Linear Algebra

1. a. *If a matrix H satisfies the property $H^TH = I$, show that the columns of H have length 1 and they are perpendicular to each other.*
Let
$$H = \begin{pmatrix} \uparrow & & \uparrow \\ h_1 & ... & h_n \\ \downarrow & & \downarrow \end{pmatrix}$$
Then
$$H^T = \begin{pmatrix} \leftarrow & h_1 & \rightarrow \\ & : & \\ \leftarrow & h_n & \rightarrow \end{pmatrix}$$
H is orthogonal, therefore $H^TH = I$.

So the $(i,j)^{th}$ entry of $H^TH = h_i \cdot h_j = \delta_{ij}$
If $i = j$ then $h_i \cdot h_i = |h_i|^2 = 1$ Therefore all columns are of unit length.
If $i \neq j$ then $h_i \cdot h_j = 0$ Therefore all column vectors are perpendicular with one another.

b. *Given two vectors v_1 and v_2 in R_3, show that their scalar product is invariant to rotations when the same rotation R is applied on both v_1 and v_2, i.e. their scalar product does not depend on the reference frame where v_1 and v_2 are defined.*
$v_1' = Rv_1$, $v_2' = Rv_2$
$v_1' \cdot v_2' = v_1'^T v_2' = (Rv_1)^T(Rv_2) = v_1^T R^T R v_2 = v_1 \cdot v_2$
$(Rv_1)_i(Rv_2)_i = R_{ij}R_{ik}v_{1j}v_{2k} = \delta_{jk}v_{1j}v_{2k} = v_{1j}v_{2j}$

c. *Show that the length of a vector v does not change when a rotation is applied to it.* $||v|| = ||Rv||$
If $\mathsf{v} = (v_1, v_2)$, then the length of $\mathsf{v} = |\mathsf{v}| = \sqrt{v_1^2 + v_2^2}$
If $v = (v_1, v_2)$ is rotated by
$$R = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix}$$
Then $v_1' = v_1 cos\theta - v_2 sin\theta$
$v_2' = v_1 sin\theta + v_2 cos\theta$
Length of vector is
$$\sqrt{v_1'^2 + v_2'^2} = \sqrt{v_1^2cos^2\theta + v_2^2sin^2\theta + v_1^2sin^2\theta + v_2^2cos^2\theta} = \sqrt{(sin^2\theta + cos^2\theta)(v_1^2 + v_2^2)}$$
And as $sin^2\theta + cos^2\theta = 1$
Length of transformed vector $|Rv| = |v|$

d. *Show that the distance between two points p_1 and p_2 does not depend on the reference frame in which they are defined: $||p_1 - p_2|| = ||Rp_1 - Rp_2||$*

Rodrigues' Formula states that $b = acos\theta + (u \times a)sin\theta + u(u \cdot a)(1 - cos\theta)$
To rotate from x_1 to x_2, y_1 to y_2 and z_1 to z_2 :

$$x_2 = \begin{bmatrix}1\\0\\0\end{bmatrix} cos\theta + \begin{bmatrix}0\\u_z\\-u_y\end{bmatrix} sin\theta + \begin{bmatrix}u_x\\u_y\\u_z\end{bmatrix} u_x(1 - cos\theta)$$

$$y_2 = \begin{bmatrix}0\\1\\0\end{bmatrix} cos\theta + \begin{bmatrix}-u_z\\0\\u_x\end{bmatrix} sin\theta + \begin{bmatrix}u_x\\u_y\\u_z\end{bmatrix} u_y(1 - cos\theta)$$

$$z_2 = \begin{bmatrix}0\\0\\1\end{bmatrix} cos\theta + \begin{bmatrix}u_y\\-u_x\\0\end{bmatrix} sin\theta + \begin{bmatrix}u_x\\u_y\\u_z\end{bmatrix} u_z(1 - cos\theta)$$

Mapping a vector between the two coordinate systems is $p_1 = Rp_2$ and as R is an orthonomal matrix with unit column lengths = 1, and $||x|| = 1, ||y|| = 1, ||z|| = 1$, and axes are orthogonal unit vectors, the transformed $||p_1 - p_2|| = ||Rp_1 - Rp_2||$ holds, as R merely rotates and does not scale.

