MEAM520 Final Report

Archit Hardikar, Mingyan Zhou, Jiacheng Song, Haochen Wang

April 2022

1 Methods

The method section only includes all the theories and knowledge background for controlling the robot to finish the task. Detailed code implementation is in the next section.

IK solver and homogeneous transformation are two main technical components for grabbing and placing blocks. The complete process are the following:

- **Step1**–get all tag information under world frame(Robot frame): Function "`detector.get_detections()`" returns all the homogeneous transformation matrices from tag 0 to tag 12 under camera frame (H^c_{tagi}). By using H^c_{tagi}, Function "`tag.getTagCenterPos()`" firstly computes all H^w_{tagi} which are all the homogeneous transformation matrices from tag 0 to tag 12 under world frame. Then, it move all tag frames to the center of the block and returns H^w_{tagic}.
- **Step2**–create proper virtual frame: Virtual frames will be created, and the end-effector can grab and place the block smoothly when it match up with all ordered virtual frames in sequence.Virtual frames need to be properly designed. Two important notes to guarantee a good virtual frame. First, it is crucial to figure out how is the end-effector frame(ee frame) being placed on the real end-effector of the robot so that the meaning of the positive direction of x,y, and z axes of the end-effector frame are clear. Secondly, designing proper grabing and placing strategy for different blocks by the position of the white surface. Detailed procedure will be disccused in the following subsection.
- **Step3**–generate H_{target}: By the relation between tag frames and corresponding crated virtual frames, one can generate every H_{rotate} and post-multiply H^w_{tagic} by it to get H_{target}.
- **Step4**–control the robot by IKsolver: Find a good seed and input H_{target} to IKsolver to get a configuration solution q. Finally, use `arm.safe_move_to_position(q)` to control the robot.

1.1 Homogeneous Transformation Matrices [1]

"`tag.getTagCenterPos()`" uses the equation

$$H^w_{tagic} = H^w_{tag0}(H^c_{tag0})^{-1}H^c_{tagi}H^{tagi}_{tagic}$$

to return all H^w_{tagic}. The first term can be computed directly because tag 0 frame and world frame are fixed.

$$H^w_{tag0} = ([1,0,0,-0.500],[0,1,0,0],[0,0,1,0],[0,0,0,1])$$

The second and the third term are obtained from "`detector.get_detections()`". The last term can move the frame to the center of the block.

$$H^{ti}_{tagic} = ([1,0,0,0],[0,1,0,0],[0,0,1,-0.02],[0,0,0,1])$$

1.1.1 Pick and place static blocks

The overall controlling mode of each pick and place iteration of the robot follows the procedure:

Tag 6 and Tag 1-4: move to static-up position, line down, line up, move to tower-up position, line down, line up.

Tag 5: move to static-up, line down, line up, rotate 90 degrees, line down, execute side case procedure.

- Top case for white surface(Tag 6):

  Move to static-up point: $H_{target} = H^w_{tag6c}H^{tag6c}_{tag6c-up}H_{rotate}$ where

  $$H^{tag6c}_{tag6c-up} = ([1,0,0,0],[0,1,0,0],[0,0,1,-0.1],[0,0,0,1])$$

  and

  $$H_{rotate} = ([-1,0,0,0],[0,1,0,0],[0,0,-1,0],[0,0,0,1])$$

  Note: Theoretically there are multiple choices for designing the virtual frames and multiple corresponding H_{rotate}. Because some of them might be hard for end frame to reach which can cause IK solver failure, it is necessary to

change the direction of x and y axes of the virtual frame to solve it. However, instead of computing new H_{rotate} and applying it, there is a simpler way to do it. The only joint that will result in bad joint angle is the joint 6(end effector joint variable). Each time a bad end-effector joint variable shows, simply adding or subtracting n*pi to this joint variable can solve it. Specifically, the relation between the virtual frame(ee frame) and the surfaces of the block will change after changing the end-effector joint variable, but it is still feasible to use old H^w_{twr} because for Tag 6 case white surface is always facing up(Switch end-effector joint variable by n*pi only change x and y axes, z axis remains the same).

Line down movement: $H_{target} = H^w_{tag6c} H_{rotate}$

Line up movement: Use same q obtained in static-up point.

Move to tower-up point: $H_{target} = H^w_{twr} H^{twr}_{up}$ where

$$H^w_{twr} = ([1, 0, 0, .562], [0, -1, 0, isRed * .169], [0, 0, -1, .2 + (i + 1) * 0.05 - 0.015], [0, 0, 0, 1])$$

and

$$H^{twr}_{up} = ([1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, -0.1], [0, 0, 0, 1])$$

Note: $isRed$ is one for red team and negative one for blue team, i is a counter for the floor.

line down movement: $H_{target} = H^w_{twr}$

line up movement: Use same q obtained in tower-up point.

