Robotic Systems Engineering
Coursework 1: Linear Algebra and Forward Kinematics

Kajan Subakannan
kajan.subakannan.21@ucl.ac.uk

December 8th, 2024

Section 1: Jacobian and Inverse Kinematics

1. A 4R planar manipulator is a manipulator consisting of 4 revolution joints, thus giving it 4 degrees of freedom in the plane it is constrained to (for instance $x - y$. The pose x_e of the end effector generally includes a position in the plane (x, y), and an orientation θ_e, which is the angle of the end effector in the plane. The total constraints described are 3, which are 2 position constraints x and y, and an orientation constraint θ_e. As the manipulator has 4 DOF, there is one remaining unconstrained DOF which allows for 2 possible solutions to the inverse kinematics problem.

   The 2 configurations are elbow-up and elbow-down, which are the 2 different ways the manipulator's joints can be positioned whilst still achieving the same end-effector pose. In each of these positions, the elbow formed by the second joint bends upwards or downwards respectively, relative to the base of the manipulator. There are special cases such as if x_e lies on the boundary of the reachable workspace of the manipulator If this is the case, there may only be one solution depending on the orientation and the singularity condition.

2. Key fctors to consider are:

   - Joint Limit Avoidance
     Ensure that the selected solution avoids positions near the robot's joint limits to prevent mechanical stress and ensure a greater range of motion for future movements.
   - Energy Efficiency
     Choose a solution that minimizes the energy consumption, often approximated by minimizing joint movements or the norm of joint velocities.
   - Smoothness of Motion
     Prefer solutions that minimize sudden changes in joint angles, especially if the robot is following a trajectory, to ensure smooth motion and reduce wear and tear on actuators.
   - Collision Avoidance
     Internal collisions between robot links should be avoided therefore solutions that keep the robot far from self-collision are preferred.
   - Computational Simplicity
     For real-time applications, choose a solution that is computationally efficient for optimal responsiveness
   - Task-Specific Constraints
     Consider any task-specific requirements, such as maintaining a certain orientation, avoiding certain postures (eg to prevent payload instability), or best aligning the end-effector for the task.

- Redundancy Exploitation
  If the robot has redundant degrees of freedom (e.g., a 7-DOF arm), use the redundancy to optimize additional criteria, such as avoiding joint limits, minimizing joint velocity, or positioning the robot for easier subsequent tasks.

3. The output of atan2(y, x) differs from atan(y/x) because atan2 considers both y and x to determine the correct quadrant, while atan(y/x) only uses their ratio. Key differences:

   Quadrant Distinction:

   - atan2(y, x) covers all four quadrants, returning values in the range $[-\pi, \pi]$, ensuring the angle reflects the signs of both y and x.
   - atan(y/x) identifies angles only in $(-\pi/2, \pi/2)$, leading to quadrant ambiguity.

   Division by Zero:

   - atan2(y, x) handles $x = 0$, returning $\pm\pi/2$ for vertical lines.
   - atan(y/x) cannot return a solution when $x = 0$ because division by zero is undefined.

   In *inverse kinematics*, atan2(y, x) is preferred for robustness and accurate quadrant determination, while atan(y/x) is limited to cases where the angle lies strictly in the first or fourth quadrant and $x \neq 0$.

4. a. `get_jacobian()`
   - Initializes the transformation matrix T as the identity matrix and computes the end-effector position p_e using `forward_kinematics`.
   - Iterates through each joint to compute the Jacobian:
     - For revolute joints: Computes the linear velocity component (J_p) using the cross product of the z-axis of the previous frame and the vector $p_e - o_i$. Then it assigns the angular velocity component (J_o) as the z-axis of the previous frame.
     - For prismatic joints: Assigns the linear velocity (J_p) along the z-axis of the previous frame. Then sets the angular velocity (J_o) to zero.
   - Updates z-axis and origin of the current frame for subsequent iterations.
   - Returns the Jacobian matrix of size 6×5.

   b. To solve the inverse kinematics of the YouBot manipulator, the joint angles $\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$ are derived based on the desired pose of the end-effector represented by the transformation matrix ${}^0T_5$. The solution uses a kinematic decoupling approach, separating the problem into position and orientation components.

   Assumptions and Parameters: The Denavit-Hartenberg (DH) parameters of the YouBot manipulator are:

   $$a = [-0.033, 0.155, 0.135, 0.002, 0.0],$$
   $$\alpha = [\pi/2, 0, 0, \pi/2, \pi],$$
   $$d = [0.145, 0, 0, 0, -0.185],$$
   θ represents the joint angles.

