Lab 3 - Inverse Kinematics

1) Learning Goals

By the end of this lab you will be able to:

- Convert a desired **relative target pose** (x,y,θ) into motion commands for a differential-drive robot.
- Implement two inverse-kinematics strategies:
    1. **Turn–Straight–Turn (TST):** rotate to face the target, drive straight, rotate to desired orientation.
    2. **Straight–Turn–Straight–Turn (STST):** move along an L-shaped path by combining two straight-line motions separated by an intermediate turn, then finish with a final turn to match the orientation.
- Compare the **accuracy** and **time** of the two approaches and explain the conceptual/mathematical differences.

> Throughout, the target pose (x,y,θ) is given **in centimeters and degrees**, relative to the robot's **current** pose, which we treat as the coordinate frame origin (0,0,0) with initial heading along +y.

---

2) Preliminaries & Coordinate Frames

We adopt the common differential-drive planar frame: the robot starts at (0,0) pointing along the +y axis. A target pose is (x,y,θ), where x and y are the desired position in cm, and θ is the desired final orientation in **degrees**, measured CCW from +x.

You may find the DifferentialDrive class to be helpful for controlling the robot throughout this lab. From `lib/XRPLib`, this class is instantiated in `defaults.py` and is imported with the variable name `drivetrain`. Please refer to the XRP API reference website for more information about the functions in the DifferentialDrive class.

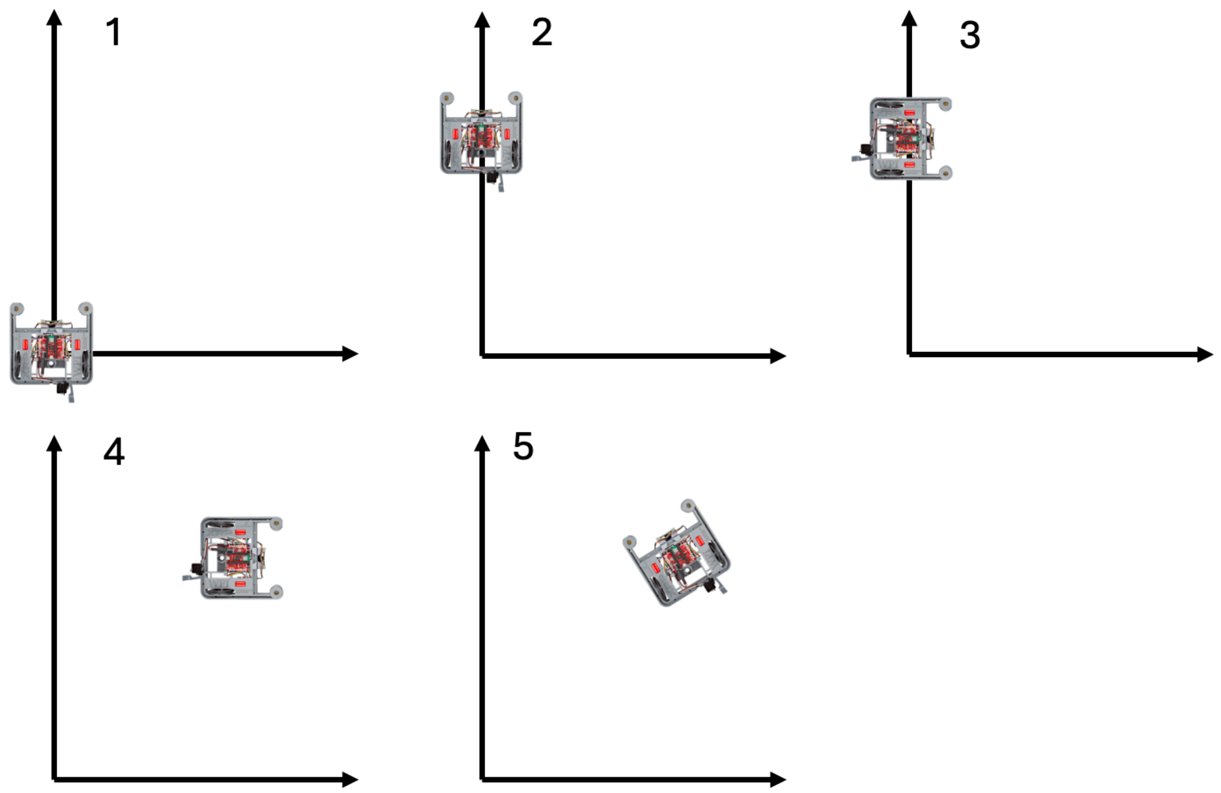
---

3) Method B: Straight–Turn–Straight–Turn (STST)

