MEAM 520 FINAL PROJECT REPORT

Zhanqian Wu, Jie (Jim) Mei, Enlin Gu, Jiani Wang
University of Pennsylvania
Philadelphia, PA 19104
{Zhanqian, jiemei, guenlin}@seas.upenn.edu

1 Abstract & Scope

This project introduces a comprehensive solution for autonomous pick-and-place operations using the Franka Panda robotic arm. The system incorporates advanced features such as collision detection, path planning, and dynamic block manipulation. The collision detection algorithm ensures real-time monitoring and response, crucial for safeguarding the robot and its environment. A hybrid path planning approach, integrating Rapidly-exploring Random Trees (RRT) and linear interpolation, showcases superior efficiency. The system demonstrates success in stacking both static and dynamic blocks, with physical tests revealing reliable performance, despite occasional mismatches. Valuable lessons learned from challenges encountered during a competition underscore the importance of designing robust, adaptable systems for real-world applications.

***K*eywords** Robotic Arm · Path Planning · Autonomous Manipulation

2 Methods

2.1 Objection Detection (AprilTag)

The AprilTag System is employed in our object detection processes, which utilizes a set of distinguishable tags for identification through computer vision algorithms. Each tag carries a unique ID within its pattern, ensuring unambiguous recognition among others within the camera's field of view. The AprilTag library is instrumental in processing the visual data captured by the robot's camera, pinpointing these tags, and calculating their orientations relative to the camera. The `get_H_ee_camera(self)` function is crucial, encapsulating the complex transformations from the camera's coordinate system to the end-effector's reference frame. After the end-effector's pose is ascertained in relation to the camera, it is transformed to the world frame through a sequence of homogeneous transformations, an essential component of the robot's forward kinematics. Once these transformation matrices are determined, they are integrated with the robot's control system, thus enabling precise positioning of the end-effector for efficiently executing tasks such as object retrieval.

2.2 Forward Kinematics

2.2.1 Establishment of the DH coordinate

To calculate forward kinematics for the Panda arm, we first need to set up DH coordinates and get DH parameters, which is the same as we did in LAB 1. The DH coordinate is shown in figure 1 and the DH parameters are shown in table 1.

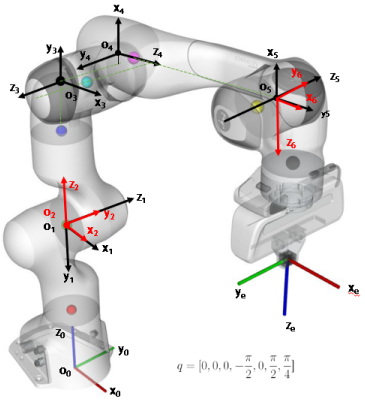


Figure 1: The assigned coordinates, where q=(0,0,0,-90,0,90,45)

	a	α	d	θ
joint0	0	-pi/2	0.141+0.192	θ_1
joint1	0	pi/2	0	θ_2
joint2	0.0825	pi/2	0.121+0.195	θ_3
joint3	-0.0825	-pi/2	0	θ_4
joint4	0	pi/2	0.259+0.125	θ_5
joint5	0.088	pi/2	0	θ_6
joint6	0	0	0.051+0.159	$\theta_7 - pi/4$

Table 1: Standard DH parameters

2.2.2 Calculating Joint positions and transformation matrix

Then, based on the DH table, we can get the transformation matrix from frame i-1 to frame i $\mathbf{T}_i^{i-1}$ in Eq (1)

$$T_i^{i-1} = A_i\left(\theta_i, d_i, a_i, \alpha_i\right) = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

And we can get the transformation matrix from base 0 to e as shown in (2)

$$T_e^0 = T_1^0 T_2^1 T_3^2 T_4^3 T_5^4 T_6^5 T_e^6 \tag{2}$$

Next, we will calculate the joint positions, here we will discuss two different scenarios.

(1) if the center of the joint coincides with the origin of its coordinates, the positions ***i*** is shown in Eq (3)

$$T_n^0 = T_1^0 T_2^1 ... T_n^{n-1} \tag{3}$$

(2) if the center of the joint does not coincide with the origin of its coordinates, that is to say, the joint center has an offset in its attached coordinates, we set the offset as (0,0,z_{offset}),the problem can be converted to determine the coordinates of a point with coordinates (0,0,z_{offset}) in the i-coordinate system under the world coordinate system. Hence, the positions ***i*** is shown in Eq (4)

$$P^0 = T_i^0 P^i \tag{4}$$

2.3 Inverse Kinematics

2.3.1 Angular Velocity Tracking

First of all, we want to calculate target angular velocity given the current and the target end-effector orientations. The rotation transformation between target and current configuration can be obtained using the Eq (5)

$$R_{error} = R_{Current}^T R_{target} \tag{5}$$

Given a single rotation R_{error}, the corresponding axis of rotation with magnitude sin θ can be found by first computing the skew symmetric part of R as

$$S = \frac{1}{2}\left(R - R^T\right) \tag{6}$$

$$[a]_\times = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{7}$$

By this, we can extract the coefficients of the corresponding vector under the skew operation.