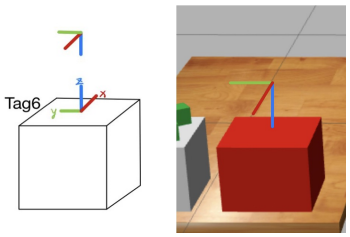

Figure 1: Virtual frames for static-up point and tower-up point of Tag 6

- Side case for white surface(Tag 1-4):

  Move to static-up point: $H_{target} = H^w_{tag1-4c} H^{tag1-4c}_{tag1-4c-up} H_{rotate}$ where

  $$H^{tag1-4c}_{tag1-4c-up} = ([1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, -0.1], [0, 0, 0, 1])$$

  and

  $$H_{rotate1} = ([0, 1, 0, 0], [1, 0, 0, 0], [0, 0, -1, 0], [0, 0, 0, 1])$$
  $$H_{rotate2} = ([0, -1, 0, 0], [-1, 0, 0, 0], [0, 0, -1, 0], [0, 0, 0, 1])$$

  Note: Similar to Tag 6 case, theoretically we have multiple choices for designing the virtual frames and multiple corresponding H_{rotate}. However, we cannot use the simple way to solve it as before, because white surface relates to x and y direction of ee frame for side case. After switching the ee joint variable by n*pi, using old H^w_{twr} will cause white surface facing downwards after placing it. Thus, we need to figure out corresponding H^w_{twr} for different H_{rotate}.

  Line down movement: $H_{target} = H^w_{tag1-4c} H_{rotate}$

  Line up movement: Use same q obtained in static-up point.

  Move to tower-up point: $H_{target} = H^w_{twr} H^{twr}_{up}$ where

  $$H^w_{twr1} = ([0, 1, 0, .562], [0, 0, 1, isRed * .169], [1, 0, 0, .2 + (i + 1) * 0.05 - 0.015], [0, 0, 0, 1])$$

  $$H^{twr1}_{up} = ([1, 0, 0, 0.15], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1])$$

  and

  $$H^w_{twr2} = ([0, -1, 0, .562], [0, 0, 1, isRed * .169], [-1, 0, 0, .2 + (i + 1) * 0.05 - 0.015], [0, 0, 0, 1])$$

  $$H^{twr2}_{up} = ([1, 0, 0, -0.15], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1])$$

  line down movement: $H_{target} = H^w_{twr}$

  line up movement: Use same q obtained in tower-up point

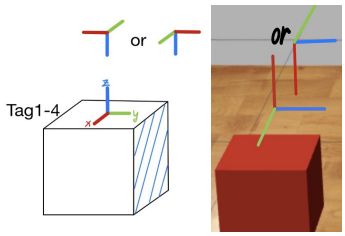

Figure 2: Virtual frames for static-up point and tower-up point of Tag 1-4

- Bottom case for white surface(Tag 5):

  Move to static-up point: $H_{target} = H^w_{tag5c} H_{rotate}$ where $H_{rotate} = H_1 H_{rot45} H_{up}$

$$H_1 = ([1,0,0,0],[0,-1,0,0],[0,0,-1,0],[0,0,0,1])$$

$$H_{rot45} = ([(1/2)^{0.5},0,(1/2)^{0.5},0],[0,1,0,0],[-(1/2)^{0.5},0,(1/2)^{0.5},0],[0,0,0,1])$$

$$H_{up} = ([1,0,0,0.1*(1/2)^{0.5}],[0,1,0,0],[0,0,1,-0.1*(1/2)^{0.5}],[0,0,0,1])$$

  Line down movement: $H_{target} = H^w_{tag5c} H_{rotate} H_{down}$ where

$$H_{down} = ([1,0,0,-0.1*(1/2)^{0.5}],[0,1,0,0],[0,0,1,0.1*(1/2)^{0.5}],[0,0,0,1])$$

  Line up movement: Use same q obtained in static-up point.

  Rotate 90: $H_{target} = H^w_{tag5c} H_{rotate} H_{rot90}$ where

$$H_{rot90} = ([cos(-pi/2),0,sin(-pi/2),0],[0,1,0,0],[-sin(-pi/2),0,cos(-pi/2),0],[0,0,0,1])$$

  Line down movement: $H_{target} = H^w_{tag5c} H_{rotate} H_{rot90} H_{down2}$

$$H_{down2} = ([1,0,0,0.1*(1/2)^{0.5}],[0,1,0,0],[0,0,1,0.1*(1/2)^{0.5}],[0,0,0,1])$$

  Finally, treat it as a side case and follow side case procedure.

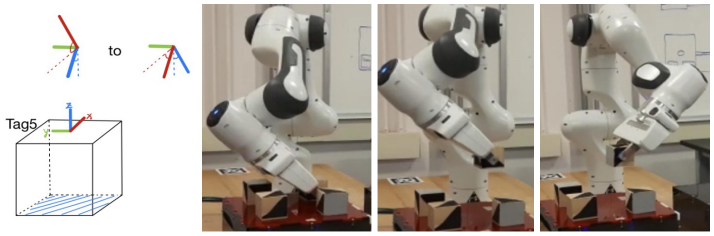

Figure 3: Virtual frames for changing Tag 5 to a side case

1.1.2 Pick and place dynamic blocks

The overall controlling mode of each pick and place iteration of the robot follows the procedure:

Tag7-12: Move to ready position, move to catch position, move to tower-up, line down, and line up.

