Final Report

Dynamic Blocks

In order to sweep the blocks into the end effector we used inverse velocity kinematics to trace a counter clockwise circle around the table in a direction that opposed the table's rotation. This was very similar to the evaluation test done in lab 4, but we had to orient the circle so that it had a normal in the z plane. To find a good start position for the tracing out a circle, we had to use inverse position kinematics to find a feasible pose. This feasible pose that traced out a circle will henceforth be referred to as the velocity strike pose aka the strike pose. The velocity strike pose was then shifted in the radial direction to intercept the path of a block. We shifted the strike pose in the radial direction by using inverse kinematics and forward kinematics. The procedure involved first finding the pose of the end effector by using lynx.get_state(). We then plugged the configuration into forward kinematics to find the exact pose of the end effector. The pose of the end effector was then modified using the findperfect function, which shifted the velocity strike pose in the radial direction so that it perfectly aligned with the path of a desired block. This pose will be henceforth known as the perfect strike pose. Shifting the pose was a matter of adding a radial vector to the x and y component of the end effector pose and then plugging this new pose into inverse kinematics to get the desired configuration.

We then used the moveincircle command to calculate a path that traced out a circle along the rotating table in the counterclockwise direction. The moveincircle command used the following inverse velocity kinematic equations to trace our a counterclockwise circle along the table:

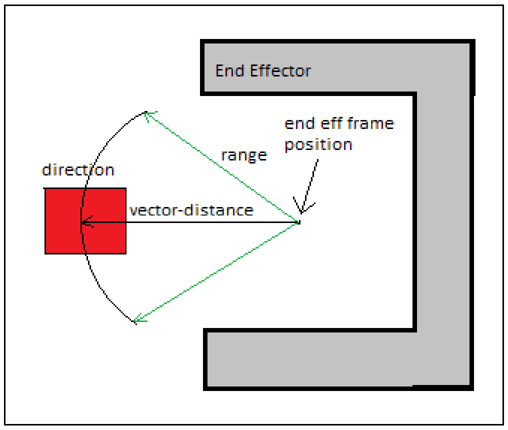
$$\dot{q} = J^{-1} v_i^0$$

$$J_{vi} = S(z_{i-1}^0)(o_n^0 - o_{i-1}^0), J_{\omega i} = z_{i-1}^0$$

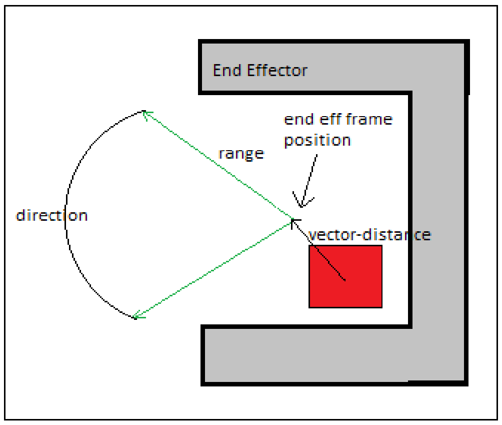
We then used euler integration, which involved multiplying dq by dt and adding this product to the current configuration to find the next configuration.At first we tried to walk along the path using a time step and velocity control because initially we found velocity control to be far faster than position control using lynx.set_pos(). However, after the TA's modified gazebo to add the lynx.command function, we found that the imprecision caused by controlling velocity using rostime was no longer worth it since lynx.command was so fast. As a result, we simply fed in the inverse velocity kinematics path into lynx.command function to force the robot to walk along the desired path from the perfect strike pose.

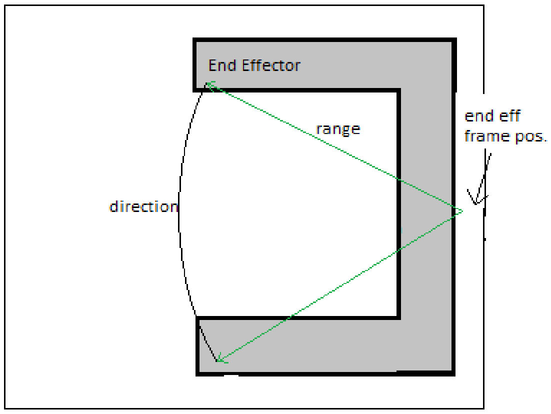
A new problem was determining when to break the end effector from the circle path and grab a block. To accomplish this, we created a function called InUrFace, which used linear algebra to determine whether a block was in the end effector's face. InUrFace worked by first transforming the block pose into the robot's frame and drawing a vector from the end effector to the position of the box that we will call vector-distance. If the vector-distance aligned with the z axis of the end effector passed a certain threshold and the length of vector-distance was small enough, then a box was indeed within the end effector's face. Alignment was determined using a dot product between vector-distance and the z axis of the end effector. One issue that occurred is that blocks could get behind the position of the end effector because the default location of the end effector is located within the center of the gripper, see diagram 1. To solve this problem, we shifted the end effector location backwards along the z axis to solve this edge case.

