Robotic Systems Engineering
Coursework 2: Jacobian, Inverse Kinematics and Path Planning

Harry Yao
ucabry0@ucl.ac.uk

November 27, 2024

Section 1

1. **The 2D 4R planar manipulator has infinite inverse kinematic solutions due to redundancy**

   Here's the proof:

   The 2D 4R planar manipulator consists of four rotational joints $(\theta_1, \theta_2, \theta_3, \theta_4)$. Given achievable pose of the end-effector $\mathbf{x}_e = [p_{ex}, p_{ey}, \phi_e]$.

   For a 4R planar manipulator, the forward kinematics are given by:

$$\begin{aligned} p_{ex} &= l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) + l_3 \cos(\theta_1 + \theta_2 + \theta_3) + l_4 \cos(\phi_e), \\ p_{ey} &= l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) + l_3 \sin(\theta_1 + \theta_2 + \theta_3) + l_4 \sin(\phi_e), \\ \phi_e &= \theta_1 + \theta_2 + \theta_3 + \theta_4. \end{aligned}$$

   Therefore we need to solve the possible joint angles $(\theta_1, \theta_2, \theta_3, \theta_4)$ using inverse kinematics, given the known achievable pose $\mathbf{x}_e$. However, due to redundancy (4 DOF controlling a 3-dimensional task space), there exists an infinite number of solutions. Below is the detailed derivation.

   (1) Express θ_3 from the Orientation Equation:

$$\theta_3 = \phi_e - \theta_1 - \theta_2 - \theta_4.$$

   (2) Substitute θ_3 into the Position Equations:

$$\begin{aligned} p_{ex} &= l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) + l_3 \cos(\phi_e - \theta_4) + l_4 \cos(\phi_e), \\ p_{ey} &= l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) + l_3 \sin(\phi_e - \theta_4) + l_4 \sin(\phi_e). \end{aligned}$$

   Define:

$$\begin{aligned} P_{ex} &= p_{ex} - l_3 \cos(\phi_e - \theta_4) - l_4 \cos(\phi_e), \\ P_{ey} &= p_{ey} - l_3 \sin(\phi_e - \theta_4) - l_4 \sin(\phi_e). \end{aligned}$$

(3) Expand $\cos(\theta_1 + \theta_2)$ and $\sin(\theta_1 + \theta_2)$:

$$\cos(\theta_1 + \theta_2) = \cos(\theta_1)\cos(\theta_2) - \sin(\theta_1)\sin(\theta_2),$$
$$\sin(\theta_1 + \theta_2) = \sin(\theta_1)\cos(\theta_2) + \cos(\theta_1)\sin(\theta_2).$$

Substitute into the equations:

$$P_{ex} = l_1\cos(\theta_1) + l_2\big[\cos(\theta_1)\cos(\theta_2) - \sin(\theta_1)\sin(\theta_2)\big],$$
$$P_{ey} = l_1\sin(\theta_1) + l_2\big[\sin(\theta_1)\cos(\theta_2) + \cos(\theta_1)\sin(\theta_2)\big].$$

(4) Combine Terms:

$$P_{ex} = [l_1 + l_2\cos(\theta_2)]\cos(\theta_1) - l_2\sin(\theta_2)\sin(\theta_1),$$
$$P_{ey} = [l_1 + l_2\cos(\theta_2)]\sin(\theta_1) + l_2\sin(\theta_2)\cos(\theta_1).$$

Define:

$$A = l_1 + l_2\cos(\theta_2), \quad B = l_2\sin(\theta_2).$$

Then:

$$P_{ex} = A\cos(\theta_1) - B\sin(\theta_1),$$
$$P_{ey} = A\sin(\theta_1) + B\cos(\theta_1).$$

(5) Solve for θ_1: Express the equations in complex form:

$$P_{ex} + jP_{ey} = (A + jB)(\cos(\theta_1) + j\sin(\theta_1)) = (A + jB)e^{j\theta_1}.$$