- **Red team:**

  Move to ready position: $H_{target} = H_{ready}$ where

$$H_{ready} = ([0,0,-1,0.12],[0,-1,0,0.65],[-1,0,0,0.24],[0,0,0,1])$$

  Move to catch position: $H_{target} = H_{catch}$ where

$$H_{catch} = ([0,0,-1,0.12],[0,-1,0,0.72],[-1,0,0,0.24],[0,0,0,1])$$

  Move to tower-up: $H_{target} = H^w_{twr} H^{twr}_{up}$ where

$$H^w_{twr} = ([0,-1,0,.562],[0,0,1,.18],[-1,0,0,.2+(i+1)*0.05-0.01],[0,0,0,1])$$

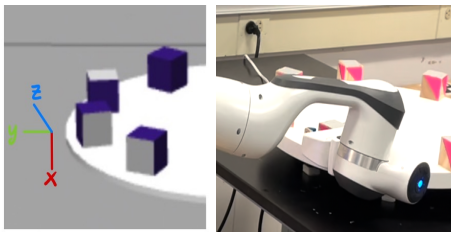

Figure 4: Virtual frames for ready and catch position of Tag 7-12 in Red team

and

$$H_{up}^{twr} = ([1,0,0,-0.2],[0,1,0,0],[0,0,1,0],[0,0,0,1])$$

line down movement: $H_{target} = H_{twr}^{w}$

line up movement: Use same q obtained in tower-up point.

- **Blue team:**

Move to ready position: $H_{target} = H_{ready}$ where

$$H_{ready} = ([0,0,1,-0.12],[0,1,0,-0.6],[-1,0,0,0.21],[0,0,0,1])$$

Move to catch position: $H_{target} = H_{catch}$ where

$$H_{catch} = ([0,0,1,-0.12],[0,1,0,-0.7108],[-1,0,0,0.24],[0,0,0,1])$$

Move to tower-up: $H_{target} = H_{twr}^{w} H_{up}^{twr}$ where

$$H_{twr}^{w} = ([0,-1,0,.562],[0,0,1,-0.158],[-1,0,0,.2+(i+1)*0.05-0.01],[0,0,0,1])$$

and

$$H_{up}^{twr} = np.array([[1,0,0,-0.2],[0,1,0,0],[0,0,1,0],[0,0,0,1]])$$

line down movement: $H_{target} = H_{twr}^{w}$

line up movement: Use same q obtained in tower-up point.

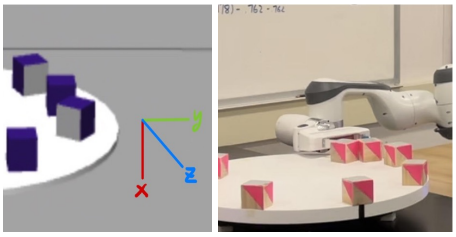

Figure 5: Virtual frames for ready and catch position of Tag 7-12 in Blue team

1.2 IK solver

The robot is being executed by `arm.exec_gripper_cmd(0.045, 10)`, `arm.safe_move_to_position(q)`, and `arm.open_gripper()`. The IK solver provides q as input to `arm.safe_move_to_position(q)`. IK solver need a seed and a target for each iteration.

1.2.1 A seed and a target

Initially, neutral position was selected as seed for every target. However, it is not the optimized choice for two reasons: First, some of the execution can be pre-computed and hard-code. Second, for movements cannot be pre-computed , a better seed candidate can be found.

Dynamic grabbing movement can be completed by pre-computation. It is because ready and catch position are fixed. Thus, after tuning the parameters for the corresponding matrices and using IK solver to get q, this q will be recorded and input to the `arm.safe_move_to_position(q)` directly for competition. Besides, all "Line up movements" will use the "Move to static-up" and "Move to tower-up" configuration q directly. It is also a hard-code part.

Static pick and place and dynamic place cannot be pre-computed. However, better seeds are selected for those movements. For all static-up points, the configuration q for the first static picking movement will be selected as a seed. For tag6 tower-up points, the configuration q for the first Tag 6 "Move to tower-up" will be selected as a seed.For Tag1-5 and dynamic tower-up points, the configuration q for the first side case "Move to tower-up" movement will be selected as a seed.

1.3 Other function parts

1.3.1 Detect gripper

"`arm.exec_gripper_cmd(0.045, 10)`" and "`arm.open_gripper()`" are execution command for gripper. In order to check whether the gripper has successfully grab the block or not. "`arm.get_gripper_state()`" was used for getting the gripper position. The gripper is composed of two parts and it will return two gripper positions. Setting gripper-dist as the sum of two gripper positions. The gripper-dist is 0.8 when they are open, and equals 0 when it is fully closed. For gripper-dist range from 0.03 to 0.05, it will be considered as a successful grabbing. Besides, in order to increase the stability of grabbing dynamic blocks, each time we detect a successful grabbing, the gripper will open again and wait for one second and then close for the second time. It can make sure the end-effector grab the same position of the dynamic blocks(This two grabbing mode is also discussed in the following evaluation section).

1.3.2 Checking for collision with the environment

First, since the command "`arm.safe_move_to_position(q)`" was selected for controlling the robot. We do not need to worry about self-collision. For collision with the environment, we directly used Gazabo simulation environment to detect collision, and find proper parameters for all homogeneous transformation matrices.

