Assignment_1 Report

Sami Munir - sm2246

September 26, 2024

1 Component I - *Validating Rotations*

1.1 function *check_SOn()*

1.1.1 Objective

This function checks if a given matrix *m* is an element of the special orthogonal group $SO(n)$ within a given *epsilon* tolerance.

1.1.2 Key Conditions

Some key conditions to verify that *m* is in $SO(n)$ include the matrix must be a square, the matrix must be orthogonal, and the determinant of the matrix should be close to 1.

1.1.3 Logic & Code

- Check if the matrix is a square by examining its shape.

```
if m.shape[0] != m.shape[1]:
    return False
```

- Check if the matrix multiplied by its transpose results in the identity matrix.

```
identity_matrix = np.eye(m.shape[0])
orthogonality_check = np.allclose(np.dot(m.T, m), identity_matrix,
```

```
    atol = epsilon)
```

- Ensure that the determinant of matrix is close to 1.

```
determinant_check = np.isclose(np.linalg.det(m), 1.0,
    atol = epsilon)
```

1.1.4 Important Points

This function utilizes *np.allclose()* to compare matrices within a tolerance *epsilon*, allowing for numerical precision errors.

1.2 function *check_quaternion()*

1.2.1 Objective

This function checks if a given vector v is a valid unit quaternion within a given *epsilon* tolerance.

1.2.2 Key Conditions

Some key conditions to verify that the vector is a valid quaternion include confirming that it is 4-dimensional, and that it must have a magnitude of 1 (within the *epsilon* tolerance), ensuring it lies on the 3-sphere S^3.

1.2.3 Logic & Code

- Magnitude check: It calculates the square of the magnitude of the vector and verifies if it is close to 1 (with *epsilon* tolerance.

```
if len(v) != 4:
    return False

magnitude_squared = np.sum(np.square(v))

return np.abs(magnitude_squared - 1) < epsilon
```

1.3 function *check_SEn()*

1.3.1 Objective

This function checks if a given matrix m belongs to the special Euclidean group $SE(n)$, which represents rigid body transformations.

1.3.2 Key Conditions

Some key conditions to check are that the matrix must be a square with a size of $(n+1) \times (n+1)$, where $n = 2$ or $n = 3$. It also essential to check that the top-left $n \times n$ sub-matrix must be an element of $SO(n)$ (i.e. a valid rotation matrix). The last check is to confirm that the last row of m should be of the form $[0, 0, ..., 1]$.

1.3.3 Logic & Code

- Check if the matrix is a square by examining its shape.

```
n = m.shape[0] - 1

if m.shape[0] != m.shape[1] or (n not in [2, 3]):
    return False
```

- Rotation check: It extracts the top-left sub-matrix and verifies if it is in $SO(n)$.

```
rotation_matrix = m[:n, :n]

if not check_SOn(rotation_matrix, epsilon):
    return False
```

- Last row check: It verifies if the last row is composed of zeros, with a 1 in the last position.

```
last_row_check = np.allclose(m[n, :], np.append(np.zeros(n), 1),
    atol = epsilon)
```

1.3.4 Important Points

- $SE(n)$ matrices represent both rotation and translation, where the top-left part is the rotation and the last row encodes translation in homogeneous coordinates.
- The use of the *epsilon* parameter allows for flexibility in handling floating-point precision, which is especially important in numerical computations.

All functions take an *epsilon* parameter to handle floating-point inaccuracies, making the checks robust against minor numerical errors. These functions are critical in verifying properties like orthogonality, unit determinant, and magnitude, all of which are fundamental in transformations and rotations. The *check_SEn()* function utilizes *check_SOn()* as a helper to validate the rotation part of the transformation matrix, demonstrating module-based in design.

2 Component II - *Random Uniform Rotations*

2.1 function *random_rotation_matrix()*

2.1.1 Purpose

This function generates a random rotation matrix $RinSO(3)$, with the option of using a naive or more sophisticated method based on the *naive* boolean flag.

2.1.2 naive: True Implementation

- Random Euler angles are generated for yaw, pitch, and roll. Each angle corresponds to a rotation around one of the three axes (z, y, and x respectively).
- Rotation matrices for yaw, pitch, and roll are created and then multiplied to form the final rotation matrix.

