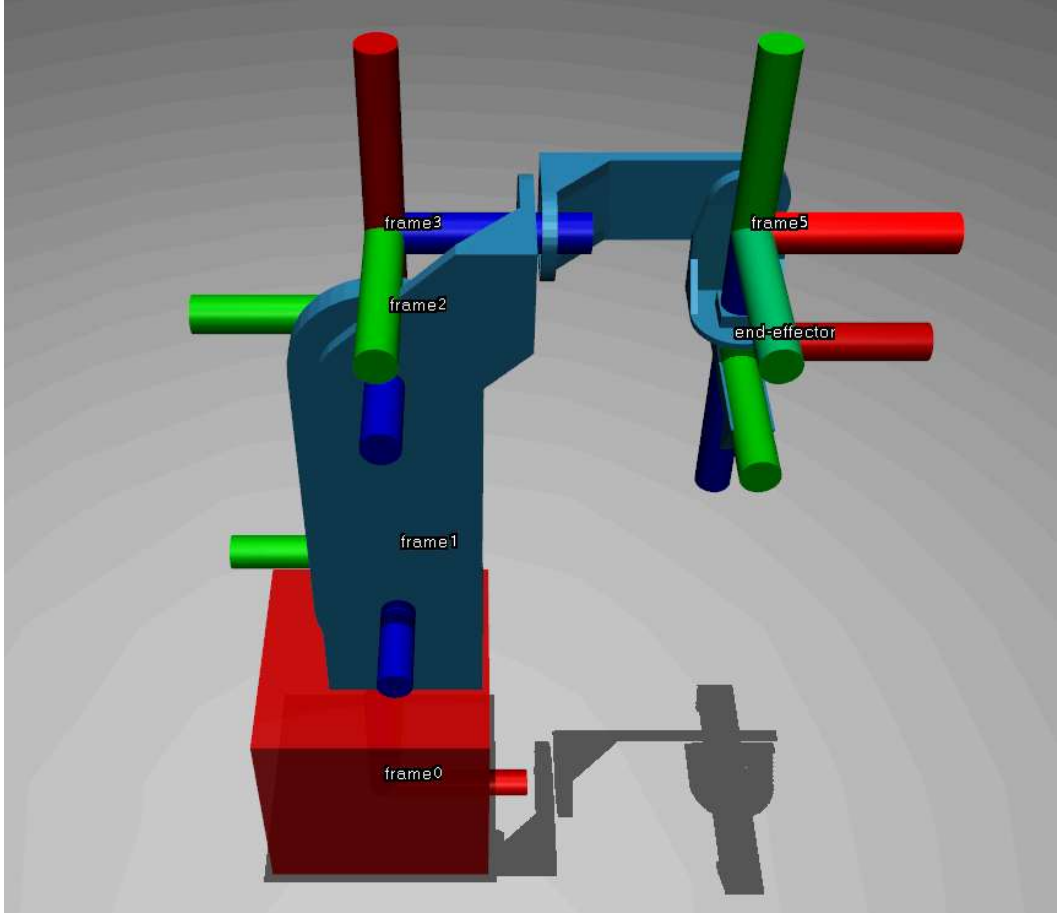


Figure 6: The mujoco model in position for question 1a

In [13]:
```
data.qpos[:len(question_1_angles)] = question_1_angles
mj.mj_forward(model, data)
mujoco_question_one_pose = get_pose(data, end_effector_body_id)
print(f"The transformation as produced by mujoco is:")
print(mujoco_question_one_pose)
if np.allclose(np.array(mujoco_question_one_pose), np.array(question_one_transformation), atol=1e-8):
    print("Question 1 part A is correct.")
else:
    print("Question 1 part A is not correct. ")
```

```
The transformation as produced by mujoco is:
   1         0         0         277.5
   0        -1         0         0
   0         0        -1         564
   0         0         0         1

Question 1 part A is correct.
```

Question 1b Verification

To verify question 1b we will set the mujoco model to the calculated joint angles, read off the transformation and verify that it matches the one given in the project instructions.