2 Implementation

In this part, we will basically talk about the implementation of the previous approach, which is, using python code to express the logic of the approach in every aspect, and then to provide an effective, executable project for further testing and modification.

2.1 Software

We used PyCharm, Atom, VS Code for code reading and updating. PyCharm has great highlighting and grammar checking for our code in Windows system. We edited the code mainly in PyCharm. In Ubuntu Linux system, we used Atom and VS Code for code reading and some minor error-checking, so that debugging and testing in simulation platform and for real robot can be more efficient.

For code updating between the teammates, we use GitHub as our version control tool, which is very effective and tolerant for mis-operation. Under Windows system, we use GitHub Desktop, this facilitates us to finish all the update works with just mouse-clicking. In ubuntu Linux system, we use git command line for pushing and pulling code. The project is at `https://github.com/architnh/meam520_finalproject`

2.2 Code Architecture and Contents

The code contains mainly 2 parts: the final folder that includes final.py, and lib folder contains modules of all features, including classes and sub functions that used in the main function. The picture below shows the files we used in this project. For RRT and A* code, we submitted these in gradescope, but will not discuss them because we didn't use these files at all.

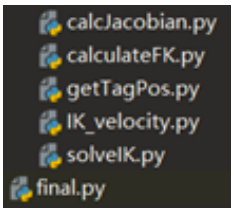

Figure 6: Code Structure

2.2.1 Previous Library Files

Before we start final project, we have already done several useful functions in previous labs, including calculateFK.py, calcJacobian.py, `IK_velocity.py` and solveIK.py. We will only talk about the major tasks they can handle.

- **calculateFK.py**
  This file was completed in lab 1, it contains class FK, which provides the D-H parameters data, calculation of transformation matrices for robot, and forward kinematics calculation. The most important function is forward(), which input is q, the 1 x 7 vector of robot joint angle configurations, and outputs are 8 x 3 matrix of position that

corresponds to each rotation joint, and a 4 x 4 homogeneous transformation matrix that represents the end effector frame w.r.t. (with respect to) the world frame.

- **calcJacobian.py**
  This file was accomplished in lab 2. It contains only one function called calcJacobian(), which imports some sub functions of FK, inputs robot configuration, and outputs the 6 x 7 Jacobian matrix of robot arm. The Jacobian matrix will facilitate us to calculate the velocity forward kinematics and the pseudoinverse of velocity inverse kinematics.

- **IK_velocity.py**
  This file was also done in lab 2. It imports calcJacobian and only has 1 function, "`IK_velocity()`". This function inputs current robot configuration, and desired linear and angular velocities of end effector in world frame, and outputs dq, 1 x 7 vector corresponding to joint velocities.

- **solveIK.py**
  This file was finished in lab 3. It has class IK and provides the function inverse(), which use gradient descent algorithm to solve the full inverse kinematics of the robot. The function inputs a 4 x 4 target matrix that represents the desired transformation from end effector to world, and a robot configuration. It outputs a target configuration, the result of try and the rollout as a log.

  For the final project, we modified the function by using projection to get the dq instead of calculating the pseudoinverse, and changing the weight of dq_center and the step size of dq to

$$dq = 0.5 * (dq\_ik + 2 * project\_dq\_center)$$

  This could rise the success rate of the solver. We also added the neutral position seed as a backup, in case the solver failed, it has another chance to calculate from this backup seed.

2.2.2 New Library File – getTagPos.py

This library file was created for the final project. It imports ObjectDetector() to get the data. It contains a class called Tag. It basically provides all the data of transformation matrices and methods to calculate specific positions of transformation matrices.

- **__init__(self)**
  This function is the initialization of the Tag class. It provides H_t0_w – homogenous transformation matrix (we use matrix instead later on) of tag0 w.r.t. world frame (robot's world frame, not the one in handbook), H_tic_ti – matrix of any tag's box center w.r.t. the corresponding tag, and H_upc_tic – matrix of position that is upper than box center for 10 cm w.r.t. the corresponding tag's box center.

- **get_H_t0_c(self)**
  This function helps us get the H matrix of the tag0 w.r.t. the camera frame. By traversing the data from the detector, we find the tag0 name and return the corresponding pose.

- **getTagCenterPos(self, H_t0_c, H_ti_c)**
  The function inputs the matrix of any tag w.r.t. camera frame and by using the previous calculation mentioned in approach section to output the matrix of any tag's box center w.r.t. the world frame.

- **get_tag_data(self, H_t0_c)**
  By traversing the detector data, we put the name and the pose of each loop into the list of tag_name and H_tic_w. This can help us to sort the detector data and for easy manipulation instead of using original chaotic dictionary data.

- **get_H_staticRot(self, tag_name)**
  Although we can drive the end effector of the robot to the static box center, however, we cannot make sure that white face is up after we pick and place the static box, since the frame of box center and the end effector frame is not correct in direction, though position is correct. So, we consider add a rotation matrix case by case, when the static box is tag1 – 4, we deal with the case as side case and returns a specific static rotate matrix, so does the tag5 and tag6.

- **get_H_twr_w(self, team, tag_name, i)**
  we regard the placing as "towering". We deal the static boxes case by case as usual to place the static box as white face up. We also have a variable called i, which indicates the tower level. However, since the catching direction and offset are different between the red and blue team case, we need to also judge the team case when placing the dynamic case. Finally, we return specific H_twr_w, the matrix of towering position frame w.r.t. world frame.