- The naive method generates random rotations by first choosing random angles in radians between 0 and 2π.
- The final random rotation matrix is obtained by multiplying the yaw, pitch, and roll matrices: $R_R = R_z \cdot R_y \cdot R_x$

```
yaw = np.random.uniform(0, 2 * np.pi) # rotation around z-axis
pitch = np.random.uniform(0, 2 * np.pi) # rotation around y-axis
roll = np.random.uniform(0, 2 * np.pi) # rotation around x-axis

# Rotation matrix around z-axis (yaw)
R_z = np.array([
    [np.cos(yaw), -np.sin(yaw), 0],
    [np.sin(yaw), np.cos(yaw), 0],
    [0, 0, 1]
])

# Rotation matrix around y-axis (pitch)
R_y = np.array([
    [np.cos(pitch), 0, np.sin(pitch)],
    [0, 1, 0],
    [-np.sin(pitch), 0, np.cos(pitch)]
])

# Rotation matrix around x-axis (roll)
R_x = np.array([
    [1, 0, 0],
    [0, np.cos(roll), -np.sin(roll)],
    [0, np.sin(roll), np.cos(roll)]
])

random_rotation_matrix = np.dot(R_z, np.dot(R_y, R_x))

return random_rotation_matrix
```

2.1.3 naive: False Implementation

- Instead of generating Euler angles, the function generates a random quaternion.

- A quaternion (q_0, q_1, q_2, q_3) is generated using uniform random values u_1, u_2, u_3.
- The quaternion is then converted to a rotation matrix using the function *utils.quaternion_to_rotation_matrix()*.
- This method generates uniform random rotations by utilizing a more efficient and accurate method based on quaternions, ensuring uniform sampling from $SO(3)$.

```
    # Generate a random quaternion
    u1 = np.random.uniform(0, 1)
    u2 = np.random.uniform(0, 2 * np.pi)
    u3 = np.random.uniform(0, 2 * np.pi)

    # Convert to quaternion (q0, q1, q2, q3)
    q0 = np.sqrt(1 - u1) * np.cos(u2)
    q1 = np.sqrt(1 - u1) * np.sin(u2)
    q2 = np.sqrt(u1) * np.cos(u3)
    q3 = np.sqrt(u1) * np.sin(u3)

    # Convert quaternion to a rotation matrix
    random_rotation_matrix = quaternion_to_rotation_matrix(np.array([q0, q1, c
```

```
def quaternion_to_rotation_matrix(q: np.array) -> np.ndarray:
    # Qaternion elements
    q0, q1, q2, q3 = q

    # Rotation matrix from quaternion
    rotation_matrix = np.array([
        [1 - 2 * (q2 ** 2 + q3 ** 2), 2 * (q1 * q2 - q0 * q3), 2 * (q1 * q3 + q0 *
        [2 * (q1 * q2 + q0 * q3), 1 - 2 * (q1 ** 2 + q3 ** 2), 2 * (q2 * q3 - q0 *
        [2 * (q1 * q3 - q0 * q2), 2 * (q2 * q3 + q0 * q1), 1 - 2 * (q1 ** 2 + q2 *
    ])

    return rotation_matrix
```

2.1.4 Validation & Visualization

- After generating the rotation matrix, the function calls *check_SOn()* to

ensure that the matrix belongs to $SO(3)$. This ensures the result is a valid rotation matrix.

- The function also calls *visualize_rotation()* to visualize the rotation in 3D space. One vector represents, the original, the other represents the translation, and the other represents the rotation.

2.2 function *utils.visualize_rotation()*

This function visualizes the effect of a rotation matrix m on vectors in 3D space. Specifically, it shows how the rotation affects two vectors: one at the "north pole" of a unit sphere and another slightly displaced from the pole.

Two vectors are defined in 3D space:

- v0: a point on the positive z-axis.
- v1: a point slightly displaced from the north pole in the y-direction. This vector is used to visualize how a small perturbation from the north pole moves under rotation.

The rotation matrix m is applied to both vectors using matrix multiplication.

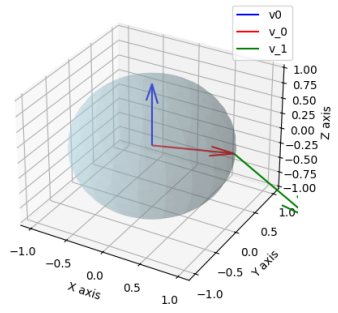

Figure 1: Random rotation matrix applied on vector (1).

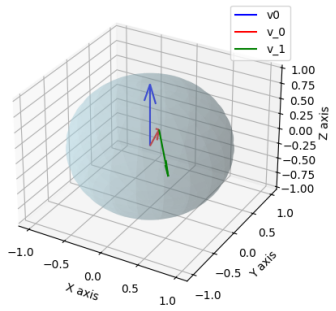

Figure 2: Random rotation matrix applied on vector (2).

A sphere is created using mesh grids generated by parameterizing the surface of the sphere with angles u and v. This helps to easily visualize and understand the cause and effect of the rotation matrix m on vectors v0 and v1.