Thus:

$$e^{j\theta_1} = \frac{P_{ex} + jP_{ey}}{A + jB}.$$

The angle θ_1 is:

$$\theta_1 = \arctan 2(P_{ey}, P_{ex}) - \arctan 2(B, A).$$

(6) Solve for θ_2: Using the magnitude of P_{ex} and P_{ey}:

$$P_{ex}^2 + P_{ey}^2 = A^2 + B^2 = (l_1 + l_2\cos(\theta_2))^2 + (l_2\sin(\theta_2))^2.$$

Simplify:

$$P_{ex}^2 + P_{ey}^2 = l_1^2 + l_2^2 + 2l_1l_2\cos(\theta_2).$$

Solve for $\cos(\theta_2)$:

$$\cos(\theta_2) = \frac{P_{ex}^2 + P_{ey}^2 - l_1^2 - l_2^2}{2l_1l_2}.$$

Thus:

$$\theta_2 = \arccos\left(\frac{P_{ex}^2 + P_{ey}^2 - l_1^2 - l_2^2}{2l_1l_2}\right).$$

(7) Solve for θ_3: Using the orientation equation:

$$\theta_3 = \phi_e - \theta_1 - \theta_2 - \theta_4.$$

(8) Final Expressions:

$$\theta_1 = \arctan 2\big(p_{ey} - l_3\sin(\phi_e - \theta_4) - l_4\sin(\phi_e),\, p_{ex} - l_3\cos(\phi_e - \theta_4) - l_4\cos(\phi_e)\big)$$
$$- \arctan 2\big(l_2\sin(\arccos(\frac{(p_{ex} - l_3\cos(\phi_e - \theta_4) - l_4\cos(\phi_e))^2 + (p_{ey} - l_3\sin(\phi_e - \theta_4) - l_4\sin(\phi_e))^2 - l_1^2 - l_2^2}{2l_1l_2})),$$
$$l_1 + l_2\cos(\arccos(\frac{(p_{ex} - l_3\cos(\phi_e - \theta_4) - l_4\cos(\phi_e))^2 + (p_{ey} - l_3\sin(\phi_e - \theta_4) - l_4\sin(\phi_e))^2 - l_1^2 - l_2^2}{2l_1l_2}))),$$
$$\theta_2 = \arccos\left(\frac{(p_{ex} - l_3\cos(\phi_e - \theta_4) - l_4\cos(\phi_e))^2 + (p_{ey} - l_3\sin(\phi_e - \theta_4) - l_4\sin(\phi_e))^2 - l_1^2 - l_2^2}{2l_1l_2}\right),$$
$$\theta_3 = \phi_e - \theta_1 - \theta_2 - \theta_4.$$

(9) Infinite Solutions Due to Redundancy:
Since θ_4 can be arbitrarily chosen within its joint limits, for each θ_4, there exists a corresponding set of $\theta_1, \theta_2, \theta_3$ that satisfy the equations. Therefore, infinite solutions exist.

Redundancy and Infinite Solutions:

1. The 4R planar manipulator has 4 DOF, but the task space (defined by (p_{ex}, p_{ey}, ϕ_e)) only has 3 dimensions. This leaves one redundant DOF, meaning the system can achieve the same end-effector pose in infinitely many ways by varying θ_4 (or redistributing the angles among the joints).

2. The redundancy allows for a continuous solution space. For example, given a fixed $\theta_1, \theta_2, \theta_3$, θ_4 can take any value that satisfies:

$$\phi_e = \theta_1 + \theta_2 + \theta_3 + \theta_4.$$

3. While the above derivation considers discrete elbow-up and elbow-down configurations, the actual solution space is a continuum due to the redundancy. Therefore, the manipulator has an infinite number of inverse kinematic solutions for a given pose $\mathbf{x}_e$.

Conclusion: The 2D 4R planar manipulator has infinite inverse kinematic solutions for a given achievable pose $\mathbf{x}_e$, due to its redundancy (4 DOF controlling a 3-dimensional task space).

2. Criteria need to be considered:

(a) Minimize total joint movement:

$$\text{Total Joint Movement} = \sum_{i=1}^{n} |\Delta\theta_i|,$$

$\Delta\theta_i$ represents the angular change for each joint.
Minimum total joint movement means the robot achieve the pose using least movement, which least power consumption of the robot, extends the life of the robot.