- **get_H_twrUp(self, tag_name)**
  Similar to the H_twr_w, however, when we place the box, we set our approach to first reach the position that are directly above of the tower. Therefore, before we arrive the towering position, we need to arrive the towering-up

position first. This provides a safe towering. We also need to judge the tag_name case by case, since the robot may either horizontally or vertically tower the box. After considering this, we give the specific H_twrUp matrix.

- **get_H_dynamicRot(self, team)**
  Also, similar to get_H_staticRot(), we shall consider the frame matching between the end effector and the dynamic box while catching. Therefore, we need to calculate the matrix H_dynamicRot. Also, since the robot have different box-catching poses and positions, we need to provide different matrices for two team cases.

2.2.3 final.py

Now we have all the library classes and functions, we can discuss the final.py deeper. But before we move on to the main function, we also introduced several sub-functions that are called by main function. These functions encapsulate the code as much as we can in order to improve the readability and the refinement of the code.

- **my_ik_move(target, seed)**
  The original inverse() function returns q but also the success and rollout, but actually the last two are only for logging and debugging. So, we encapsulate the original function again, withing printing success result, q, and time cost, and safe move to q, then finally return q only.

- **pick_and_place_tagx(team, tag_name, H_tic_w, i)**
  We encapsulate the pick and place operation of tag 1 - 4 condition. The basic logic is:
  – **1** open gripper and move to static-up
  – **2** line down
  – **3** catch and back to static-up
  – **4** go to tower-up
  – **5** line down and place
  – **6** place and back to tower up
  and finally return the final q configuration and matrix for further use, and also returns True, which indicates it's a side case.

  In this function, we provide the tested value, q_static_up_ref and q_tower_up_ref_side for both red and blue case, since they can be the seeds for calculating the static-up and tower-up much faster.

- **pick_and_place_tag6(team, H_tic_w, i, is_first_box=False)**
  We also encapsulate the pick and place operation of tag 6 condition. The logic process is almost the same, but since the robot will only need to place horizontally, some of the rotation matrices need to be changed to tag6 case. Also, we have a default variable, is_first_box. Only when we place the first box, we shall turn this to true, since the value we tested and used of q_tower_up_ref_up is the same as q_tower_up when placing the first box. It also returns same value, instead of side_flag is false since this is tag6 case. We will use this function for tag5 too because we give up using 2-step method to pick the time-costing tag5.

- **pick_and_place_first_box(team, tag_name, H_tic_w)**
  Because we deal with the first box using the tag6 method to avoid bumping with the towering platform, it's necessary to use a new function to express this logic as well. We use python "if" and "flag" to indicate:
  – **1** if there exists tag5, we pick and place it
  – **2** otherwise we check if there exists tag6 and pick and place it
  – **3** otherwise we pick the first one in our list.

- **dynamic_pick_and_place(team, i, dynamic_start)**
  Since the dynamic operations are different between blue or red case, we will provide the different matrices in calculation with variable team. Also, we follow the logic:
  – **1** go to the q_ready
  – **2** sweep to q_catch
  – **3** catch or wait
  – **4** go to tower-up
  – **5** line down and place
  – **6** place and back to tower-up
  The gripper's logic at step 3 is a bit complex:
  – **1** open the gripper and wait for 3 seconds
  – **2** close the gripper
  – **3** judge the catching. In step3, If we caught, reopen the gripper, wait for 1.5 seconds so that the box will reach the edge side of the gripper; if not, then keep opening.
  These 3 steps form a loop and finally we check if it exceeds the 60 seconds. If exceeds then back to tower-up, if not then keep looping.

- **main**
  Based on the code that the lib and sub-functions has already been provided, we implemented finally in main.
  – **1** initialize classes we need and argue the variables we need to use later
  – **2** get the center of the boxes, dealing with the first static box
  – **3** do 60-seconds dynamic catching, catching the rest 3 static boxes, catch the rest dynamic boxes.

3 Evaluation

For most portion of the evaluation process, we firstly tested the correctness of our code in Gazebo and debugged every syntax issues that may have. Also, we evaluated the rough performance of our code in the virtual condition at this first stage.

However, there truly have some gaps between the Gazebo and the real-world lab (will be illustrated in the following content of this section).
So, we went on designing a few tests in the real-world lab to further testify our implementation.
After we got the bug-free code tested from the Gazebo, the most important task in the real-world lab is to run and modify the parameters, letting which to be the best fit to real-world conditions.

In conclusion, we did the evaluation section beginning from the Gazebo, then to the real-world lab. In the real-world lab, we designed two main kind of tests:

−The first one is the static blocks test.
−The second one is the dynamic blocks test.

For each test, the code went through a loop and executed a series of moves corresponding to the subroutines that formed the loop.