3 Component III - *Rigid Body in Motion*

3.1 function *interpolate_rigid_body()*

3.1.1 Purpose

This function performs linear interpolation between the start pose and the goal pose of the robot. It computes a sequence of poses (position & orientation) that smoothly transitions from the start to the goal.

3.1.2 Logic & Code

- The function uses linear interpolation with 100 steps to generate intermediate positions and orientations between the start and goal poses.
- The result is a path that consists of 100 poses that smoothly transition from the initial pose to the goal pose.

3.1.3 Important Points

- Linear interpolation works well for simple, direct motion without complex dynamics or changes in velocity.
- The *steps* variable is set to 100, giving a fine-grained resolution of the robot's path.

3.2 function *forward_propagate_rigid_body()*

3.2.1 Purpose

This function computes the path of the rigid body robot based on a sequence of velocity commands and durations. It simulates the forward propagation of the robot through its environment.

3.2.2 Logic & Code

- For each velocity and duration in the plan, the function updates the robot's position and orientation over small time steps (assuming 100 updates per second).
- The x and y coordinates are updated based on the velocity in the x and y directions.
- The orientation theta is updated based on the angular velocity $v\theta$.
- The poses are continuously updated and appended to the result list, creating a detailed path which can later be visualized using function *visualize_path()*.

3.2.3 Important Points

- The time step of 0.1 seconds (for 100 Hz) is small enough to ensure smooth and continuous motion of the robot.
- The method accounts for velocity, allowing more dynamic motion compared to linear interpolation.
- Each pose is computed based on the prior pose, making it a recursive motion simulation.

3.3 function *visualize_path()*

3.3.1 Purpose

This function visualizes the path of the robot in a 20×20 environment using Maplotlib. It also animates the robot's movement along the path.

3.3.2 Logic & Code

- The plot is set up with x and y limits from [-10, 10], representing a 20×20 environment.
- The path is visualized as a blue line on the plot, showing the robot's trajectory.
- The robot itself is visualized as a red rectangle with dimensions 0.5×0.3.
- The function uses Matplotlib's FuncAnimation to animate the robot's movement along the path. The robot's position and orientation are updated at each frame.
- The update() function is called for each frame to change the robot's position and rotation.

3.3.3 Important Points

Linear interpolation provides a direct, constant speed path between the start and goal. It is simple and fast but does not account for dynamic changes in velocity.

Forward propagation is more flexible and realistic, allowing velocity changes and varying durations, giving a more dynamic and complex trajectory. This method better simulates real-world robot motion where velocity is not constant.

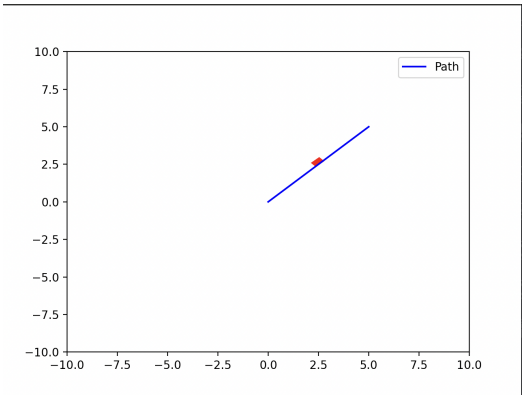

Figure 3: Interpolated Path Behavior

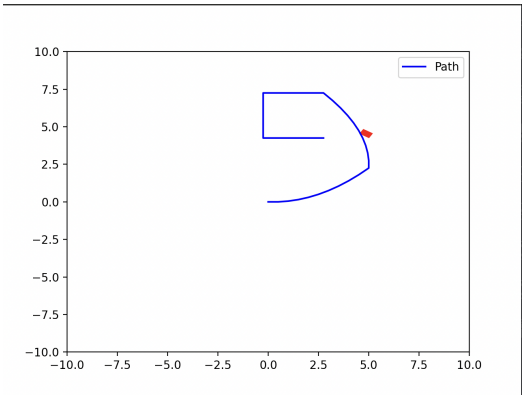

Figure 4: Forward Propagated Path Behavior

4 Component IV - *Movement of an Arm*

4.1 function *interpolate_arm()*

4.1.1 Purpose

This function generates a linear interpolation between two configurations of a two-link arm in 2D space. The configurations represent the angles of the two joints of the arm.

4.1.2 Logic & Code

- The function computes the difference between the start and goal configurations (delta_theta) to determine how much the angles change.
- Based on this change, it computes a set number of steps (proportional to the magnitude of the angle difference) and linearly interpolates between the start and goal angles using *np.linspace()*.
- Each step is a new arm configuration that smoothly transitions between the start and goal.