2. a. *Provide a matricial example, i.e. a succession of 3 matrices along the 3 different axes, of gimbal lock for the YZY (Euler, extrinsic) and XYZ (Tait-Bryan, intrinsic) rotations. Why do we need to avoid gimbal lock when controlling robotic arms? How is this achieved?*

YZY (Euler, extrinsic) rotation: Two elemental rotations around the same axis with and elemental rotation around a different axis in between.

$$(\alpha, \beta, \gamma)_{yzy} = R_y(\alpha)R_z(\beta)R_y(\gamma) = \begin{pmatrix} cos\alpha & 0 & sin\alpha \\ 0 & 1 & 0 \\ -sin\alpha & 0 & cos\alpha \end{pmatrix} \begin{pmatrix} cos\beta & -sin\beta & 0 \\ sin\beta & cos\beta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cos\gamma & 0 & sin\gamma \\ 0 & 1 & 0 \\ -sin\gamma & 0 & cos\gamma \end{pmatrix}$$

Tait-Bryan Rotation (intrinsic): Rotation around 3 different axis. $\beta = 90^o$

$$(\alpha, \beta, \gamma)_{xyz} = R_x(\alpha)R_y(\beta)R_z(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha \\ 0 & sin\alpha & cos\alpha \end{pmatrix} \begin{pmatrix} cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{pmatrix} \begin{pmatrix} cos\gamma & -sin\gamma & 0 \\ sin\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We need to avoid gimbal lock when controlling robotic arms as one degree of freedom is lost, leaving an inability to distinguish between roll and yaw. This is because in a z-x-z rotation, any rotation θ around the z-axis has infinite Euler representations, as shown below. This may cause unpredictable motions. Having a fourth gimbal can get around this problem, but this gives an over constrained system. The only other way to avoid gimbal lock is to use Quaternion instead of Euler to represent rotations.
$R_z(\theta) \rightarrow (\alpha, 0^o, \gamma)_{zyz}$
α, γ can be any value such that $\alpha + \gamma = \theta$

b. *Show mathematically how to pass from Quaternion representation to rotation matrix representation.*

$$R = \begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_x 2q_y - 2q_z q_w & 2q_x q_z + 2q_y q_w \\ 2q_x q_y + 2q_z q_w & 1 - 2q_x^2 - 2q_z^2 & 2q_y q_z - 2q_x q_w \\ 2q_x q_z - 2q_y q_w & 2q_y q_z + 2q_x q_w & 1 - 2q_x^2 - 2q_y^2 \end{bmatrix}$$

c. *What rotation representation would you suggest to use in the following cases:*

- *Nano-robot with very limited memory storage*
  Axis angle
- *Nano-robot with very limited computational power*
  Quaternion
- *Iphone navigation system*
  Euler angle
- *Robotic arm with 6 DOF*
  Quaternion to avoid gimbal lock.

3. a. *Why do we work with decompositions? What are they useful for?*

Decompositions change the basis set s and in doing so, expose the structure of problems much more easily. Both Eigen decompositions and Singular Value Compositions help solve Systems of Linear Equations.

b. *Describe Singular Value Decomposition (SVD) and highlight pros and cons w.r.t. eigen decomposition.*

In Single Value Decomposition the scale factors are all non-negative. The inputs and outputs of vectors are orthonormal. The SVD, like the eigendecomposition, can be viewed as a succession of three transformations:

- A change of basis from the n-dimensional “input” space to an ndimensional “singular value” space.
- A scaling and a projection from the input singular value space to the output space.
- A transformation from the output singular value space to the output space.

However, the input and output spaces are of different dimensions.

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

Where: $\mathbf{U}$ is an orthonomal m by m matrix.
$\mathbf{\Sigma}$ is an m by m diagonal matrix of singular values which are non-negative.
$\mathbf{V}$ is an orthonomal n by n matrix.

SVD generlises eigen decompositions to non-square matrices where the dimensions of x and b are different, SVD also uses the orthonormal basis and thus are much more elegant and easy to work with.
Eigen decomposition is conceptually simpler, however only exists for the case where x and b are of the same dimension and matrix $\mathbf{A}$ is square.