   The homogeneous transformation matrix describing the pose of the end-effector in the base frame is ${}^0T_5$.

Position Analysis

The position of the end-effector is extracted from T:

$$p_e = \begin{bmatrix} p_{ex} \\ p_{ey} \\ p_{ez} \end{bmatrix}.$$

Solve for θ_1: The angle θ_1 determines the rotation about the base z-axis:

$$\theta_1 = \text{atan2}(p_{ey}, p_{ex}).$$

Solve for θ_2 and θ_3: These angles define the position of the manipulator in the x-y-z space. Define the wrist center W:

$$W = p_e - d_5 \cdot z_5,$$

where z_5 is the orientation of the end-effector along its z-axis.
Using the law of cosines:

- Compute the distance r from the origin to W:

$$r = \sqrt{W_x^2 + W_y^2 + W_z^2}.$$

- Solve for θ_3:

$$\theta_3 = \pm \cos^{-1}\left(\frac{r^2 - a_2^2 - a_3^2}{2a_2a_3}\right).$$

- Solve for θ_2:

$$\theta_2 = \text{atan2}(W_z, W_x) - \tan^{-1}\left(\frac{a_3 \sin(\theta_3)}{a_2 + a_3 \cos(\theta_3)}\right).$$

Orientation Analysis

The orientation is described by the rotation matrix $R = {}^0R_5$, extracted from T.

Solve for θ_4 and θ_5: Given the decoupling property of the spherical wrist:

$$\theta_4 = \text{atan2}(R_{23}, R_{13}),$$
$$\theta_5 = \text{atan2}(\sqrt{R_{13}^2 + R_{23}^2}, R_{33}).$$

The final closed-form solutions for each joint are:

$$\begin{aligned}
\theta_1 &= \text{atan2}(p_{ey}, p_{ex}), \\
\theta_2 &= \text{atan2}(W_z, W_x) - \tan^{-1}\left(\frac{a_3 \sin(\theta_3)}{a_2 + a_3 \cos(\theta_3)}\right), \\
\theta_3 &= \pm \cos^{-1}\left(\frac{r^2 - a_2^2 - a_3^2}{2a_2a_3}\right), \\
\theta_4 &= \text{atan2}(R_{23}, R_{13}), \\
\theta_5 &= \text{atan2}(\sqrt{R_{13}^2 + R_{23}^2}, R_{33}).
\end{aligned}$$

c. `check_singularity()`

- Computes the Jacobian matrix using `get_jacobian`.
- Checks the rank of the Jacobian matrix using `np.linalg.matrix_rank`.
- Returns `True` if the Jacobian rank is less than $\min(\text{rows}, \text{columns})$, indicating a singularity. Otherwise, returns `False`.

Section 2: Path and Trajectory Planning

5. a. The proposed drive system for the cleaning robot is a differential drive system, consisting of 2 independently controlled wheels on either side of the robot, and a passive caster wheel for support.

This drive system is very effective for the task due to several reasons.

- It's very compact and only requires 2 motors, which gives more space for the robot to have its other functions and cleaning components, while keeping the size of the robot small to navigate tight spaces.
- The system allows the turn in place, ie a zero turning radius, which is crucial for reaching specific points efficiently without wasted movement and pathing, and allows easy avoidance of obstacles.
- As the vacuum is at the back and the brush is at the front, having precise orientation control is important. The differential drive system allows accurate control over the robot's heading.

The differential drive system is non-holonomic, ie the robot has constraints on its motion. More specifically, the robot can only move forward or backwards along its current orientation, and must rotate to change direction, thus the robot cannot achieve instantaneous motion in an arbitrary direction.

This isn't an issue however, as with the use of effective odometry or external tracking, the need for instantaneous sideways movement is not needed, as the robot can simply rotate in place and follow planned and optimised trajectories to ensure coverage. Furthermore due to the fact that there is a 'front' and 'back' of the robot due to the brush and vacuum placement, having the robot move in a direction that is not aligned with the robot's orientation is not desirable.

The configuration space of the robot is $q = (x, y, \theta)$ where:

- x and y represent the position of the robot in the 2D cartesian map as shown in the coordinates diagram (b)
- θ represents the robot's orientation in radians

b. ## Representing the Environment

The environment is represented as a 2D grid with:

- **X-axis:** Ranges from 0 to 480.
- **Y-axis:** Ranges from 0 to 480.
- **Obstacles:**
  - Hot dog stand 1: Rectangular obstacle from $(220, 100)$ to $(260, 140)$.
  - Hot dog stand 2: Rectangular obstacle from $(160, 160)$ to $(200, 200)$.
  - Fountain: Circular obstacle with center at $(240, 240)$ and radius 40.
  - Escalator 1: Rectangular obstacle from $(135, 375)$ to $(185, 425)$.
  - Escalator 2: Rectangular obstacle from $(296, 375)$ to $(345, 425)$.