And finally, we need to express the axis relative to the world frame using the follwing (8)

$$\vec{\omega}_{1,2}^0 = R_1^0 \vec{\omega}_{1,2}^1 \tag{8}$$

If we want to calculate magnitude of the rotation angle between two homogeneous transformations (G and H). We will start by calculating the rotation matrix as Eq 5

Then, the magnitude of the angle can be obtained by using trace in Eq (9)

$$|\theta| = \arccos\left(\frac{\mathrm{tr}(R) - 1}{2}\right) \tag{9}$$

2.3.2 Velocity IK/Primary Task

To calculate the velocity forward kinematics, it is essential to calculate the Jacobian J for the Panda arm with 7-DOF, which can be expressed as Eq 10:

$$J = \begin{bmatrix} \mathbf{J_v} \\ \mathbf{J}_\omega \end{bmatrix} \tag{10}$$

where:

- $\mathbf{J_v}$ is the top 3×7 matrix portion of the Jacobian which pertains to linear velocities of the end effector.
- $\mathbf{J}_\omega$ is the bottom 3×7 matrix portion of the Jacobian which pertains to angular velocities of the end effector.

Thus, for each joint i, the linear and angular velocity components are shown in Eq 11 and Eq 12,respectively.

$$\mathbf{J_v}[:, i] = z_{i-1} \times (O_{EE} - O_{i-1}) \tag{11}$$

$$\mathbf{J}_\omega[:, i] = z_{i-1} \tag{12}$$

Therefore, the linear velocity Jacobian matrix becomes the form shown in Eq 13:

$$\mathbf{J} = \begin{bmatrix} z_0 \times (O_{EE} - O_0) & z_1 \times (O_{EE} - O_1) & \cdots & z_6 \times (O_{EE} - O_6) \\ z_0 & z_1 & \cdots & z_6 \end{bmatrix} \tag{13}$$

It should be noted that, if we choose pseudo jacobian

$$dq = J^+ \cdot V \tag{14}$$

if we choose transpose jacobian

$$dq = J^T \cdot V \tag{15}$$

where V denotes ***[displacement,axis]***

2.3.3 Null-Space Velocity IK / Secondary Task

In the previous Section, we only controlled the end-effector position but not the orientation, by using secondary task in this part, we will add the angular velocity target part to achieve the full velocity inverse kinematics for tracking. Null-space velocity inverse kinematics (IK) is a technique used in robotics to compute joint velocities that achieve a desired end-effector motion while considering additional objectives or constraints. The "null-space" refers to the space of joint configurations that do not affect the end-effector position or orientation. We can get null space in the following Equation.

$$null = \left(I - J^{+}J\right) \tag{16}$$

And then, add the null (second task) to the primary task to get the joint velocity.

$$\dot{\vec{q}} = J^{+}\vec{\xi} + \left(I - J^{+}J\right) b \tag{17}$$

In this paper, the second task space is to keep the joints centered. We can describe it with the following equation.

$$\text{offset} = \frac{2 \cdot (q - \text{IK.center})}{\text{IK.upper} - \text{IK.lower}} \tag{18}$$

$$dq = \text{rate} \cdot (-\text{offset}) \tag{19}$$

2.3.4 Iterative Optimization Function

To solve the inverse kinematics numerically for a 7-degree-of-freedom robotic arm, we shall use the following gradient descent method:

1. **Define the Goal**: define the target position and orientation of the end-effector.
2. **Define the Error Function**: Error function ***IK.distance and angle(...)*** and ***IK.displacement and axis(...)*** will quantify the difference between the current position/orientation of the end-effector and the target position/orientation.
3. **Calculate the Gradient:** Compute the gradient of the error function with respect to the joint angles.
4. **Initialize Joint Angles:** Choose an appropriate initial set of joint angles as a starting point. We start by setting a configuration seed at ***(0,0,0,-pi/2,0,pi/2,pi/4)***
5. **Iterative Optimization**: Minimize the error function by iteratively updating the joint angles. In each iteration, multiply the gradient by a learning rate and add the result to the current joint angles using Eq (20).

$$q = q + \alpha \nabla q \tag{20}$$

6. **Termination Condition:** stop when the error is sufficiently small or after a maximum number of iterations.