4. *Complete the following tasks by filling in the python code templates in the packages ”cw1/cw1q4 srv” and ”cw1/cw1q4” to create services that perform representation transformations.*
a. *Fill in the template in package ”cw1/cw1q4 srv” with the appropriate request and response message types for each service.*

The appropriate request response message types for each services for quaternion to rodrigues transformations and quarternion to zyx transformations are shown in the code listing below. As the appropriate request response message types are 64 bit Floating points in (x, y, z) in quaternion to rodriguez conversion, and in (z, y, x) quaternion to zyx conversion.

```
geometry_msgs/Quaternion q
---
std msgs/Float64 x
std msgs/Float64 y
std msgs/Float64 z
```

Listing 1: quat2rodrigues.srv

```
geometry_msgs/Quaternion q
---
std msgs/Float64 z
std msgs/Float64 y
std msgs/Float64 x
```

Listing 2: quat2zyx.srv

b. *Fill in the template in package "cw1/cw1q4" to create a service that converts a quaternion representation to an euler angle representation $R_zR_yR_x$. Your request should contain the quaternion you need to convert, whereas your response should store the requested euler angles.*

To convert from Quaternion to Euler, the function $convert_quat2zyx$ was written. This function inputs a request service message that contains the quaternion needed to convert, and outputs a service response, in which the Euler angle is stored. The x axis rotation is calculated as shown below.

$$roll = arctan2(2(Q_w * Q_x + Q_y * Q_z), 1 - 2(Q_x * Q_x + Q_y * Q_y))$$

$$pitch = arctan(2 * (Q_w * Q_y - Q_z * Q_x))$$

As pitch only operates $< 90^o$, if the value of the argument is $> 90^o$, then 90^o is assigned.

$$yaw = arctan2(2 * (Q_w * Q_z + Q_x * Q_y), 1 - 2(Q_y * Q_y + Q_z * Q_z))$$

```python
def convert_quat2zyx(request):
    # TODO complete the function
    """Callback ROS service function to convert Quaternion to Euler z-y-x representation

    Args:
        request (quat2zyxRequest): cw1q4_srv service message, containing
        the quaternion you need to convert.

    Returns:
        quat2zyxResponse: cw1q4_srv service response, in which
        you store the requested euler angles
    """
    assert isinstance(request, quat2zyxRequest)

    # Your code starts here ----------------------------
    m = np.array(request.R.data).reshape(3,3)
    tr = np/trace(m)
    response = quat2zyxResponse()
    theta = np.arccos((tr - 1) / 2

    # roll (x-axis rotation)
    sinr_cosp = 2 * (request.q.w * request.q.x + request.q.y * request.q.z)
    cosr_cosp = 1 - 2 * (request.q.x * request.q.x + request.q.y * request.q.y)
    response.roll = np.arctan2(sinr_cosp, cosr_cosp)

    # pitch (y-axis rotation)
    sinp = 2 * (request.q.w * request.q.y - request.q.z * request.q.x)
    if (abs(sinp) >= 1):
        response.pitch = math.copysign(np.pi / 2, sinp) # use 90
   degrees if out of range
    else:
        response.pitch = np.arcsin2(sinp)
    # yaw (z-axis rotation)
    siny_cosp = 2 * (q.w * q.z + q.x * q.y)
    cosy_cosp = 1 - 2 * (q.y * q.y + q.z * q.z)
    response.yaw = np.arctan2(siny_cosp, cosy_cosp)
    # Your code ends here ------------------------------
    assert isinstance(response, quat2zyxResponse)
    return response
```

Listing 3: Quaternion to Euler angle conversion

c. *Fill in the template in package "cw1/cw1q4" to create a service that converts a quaternion representation to a rodrigues representation. Your request should contain the quaternion you need to convert, whereas your response should store the requested rodriguez representation*

To convert from Quaternion representation to Rodrigues, the function $convert_quat2rodrigues$ was written, this function inputs a request that contains the quaternion to be converted, and outputs the corresponding Rodrigues angle.