(b) Avoiding Joint Limits:

$$\theta_{\text{min}} \leq \theta_i \leq \theta_{\text{max}}.$$

Ensure that all joint angles θ_i stay within their operational limits, avoiding robot joint damage.

(c) Avoiding Singularities: Avoid configurations where the Jacobian matrix J becomes singular, determinant close to zero, leading to loss of control or instability.

$$\det(J) \neq 0.$$

Singularities can result in infinite joint velocities, which can cause robot damage.

(d) Smoothness of Motion: Choose solutions that minimize abrupt changes in joint configurations. This make sure the robot's smoothness when operating high accuracy required tasks.

(e) Task-Specific Requirements: Select configurations based on the task's specific needs. For certain tasks, an "elbow-up" or "elbow-down" configuration may be preferable to achieve specific moving path and task.

3. The function atan(y/x) is defined as:

$$\text{atan}(y/x) = \arctan\left(\frac{y}{x}\right),$$

where the output range is:

$$\text{atan}(y/x) \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right).$$

The function atan2(y, x) is defined as:

$$\text{atan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right), & \text{if } x > 0, \\ \arctan\left(\frac{y}{x}\right) + \pi, & \text{if } x < 0 \text{ and } y \geq 0, \\ \arctan\left(\frac{y}{x}\right) - \pi, & \text{if } x < 0 \text{ and } y < 0, \\ \frac{\pi}{2}, & \text{if } x = 0 \text{ and } y > 0, \\ -\frac{\pi}{2}, & \text{if } x = 0 \text{ and } y < 0, \\ \text{undefined}, & \text{if } x = 0 \text{ and } y = 0. \end{cases}$$

The output range is:

$$\text{atan2}(y,x) \in [-\pi, \pi].$$

The fundamental difference is that atan2(y, x) uses both x and y to determine the angle and its quadrant, while atan(y/x) only considers the ratio y/x, losing quadrant information.

Base on the fundamental difference, the different situation of output between atan2(y, x) and atan(y/x) are as follow:

(a) **Quadrant Identification:**

atan2(y, x) determines the correct quadrant based on the signs of x and y, while atan(y/x) cannot.

Example: For $(x, y) = (-1, 1)$,

$$\text{atan2}(1,-1) = \frac{3\pi}{4}, \quad \text{atan}\left(\frac{1}{-1}\right) = -\frac{\pi}{4}.$$

(b) **Division by Zero:**
atan2(y, x) handles $x = 0$ without errors, but atan(y/x) is undefined for $x = 0$.
Example: For $(x, y) = (0, 1)$,

$$\text{atan2}(1, 0) = \frac{\pi}{2}, \quad \text{atan}\left(\frac{1}{0}\right) \text{ is undefined.}$$

(c) **Symmetry Issues:**
atan2(y, x) distinguishes symmetric points like $(-1, -1)$ and $(1, 1)$, while atan(y/x) gives the same result for both.

4. a. 1. Initialization:

```
z_prev = np.array([0, 0, -1])   # Base Z-axis
o_prev = np.array([0, 0, 0])    # Base origin
```

Initialize the base frame's Z-axis and origin.

2. Compute End-Effector Position:

```
o_end = T_matrices[-1][:3, 3]
```

Extract the translation component (position) of the last transformation matrix $T_{\text{matrices}}[-1]$.

3. Iterate Over Joints:

```
for i in range(5):
    z_i = T_matrices[i][:3, 2]   # Current Z-axis
    o_i = T_matrices[i][:3, 3]   # Current origin
```

For each joint i, compute its Z-axis direction z_i and origin position o_i from the transformation matrix T_i.