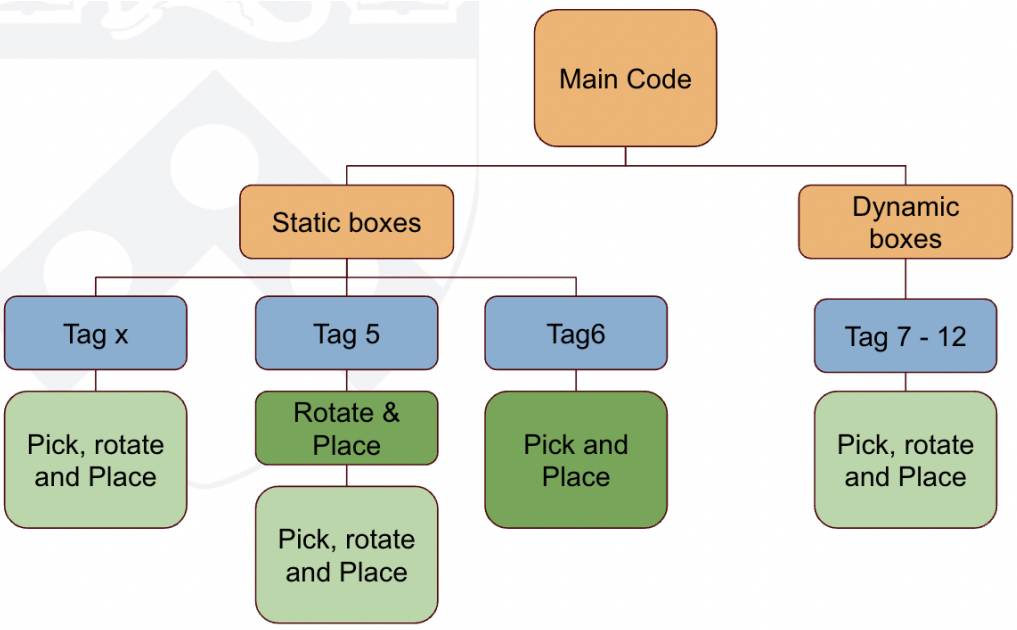

Figure 7: Static test result

3.1 Static block test

Before executing the static block test, there are some issues that need to be considered before and during the experiment. Firstly, the offset of the block tower is the most important factor that could make a huge difference.
When placing blocks, we wanted the blocks to be erected as a stable and safe tower. However, in the real world, the blocks may bounce, as the mechanics of interaction between gripper, blocks and the table cannot be accurately modeled. Also, gripper has offsets, thus causing the unstable catching towering.
We implemented some fixes that rectified this issue.

3.1.1 Static block test result

From the static block test, we discovered that there truly have some offset of the tower of blocks, leading to the potential collapse of the tower and the falling of the blocks. Below is the comparing figure (figure 8) that shows the difference.

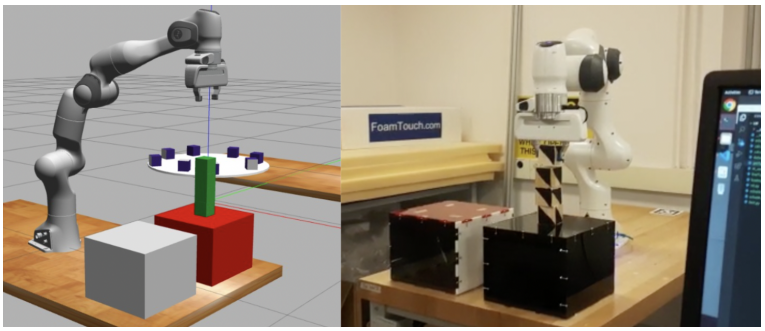

Figure 8: Static test result

3.2 Dynamic block test

For the gap between Gazebo and lab: Gazebo Simulation vs. Real-World Robot.
In Gazebo, blocks have **huge mass and friction** with the plate, which prevents them from being pushed by the robot arm.
In Gazebo, the simulation environment is very **slow and lagging**, which may affect the performance of the robot system.
For these mentioned reasons, we should also take a dynamic block test in the lab.

3.2.1 Dynamic block test result

As was mentioned in the earlier section, the blocks in the Gazebo has rather heavy weight and massive friction, preventing being pushed by the robot arm. While in the real-world lab, this problem no longer exists. The robot arm could push the block on its side, without crushing into it.
Moreover, the operation speed of the robot arm is also faster than that in the simulation environment. By extensive testing in the lab, on the real robot, we systematically implemented checks to make sure the robot operation was stable.

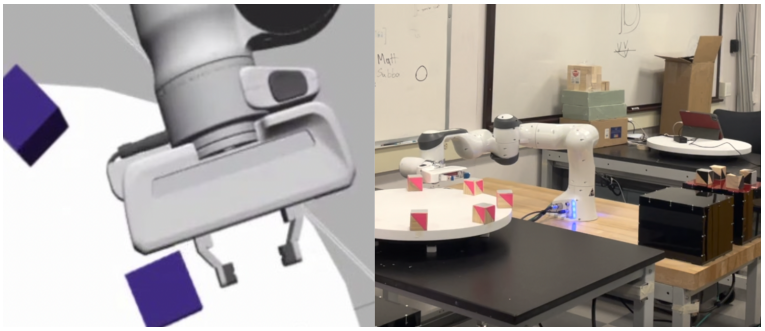

Figure 9: Dynamic test result

3.2.2 Safe-grabbing method of the dynamic blocks

During the process of testing our dynamic code, we found an new problem that has never been considered before coming to lab: Since we coded to pick the dynamic blocks **in a rather blind way**, we could not make sure if we grabbed the same place of the blocks every time, which will lead to the offset of the tower, thus increasing the possibility of collapse.