The robot is non-holonomic with a square size of 20×20, and its centre position is used for planning. The robot's configuration is defined as:

$$q = (x, y, \theta)$$

where:

- x, y: Center position of the robot.
- θ: Orientation of the robot.

Path Planning Algorithm

The best algorithm to use for path planning would be a modified **A* algorithm**, because of its ability to find an optimal path between waypoints while avoiding obstacles.

Steps:

1. **Discretize the Space:**
   - Use a fine resolution (e.g., 1×1) grid for planning.
   - Mark obstacles based on their geometry:
     - Hot dog stands and escalators: Mark all grid cells within their bounding rectangles as obstacles.
     - Fountain: Mark all grid cells within a radius of 40 around $(240, 240)$ as obstacles.
2. **Define Waypoints:**
   - The waypoints are given in order:

     Start: $(260, 60)$, $1 : (360, 170)$, $2 : (150, 70)$, $3 : (150, 300)$, $4 : (310, 330)$, End: $(230, 430)$

   - Connect each pair of consecutive waypoints (e.g., Start $\rightarrow 1$, $1 \rightarrow 2$) using the A* algorithm.
3. **Implement the A* Algorithm:**
   - The robot moves on the grid, and each move transitions to one of its neighboring cells.
   - The cost function is defined as:

     $$f(n) = g(n) + h(n)$$

     where:
     - $g(n)$: Actual cost from the start to the current node n.
     - $h(n)$: Heuristic cost to the target node, calculated using **Euclidean distance**.
   - Add penalties for:
     - Sharp turns: Penalize changes in θ beyond a certain threshold to respect the robot's non-holonomic constraints.
     - Proximity to obstacles: Add a safety buffer around obstacles (e.g., inflate their boundaries by 10 units to ensure the robot does not collide due to its size).
4. **Smooth the Path:**
   - Once the A* algorithm finds a path, use cubic splines or Bézier curves to smooth the path for realistic execution by the robot.

The robot follows the waypoints in the specified order (Start $\rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow$ End). For each segment:

- Plan the path independently between consecutive waypoints, ensuring obstacle avoidance.
- Merge the paths to create a complete trajectory.

Planning When Ordering Does Not Matter

If the order of waypoints does not matter, the problem becomes a **Traveling Salesman Problem (TSP)** with obstacles. The steps are as follows:

1. Generate all possible orders of waypoints (permutations of points $1, 2, 3, 4$, and End).
2. For each order, compute the total path cost (using A* for each segment).
3. Select the order that minimizes the total path cost.

Optimization:

- Use heuristics (e.g., nearest neighbor) to reduce computational complexity.
- Precompute distances between waypoints without obstacles to get an initial estimate of the order.

c. To ensure smooth transitions between waypoints and maintain continuity in position, velocity, and acceleration, we use a **quintic polynomial time-scaling function** $s(t)$. The function is expressed as:

$$s(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

where:

- $s(t)$ represents the progress of the robot along the trajectory at time t, normalized to range between 0 and 1.
- The coefficients $a_0, a_1, \ldots, a_5$ are determined based on boundary conditions.

Boundary Conditions

The boundary conditions ensure that the trajectory is smooth and physically feasible. For each segment (e.g., between two waypoints), the conditions are defined as follows:

1. **At the Start of the Segment ($t = 0$)**:

$$\begin{aligned} s(0) &= 0, \quad \text{the trajectory starts at the first waypoint.} \\ \dot{s}(0) &= v_0, \quad \text{the initial velocity is specified.} \\ \ddot{s}(0) &= a_0, \quad \text{the initial acceleration is specified.} \end{aligned}$$

2. **At the End of the Segment ($t = T$)**:

$$\begin{aligned} s(T) &= 1, \quad \text{the trajectory ends at the second waypoint.} \\ \dot{s}(T) &= v_f, \quad \text{the final velocity is specified.} \\ \ddot{s}(T) &= a_f, \quad \text{the final acceleration is specified.} \end{aligned}$$

3. **At the Via Points**:
   - The velocity and acceleration at the end of one segment must match the start of the next segment to ensure smooth transitions.