This gradient descent process can be visualized in a flowchart shown in Fig 2a.

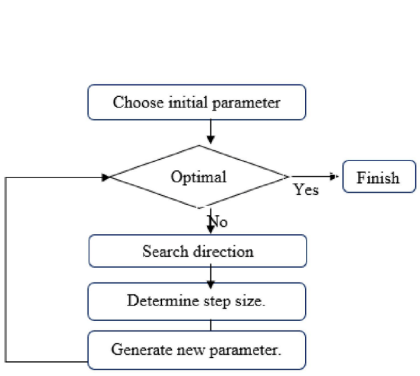
2.4 Collision Detection

We employ a 2D spatial analysis to prevent the robot's gripper from colliding with surrounding objects, ensuring safe operation within its environment. As shown in Fig 2b, the gripper is represented as an array of eight small circles, each corresponding to a potential contact point that allows for precise spatial positioning. This discretization into multiple points simplifies the model while maintaining a high-resolution representation of the gripper's spatial occupancy. Each block is represented by a circle with specified attributes $[x, y, \text{Radius}, \text{Orientation}]$, which denote their respective positions, sizes, and orientations in the Cartesian plane.

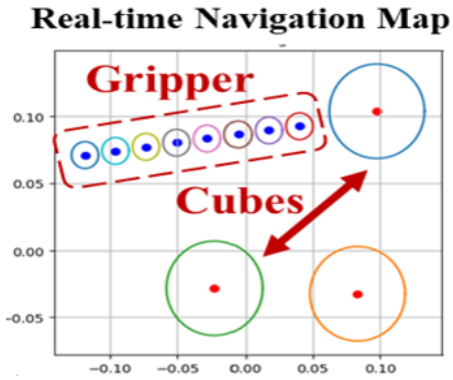
The collision detection operates on the principle of intersecting circles detection, a method chosen for its computational efficiency and robustness in handling 2D geometric shapes. The algorithm proceeds as follows:

1. Iteration through the `obstacle_map`, which contains the positional data for each block in the environment.
2. Concurrent iteration over the gripper's circles, evaluating their positional relationship with the obstacles.
3. Calculation of the Euclidean distance between the centers of each gripper circle and each obstacle.
4. Summation of the radii of the gripper circle and the obstacle (`radius_sum`) to define a collision boundary.
5. Comparison of the calculated distance with the `radius_sum`. If the distance is less than or equal to `radius_sum`, a collision is detected.

Upon detection, a counter is incremented, which can be used for logging collision incidents and triggering responsive actions. The implementation of this collision detection algorithm allows for real-time monitoring and response, crucial for autonomous navigation and manipulation tasks. It ensures that the robotic system can recognize and react to potential collisions, thereby safeguarding the robot and its environment from unintended contact.



(a) Gradient decent algorithm



(b) Real-time Navigation Map

Figure 2: Gradient Descend and Real-time Navigation Map

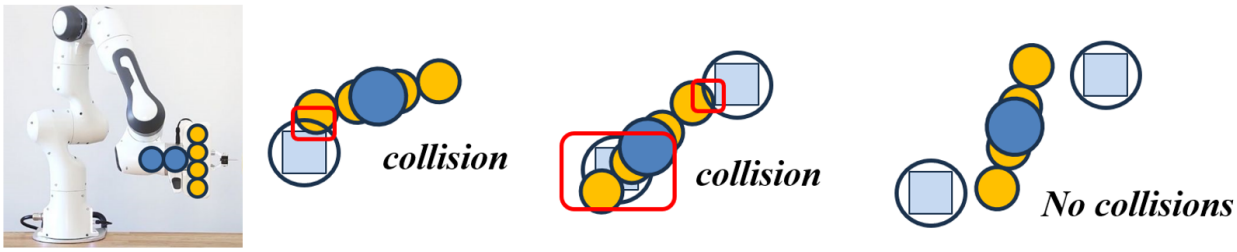


Figure 3: Collision Detection Method

2.5 Path planning

2.5.1 Linear Interpolation

Linear interpolation is a common technique used in robotics to generate smooth and controlled motion between two points. In the Franka Panda Robotic arm, linear interpolation can be applied to interpolate between two joint configurations or Cartesian positions.

