

Final Project Report

Vibha Kamath, Berk Kasimcan, Shreya Terala, Yvonne Zhang

Jin Seob Kim

EN.530.646.FA25 RDKDC

12.15.2025

1: Task & Workflow

Push-and-Place Task

Our goal for this project assignment was to program a UR5(e) to complete a simple manipulation task. The project is teaching us how to use our kinematics knowledge to drive a “push-and-place” routine sequence of desired rigid body poses in $SE(3)$. Initially, the robot is taught a start contact pose, followed by automatically accomplishing a series of motions to softly push a foam cube about 3 cm to the right, which is defined as in the +x direction of the robot’s tool frame. Translation to a second determined contact pose is executed to push the cube back towards its original position.

The second goal of this project is to directly compare resolve-rate and inverse-kinematics control as two different ways of executing the same Cartesian task. Both RR control and IK execute the same sequence of poses in $SE(3)$, but they do use different mathematical viewpoints to show our understanding of the concepts. This allows us to study how local, differential kinematics feedback compares to solving the full IK problem at discrete waypoints. Since it’s under the identical task geometry and safety constraints, it gives us a parallel learning opportunity.

This comparison is useful because the two approaches fail and succeed for different numerical reasons. RR’s update is driven by the Jacobian (J^{\dagger}), so performance depends on step size & gain. Also, the numerical conditioning of J along the trajectory is an important thing to consider. When the Jacobian is well-conditioned, small modeling or pose errors are corrected over successive iterations. However, as the robot approaches singularity configurations, the J becomes poorly conditioned. This can lead to large or unstable joint updates even for small Cartesian errors. On the other hand, IK can satisfy the pose exactly at each waypoint but returns multiple valid solutions. The motion quality can depend on selecting a consistent kinematics branch. What we do is we choose the solution nearest to the current joints using wrapped joint distance to avoid discontinuous flips.

2: Control Methods

Algorithms

The following control schemes are implemented in the `ur_project.m` and `ur_project_rtde.m` files. Each control method + robot environment code was also separated into individual files for easier debugging; these files are: `RR_ROS.m`, `ur_project_rtde.m`, and `IK_ROS.m`.

Resolved-Rate Control (Implemented in ROS & RTDE Environments)

Our resolved rate (RR) controlled a differential kinematics-based solution to push-and-place tasks. This helps us bypass the need for full Inverse Kinematics (IK) solutions at every waypoint. The method we use is consistent with the basic idea of RR control as described in the project and the MLS textbook. It's defined as the process of alternating computing a small Cartesian correction and propagating it backwards via Jacobian to derive a joint space correction.

Also, the implementation features the ability to teach the initial pose via pendant control, also known as freedrive, followed by a pause, reading of joint angles, and then resumption of control. After we have the start frame g_{start} , we build a short sequence of physically meaningful $SE(3)$ keyframes: (i) push forward by $d_{\text{push}} = 0.03$ m along the tool's local $+\hat{x}$ direction, (ii) lift by a fixed height to clear the cube, (iii) translate above the far-side contact point, (iv) descend to contact, (v) push back by 0.03 m along local (\hat{x}) , (vi) lift again, then return home. Each keyframe is then controlled by calling our RR controller, `urRRcontrol`, to converge to that desired pose before moving on to the next one.

From Lab 3, inside `urRRcontrol`, we use the discrete-time RR update to use the body Jacobian $J_b(q)$ and the twist representation of the pose error. Concretely, given a desired end-effector pose g_d and the current pose $g(q)$ from forward kinematics, we form an error transform $g_{\text{err}} = g_d^{-1}g(q)$ and extract the corresponding twist $\xi_k \in \mathbb{R}^6$ via the matrix `log` (our `getXi` routine). Then we update joints using

$$q_{k+1} = q_k - K T_{\text{step}} J_b(q_k)^{-1} \xi_k$$

with K set by our RR gain (we used $K_{\text{rr}} = 0.15$) in the main script. We continue iterating until the rotational component of the twist, as well as the translational component, is less than the corresponding threshold. Especially since they are in different units. Then, we safely abort when we realize that we are approaching a singularity. This means it returns to (-1) , which is the same as the termination/singularity in the RR lab.