It is worthwhile for us to come up with **an urgent implementation** to deal with this dilemma.

- Then, we decide to code our gripper to do a few more steps of operation:
- When something is detected being grabbed, open the gripper for 0.5 second then close it again.
- In this way, the dynamic blocks will eventually being grabbed in the same position by the gripper, guaranteeing the perfection of each attempt.

3.3 Tag5 (white-side-down) implementation performance

To get full scores for all of the static blocks, we have introduced the method of dealing with the Tag5 blocks. Although it may take longer time to finish the operation, but the overall outcome was pretty stable and reliable. The robot picked up the block and placed it on the table after rotating. Then the block was treated as a side facing case.
The average time of picking then placing such a Tag5 block is about 45 to 50 seconds, making it probably a undesirable move to be done in the real competition process. So, we may reconsider the necessity of the Tag5 implementation strategy, ensuring we can get the best of our result in precious three minutes.

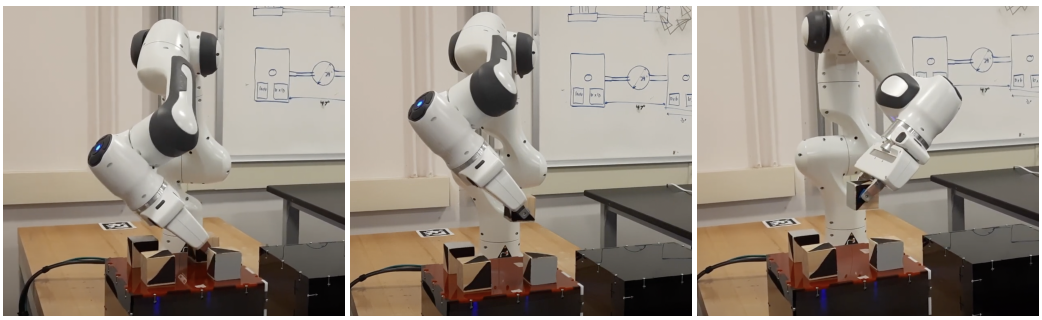

Figure 10: Tag5 cases result

3.4 Final output in the real-world lab

For ordinary scenarios, our pick-and-place algorithm could finish the work in 3 minutes with 4 static blocks and 2 dynamic blocks.

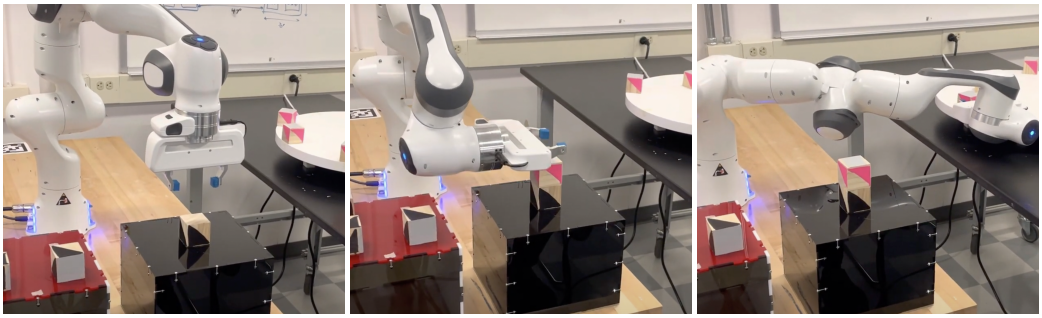

Figure 11: Final output of the algorithm part one

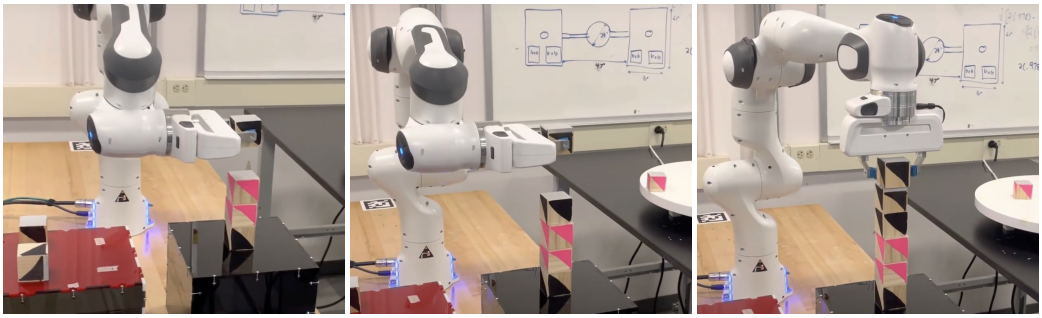

Figure 12: Final output of the algorithm part two

As can be seen in the figure 6, the evaluation and testing, helped mitigate the offset issue of the tower. We were able to pile up the blocks perfectly. The static blocks aligned perfectly along with the dynamic blocks.
We considered multiple test cases for both red and blue sides separately. The co-ordinate axes for blue and red side robots were flipped. It was difficult for the IK solver to perform in some scenarios if the seed was not declared accurately. Picking the right value of start and goal positions for each steps was necessary for both red and blue side. We implemented some necessary transformations to ensure that the robot motion was as desired.