For joint interpolation between an initial joint configuration:

$$\theta_{i,j} = (1 - \alpha) \cdot \theta_{\text{initial},j} + \alpha \cdot \theta_{\text{final},j} \tag{21}$$

where:

- $theta_{i,j}$ is the interpolated joint configuration at step
- j represents the joint index (1 to 7).
- α is the interpolation parameter that varies from 0 to 1 as you move from the initial to the final configuration.

For joint interpolation, we would linearly interpolate each joint angle between start configuration and target configuration.

2.5.2 Rapidly-exploring Random Trees (RRT)

The RRT planner is a path planning algorithm that rapidly explores the space by randomly building a space-filling tree. It's particularly effective in high-dimensional spaces. The pseudocode shown in Algorithm 1 describes an RRT planner for the Panda robot. The algorithm initializes two trees, one starting at the initial configuration q_{start} and one at the goal configuration q_{goal}. It iterates a specified number of times, generating a random configuration in the free space Q_{free}, finding the closest existing node in both the start and goal trees, and attempting to add a new node if there is no collision on the path. The process continues until a connection between T_{start} and T_{goal} is established or the maximum number of iterations is reached.

Algorithm 1 Rapidly-exploring Random Tree (RRT) Planner for Panda Robot

$T_{\text{start}} \leftarrow \{(q_{\text{start}}, \emptyset)\}$, $T_{\text{goal}} \leftarrow \{(q_{\text{goal}}, \emptyset)\}$
for $i = 1$ to N_{iter} **do**
    $q \leftarrow$ random configuration in Q_{free}
    $q_a \leftarrow$ closest node in T_{start}
    if NOT collide(q, q_a) **then**
        Add (q, q_a) to T_{start}
    end if
    $q_b \leftarrow$ closest node in T_{goal}
    if NOT collide(q, q_b) **then**
        Add (q, q_b) to T_{goal}
    end if
    if q connected to T_{start} and T_{goal} **then**
        break
    end if
end for

Algorithm 2 Rapidly-exploring Random Tree (RRT) Algorithm

procedure RRT($map, start, goal, N\_samples$)
    if IsRobotCollided($map, start$) or IsRobotCollided($map, goal$) **then**
        return $\emptyset$
    end if
    if not IsPathCollided($map, start, goal$) **then**
        return $[start, goal]$
    end if
    $lowerLim, upperLim \leftarrow$ Joint limits
    $tree \leftarrow [\{'config' : start, 'parent' : None\}]$
    for $n \leftarrow 1$ **to** $N\_samples$ **do**
        $q\_rand \leftarrow$ Random configuration within limits
        if IsRobotCollided(map, q_rand) **then**
            continue
        end if
        $nearest\_node \leftarrow$ Find nearest node in $tree$ to q_rand
        if IsPathCollided($map, nearest\_node.config, q\_rand$) **then**
            continue
        end if
        Add q_rand to $tree$
        if not IsPathCollided($map, q\_rand, goal$) **then**
            Add $goal$ to $tree$
            $path \leftarrow$ Construct path from start to goal in $tree$
            return $path$
        end if
    end for
    return $\emptyset$
end procedure
function IsRobotCollided(map, q)
    Check if robot configuration q collides with any obstacle in map
    return True or False
end function
function IsPathCollided($map, q1, q2$)
    Check if path between $q1$ and $q2$ collides with any obstacle in map
    return True or False
end function

3 Game Strategy

3.1 Static-Block Strategy

Our code structure can be seen in Algorithm 3. The strategy for static block gripping and placing is very straightforward: we first need to get the frame of one block, and generate a path to grip it; after that, generate another path to the proper position and release the block. However, there are a lot of issues to consider in practice.
Firstly, there exists a possibility that the blocks are not placed properly so the gripper pose should be carefully determined to avoid collision with other blocks. Also, though some gripping poses are collision-free, some may have longer paths that may take more time. In this case, we introduce our **Sequencing Algorithm** to determine the order of gripping blocks and corresponding gripper orientation. We introduce a loss function that describes the length of the path in joint space as 'energy':

$$Cost = \mu\frac{1}{2}||q_{current} - q_{target}||, \text{ where } \mu = 1 \text{ when collision-free and otherwise } \mu = -1$$

Then we can sort the four blocks and corresponding gripping poses in sequence and pick the block with least cost. With the sorting algorithm, we were able to achieve a more efficient task. The summary of our algorithm is shown in figure 4.