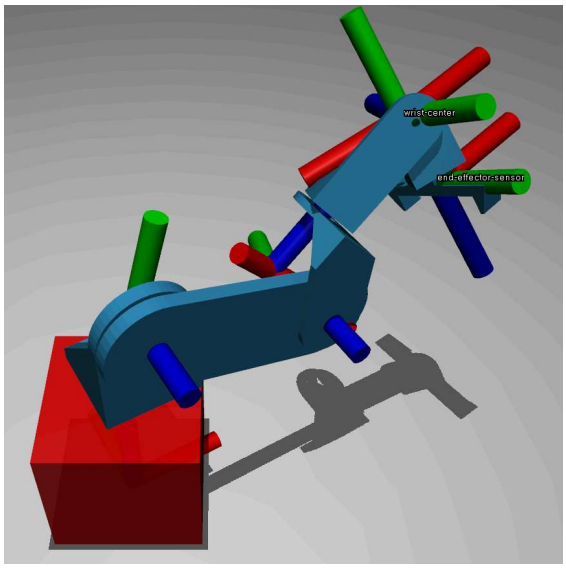


Figure 7: Inverse Kinematics Solution 1

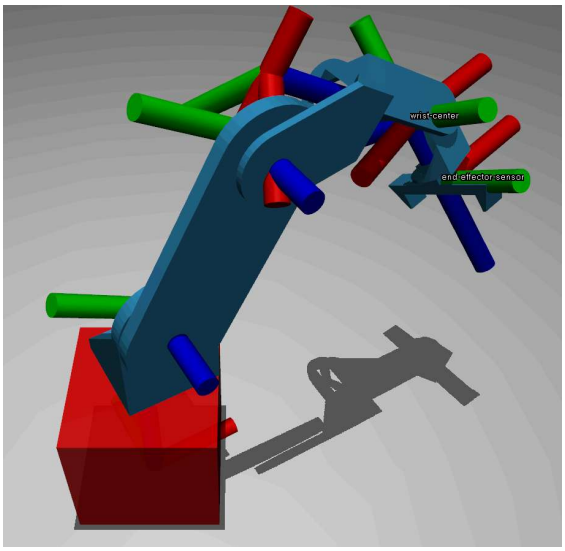


Figure 8: Inverse Kinematics Solution 2

Below is the verification.

In [14]:
```
for i, solution in enumerate(inverse_kinematics_solutions):
    data.qpos[:len(solution)] = solution
    mj.mj_forward(model, data)
    pose = get_pose(data, end_effector_body_id)
    if np.allclose(np.array(pose), np.array(given_transformation), atol=1e-3):
        print(f"Question 1 part B solution {i + 1} is correct.")
    else:
        print(f"Question 1 part B solution {i + 1} is not correct. ")
        print(f"The transformation as produced by mujoco is:")
        print(pose)
```

```
Question 1 part B solution 1 is correct.
Question 1 part B solution 2 is correct.
```

Question 2 Verification

Here we will check if the jacobian function works correctly by setting joint positions and velocities in mujoco, reading the twist, and comparing it to what `DHKinematics.jacobian` yields.

In [15]:
```
correctnesses = []
for i, rates in enumerate(rate_solutions):
    solution = inverse_kinematics_solutions[i]
    mj.mj_resetData(model, data)
    data.qpos[:len(solution)] = solution
    data.qvel[:len(rates)] = rates
    mj.mj_forward(model, data)
    body_lin_vel = data.sensordata[0 : 3]
    body_ang_vel = data.sensordata[3 : 6]
    measured_twist = np.concatenate([body_lin_vel, body_ang_vel])
    measured_twist[:3] = get_pose(data, end_effector_body_id).R @ measured_twist[:3] * 1000 # Convert linear units to mm
    measured_twist[3:] = get_pose(data, end_effector_body_id).R @ measured_twist[3:]
    correctnesses.append(np.allclose(measured_twist, question_2_twist, 1e-3))
    print(f"Measured Twist for joint solution {i + 1}: {np.round(measured_twist, 3).tolist()} is {"correct" if correctnesses[i] el
if np.all(np.array(correctnesses)):
    print("Question 2 is correct.")
```

```
Measured Twist for joint solution 1: [100.0, -200.0, -300.0, 2.0, -1.0, 0.5] is correct
Measured Twist for joint solution 2: [100.0, -200.0, -300.0, 2.0, -1.0, 0.5] is correct
Question 2 is correct.
```

Appendix A: Robot Arm Control

Developing foreward and inverse kinematics is really cool, so why not put it to use? Here the code developed for the rest of project is used as the backbone needed to plan a trajectory across space with minimal high level inputs as to how it should look. High level functions like `get_linear_path` and `get_three_point_arc_path` have been created which take in basic information and produce a series of poses that the robot needs to hit to make the motion. Spherical linear interpolation with `scipy.spatial.transform.Slerp` is used for the rotational interpolation. These functions are not good however, they are naive and if the inputs are not carefully chosen, will happily make the robot go through itself and snap 180° in an instant, which obviously cannot happen in real life. Below is an animation that could be imagined as the robot picking something up and moving it. Its carefully picked so as to be physically possible but I repeat, the code used to do this is *not* robust and does not take into consideration many important things. That said, below is an animation and the high level code that produced it. If this is run as a jupyter notebook, you will need to press enter to start the animation, this is to help get the viewer configured for viewing before it starts and can be disabled. The notebook will not finish until the viewer is closed.