4. Compute Jacobian Linear Velocity Component:

```
jacobian[:3, i] = np.cross(z_prev, o_end - o_prev)
```

The linear velocity part is calculated using:

$$J_{v_i} = z_{\text{prev}} \times (o_{\text{end}} - o_{\text{prev}})$$

5. Compute Jacobian Angular Velocity Component:

```
jacobian[3:, i] = z_prev
```

The angular velocity part is the previous Z-axis:

$$J_{\omega_i} = z_{\text{prev}}$$

6. Update States:

```
z_prev = z_i
o_prev = o_i
```

Update z_{prev} and o_{prev} for the next joint iteration.

7. Return Result:

```
    return jacobian
```

Return the final Jacobian matrix:

$$J = \begin{bmatrix} J_v \\ J_\omega \end{bmatrix}$$

b.

$${}^0T_5 = \begin{pmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

We first compute the position of the wrist center. The end-effector frame is offset by d_5 along its z_5-axis. Thus,

$$p_{\text{wrist}} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} - d_5 \begin{pmatrix} r_{13} \\ r_{23} \\ r_{33} \end{pmatrix} = \begin{pmatrix} p_x - d_5 r_{13} \\ p_y - d_5 r_{23} \\ p_z - d_5 r_{33} \end{pmatrix}.$$

Since joint 1 typically rotates about a vertical axis, we have

$$q_1 = \arctan 2(p_{\text{wrist},y}, p_{\text{wrist},x}).$$

After determining q_1, we focus on the planar 2R subproblem for joints 2 and 3. Define

$$x' = \sqrt{p_{\text{wrist},x}^2 + p_{\text{wrist},y}^2} - a_1, \quad z' = p_{\text{wrist},z} - d_1.$$

Let

$$R = \sqrt{x'^2 + z'^2}.$$

Using the law of cosines for the triangle formed by a_2 and a_3:

$$\cos(q_3) = \frac{R^2 - a_2^2 - a_3^2}{2a_2a_3}.$$

Thus,

$$q_3 = \arccos\left(\frac{R^2 - a_2^2 - a_3^2}{2a_2a_3}\right).$$

Next, we find q_2:

$$q_2 = \arctan 2(z', x') - \arctan 2(a_3 \sin(q_3), a_2 + a_3 \cos(q_3)).$$

Having determined q_1, q_2, q_3, we know the orientation of frame 3. We now solve for q_4 and q_5 using the desired end-effector orientation. Let ${}^0R_5$ be the rotation part of ${}^0T_5$. Compute

$${}^3R_5 = ({}^0R_3(q_1, q_2, q_3))^T {}^0R_5.$$

If the last two joints form a wrist such that

$$^3R_5 = R_y(q_4)R_z(q_5),$$

then comparing the elements, we got

$$q_5 = \arcsin(-(^3R_5)_{31}),$$

$$q_4 = \arctan 2((^3R_5)_{21}, (^3R_5)_{11}).$$

These processes provide a closed-form solution for q_1, q_2, q_3, q_4, q_5.

c. 1. Compute the Jacobian Matrix: Use the `get_jacobian` function to compute the Jacobian matrix for the given joint angles.

```
jacobian = self.get_jacobian(joint)
```

2. Extract the Linear Velocity Component: The linear velocity component is represented by the first 3 rows of the Jacobian matrix.

```
linear_velocity = jacobian[:3, :]
```

3. Compute the Determinant: Calculate the determinant of the linear velocity component using `np.linalg.det`.

```
determinant = np.linalg.det(linear_velocity)
```

4. Check for Singularity: Use `np.isclose` to determine if the determinant is approximately zero, indicating a singular configuration.

```
singularity = np.isclose(determinant, 0)
```

5. Output Validation and Return: Ensure the output is of type `bool` and return the singularity status.

```
assert isinstance(singularity, bool)
return singularity
```

Section 2

5. a. A differential-drive robot with a front-mounted brush and rear-mounted vacuum is chosen. This non-holonomic configuration is cost-effective and well-established. The configuration space is $q = (x, y, \theta)$, where x, y define the position and θ defines the orientation.

b. Represent the environment as graph, points as nodes and path between them as edges first. If order matters, use Dijkstra's algorithm to find the shortest path. If the order does not matter, consider the problem as Traveling Salesman Problem-like optimization to find an efficient visiting sequence.

c. Use a cubic polynomial time-scaling function:

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

which can make sure that:

1. Zero initial and final velocities and accelerations:

$$q(t_s) = q_s, \quad q'(t_s) = 0, \quad q(t_f) = q_f, \quad q'(t_f) = 0$$

in which s means start, f means final.