$$angle = 2 * acos(q_w)$$

$$x = \frac{qx}{sqrt(1 - q_w * q_w)}$$

$$x = \frac{qx}{sqrt(1 - q_w * q_w)}$$

$$x = \frac{qx}{sqrt(1 - q_w * q_w)}$$

If the inputted angle $\theta = 0$, the output Rodrigues represntation $(x, y, z, w) = (0, 0, 0, 1)$ In the code listing below, this avoids division by zero singularity. At angle $= 0^o$, $x, y, z = \text{inf}$ due to division by zero.

$$x = \frac{qx}{sqrt(1 - q_w * q_w)} = \text{inf}$$

$$y = \frac{qy}{sqrt(1 - q_w * q_w)} = \text{inf}$$

$$z = \frac{qz}{sqrt(1 - q_w * q_w)} = \text{inf}$$

The angle input = 180 would also produce a singularity.

$$q = 0 + ix + iy + kz$$
$$q_w = 0$$
$$angle = 2 * acos(q_w) = 2 * 90^o = 180^o$$
$$x = \frac{qx}{sqrt(1 - q_w * q_w)} = q_x$$
$$y = \frac{qy}{sqrt(1 - q_w * q_w)} = q_y$$
$$z = \frac{qz}{sqrt(1 - q_w * q_w)} = q_z$$

```python
def convert_quat2rodrigues(request):
    # TODO complete the function
    """Callback ROS service function to convert quaternion to
rodrigues representation
    Args:
        request (quat2rodriguesRequest): cw1q4_srv service message,
containing
        the quaternion you need to convert
    Returns:
        quat2rodriguesResponse: cw1q4_srv service response, in which
        you store the requested rodrigues representation
    """
    assert isinstance(request, quat2rodriguesRequest)
    # Your code starts here ----------------------------
    m = np.array(request.R.data).reshape(3,3)
    tr = np/trace(m)
    response = quat2rodriguesResponse()
    theta = np.arccos((tr - 1) / 2
    if theta == 0:   # avoid division by zero
        response.x = 0
        response.y = 0
        response.z = 0
        response.w = 1
    elif theta == np.pi or theta == -np.pi:
    # conversion is the same as x= qx / sqrt(1 - qw**2)=qx etc
        response.x = request.q.x
        response.y = request.q.y
        response.z = request.q.z

    else:
        response.angle = 2 * np.arccos(request.q.w)
        response.x = request.q.x /((1 - request.q.w ** 2 ) ** 0.5)
        response.y = request.q.y /((1 - request.q.w ** 2 ) ** 0.5)
        response.z = request.q.z /((1 - request.q.w ** 2 ) ** 0.5)
    # Your code ends here ------------------------------
    assert isinstance(response, quat2rodriguesResponse)
    return response
```

Listing 4: Quaternion to Rodrigues representation conversion function

Forward Kinematics

5. *Apply forward kinematics on the KUKA YouBot manipulator.*

a. *Identify the standard Denavit-Hartenberg parameters for the simplified dimensions of the Youbot shown in Figure 1. Your report should include a picture with the frames on the robot joints, as well as a brief explanation of how the parameters were derived.*

Denavit-Hartenberg (DH) parameters allow succinct description of the motion of a series of ridid joints, useful for forward and inverse kinematics. The DH parameters were defined by the frames shown in the youBot Arm Dimensions Figure. The z axis is placed along the axis of rotation for each joint. For the joint 0, the x axis is a free choice, however for later joints, the x axis is the common normal to the current and previous z axis, according to the right hand co-ordinate system. The corresponding y axis then completes the ritgh handed reference frame for each joint.

The DH parameter d is the depth along the previous joint's z axis, from the origin, to the common normal. Theta θ is the angle about the previous z axis to align its x_{i-1} with the new origin x_i. a is the distance along the rotated x axis from i-1 to i. Finally, α rotates about the new x axis, to put z in its desired orientation. In this case when some of the z axis are parallel, this means there's an infinite number of common normals, thus, the d is a free parameter. The calculated DH parameters are shown in Table 1, and the axis are shown in

Table 1: DH parameters for Kuka YouBot

i	a_i	α_i	d_i	θ_i
0	0	0	0	0
1	75	$\frac{\pi}{2}$	75	0
2	155	0	155	0
3	135	0	135	0
4	113	$\frac{\pi}{2}$	113	0