Obtaining Boundary Values

1. **Position**: The position values (x, y) at waypoints are explicitly given:

$$(260, 60), (360, 170), (150, 70), (150, 300), (310, 330), (230, 430).$$

   These values define the trajectory in Cartesian space, and the time-scaling function determines how quickly the robot moves along the trajectory.
2. **Velocity**:
   - Initial velocity at the start of the trajectory is typically zero: $\dot{s}(0) = 0$.
   - Final velocity at the end waypoint is also zero: $\dot{s}(T) = 0$.
   - For intermediate waypoints, velocities are estimated based on the path length and desired timing. For example:
     – Higher velocities are used for segments with fewer obstacles.

- Lower velocities are used for segments requiring precise navigation near obstacles.

3. **Acceleration**:
   - Initial and final accelerations are typically zero: $\ddot{s}(0) = 0, \ddot{s}(T) = 0$.
   - At via points, accelerations depend on the curvature of the path:
     - For high curvature (e.g., sharp turns), lower accelerations are used to ensure stability.
     - For low curvature, higher accelerations are acceptable.

Once $s(t)$ is determined for a segment, the robot's Cartesian trajectory can be computed as:

$$x(t) = x_0 + (x_1 - x_0) \cdot s(t)$$
$$y(t) = y_0 + (y_1 - y_0) \cdot s(t)$$

where (x_0, y_0) and (x_1, y_1) are the start and end waypoints of the segment. The trajectory in $x(t)$ and $y(t)$ space is parameterized using $s(t)$.

At via points:

- Continuity in position, velocity, and acceleration is enforced. This ensures smooth transitions and prevents abrupt changes in robot motion.
- If the via point represents a sharp turn, the velocity is reduced to ensure safe navigation.

d. To ensure the robot's water vacuum effectively picks up water during turns and rotations, the path and trajectory design must account for the vacuum's position relative to the robot's movement. It is discernible that abrupt turns and rotations are what would cause the vacuum to misalign with the robot's cleaning area, resulting in the vacuum missing water pick-ups.

Trajectories using curvature-constrained paths such as cubic splines, quintic polynomials or bezier curves. Minimising the robot's angular velocity during sharp turns by enforcing a maximum turning rate where water is guaranteed to be picked up will prevent water from being missed. To further ensure no water is missed, incorporating overlapping coverage during turns, as well as reverse movements will give the vacuum a second chance to pick up any missed water.

This method can be incorporated for areas where the previously imposed maximum turning rate must be exceeded, for instance when the robot is cleaning a 90 degree corner. The disadvantages with both of these methods is that they add path redundancy, as the path lengths would have to be increased and the same areas may be cleaned again. Implementing reversing will also mean the robot must pause, thus slowing the process down. Having to repeatedly accelerate from 0 is also more energy consuming.

e. **Mall Layout**

The shopping mall is divided as follows:

- **Dimensions**: 480×480.
- **Stores**:
  - Left-side stores (1–4): Located from $x = 0$ to $x = 120$, entered from the right-most wall.
  - Right-side stores (5–8): Located from $x = 360$ to $x = 480$, entered from the left-most wall.
- **Central Hallway**: Open space between $x = 120$ and $x = 360$.

The robot avoids static obstacles and padded no-go zones:

- **Obstacles**:
  - Hot Dog Stand 1: Padded area from $(210, 90)$ to $(270, 150)$.
  - Hot Dog Stand 2: Padded area from $(150, 150)$ to $(210, 210)$.

  - Fountain: Centered at $(240, 240)$, with a radius of 50.
  - Escalator 1: Padded area from $(125, 365)$ to $(195, 435)$.
  - Escalator 2: Padded area from $(286, 365)$ to $(355, 435)$.
- **No-Go Zones**:
  - Zone 1: Padded area from $(110, -10)$ to $(370, 50)$.
  - Zone 2: Padded area from $(110, 430)$ to $(370, 490)$.

The robot should follow a horizontal zigzag motion that covers the entire floor, transitioning between the central hallway and the stores. A zigzag path is optimal for covering the entire floor as it allows the robot to maintain maximum velocity as much as possible as it has to change direction less frequently. The steps are as follows:

Zigzag Motion Across the Entire Mall

1. **Start Position**: The robot begins at $(10, 10)$, on the bottom-left corner of Store 1.
2. **Horizontal Motion**:
   - The robot moves horizontally from left to right, cleaning up to the right wall of Store 5.
   - When it reaches the right wall, it shifts upward by 20 units (robot width) and moves from right to left, cleaning back to the left wall of Store 1.
3. **Store and Hallway Cleaning**:
   - The robot continues this zigzag motion across the entire width of the mall, alternating between the stores and the central hallway.
   - For left-side stores (1–4), the robot cleans horizontally from left to right and transitions seamlessly into the central hallway when reaching the right-most wall.
   - For right-side stores (5–8), the robot transitions from the central hallway to the left-most wall of the stores and cleans horizontally.
4. **Obstacle Avoidance**:
   - The robot dynamically adjusts its path to detour around obstacles and padded no-go zones. For example, it skips over the fountain's exclusion zone and resumes the zigzag pattern afterward.
   - Detours are implemented using local adjustments, ensuring smooth transitions back to the zigzag pattern.