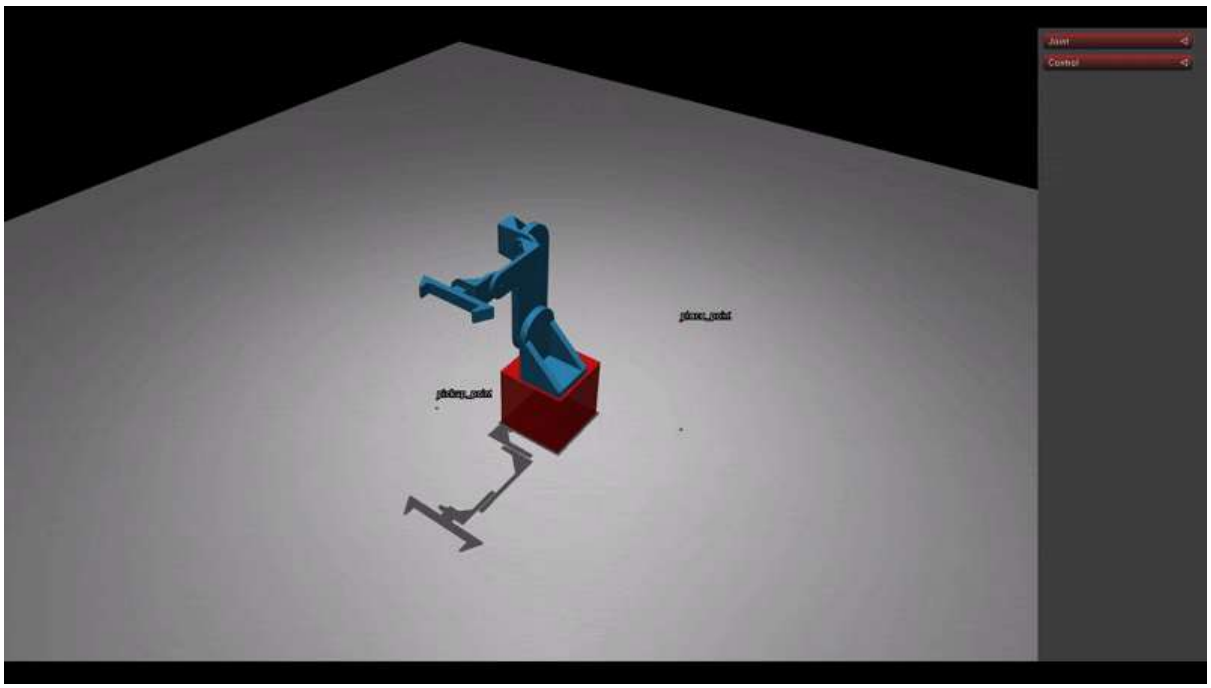


Figure 9: Animation of arm moving through trajectory (Does not work in pdf form, see resources/Animation.mp4 in project files)

In [16]:
```python
from kinematics.trajectory import *
# Define pickup and place poses
pickup_pose = SE3().CopyFrom([
    [1, 0, 0, 340 * np.cos(deg2rad(60))],
    [0, -1, 0, -340 * np.sin(deg2rad(60))],
    [0, 0, -1, 40],
    [0, 0, 0, 1]
], check=False)
place_pose = SE3().CopyFrom([
    [0, 1, 0, -340 * np.cos(deg2rad(60))],
    [0, 0, 1, 340 * np.sin(deg2rad(60))],
    [1, 0, 0, 400],
    [0, 0, 0, 1]
], check=False)

# Create trajectory
speed = 400
time_step = 0.02
pose_path = get_linear_path(home_pos, pickup_pose, speed, time_step)
pose_path += get_hold_path(pickup_pose, 0.375, time_step)
pose_path += get_directional_linear_path(pickup_pose, np.array([0,0,1]), 200, speed, time_step)
pose_path += get_three_point_arc_path(pose_path[-1], np.array([150, 200, 600]), place_pose, speed, time_step)

# Animate
animate_path(pose_path, mini_bot_kinematics, home_angles, model, data, time_step, prompt_for_enter=True)
```

```
Animation finished or viewer closed.
Viewer is closed. Exiting...
```