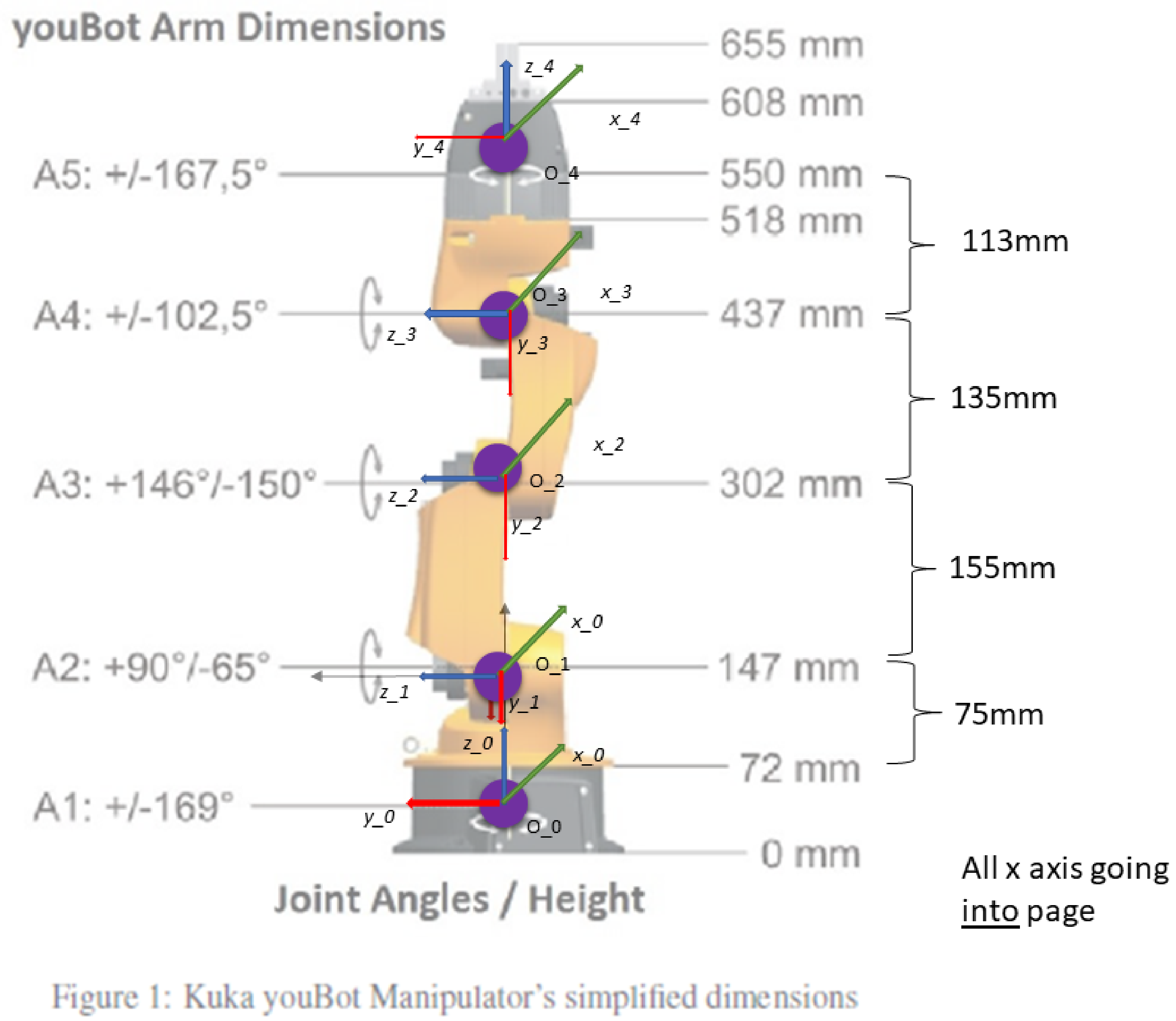

Figure 1: Kuka youBot Manipulator's Simplified Dimensions and axis frames

b. *Complete this task by filling in the 'cw1q5b node.py' code template, inside the the package "cw1/cw1q5". Write a ROS script to compute the forward kinematics using the standard Denavit-Hartenberg convention.*

