# PART_A

December 11, 2023

[2]:
```python
import sympy as sp
from sympy.matrices import Matrix
```

[3]:
```python
# Create class to calculate end effector data
class CalcEndEffector():
    def __init__(self, dh_params : list) -> None:

        # Useful attributes
        self.eye = Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]]) # Identity matrix
        self.zero_matrix = Matrix([[0, 0, 0], [0, 0, 0], [0, 0, 0]]) # 0 Matrix

        # Math for end effector
        self._original_dh_params = dh_params
        self._dh_params = self.make_dh_copy()
        self._transforms, self.rotation_matrices = self.
↪compute_transforms_rot_matrices()
        self._ee_transform = self.compute_ee_transform(self._transforms)
        self._jacobian = self.calc_jacobian(self._ee_transform)

    def make_dh_copy(self):
        '''
        Makes a copy of the original DH table with all the thetas being symbols␣
↪for
        computation sake

        This allows us to differentiate with a symbol and then plug in the␣
↪original theta
        to give us the correct answer
        '''

        # Init variables
        dh_copy = self._original_dh_params
        num_rows = len(self._original_dh_params)
        self._joint_symbols = sp.symbols('q1:{}'.format(num_rows+1))

        # Loop through
        for i in range(num_rows): dh_copy[i][3] = self._joint_symbols[i]
```

```
        # return copy
        return dh_copy

    # Compute the end effector matrix
    def compute_transforms_rot_matrices(self):
        # Compute transformation matrices for each joint
        transforms = [self._dh_transform(*params) for params in self._dh_params]

        # Extract rotational part of each transformation matrix
        rotation_matrices = [transform[:3, :3] for transform in transforms]
        return transforms, rotation_matrices

    # Helper function to calculate D-H transformation matrix
    def _dh_transform(self, a, alpha, d, theta, j_type):
        """
        Compute the Denavit-Hartenberg transformation matrix.

        Parameters:
        - a: Link length.
        - alpha: Link twist in radians.
        - d: Link offset.
        - theta: Joint angle in radians.
        - j_type: the type of joint, either revolute or prismatic 'P' or 'R'

        Returns:
        Homogeneous transformation matrix representing the transformation from␣
↪the
        current joint to the next joint in the Denavit-Hartenberg convention.
        The matrix is a 4x4 matrix representing both rotational and␣
↪translational components.
        """

        if j_type == "R": # If revolute
            return Matrix([
                [sp.cos(theta), -sp.sin(theta) * sp.cos(alpha), sp.sin(theta) *␣
↪sp.sin(alpha), a * sp.cos(theta)],
                [sp.sin(theta), sp.cos(theta) * sp.cos(alpha), -sp.cos(theta) *␣
↪sp.sin(alpha), a * sp.sin(theta)],
                [0, sp.sin(alpha), sp.cos(alpha), d],
                [0, 0, 0, 1]
            ])
        else: # If prismatic
            return Matrix([
                [sp.cos(alpha), -sp.sin(alpha), 0, a * sp.cos(alpha)],
                [sp.sin(alpha), sp.cos(alpha), 0, a * sp.sin(alpha)],
                [0, 0, 1, theta],
```

```python
                [0, 0, 0, 1]
            ])

    def compute_ee_transform(self, transforms):
        # Compute end-effector transformation matrix
        end_effector_transform = sp.simplify(sp.prod(transforms))
        return end_effector_transform

    def calc_jacobian(self, end_effector_transform):
        """
        Calculate the Jacobian matrix for the robot's end effector position.

        Parameters:
        - end_effector_transform: Homogeneous transformation matrix␣
↪representing the end effector's pose.

        Returns:
        Jacobian matrix (3xN) where N is the number of joints.
        The Jacobian maps joint velocities to the linear velocity of the end␣
↪effector in Cartesian space.
        """

        # Define end-effector position
        end_effector_position = end_effector_transform[:3, 3]
        self._ee_position = end_effector_position

        # Calculate Jacobian matrix
        num_joints = len(self._dh_params)
        jacobian_linear = sp.zeros(3, num_joints)
        jacobian_angular = sp.zeros(3, num_joints)

        # Calculate the linear velocities
        for i in range(num_joints):
            for j in range(3):

                # If the ORIGINAL DH table has a symbol at that value
                if isinstance(self._original_dh_params[i][3], sp.core.symbol.
↪Symbol):
                    jacobian_linear[j, i] = sp.diff(end_effector_position[j],␣
↪self._joint_symbols[i])

                # else we differentiate to the temp value and then sub in the␣
↪real value
                else:
                    x = sp.diff(end_effector_position[j], self.
↪_joint_symbols[i])
```

```
                    jacobian_linear[j, i] = x.subs(self._dh_params[i][3], self.
↪_joint_symbols[i])

        # Calculate the angular velocities
        for i in range(num_joints):

                # If joint is prismatic rotation is 0
                if self._original_dh_params[i][4] == "P":
                    for j in range(3):
                        jacobian_angular[j, i] = 0

                    # If first joint is prismatic need to include it in rotation
                    # sequence
                    if i == 0:
                        self.rotation_matrices.insert(0, self.zero_matrix)

                # if revolute
                else:
                    # if first column and is revolute
                    if i == 0:
                        jacobian_angular[0, i] = 0
                        jacobian_angular[1, i] = 0
                        jacobian_angular[2, i] = 1

                        # Need to add identity matrix as first matrix in
                        # sequence
                        self.rotation_matrices.insert(0, self.eye)

                    # Loop through each rotational matrix and multiply
                    # to get R0 of i-1 for end rotation
                    # for that joint frame relative to the base frame
                    else:
                        # Calculate the current rotational matrix
                        selected_matrices = self.rotation_matrices[:i]
                        m = sp.simplify(sp.prod(selected_matrices))

                        # Allocate the values
                        for j in range(3):
                            jacobian_angular[j, i] = m[j, 2]

        # Stitch matrices for correct usage
        jacobian = self._stitch_matrices(jacobian_linear, jacobian_angular,␣
↪num_joints)

        return jacobian

    def _stitch_matrices(self, j_lin, j_ang, num_joints):
```

```
        '''
            Because angular velocities are calculated as rows, we need to␣
↪convert to columns and then
            append those to the linear velocities for correct formatting of the␣
↪jacobian

            EXAMPLE: [0, 0, 1], [0, 0, 0], [0, 0, -1] needs to become:

            [0, 0, 0], [0, 0, 0], [1, 0, -1] essentially transposing and␣
↪putting back together

            Return: concatenated arrays
        '''
        # create list for new angular velos
        new_j_ang = sp.zeros(3, num_joints)

        # loop through all and reformat
        for i in range(3):
            # loop thru and get values
            for j in range(num_joints):
                new_j_ang[i , j] = j_ang[i , j]

        # with new format, we can now concat arrays
        jacobian = j_lin.col_join(new_j_ang)

        # return
        return jacobian

    def print_results(self):
        # Display the results
        print("End-effector transformation matrix:")
        display(self._ee_transform)
        print("\nEnd-effector position:")
        display(self._ee_position)
        print("\nJacobian matrix:")
        display(self._jacobian)
```