2. Maintaining continuity in velocity and acceleration creates smooth deceleration and acceleration when passing points.

d.
1. Use smooth, wide turns to avoid misalignment.
2. Prioritize forward motion over rotations.
3. Overlap paths in turns for complete coverage.
4. Reduce speed during tight corners.
5. Align robot's orientation with cleaning direction.

e.
1. Use PRM to sample $\mathcal{Q}_{\text{free}}$ and construct a roadmap.
2. Optimize paths in subregions with a systematic lawnmower pattern.
3. Incorporate obstacles by excluding $\mathcal{Q}_O$ from sampling.
4. Smooth paths using Bézier curves or polynomial splines.
5. Combine global PRM paths with TSP-optimized subregion sequences for efficient cleaning.

6. a. In the `load_targets()` method:

```
for topic, msg, t in
   bag.read_messages(topics=['/target_joint_positions']):
for i in range(len(msg.points)):
    if i < 4:
        target_joint_positions[:, i+1] =
           msg.points[i].positions
for i in range(1, 5):
    target_cart_tf[:,:,i] =
       self.kdl_youbot.forward_kinematics
    (target_joint_positions[:,i])
```

I directly read from the `/target_joint_positions` topic in the bag file. For each incoming message, I extract up to four sets of target joint positions (since we have four target checkpoints). After populating the `target_joint_positions` array with these values, I use the forward kinematics function to convert these joint configurations into their corresponding end-effector poses. By doing this, I ensure that each of the four target joint positions now has a known Cartesian position, giving me the full set of transformations (one initial plus four targets) that I need for the planning process.

b. In the `get_shortest_path()` method:

```
from itertools import permutations
def dist(a,b):
    return np.linalg.norm(checkpoints_tf
    [:3,3,a]-checkpoints_tf[:3,3,b])
min_dist = float('inf')
best_order = None
for perm in permutations([1,2,3,4]):
    order = [0]+list(perm)
    d=0.0
    for i in range(len(order)-1):
        d+=dist(order[i],order[i+1])
    if d<min_dist:
        min_dist=d
        best_order=order
sorted_order=np.array(best_order)
```

I focus on finding the shortest order to visit all checkpoints. First, I define a function that computes the Euclidean distance between any two checkpoints. Then, I go through every possible permutation of the four target checkpoints (1 to 4) while always starting from the initial checkpoint (0). For each possible order, I sum up the distances between consecutive checkpoints. I keep track of the minimum total distance I find and, once I've checked all permutations, I pick the one with the shortest path. This gives me the final sorted order of checkpoints to visit and the minimum total travel distance.

c. For `decoupled_rot_and_trans()`:

```
pos_a=checkpoint_a_tf[:3,3]
pos_b=checkpoint_b_tf[:3,3]
R_a=checkpoint_a_tf[:3,:3]
R_b=checkpoint_b_tf[:3,:3]
Ra=PyKDL.Rotation(R_a[0,0],R_a[0,1],R_a[0,2],
R_a[1,0],R_a[1,1],R_a[1,2],R_a[2,0],R_a[2,1],R_a[2,2])
Rb=PyKDL.Rotation(R_b[0,0],R_b[0,1],R_b[0,2],
R_b[1,0],R_b[1,1],R_b[1,2],R_b[2,0],R_b[2,1],R_b[2,2])
qa=Ra.GetQuaternion()
qb=Rb.GetQuaternion()
total_steps=num_points+2
tfs=np.zeros((4,4,total_steps))
for i in range(total_steps):
    alpha=float(i)/(total_steps-1)
    p=(1-alpha)*pos_a+alpha*pos_b
    dot=qa[0]*qb[0]+qa[1]*qb[1]+qa[2]*qb[2]+qa[3]*qb[3]
    if dot<0.0:
        qb=(-qb[0],-qb[1],-qb[2],-qb[3])
        dot=-dot
    if dot>0.9995:
```

```
        q_interp=[qa[j]+alpha*(qb[j]-qa[j]) for j in
            range(4)]
        q_interp=q_interp/np.linalg.norm(q_interp)
    else:
        theta_0=np.arccos(dot)
        sin_theta_0=np.sin(theta_0)
        theta=theta_0*alpha
        sin_theta=np.sin(theta)
        s0=np.cos(theta)-dot*(sin_theta/sin_theta_0)
        s1=sin_theta/sin_theta_0
        q_interp=[s0*qa[j]+s1*qb[j] for j in range(4)]
    q_interp/=np.linalg.norm(q_interp)
    Rint=PyKDL.Rotation.Quaternion(q_interp[0],q_interp[1],
    q_interp[2],q_interp[3])
    R_mat=np.array([[Rint[0,0],
    Rint[0,1],Rint[0,2]],[Rint[1,0],Rint[1,1],Rint[1,2]],
    [Rint[2,0],Rint[2,1],Rint[2,2]]])
    tf=np.eye(4)
    tf[:3,:3]=R_mat
    tf[:3,3]=p
    tfs[:,:,i]=tf
```