The DH Parameters were input into the youbot_dh_parameters Python dictionary.

```
youbot_dh_parameters = {'a':[0, 75 , 155, 135, 113 ],
                        'alpha': [0, math.pi /2 , 0, 0, -math.pi /2],
                        'd' : [0, 75, 155, 135, 113],
                           'theta' : [0, 0, 0, 0, 0]}
```

Listing 5: DH Parameters dictionary

The rotmat2q function then converts the rotation matrix to quaternion.

```python
def rotmat2q(T):
    q = Quaternion()
    tr = np.trace(T)
    if tr == 4:
        q.w = 1.0
        q.x = 0.0
        q.y = 0.0
        q.z = 0.0
        return q
    angle = np.arccos((T[0, 0] + T[1, 1] + T[2, 2] - 1)/2)
    xr = T[2, 1] - T[1, 2]
    yr = T[0, 2] - T[2, 0]
    zr = T[1, 0] - T[0, 1]
    x = xr/np.sqrt(np.power(xr, 2) + np.power(yr, 2) + np.power(zr, 2))
    y = yr/np.sqrt(np.power(xr, 2) + np.power(yr, 2) + np.power(zr, 2))
    z = zr/np.sqrt(np.power(xr, 2) + np.power(yr, 2) + np.power(zr, 2))
    q.w = np.cos(angle/2)
    q.x = x * np.sin(angle/2)
    q.y = y * np.sin(angle/2)
    q.z = z * np.sin(angle/2)
    return q
```

Listing 6: Rotation matrix to Quaternion conversion.

The function 'standard_dh' computes the homogenous 4x4 transformation matrix T_i based on the four standard DH parameters associated with link i and joint i. The T array outputted is calculated by:

$$T = \begin{bmatrix} cos(\theta) & -sin(\theta)cos(\alpha) & sin(\theta)sin(\alpha) & a * cos(\theta) \\ sin(\theta) & cos(\theta)cos(\alpha) & cos(\theta)sin(\alpha) & a * sin(\theta) \\ 0 & sin(\alpha) & cos(\alpha) & d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```python
def standard_dh(a, alpha, d, theta):
    # TODO complete the function

    """This function computes the homogeneous 4x4 transformation matrix T_i based
    on the four standard DH parameters associated with link i and joint i.

    Args:
        a ([int, float]): Link Length. The distance along x_i ( the
    common normal) between z_{i-1} and z_i
        alpha ([int, float]): Link twist. The angle between z_{i-1}
    and z_i around x_i.
        d ([int, float]): Link Offset. The distance along z_{i-1}
    between x_{i-1} and x_i.
        theta ([int, float]): Joint angle. The angle between x_{i-1}
    and x_i around z_{i-1}

    Returns:
        [np.ndarray]: the 4x4 transformation matrix T_i describing  a
    coordinate
        transformation from the concurrent coordinate system i to the
    previous coordinate system i-1
    """
    assert isinstance(a, (int, float)), "wrong input type for a"
    assert isinstance(alpha, (int, float)), "wrong input type for =
alpha"
    assert isinstance(d, (int, float)), "wrong input type for d"
    assert isinstance(theta, (int, float)), "wrong input type for
theta"
    A = np.zeros((4, 4))

    # your code starts here -----------------------------
    A[0, 0] = np.cos(theta)
    A[0, 1] = -np.sin(theta)*np.cos(alpha)
    A[0, 2] = np.sin(theta)*np.sin(alpha)
    A[0, 3] = a * np.cos(theta)

    A[1, 0] = np.sin(theta)
    A[1, 1] = np.cos(theta)*np.cos(alpha)
    A[1, 2] = -np.cos(theta)*np.sin(alpha)
    A[1, 3] = a * np.sin(theta)

    A[2, 1] = np.sin(alpha)
    A[2, 2] = np.cos(alpha)
    A[2, 3] = d

    A[3, 3] = 1.0

    # your code ends here ------------------------------
    assert isinstance(A, np.ndarray), "Output wasn't of type ndarray"
    assert A.shape == (4,4), "Output had wrong dimensions"
    return A
```

The forward kinematics were then calculated in the function forward_kinematics. This function solves the forward kinematics trasnformation model by multiplying frame transformations up to the specified frame number 'up_to_frame' by computing transformation T01 from joint 0 to 1, T12 from joint 1 to 2 etc. and multiplying the resulting matrices.