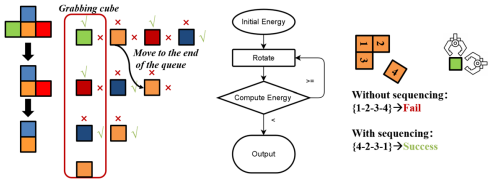


Figure 4: Sequencing Algorithm & Gripper orientation

Besides, path planning using RRT or APF is very time-consuming. It is clear that the longer the path is, the greater the time it takes for the robot to find a path. As the time complexity of RRT is $O(nlogn)$ and for APF is $O(n^D)$, which means that planning time increases dramatically as the path lengthens. To avoid this, we can add several "Control Points" along the path and use linear motion between them, which reduces the need for planning. We only use planning for the start and end points of the trajectory, where collisions are more likely to occur during grabbing or placing the block. By choosing proper control points, we can ensure a collision-free linear motion, while generating a proper trajectory at the grabbing and gripping process. Our method is shown in figure 5.

Algorithm 3 Static Tasks Pseudocode

```
MOVE_ TO_ STATIC _INITIAL_ SEARCH_ POSITION(arm, team_ color)
Block _ num ← len(detector.get_detections())
Remaining block=Block _ num
for n = 1 to Block_num do
    all_block_pose←[pose in detector.get_detections()]        ▷ All block pose we can detect now
    if len(all_block_pose)>1 then
        xvalue, yvalue, zvalue ← extract_pose_values(all_block_pose)        ▷ We only need x,y,z
        Pose ← The no-collision gripper orientation pose with minimum cost by our Sequencing Algorithm
    else
        Pose ← all_block_pose[0]        ▷ Only one block left
    end if
    H_ee_camera ← EE_cam_offset(detector.get_H_ee_camera(), 'x','y')        ▷ Correct camera frame error
    Block_pose ← compute_object_pose(Pose,H_ee_camera)        ▷ Compute block pose in world frame
    STATIC_PRE_GRAB(arm,Block_pos)        ▷ Move to pre-grab pose
    STATIC_GRAB(arm,Block_pos)        ▷ Plan & grab the block
    STATIC_PLACE(arm)        ▷ Safe place block
    STATIC_LEAVE(arm)        ▷ Safe leave and move to detect position for the next block
end for
Remaining block -= 1
if Remaining block == 0 then break the loop and go down to dynamic part
end if
```

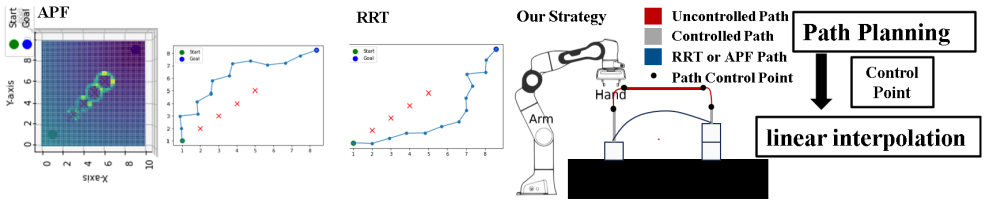


Figure 5: Our method of Path Planning **VS** APF&RRT

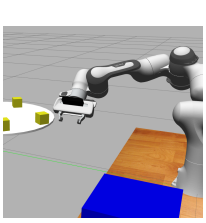
3.2 Dynamic-Block Strategy & Method

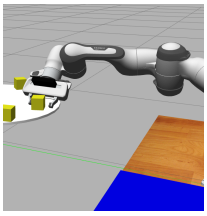
After completing the the stacking task for the static blocks, we will turn to the dynamic blocks. For the dynamic blocks, the ideal approach consists of a series of steps designed to efficiently iterate between a robot and a set of blocks. Initially, the process begins with the detection of the blocks, which returns the layout of the blocks place on the turntable. From this layout, the team can determine the center of the "circle" formed by the dynamic blocks. By measuring the velocity of the turntable in the lab, the angular velocity of a particular block can then be determined. Following the detection, the robot computes the blocks' poses within its own world frame, which helps establish the blocks' locations. The cube trajectory is given by the equations shown in Eq 22.

$$\begin{aligned} x &= c_x + r \cdot \cos(a + \omega t) \\ y &= c_y + r \cdot \sin(a + \omega t) \\ z &= z_0 \\ \phi &= \phi_0 + \omega t \end{aligned} \tag{22}$$

Using these equations, the coordinates of the blocks can be estimated. Thus, the team can use `IK_velocity_null.py` and `follow.py` to follow and then grab the blocks. The final step of the strategy is to place the dynamic blocks on to the top of the stacked tower.