4 Analysis

4.1 Analysing the problem statement and approach

For this problem, we identified that the robot needs to perform two major tasks- to move around the environment in a repeatable manner and in the shortest possible time, and to avoid collisions with obstacles.
We tried multiple approaches for solving the problem-

Algorithm 6: RRT*

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
    $V \leftarrow V \cup \{x_{\text{new}}\}$;
    $x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Connect along a minimum-cost path
      if $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
        $x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
    $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Rewire the tree
      if $\texttt{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \wedge \texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}})) < \texttt{Cost}(x_{\text{near}})$ **then** $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
      $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
return $G = (V, E)$;

Figure 13: RRT* Pseudo-code

1. **RRT and RRT*:** RRT (rapidly exploring random trees) [2] involves initializing random configurations as nodes. Although it converges onto a solution faster than the IK solver approach, it does not ensure a repeatable solution. Also, the ”waypoints” between start and goal are random and not intuitive.
   This made RRT undesirable, from the competition point of view. We also considered the use of RRT* which includes implementing a cost function to check the cost for each node and its neighboring nodes. If nodes in vicinity are closer, they get rewired to the nearest nodes and all of the upstream nodes are updated. This results in a much cleaner path. RRT* is computationally more expensive than regular RRT, but provides a more direct path from start to goal, rather than a random indirect path given by regular RRT.

   We also observed, that we could save a lot of computational time, by pre-computing the path using IK Solver, and saving the configurations (q values) and feeding them on the competition day, to get a faster motion. RRT would have been necessary, had there been a cluster of obstacles in the 3D space around the robot, but because the only existing obstacles are the tables, rotating circular table, and the blocks themselves, implementing RRT complicated and over-designed the solution for this problem. Thus we tested, but systematically removed both RRT and RRT* from the list of viable approaches.

2. **Line trajectory follower:** A second approach that we considered, and also tested was the trajectory following approach. The line trajectory following approach uses the parametric solution of a line. But because the solution has steps involving picking up a block, moving above the already placed blocks, and placing it, it involves a lot of steps which could be iterated by calling the function that executes it. This can be done much more efficiently with the IK solve methodology.
   Also, it is much easier to control the joint velocities, joint limits. The code for IK solve is not just cleaner, but more optimized.
   Thus we selected IK Solve as the most appropriate approach for solving our problem.

4.2 Competition performance

In the MEAM 520 Final pick-and-place competition, we observed a few bugs which we had not previously observed before. We believe these to be rare cases. We brainstormed, and resolved these issues soon, and updated the code.

1. The first issue was that IK solver had some errors when trying to reach a certain configuration to pick the static blocks.
   This issue happened because it became a bit difficult for the end-effector of the robot to reach the end-effector frame that we defined. Sometimes the seed needs to have the end-effector rotated by 90° about the z-axis to provide the

Figure 14: Figure 13: Competition result

possible ee frame for the robot to operate. This issue has been resolved and discussed in the method section.

2. The second issue was that the end-effector of the robot would slightly brush the dynamic blocks from above sometimes, when attempting to reach the grabbing configuration in the blue configuration. This issue was because of the seeds and the intermediate configurations that the IK solver would utilize to move from the start to the goal in the dynamic blocks case. The solution was to change the values of the start position and end positions such that the end effector would not brush the blocks from above when moving inwards over the revolving table.

4.3 Future improvements

The code developed in this section was largely aimed form the final pick-and-place challenge point of view. Further improvements (specific to this task) would be to chose more optimal path, to optimize and save time. The other improvements would be to try other seed values and perform extensive testing on the actual robots to maximize the points scored. We could also try different orientations of the blocks and perform much more comprehensive testing for other scenarios (e.g block lying in oblique fashion with one edge touching the table (instead of one face).
The dynamic block pick and place problem can be solved in multiple ways. Because we don't have feedback, or perception integrated in the timeline, it is difficult to accurately grasp the block. In this case, the pick and place task becomes a bit complex. It can be achieved by grabbing the block in different orientations.

1. We moved the end-effector to a position above the table. Then it swept in towards the revolving table. It swept in at a $45°$ angle. This can help us catch the dynamic box safely without collide with the rotating plate, and effectively catch the box since we don't need to spend much time on waiting the box to move into the gripper.
2. The second method that we devised was to move the robot to a position in plane with the revolving table. Then the end effector (positioned at 0 with the revolving table) slides inward and waits to receive the dynamic block.

After evaluating both approaches experimentally, we found that the second one was the most efficient dynamic block-picking approach.

5 REFERENCES

1. Robot Modelling and control, 4^{th} edition - Mark W. Spong, Seth Hutchinson, M. Vidyasagar
2. Sampling-based Algorithms for Optimal Motion Planning, Sertac Karaman and Emilio Frazzoli, 5 May 2011

This is normal size!!!—Sampling-based Algorithms for Optimal Motion Planning —- This is 30pt—-!!Sampling-based Algorithms for Optimal Motion Planning