```python
def forward_kinematics(dh_dict, joints_readings, up_to_joint=5):
    # TODO complete the function
    """This function solves the forward kinematics by multiplying frame
    transformations up until a specified frame number. The frame transformations
     used in the computation are derived from the dh parameters and joint_readings.
    Args:
        dh_dict (dict): A dictionary containing the dh parameters describing the robot.
        joints_readings (list): the state of the robot joints. For youbot those are revolute.
        up_to_joint (int, optional): Specify up to what frame you want to compute forward kinematicks. Defaults to 5.
    Returns:
        np.ndarray: A 4x4 homogeneous tranformation matrix describing the pose of frame_{up_to_joint} w.r.t the base of the robot.
    """
    assert isinstance(dh_dict, dict)
    assert isinstance(joints_readings, list)
    assert isinstance(up_to_joint, int)
    assert up_to_joint>=0
    assert up_to_joint<=len(dh_dict['a'])

    T = np.identity(4)
    # your code starts here ------------------------------
    a = youbot_dh_parameters['a']
    alpha = youbot_dh_parameters['alpha']
    d = youbot_dh_parameters['d']
    theta = youbot_dh_parameters['theta']
    # add for loop computing up to 'up_to_joint' joint number
    for i in range(up_to_joint):
        T *= standard_dh(a[i], alpha[i], d[i], theta[i] + joints_readings[i]) )
    # your code ends here ------------------------------
    assert isinstance(T, np.ndarray), "Output wasn't of type ndarray"
    assert T.shape == (4,4), "Output had wrong dimensions"
    return T
```

The function fkine_wrapper integrates robotics code with ROS. The function gets joint angle from rostopic and broadcasts joint data to tf2.

```python
def fkine_wrapper(joint_msg, br):
    # TODO complete the function
    """This function integrates your robotics code with ROS and is responsible
        to listen to the topic where joint states are published. Based on this,
        compute forward kinematics and publish it.
        In more detail this function should perform the following actions:
        - get joint angles from the rostopic that publishes joint data
        - Publish a set of transformations relating the frame 'base_link' and
            each frame on the arm 'arm5b_link_i' where i is the frame, using
            tf messages.

    Args:
        joint_msg (JointState): ros msg containing the joint states of the robot
        br (TransformBroadcaster): a tf broadcaster
    """
    assert isinstance(joint_msg, JointState), "Node must subscribe to a topic where JointState messages are published"
    # your code starts here ------------------------------

    transform = TransformStamped()
    for i in range(up_to_joint=4):
        #Call forward kinematics
        T = forward_kinematics(youbot_dh_parameters, list(joint_msg.position), i)

        #define transform from joint frame 0 to new joint frame 1 using tf2 broadcaster
        transform.header.stamp = rospy.Time.now()
        #define the transform header frame (parent)
        transform.header.frame_id = 'world'
        #define tranform child frame
        transform.child.frame_id = name_link[i]
        #populate the transform field
        transform.transform.translation.x = T[0, 3]
        transform.transform.translation.y = T[1, 3]
        transform.transform.translation.z = T[2, 3] #Multiple times?
        transform.transform.rotation = rotmat2q(T)