2.1: Comparing ROS and RTDE

In the simulation, the RTDE version exhibited a greater lag between movement steps than the ROS version. However, when testing in real life, as the ROS system has more inherent lag when

communicating with the robot, we found the RR control was smoother when operating with RTDE. This indicates that the communication system used can significantly impact the robot's movements, with RTDE providing direct communication and resulting in smoother movements with minimal latency.

RR_ROS & RR_RTDE Pseudocode

```

INPUT: Robot model, frame selection method, RR gain
OUTPUT: Executed push-and-place motion, Start and target pose errors

BEGIN
    CONNECT to robot using ur_interface
    READ current joint angles
    IF robot is not at home pose THEN
        MOVE robot to home pose
    ENDIF
    COMPUTE home pose using FK

    OUTPUT "1 = sample frames, 2 = teach start"
    INPUT frame choice
    IF sample frames selected THEN
        SET start pose and target pose from predefined frames
    ELSE
        SWITCH to pendant control
        WAIT until the position of the robot in freedrive
        READ joint angles
        SWITCH back to control
        COMPUTE start pose using FK
        COMPUTE target pose automatically
    ENDIF

    COMPUTE all task poses (push, lift, travel, push back)
    STORE poses in desired pose list

    FOR each desired pose in the list
        CALL resolved-rate controller with desired pose and gain
        IF pose is start or target THEN
            COMPUTE and OUTPUT pose error
        ENDIF
    NEXT

```

MOVE robot back to home pose
END

Inverse Kinematics Control (Implemented in ROS Environment)

In the IK_ROS.m file, push-and-place is considered a pose planning problem in $SE(3)$. This can also be known as the Cartesian planning problem in $SE(3)$. These planning problems come with a desired set of end effector poses $g_d(k)$. $g_d(k)$ is specified initially and then followed by IK to transform the poses to UR5 robot joint angles.

The script we wrote interacts with the `ur_interface`, which moves the robot to a stable home joint configuration for repeatability. The script helps with calculating the corresponding home pose g_{home}/g_{start} from `urFwdKin` to enable returning to home pose at the end of the process. We later use it as the final return pose at the end of the task. Next, we decide how the start and target frames are specified. For debugging and simulation purposes, predefined “sample frames” are used. The real workflow of the start pose can be taught by manually positioning the robot in free drive. Then we can convert the measured joint angles to have a start frame by just using FK.

Once we have g_{home}/g_{start} , we can build the motion by using a few keyframes. The first push is a 3 cm translation along the robot tool’s local +x direction. The way we do is that we take `dir1=R_start(:, 1)` and set g_{start_end} by shifting the position by 3 cm * *dir1* while keeping the same orientation. If the contact of pose at the goal isn’t already known, we define a sensible far-side contact pose by translating from g_{start_end} with an offset based on the size of the cube + a guessed width for the tool in the same direction as translation. The second push was also needed. It’s a translation of 3cm in the -x direction at the goal target. It’s defined as `dir2 = -R_start(:, 1)`, followed by a final lift back to g_{home}/g_{start} .

For every desired frame $g_d(k)$, we solve the analytic inverse kinematics problem using the TA-provided `urInvKin` function. This function returns up to 8 valid joint-angle solutions for the UR5. Rather than integrating velocities through the Jacobian, the robot moves by directly executing these discrete IK solutions.

Due to the multiple solution characteristic of the IK solver, a reliable approach is adapted to facilitate a smooth trajectory. For a specific waypoint, the joint posture is determined as the solution closest to the robot’s current joining position $\|\text{wrapToPi}(q_i - q_{cur})\|_r$. Intermediate waypoints are generated between keyframes using smooth $SE(3)$ interpolation, producing the sequence. This is expressed by a wrapped joining space error. The solution closets in the joint space ensures that the robot stays on a single kinematics path with no sudden elbow/wrist flip.

By applying this procedure to every interpolated waypoint, the robot moves along the planned Cartesian path in a sequence of small joint space motions. These are calculated via IK. This provides a clean separation of Cartesian task-level planning from joint-level implementation. It also highlights the key difference from the resolve-rate controller, which relies on Jacobian-based updates rather than full IK solutions.