Smooth Transitions

To maintain smooth motion:

- The robot uses cubic splines to ensure seamless U-turns at the end of each horizontal row to maintain velocity
- When detouring around obstacles, the robot generates splines to rejoin the zigzag pattern smoothly.

Acceleration and Deceleration

- The robot maintains a near-constant velocity during straight-line motion, avoiding unnecessary accelerations or decelerations.
- Gradual acceleration and deceleration are applied during transitions and U-turns to minimize jerk and maintain stability.

6. a. `load_targets()`

   Initializes `target_joint_positions` (5x5) and `target_cart_tf` (4x4x5) to store joint positions and corresponding Cartesian transforms.

   - Reads current joint positions of the robot and calculates the Cartesian pose using forward kinematics (FK).
   - For each target in the `data.bag` file, retrieves the joint positions and computes the end-effector pose as a 4x4 transformation matrix:

   $$\mathbf{T} = \mathbf{R}(\mathbf{q})\mathbf{P}(\mathbf{q}),$$

   where $\mathbf{R}$ is the rotation matrix and $\mathbf{P}$ is the translation vector.
   - Returns the Cartesian transforms and joint positions for all targets.

b. `get_shortest_path()`

   - Uses a recursive function to generate all permutations of the checkpoint indices.
   - For each permutation, computes the total path distance:

   $$D = \sum_{k=1}^{n-1} d(\mathbf{p}_{i_k}, \mathbf{p}_{i_{k+1}}),$$

   where d is the Euclidean distance:

   $$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}.$$

   - Tracks the permutation with the smallest distance.
   - Returns the shortest path order and the minimum total distance.

c. `decoupled_rot_and_trans()` and `intermediate_tfs()`

   - Computes smooth intermediate transformations between consecutive checkpoints.
   - **Translation Interpolation:** Interpolates the translation components linearly:

   $$\mathbf{p}(t) = (1 - t)\mathbf{p}_A + t\mathbf{p}_B,$$

   where $t = \frac{i}{n-1}$ and $i = 0, 1, \ldots, n-1$.
   - **Rotation Interpolation (Slerp):** Computes interpolated rotations along the shortest angular path:

   $$\mathbf{R}(t) = \text{Slerp}(\mathbf{R}_A, \mathbf{R}_B, t).$$

   - Combines the interpolated rotations and translations into 4x4 transformation matrices:

   $$\mathbf{T}(t) = \begin{bmatrix} \mathbf{R}(t) & \mathbf{p}(t) \\ 0 & 1 \end{bmatrix}.$$

   - Concatenates all intermediate transformations to form the full trajectory.
   - Returns the full trajectory as a sequence of 4x4 transformation matrices.

d. `ik_position_only()` and `full_checkpoints_to_joints()`

   - Computes the joint configurations for each pose in the trajectory using inverse kinematics (IK).
   - **Position Error:** Computes the position error between the desired pose and current pose:

   $$\mathbf{e} = \mathbf{p}_{\text{des}} - \mathbf{p}_{\text{curr}}.$$

- **Jacobian Update Rule:** Updates joint positions iteratively:

$$\Delta \mathbf{q} = \mathbf{J}^{+}\mathbf{e},$$

  where $\mathbf{J}^{+}$ is the pseudoinverse of the Jacobian.
- **Iterative Update:** Applies updates until the error is below a threshold or a maximum number of iterations is reached:

$$\mathbf{q}_{k+1} = \mathbf{q}_k + \Delta \mathbf{q}.$$

- Uses the joint positions from the previous pose as the initial guess for the next pose to ensure faster convergence.
- Returns the joint positions for the full trajectory.

e. `q6()`

- Loads the target joint positions and Cartesian transforms using `load_targets`.
- Computes the shortest order of checkpoints using `get_shortest_path`.
- Generates intermediate poses for smooth motion using `intermediate_tfs`.
- Computes joint configurations for all poses using `full_checkpoints_to_joints`.
- Constructs a `JointTrajectory` message by assigning joint positions and timestamps to trajectory points:

$$\text{time\_from\_start} = i \cdot \Delta t,$$

  where i is the index and Δt is the time step.
- Returns the complete trajectory for execution.

END OF COURSEWORK