        #Broadcast the transform to tf2
        br.sendTransform(transform)
    return None
    # your code ends here ------------------------------
```

Listing 7: fkine_wrapper function

c. *Identify the standard D-H parameters following the complete Youbot dimensions found in the 'robot description/youbot description/urdf/youbot arm/arm.urdf.xacro' URDF file. Based on the URDF description of each joint, you should be able to come up with a new set of DH parameters, as well as the joint offsets that the xacro file incorporates. Your report should include a brief explanation of how the parameters were* *derived.*

The DH parameters were derived from the arm.urdf.xacro file, in which the joints are labelled as at positions:
Joint 0: $(0, 0, 0)$
Joint 1: $(0.024, 0, 0.096)$
Joint 2: $(0.033, 0, 0.019)$
Joint 3: $(0, 0, 155)$
Joint 4: $(0, 0, 135)$
Joint 5: $(0.002, 0, 0.130)$
Thus, the DH parameters were defined to be:

Table 2: DH parameters from URDF file for Kuka YouBot arm

i	a_i	α_i	d_i	θ_i
0	0	0	0	0
1	78	0	98	π
2	222	π	78	π
3	192	π	138	π
4	118	π	120	π
5	96	0	5	0

For example, d was defined by calculating the length of the links between the normals of the joints, a was the link length.
Joints 1 to 6 are revolute, while joint 0 is fixed, thus DH parameters for joint 0 are 0.
Joint offsets were calculated by determining the distance between the joints, and the link length, then using Pythagorean theorem to determine the offset.

Table 3: Joint Offsets determined from URDF file.

i	Offset (mm)
0	0
1	59
2	-65
3	-133
4	-116
5	-95

Complete this task by filling in the 'cw1q5d node.py' code template, inside the the package "cw1/cw1q5". Write a ROS script to compute the forward kinematics based on the URDF description.

```python
def fkine_wrapper(joint_msg, br):
    # TODO complete the function
    """This function integrates your robotics code with ROS and is responsible
        to listen to the topic where joint states are published. Based on this,
        compute forward kinematics and publish it.

        In more detail this function should perform the following actions:
        - get joint angles from the rostopic that publishes joint data
        - Publish a set of transformations relating the frame 'base_link' and
            each frame on the arm 'arm5d_link_i' where i is the frame, using
            tf messages.

    Args:
        joint_msg (JointState): ros msg containing the joint states of the robot
        br (TransformBroadcaster): a tf broadcaster
    """
    assert isinstance(joint_msg, JointState), "Node must subscribe to a topic where JointState messages are published"

    # your code starts here ------------------------------
    #depending on the dh parameters you may need to change the sign of some angles here
    transform = TransformStamped()

    #This loop should iterate through all joints. In this simplified example, we are only using joint frame 1.
    for i in range(1):


        T = forward_kinematics(youbot_dh_parameters['a'][i], youbot_dh_parameters['alpha'][i], youbot_dh_parameters['d'][i], youbot_dh_parameters['theta'][i] + joint_msg.position[i])

        #Here we define the transform from joint frame 0 to joint frame 1 using the tf2 broadcaster.
        #define the transfrom timestamp
        transform.header.stamp = rospy.Time.now()
        #define the transfrom header frame (parent frame)
        transform.header.frame_id = 'world'
        #define the transfrom child frame
        transform.child_frame_id = name_link[i]
        #populate the transform field. It consists of translation and rotation.
        transform.transform.translation.x = T[0, 3]
        transform.transform.translation.y = T[1, 3]
        transform.transform.translation.z = T[2, 3]
        transform.transform.rotation = rotmat2q(T)

        #broadcast the transform to tf2
        br.sendTransform(transform)


    # your code ends here ------------------------------
```

Listing 8: Q5d fkine_wrapper function.

In the listing below, the subscriber is initialised to the topic that publishes joint angles ('/joint_states', and is configured to have the fkine_wrapper from above as a callback , and the broadcaster is passed as an additional argument to the callback.

```
def main():
    rospy.init_node('forward_kinematic_node')
    #Initialize your tf broadcaster.
    br = TransformBroadcaster()

    # TODO: Initialize a subscriber to the topic that
    # publishes the joint angles, configure it to have fkine_wrapper
    # as callback and pass the broadcaster as an additional argument
   to the callback

    # your code starts here ------------------------------
    sub = rospy.Subscriber('/joint_states', JointState, fkine_wrapper,
    br)
    # your code ends here ----------------------

    rospy.spin()
```

Listing 9: Question 5d subscriber initialisation.

References

[1] M. Baker, “Maths - Quaternion to AxisAngle -.” https://www.euclideanspace.com/maths/geometry/ rotations/conversions/quaternionToAngle/index.htm (accessed Nov. 01, 2021).
[2] “Conversion between quaternions and Euler angles - Wikipedia.” https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles (accessed Nov. 01, 2021).