IK_ROS Pseudocode

```

INPUT: Robot model, start/target frame source, task parameters
OUTPUT: Robot joint motion and pose error results

BEGIN
    CONNECT to the robot using ur_interface
    READ CurrentJoints
    IF CurrentJoints != HomeJoints THEN
        MOVE robot to HomeJoints
    ENDIF
    COMPUTE HomePose using ForwardKinematics(HomeJoints)

    OUTPUT "1 = sample frames, 2 = teach start"
    INPUT FrameChoice
    IF FrameChoice = 1 THEN
        SET StartPose and TargetPose from the provided sampletransforms
    ELSE
        SWITCH robot to pendant control
        WAIT until I position the robot in freedrive and press Enter
        READ StartJoints
        COMPUTE StartPose using ForwardKinematics(StartJoints)
        SET TargetPose as unknown from now
    ENDIF

    COMPUTE StartAfterPush by moving 3 cm along the tool +x axis at StartPose
    IF TargetPose is unknown THEN
        Compute TargetPose by shifting to the far side using cube width + tool width
    ENDIF
    COMPUTE MidPose and AboveTargetPose by lifting 0.15m in z
    COMPUTE TargetAfterPush by moving 3 cm along tool -x axis at TargetPose
    COMPUTE LastLiftPose by lifting 0.15 m after the second push

    BUILD a waypoint list by interpolating in SE(3) between each consecutive keyframe
    REPORT start pose error using FK(CurrentJoints) vs StartPose

```

```

FOR each WaypointPose in the waypoint list
    SOLVE all IK candidates with urInvKin(WayPointPose)
    PICK the candidate closest to the current joints using wrapped joint distance
    MOVE robot to that joint solution
    UPDATE CurrentJoints
    IF WaypointPose is the far-side contact pose THEN
        REPORT target pose error using FK(CurrentJoints) vs TargetPose
    ENDIF
NEXT

MOVE robot back to HomeJoints
END

```

3: Best IK solution

The provided implementation of the inverse kinematics calculations, in `urInvKin.m`, can return up to 8 possible solutions for the joint angles that will move the end effector of the robot to the desired position and orientation. After determining the possible inverse kinematic solutions, the norm of each solution compared to the current joint angles is computed. The solution with the smallest norm, or the closest to the current robot configuration, is then picked and returned as the best solution.

Pseudocode for picking the best IK solution:

```

INPUTS: g_des = homogenous matrix of desired EE location + orientation, q_ref = current joint angles
OUTPUT: q_best = joint angles for desired end-effector position

possible joint angle solutions ← urInvKin(g_des)
for each solution:
    cost ← norm(possible solution - q_ref)
    if cost < best_cost: assign as best solution
return best joint angle solution q_best

```

4: Simulation Results

Video Links for Demos

RR in RTDE:

<https://drive.google.com/file/d/16X6B3TUASpxfi3bwaWHN8YnvE9eq9bsz/view?usp=sharing>

RR on UR5:

<https://drive.google.com/file/d/1FYQMh2KW9Rv3A3gGAT-YKX8QYoL6xuYt/view?usp=sharing>

IK on UR5:

<https://drive.google.com/file/d/1d5jaDda3MjKu2gZaqD9U8fD0ui-C3039/view?usp=sharing>

Simulation Results

Starting Position Errors

Using the resolved-rate control and ROS the following are the errors between the desired and actual starting positions:

$$d_{SO(3)} = \sqrt{\text{tr}((R - R_d)(R - R_d)^T)} = 2.4475 \text{ e-5}$$

$$d_{\mathbb{R}^3} = \|r - r_d\| = 0.0099$$

Using the resolved-rate control and RTDE the following are the errors between the desired and actual starting positions:

$$d_{SO(3)} = \sqrt{\text{tr}((R - R_d)(R - R_d)^T)} = 2.4475 \text{ e-5}$$

$$d_{\mathbb{R}^3} = \|r - r_d\| = 0.0099$$

Using inverse kinematics control and ROS the following are the errors between the desired and actual starting positions:

$$d_{SO(3)} = \sqrt{\text{tr}((R - R_d)(R - R_d)^T)} = 7.9519 \text{ e-16}$$

$$d_{\mathbb{R}^3} = \|r - r_d\| = 0.0075$$

Final Position Errors

Using the resolved-rate control and ROS the following are the errors between the desired and actual final positions:

$$d_{SO(3)} = \sqrt{\text{tr}((R - R_d)(R - R_d)^T)} = 9.7690\text{e-9}$$

$$d_{\mathbb{R}^3} = \|r - r_d\| = 0.0099$$

Using the resolved-rate control and RTDE the following are the errors between the desired and actual final positions:

$$d_{SO(3)} = \sqrt{\text{tr}((R - R_d)(R - R_d)^T)} = 1.2878\text{e-8}$$

$$d_{\mathbb{R}^3} = \|r - r_d\| = 0.0099$$

Using inverse kinematics control and ROS the following are the errors between the desired and actual final positions:

$$d_{SO(3)} = \sqrt{\text{tr}((R - R_d)(R - R_d)^T)} = 1.0008\text{e-}15$$

$$d_{\mathbb{R}^3} = \|r - r_d\| = 1.662\text{e-}04$$

5: Error Analysis

From the simulation results detailed in the previous section, the inverse kinematics controller proves to get the robot closer to the desired position and orientation for the end effector. The metrics computed above show that the inverse kinematics solution has approximately half the error as the resolved-rate controller when approaching the desired orientation for the end-effector. Additionally, the final position is almost a magnitude closer to the desired final position.

These differences in error magnitude are likely due to how the resolved rate controller is implemented; this method terminates when within the positional tolerance threshold of 0.01 m for the v portion of the twist and rotational tolerance threshold of 0.0873 rad (5 deg) for the ω position of the twist defining the current position of the robot. These thresholds can be decreased to increase the accuracy of the resolved rate controller at the expense of the time required to reach the final position. Meanwhile, the inverse kinematics controller computes the joint angles for the desired position and moves directly to the angles in the defined path. Hence, the only sources of errors here are within the internal robot controller.

6: Safety Considerations

Safety is the number one priority when we execute any robotic task on UR5 or UR5(e). For this project, the robot is moving in proximity to a table, making gentle contact with an object and also moving in the lab where students work. While testing, we ensured a group member was always prepared to activate the emergency stop on the system if needed.

We begin each test by moving the robot to the home position, ensuring predictability of the motion that was determined on the physical robot and real-life workspace. All of the motions are accomplished at low speeds to minimize the forces of impact. We choose the speeds of the joint to provide smooth movement for easier observation, but also to build a safe working environment. While teaching, we make use of the freedrive only. If necessary, we can always pause the script before manually moving the arm. Additionally, the workspace is kept clutter-free with the ability to halt the robot at any point.

Additionally, we ensure safety by using motion limits and posture analysis. Joint collision was prevented through thorough simulation testing to ensure there would be no concern that the robot would collide with itself while running. In resolved rate control, step sizes are limited to ensure stability in updating joints. While in IK, the joints that are close to the current pose are selected to ensure that there are no rapid flips in the wrist or elbow joints.

There are also two hard safety checks in the code: one for singularities and one for table collision, which are both checked prior to moving to the next waypoint in our main control loop for RR and IK. The table collision check pulls the z coordinate value from the next frame to be moved to and compares it with the table surface z coordinate (usually 0 for the robots we tested on), plus an additional 0.02 m safety factor to ensure the end effector never makes contact with the table. The singularity check uses our `urJacobian()` function to find the Jacobian J for the instantaneous joint configuration and finds $\sqrt{\text{abs}(\det(J * J'))}$, which is then used as a manipulability measure to compare to some singularity threshold defined at the beginning of our script. For most tests, we ran with this threshold in the range of 0.01 - 0.05.

Individual Contributions

Vibha Kamath	She worked on implementing, testing, and gathering results from the ROS simulation and the actual robot for both the resolved rate and inverse kinematics controllers, as well as final code documentation, modularity, and commenting. She also contributed to report writing and editing.
Berk Kasimcan	He worked on the version of the simulation code implementation of the first part project. He wrote the Section 1, 2 and 6 of the report. To show a further explanation he wrote the pseudocode for both implementations.
Shreya Terala	She worked on implementing, testing, and gathering results from the ROS simulation and the actual robot for both the resolved rate and inverse kinematics controllers. She also helped with writing the report and editing for clarity and accuracy.
Yvonne Zhang	She worked on implementing, testing and gathering results from the ROS and RTDE simulations, and the actual robot for the resolved rate controllers. She also helped review the report for submission.