However, due to the limitations of the run-time of the gradient-descent-based IK solver and the noise produced by the vision system, it is hard for the team to follow and predict the locations of the dynamic blocks accurately and efficiently. So, in the actual implementation, the team abandoned the use of the vision system. For the grabbing of the dynamic blocks, the setup involves the calculation of pre-grab and grab positions. This is followed by solving for the corresponding poses using Inverse Kinematics (IK). Consequently, the robot arm is moved to the pre-grab position and subsequently to the grab position. The end-effector was rotated 45 degrees about the y axis of the end-effector's frame so that it can hinge the blocks on the turntable. Take the blue side as an example, the pre-grab and grab positions and the corresponding transformation matrices for blue team are shown in Fig 6 .





```
pregrab_blue = np.array([[ 0.70710678, 0, 0.70710678,  4e-4],
                         [0, -1, 0, -0.58],
                         [0.70710678, 0, -0.70710678,  0.215],
                         [ 0.,  0., 0., 1. ]]
                         )

grab_blue = np.array([[ 0.70710678, 0, 0.70710678,  4e-4],
                      [0, -1, 0, -0.75],
                      [0.70710678, 0, -0.70710678,  0.215],
                      [ 0.,  0., 0., 1. ]]
                      )
```

Figure 6: The pre-grab and grab positions and the corresponding transformation matrices, respectively, from left to right

Once the arm reached the grab position, the robot actuates the gripper to open and close continuously until a block is securely grasped. The next step is to move the robot arm to a safe pre-place position, which is carefully selected to avoid any interference with the existing stack, facilitated by collision detection mechanisms mentioned in Sec 2. Lastly, the block was moved to the place position and then placed onto the top of the tower. Therefore, in this case, the place position's x-y coordinates are nearly identical with those in the static tasks besides little errors and inaccuracy. Thus, the offsets are needed. What is more, as the grabbing of the blocks tilted the end-effector with 45 degrees, we need to compensate this angle when placing the blocks. Therefore, including the offset and assuming the stack layer is zero (i.e. no static blocks are stacked), the transformation matrix of the place position (blue side example) is shown in Fig 7. Additionally, for the pre-place position, it is simply eight-block-higher than the placing platform (40cm above

```
Dyn_Block_target_robot_frame = np.array([
        [0.70710678, -1.86573745e-09, 0.70710678, 5.62000000e-01],
        [-1.86573734e-09, -1.00000000e+00, -2.44297205e-09, -1.69000000e-01],
        [0.70710678, 2.44297209e-09, -0.70710678, 2.50000000e-01],
        [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 1.00000000e+00]
    ])
```

Figure 7: Transformation matrix of the place position, assuming stack_layer=0

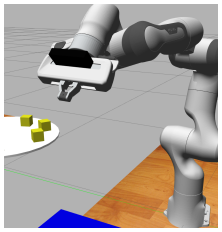


Figure 8: Pre-place position

the platform and 60cm above the table). The position in simulator is shown in Fig 8.

The code structure is shown in Algorithm 4. In this pseudo-code, the functions like `Dynamic_place` and `Dynamic_leave` are predefined functions in a separate script called `Dynamic_Basic_actions.py`. The main functionalities are to bring in the blocks smoothly with a lower speed when it is close to the target place position and leave the placing position safely.

Algorithm 4 Dynamic Tasks Pseudocode

ARM.OPEN_GRIPPER
ARM.SAFE_MOVE_TO_POSITION(Pre_grab_pos)
$t \leftarrow$ TIME_IN_SECONDS
while not rospy.is_shutdown() **do**
    ARM.SAFE_MOVE_TO_POSITION(grab_pos)
    function IS_BLOCK_GRABBED(arm)
        $gripper\_state \leftarrow arm.get\_gripper\_state()$
        $positions \leftarrow gripper\_state['position']$
        $width \leftarrow$ **abs**$(positions[1] + positions[0])$
        print("The grip width is: ", width)
        return $0.038 \leq width \leq 0.051$ ▷ Determine whether the block is grabbed successfully
    end function
    while not rospy.is_shutdown() **do**
        GRIPPER_CONTROL(arm, "close")
        if IS_BLOCK_GRABBED(arm) **then**
            print("Block grabbed successfully!")
            ARM.SAFE_MOVE_TO_POSITION(Pre_place_pos)
            DYNAMIC_PLACE
            DYNAMIC_LEAVE
            break
        else
            print("Grab unsuccessful, retrying...")
        end if
        GRIPPER_CONTROL(arm, "open")
        ROSPY.SLEEP(7.0)
    end while
    ARM.SAFE_MOVE_TO_POSITION(Pre_grab_pos)
    $Stacked\_Layers \leftarrow Stacked\_Layers + 1$
end while