[4]:

```
##### Execute code here to test Jacobian for SCARA ######

# Set the number of joints
NUM_JOINTS = 4

# Define symbolic joint variables
joint_symbols = sp.symbols('q1:{}'.format(NUM_JOINTS+1))
a1, a2 = sp.symbols('a1 a2')
d3, d4 = sp.symbols('d3 d4')
```

```
'''
Parameters:
- a: Link length.
- alpha: Link twist in radians.
- d: Link offset.
- theta: Joint angle in radians.

NOTE: There is a fourth column for P or R which indicates prismatic or revolute␣
↪joints
This makes a difference when calculating transforms

NOTE: Make sure to simplify results, a lot of trig functions cancel out because␣
↪some equal 0
( Sympy didnt do a very good job, even with the .simplify() function )
'''

# List of dh parameters to make DH table
dh_parameters = [
    [a1,    0,      0,      joint_symbols[0], 'R'],
    [a2,    180,    0,      joint_symbols[1], 'R'],
    [0,     0,      d3,     0,                'P'],
    [0,     0,      d4,     joint_symbols[3], 'R']
]

# Calc the data
test = CalcEndEffector(dh_params=dh_parameters)
test.print_results()
```

```
End-effector transformation matrix:
Matrix([[-sin(q4)*sin(q1 + q2)*cos(180) + cos(q4)*cos(q1 + q2), -sin(q4)*cos(q1
+ q2) - sin(q1 + q2)*cos(180)*cos(q4), sin(180)*sin(q1 + q2), a1*cos(q1) +
a2*cos(q1 + q2) + d4*sin(180)*sin(q1 + q2) + q3*sin(180)*sin(q1 + q2)],
[sin(q4)*cos(180)*cos(q1 + q2) + sin(q1 + q2)*cos(q4), -sin(q4)*sin(q1 + q2) +
cos(180)*cos(q4)*cos(q1 + q2), -sin(180)*cos(q1 + q2), a1*sin(q1) + a2*sin(q1 +
q2) - d4*sin(180)*cos(q1 + q2) - q3*sin(180)*cos(q1 + q2)], [sin(180)*sin(q4),
sin(180)*cos(q4), cos(180), (d4 + q3)*cos(180)], [0, 0, 0, 1]])

End-effector position:
Matrix([[a1*cos(q1) + a2*cos(q1 + q2) + d4*sin(180)*sin(q1 + q2) +
q3*sin(180)*sin(q1 + q2)], [a1*sin(q1) + a2*sin(q1 + q2) - d4*sin(180)*cos(q1 +
q2) - q3*sin(180)*cos(q1 + q2)], [(d4 + q3)*cos(180)]])

Jacobian matrix:
Matrix([[-a1*sin(q1) - a2*sin(q1 + q2) + d4*sin(180)*cos(q1 + q2) +
q3*sin(180)*cos(q1 + q2), -a2*sin(q1 + q2) + d4*sin(180)*cos(q1 + q2) +
q3*sin(180)*cos(q1 + q2), sin(180)*sin(q1 + q2), 0], [a1*cos(q1) + a2*cos(q1 +
q2) + d4*sin(180)*sin(q1 + q2) + q3*sin(180)*sin(q1 + q2), a2*cos(q1 + q2) +
```

```
d4*sin(180)*sin(q1 + q2) + q3*sin(180)*sin(q1 + q2), -sin(180)*cos(q1 + q2), 0],
[0, 0, cos(180), 0], [0, 0, 0, sin(180)*sin(q1 + q2)], [0, 0, 0,
-sin(180)*cos(q1 + q2)], [1, 1, 0, cos(180)]])
```