Concept: Instead of heading directly toward the goal point, this method moves in a Manhattan-style sequence: first drive straight along the y-axis, then turn to align with the x-axis, drive straight to match the x-coordinate, and finally turn in place to achieve the target orientation. This breaks the motion into **four easy primitives**.

Math:

1. **First straight segment:** move forward (or backward if needed) by y cm.
2. **Intermediate turn:** rotate left (if x>0) or right (if x<0) by 90°.
3. **Second straight segment:** move by |x| cm.
4. **Final orientation correction:** turn in place by $\theta - (\pm 90^\circ)$ depending on path taken.

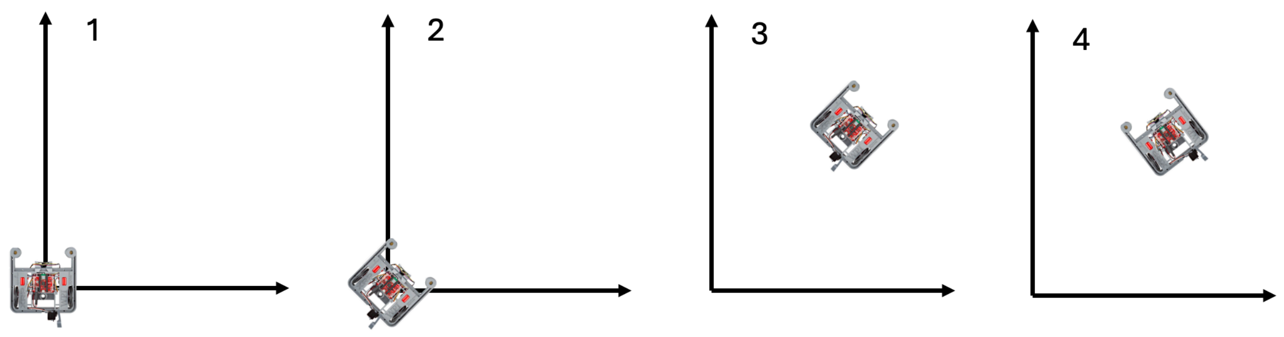


4) Method A: Turn–Straight–Turn (TST)

Concept: Decompose the motion into three simpler segments that are easy to control accurately with the XRP API: (i) rotate in place to face the target point, (ii) drive straight to that point, (iii) rotate in place to the final desired orientation.

Math:

1. **Face the goal point** (x,y): the required initial turn is $\alpha = atan2(x, y)$ (in radians; convert to degrees for commands).
2. **Straight-line distance** to the goal point is $d = \sqrt{x2 + y2}$ (in cm).
3. **Final heading correction** is $\beta = \theta - \alpha$ (be careful with angle wrap to [−180∘,180∘]).



5) Experimental Procedure

Follow the same procedure for both methods:

1. Mark an axis grid on the floor.
2. Implement both functions in a single MicroPython file.
3. Test on these three targets:
   - **Target A:** $(0, 0, 195°)$
   - **Target B:** $(25, 15, 90°)$
   - **Target C:** $(40, -15, -45°)$
4. Record timing with `time.time()`.

6) Waypoint Navigation - Draw a Circle

Now we are going to convert our code such that we can pass in a list of (x, y, θ) points and the robot will navigate between them seamlessly.