I start by extracting the positions and rotations of the start and end checkpoints. Then, I linearly interpolate the translation between these points. For the orientation, I convert both rotations to quaternions and use a spherical linear interpolation (slerp) approach, which smoothly transitions the orientation. By doing this, I produce a set of intermediate transformations that form a continuous path in both position and orientation.

In `intermediate_tfs()`:

```
full_list=[]
for i in range(len(sorted_checkpoint_idx)-1):
    A=target_checkpoint_tfs[:,:,sorted_checkpoint_idx[i]]
    B=target_checkpoint_tfs[:,:,sorted_checkpoint_idx[i+1]]
    seg=self.decoupled_rot_and_trans(A,B,num_points)
    if i<(len(sorted_checkpoint_idx)-1):
        full_list.append(seg[:,:,:-1])
    else:
        full_list.append(seg)
full_checkpoint_tfs=np.concatenate(full_list,axis=2)
```

I apply the above interpolation method to every pair of checkpoints along the shortest path I found previously. For each pair, I generate intermediate waypoints using `decoupled_rot_and_trans()`, and then I stitch all these segments together. This results in a full, smooth trajectory from the initial pose all the way through to the final target checkpoint, passing smoothly through every intermediate checkpoint in the shortest order.

d. In `ik_position_only()`:

```
max_iter=100
epsilon=1e-4
q=q0.copy()
desired_pos=pose[:3,3]
for _ in range(max_iter):
    fk=self.kdl_youbot.forward_kinematics(q)
    current_pos=fk[:3,3]
    error_vec=desired_pos-current_pos
    error=np.linalg.norm(error_vec)
    if error<epsilon:
        break
    J=self.kdl_youbot.jacobian(q)
    J_pos=J[0:3,:]
    dq=np.linalg.pinv(J_pos).dot(error_vec)
    q+=dq
```

I solve for the joint values needed to reach a desired end-effector position. I do this iteratively: first, I compute the current end-effector position from the given joint configuration, then I measure the difference to the target position. Using the Jacobian's position part, I apply a pseudo-inverse update to move the joints closer to the solution. I repeat this until the error is small enough or I hit a maximum number of iterations.

In `full_checkpoints_to_joints()`:

```
n=full_checkpoint_tfs.shape[2]
q_checkpoints=np.zeros((5,n))
q=init_joint_position.copy()
for i in range(n):
    pose=full_checkpoint_tfs[:,:,i]
    q,err=self.ik_position_only(pose,q)
    q_checkpoints[:,i]=q
```

I take the entire set of intermediate poses from `intermediate_tfs()` and run `ik_position_only()` on each one. Starting from the initial joint configuration, I solve for each subsequent pose in turn. By the end, I have a complete set of joint angle solutions corresponding to the entire Cartesian path. This gives me the final joint-space trajectory that the robot can follow to achieve the planned movement.

END OF COURSEWORK