4 Evaluation

4.1 Static Blocks

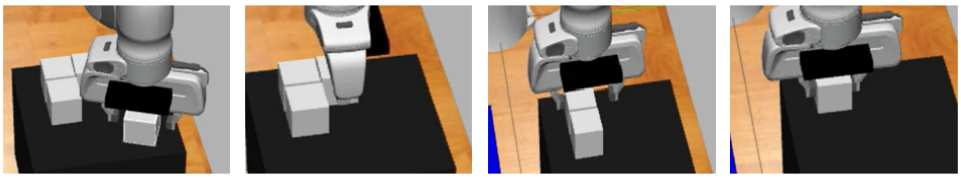


Figure 9: Collision Detection,randomly placing cubes close to each other to verify that the algorithm can efficiently grab all the cubes

For the static blocks, the test concentrated basic actions & path planning and collision detection on the simulation environment.

First, we tested collision detection by randomly placing cubes close to each other as shown in Fig 9 to verify that the algorithm can efficiently grab all the cubes. We performed three random test whose results are listed in the Tab 2, in which the robotic arm successfully clamped all the blocks, showing the effectiveness of the obstacle detection algorithm.

Table 2: Three random Collision Detection Test

Test Num	**Cube 1**	**Cube 2**	**Cube 3**	**Cube 4**
1	Yes	Yes	Yes	Yes
2	Yes	Yes	Yes	Yes
3	Yes	Yes	Yes	Yes

Then, we tested the three path planning method mentioned above. It should be noted that, our strategy as is shown in Fig 5 is to use RRT at the beginning and end stages and linear interpolation in the middle. Among the three algorithms compared in the Tab 3, Our's algorithm distinguishes itself with a significantly lower average planning time of 0.89, showcasing its efficiency in the planning phase. Although it has the highest average execution time at 10.87, the algorithm achieves the task with the lowest total time of 11.76, emphasizing its overall effectiveness. While trade-offs exist, the efficiency gains in planning and total time make Our's algorithm a compelling choice, particularly in scenarios prioritizing faster planning and task completion.

Table 3: Path Planning Performance Comparison

Algorithm	**Average Planning Time**	**Average Execution Time**	**Total Time**
APF	5.17 ↑	9.32 ↓	14.49
RRT	6.48 ↑	10.61	17.09 ↑
Our's	0.89 ↓	10.87 ↑	11.76 ↓

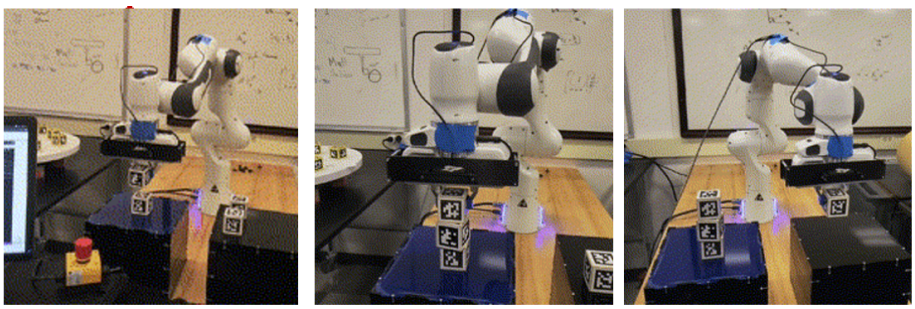


Figure 10: Static Cubes Stacking Task Test

Finally, we performed static block stacking task both on simulator and physical robot. Interestingly, in the virtual machine our path planning runs faster, while in the physical robotic arm, the robotic arm runs faster motions, so there will be a noticeable difference in run time between the two. our average simulation time for four static cubes is 167.7s, while our average physical time is 152.8s.

4.2 Dynamic Blocks

For the dynamic blocks, the test concentrated on the physical robot test. This is because the blocks in the simulator would stop moving when the end-effector grabs the block unsuccessfully and hit the edges of the blocks.

The performance of the dynamic task is evaluated using a set of defined performance metrics. The first metric assesses the accuracy of the robot's positioning by comparing the predefined grab and place positions against those achieved during physical testing; any mismatches here are critical and must be carefully analyzed. The second metric measures the time efficiency of the system by recording the duration needed to complete each cycle of picking and placing the dynamic block. The third metric is the success rate of the robot when operating in the real setting, acknowledging that the simulator provides a limited environment and may not capture all real-world complexities. The test results are shown in Tab 4.