4.1.3 Important Points

Linear interpolation ensures a smooth transition between configurations, but it assumes constant motion for both joints. This may not accurately represent real-world robotic arm motion, which can have varying speeds and velocities for different joints.

The number of steps is determined based on the angular difference between the start and goal configurations, ensuring that the interpolation adapts to the amount of motion required.

4.2 function *forward_propagate_arm()*

4.2.1 Purpose

This function simulates the forward propagation of the arm's motion using a set of velocity commands and durations, producing a series of arm configurations.

4.2.2 Logic & Code

- The function starts with the initial pose and iterates through the velocity plan.
- For each velocity and duration, it updates the joint angles proportionally based on the velocity and time.
- This method accounts for velocity, allowing the arm to follow a more dynamic and realistic trajectory, where joints can move at different rates and over varying time intervals.

4.2.3 Important Points

- Forward propagation allows for non-uniform joint motion, meaning the robot's joints can move at different speeds and over different time durations, resulting in a more flexible and realistic path.
- This method is more accurate in simulating real-world robotics, where joint velocities and durations vary, and the arm's movement is nonlinear.
- The plan provides flexibility to handle complex movements, such as accelerations and decelerations.

4.3 function *visualize_arm_path()*

4.3.1 Purpose

This function visualizes the movement of the two-link robotic arm based on the provided path of joint angles. The visualization shows the arm's motion in 2D space, where each link of the arm is represented as a segment connected by joints.

4.3.2 Key Details

- The arm is modeled as two segments: the first link has a length of 2 meters, and the second link has a length of 1.5 meters.
- The base of the arm is fixed at the origin (0, 0).

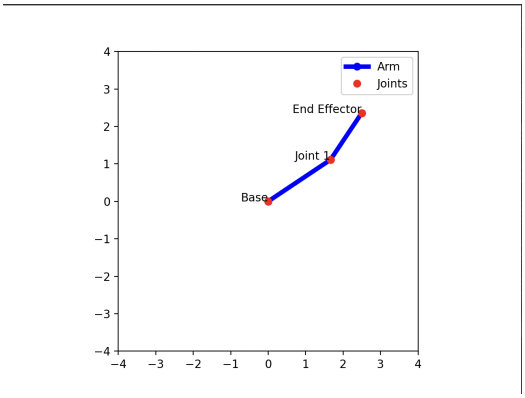

Figure 5: Interpolated Path Behavior

- The arm's joint angles are used to compute the positions of the first and second joints (end-effector), and these positions are used to draw the arm.

4.3.3 Logic & Code

- The *get_arm_position()* function calculates the Cartesian coordinates of the arm's joints based on the angles θ_0 and θ_1.
- The function sets up an animated plot using Matplotlib, where the arm's position is updated in each frame.
- The arm's motion is shown as an animation, with a red line representing the joints and links of the arm. The labels "Base", "Joint 1", and "End Effector" help in identifying the components of the two-link robotic arm system.

4.3.4 Important Points

- The choice of link lengths is important in determining how far the arm can reach and how the joints interact during movement.
- The *interval* value can be modified to change the speed of the animation.

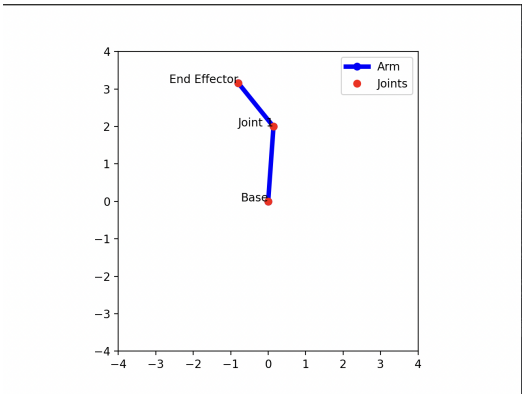

Figure 6: Forward Propagated Path Behavior

5 References

1. https://docs.python.org/3/
2. Effective Sampling and Distance Metrics for 3D Rigid Body Path Planning - James J. Kuffner
3. Fast Random Rotation Matrices - James Arvo
4. Rotations.pdf
5. Robot Motion.pdf
6. https://matplotlib.org/stable/index.html
7. https://numpy.org/doc/
8. https://matplotlib.org/stable/api/_as_gen/matplotlib.animation.FuncAnimation.html
9. https://youtu.be/bKd2lPjl92c?si=FiLiFEax229kO3xJ
10. https://youtu.be/7RgoHTMbp4A?si=89FQe7A0JGRCsHoc
11. Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace and Virtual Reality - J.B. Kuipers