Lets use this algorithm to draw a circle with our robot. Below is a python scrip that will take in a radius, R, and a number of points, N, and will generate a set of N (x, y, θ) points that lay on the circle. The idea is that if you pass these into your TST functional algorithm sequentially, the path of the robot should approximate the path of a circle as N gets bigger. When $N = 3$, the robot should make a triangle, when $N = 4$, the robot should make a square, when $N = 5$, it should make a pentagon, and so on.

```python
import math

def normalize_deg(deg):
    return (deg + 180.0) % 360.0 - 180.0

def generate_circle_path(n_points, radius_cm, closed=True, ccw=True):
    """
    Generate waypoints (x, y, theta_deg) approximating a circle of i
    radius R, starting and ending at (0,0,0). Circle center is (0, R).

    Args:
        n_points (int): number of waypoints to return.
            If closed=True, the first and last points are identical
            (closed loop).
            If closed=False, the endpoint repeat is omitted.
        radius_cm (float): circle radius in cm.
        closed (bool): include the final point equal to the start (default
        True).
        ccw (bool): traverse counterclockwise if True, clockwise if False.

    Returns:
        list[(float, float, float)]: (x, y, theta_deg) tuples.
    """
    if n_points <= 0:
        return []

    # Circle center such that start/end is (0,0,0)
    cx, cy = 0.0, float(radius_cm)

    # Start at bottom of circle so position is (0,0) and tangent is 0°
    phi0 = -math.pi / 2.0

    # How many steps along the circle?
    # If closed, we want to include both start and
    # end (same point) -> n_points-1 intervals.
    steps = n_points - 1 if closed and n_points > 1 else n_points - 1
    total_span = 2.0 * math.pi  # full circle
    if steps <= 0:
        # Degenerate cases (n_points=1)
        x0 = cx + radius_cm * math.cos(phi0)
        y0 = cy + radius_cm * math.sin(phi0)
        theta0 = normalize_deg(math.degrees(phi0 + (math.pi/2.0)))
        return [(x0, y0, theta0)]

    dphi = total_span / steps
    if not ccw:
        dphi = -dphi

    points = []
    # When closed=True, iterate i=0..steps so last equals first.
    # When closed=False, iterate i=0..steps-1 so last != first.
    imax = steps if closed else steps
    for i in range(imax + (1 if closed else 0)):
        phi = phi0 + i * dphi
        x = cx + radius_cm * math.cos(phi)
```

```python
        y = cy + radius_cm * math.sin(phi)
        # Tangent direction: perpendicular to radius.
        # For CCW, tangent angle = phi + 90°.
        # For CW, it flips sign; we can still use phi + 90° because
        # phi itself is walking CW when ccw=False.
        theta_deg = normalize_deg(math.degrees(phi + math.pi / 2.0))
        points.append((x, y, theta_deg))

        if not closed and i == steps:
            break

    # Ensure first point is exactly (0,0,0)
    points[0] = (0.0, 0.0, 0.0)
    if closed:
        points[-1] = (0.0, 0.0, 0.0)

    return points


# Example usage
if __name__ == "__main__":
    # 12 waypoints including the final return to (0,0,0), CCW
    waypoints = generate_circle_path(n_points=12,
                                     radius_cm=30,
                                     closed=True,
                                     ccw=True)

    for wp in waypoints:
        print(wp)

```

Notes

- This algorithm generates absolute (x, y, θ) way points. Depending on how you implement your TST function, you make need to pass in relative way points. You can do this subtracting the previous point from the current point.
- Wrap around - depending on how you handle wrap around, your TST algorithm might expect angles to be between $[-180°, 180°]$ or it might expect angles from $[0, 360]$. Either way, try to implement a "shortest turn" algorithm, such that your robot figures out the shortest way to get from its current orientation to the desired orientation. This will make your robot drive in a circle smoothly, and not spin around in the middle of trying to navigate the circle.
- Drift - Even though the robot should be driving in a loop, it might not end up where it started. Take note of what the final error in position is. To measure the error, put a piece of tape on the floor where the robot starts and measure the distance to where it ends.

Once you get your algorithm working, try messing around with a few different radii and different numbers of waypoints.

7) Report & Submission

- Submit the code for all three tasks: STST method, TST method, and waypoint navigation. Test the waypoint navigation with $N = 4, N = 8$, and $N = 12$.
- Report the distance between the starting point and the ending point for the circular waypoint path.
- Report the time difference between STST and TST for the three targets that you tested.
- Answer the following questions:
  - How many hours did you spend on this lab?
  - Did you face any challenges with this lab?
  - Did you use AI to assist with this lab, and if so, what did you use it for?