Table 4: Dynamic tasks test results

Test number	Poses Mismatch (m)	Time for one cycle (s)	Success Rate
1	0.013	78	50%
2	0.021	88	33.3%
3	0.009	47	100%
4	0.012	74	50%
5	0.017	98	25%
6	0.01	46	100%
7	0.012	90	33.3%
8	0.011	47	100%
9	0.018	77	50%
10	0.017	73	50%
Avg.	0.014	71.8	52%

From the test results, the poses mismatches of the grabbing position is about 0.014m, which is in an acceptable range. The position is acceptable as long as the blocks can "slide" into the gripper successfully. Note that a successful grab means the gripper grasp the two surfaces of the block, and the success rate is calculated by dividing the number of success grab (which is obviously one) by the number of attempts. One example of successful grab is shown in Fig 11. Together with the static tasks, within 297 seconds, we successfully stacked 6 blocks (4 static + 2 dynamic) in the latest physical test. The stacked tower is shown in Fig 12. Note that one of the yellow block is the static block since there are no enough static blocks during the test day.

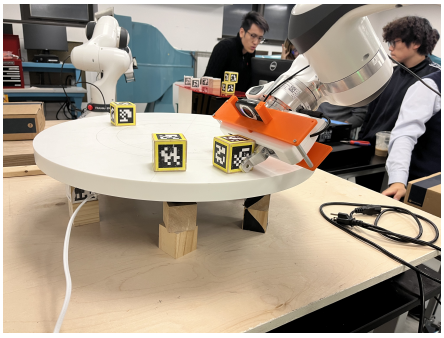


Figure 11: Successful grab of the dynamic blocks

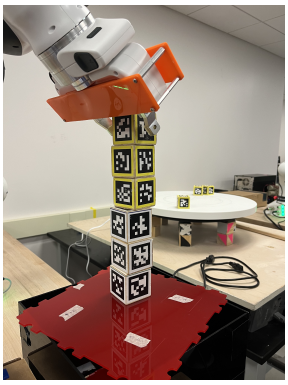


Figure 12: Tower with 6 blocks finished in 297 sec

5 Analysis

Test Video:https://youtu.be/gWPJHwZ5hco.

The evaluation illuminates the performance of our developed code, as demonstrated in the provided video references. During the lab trials, the code exhibited efficiency, successfully stacking four static blocks within 2.5 minutes, and consequently stacking two dynamic blocks within another 2.5 minutes (6 blocks within 297s). Notably, the performance in the competition varied due to unforeseen issues with the Z-axis accuracy, underscoring the importance of anticipating potential subsystem failures.

The lab trials showcased a high success rate (100% for static blocks and 52% for dynamic blocks), picking and placing blocks with remarkable accuracy. However, the competition revealed the critical need to consider subsystem reliability under diverse conditions. Despite our meticulous design and testing, the vision system's performance deviation during the competition led to inaccuracies in the Z-axis alignment. This resulted in collisions, emphasizing the necessity of contingency planning for unexpected scenarios.

This experience emphasizes a valuable lesson: the assumption that all subsystems will function flawlessly may not hold. The Z-axis misalignment issue, not encountered during lab testing, underscored the importance of robustness testing under various conditions. Future iterations of the system should incorporate enhanced fault tolerance and adaptability to ensure reliable performance across diverse environments. This highlights the dynamic nature of real-world applications and the need for continuous refinement and adaptation in robotic systems.

In summary, while the developed code demonstrated impressive performance in controlled lab settings, the competition environment revealed the importance of anticipating and mitigating unexpected challenges. This underscores the iterative nature of robotics development, emphasizing the need for adaptability and robustness to ensure success in real-world applications.

6 Conclusion (Lessons Learned)

A key takeaway from this project is the necessity to anticipate potential subsystem failures in real-world scenarios. The discrepancy in Z-axis accuracy, experienced during a competition, serves as a valuable lesson in designing resilient systems that can gracefully handle unexpected challenges. This insight contributes to the ongoing refinement of the robotic system for improved reliability.

7 Appendix

References

[1] Franka Control Interface Documentation — Franka Control Interface (FCI) documentation. (n.d.). Frankaemika.github.io. https://frankaemika.github.io/docs/

[2] Zhao, A. Z. (2018, September 13). engr027-project1. GitHub. https://github.com/zhaolebor/engr027-project1

[3] Robot and interface specifications — Franka Control Interface (FCI) documentation. (n.d.). Frankaemika.github.io. https://frankaemika.github.io/docs/control_parameters.html

[4] Mirrazavi Salehian SS, Figueroa N, Billard A. A unified framework for coordinated multi-arm motion planning. The International Journal of Robotics Research. 2018;37(10):1205-1232. doi:10.1177/0278364918765952