InUrFace with default end effector position



End effector position edge case





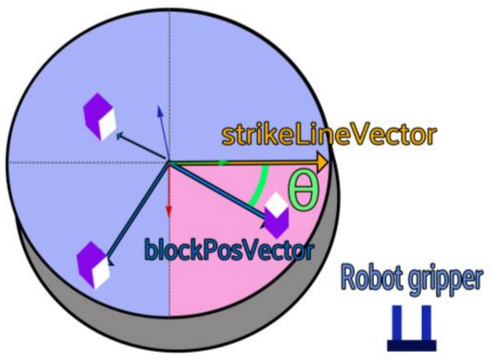
Edge case solution was shifting the end effector along the effector z axis.

Once a box was within the robot's face we broke out of movement and let the table's movement drag the box into the end effectors grasp. The force of the table on the block and the gripper on the block would tend to align the box in the end effector so once the vector-distance was small enough we forced the end effector closed. To speed up the process we would allow the robot to move linearly towards the box if the box was within a certain range and within the robot's face. This sometimes had the unintended effect of knocking other blocks off the table though.

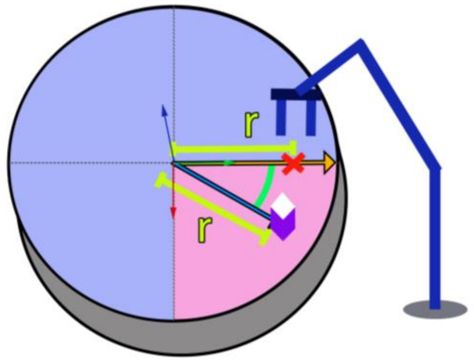
The functionality to pick up blocks on the right hand side of the turntable was implemented using the pickRightSideBlocks driver function, and the sub functions are described below. From the perspective of the robot, we can view the spinning turnstile as the face of a clock, where the 12 o'clock mark is the furthest distance away from the robot. As the turnstile always moves counter clockwise, it made sense to wait for the blocks to come to the robot and then catch them, similar to playing Tetris. The most straightforward position for the robot to catch the blocks was at the 3 o'clock position from the robot's perspective, or where strikeLineVector is in the diagram below.

The algorithm first needs a method of determining the closest dynamic block to the strikeLineVector, which is [0 1] if the blue robot and [0 -1] if the red robot (in the world frame). It then starts by calling filterOutStaticBlocks to calculate the dynamic blocks that are still moving by checking whether the twist of the block is not zero. Since some dynamic blocks could have fallen off the platform, we could not simply filter based on name.

After retrieving the list of moving dynamic blocks, it then sends the list to calculateTargetBlock which decides which block to pick up. The first thing the function does is filter out all blocks which are not in the pink shaded region (see figure on right) because we don't want the robot waiting too long before catching a block. Next, it determines the position vector of each moving dynamic block in the world frame, called the blockPosVector. After normalizing each blockPosVector, the function calculates the angle θ between each blockPosVector and the strikeLineVector. The block with the smallest angle is then determined to be the target block. If there is no block in the pink shaded region, then the function exits.



Next, provided that there is a target block, calculateStrikePosForTargetBlock takes the target block and normalizes its blockPosVector to get the radius r from the center of the turnstile. It then maps r onto the strikeLineVector by multiplying r with the strikeLineVector, which provides the location along the strikeLineVector which the robot needs to navigate to and wait (in the figure to the left, X marks the spot). The function then sets the transformation matrix



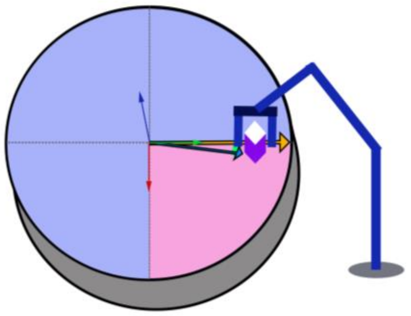
from the robot's base frame to the end effector frame with the equation below. If the robot is red, then the transformation matrix from world to robotBase would replace the -1's with 1's.

$$r_{robotBase} = T^{world}_{robotBase} \cdot r_{world} = \begin{bmatrix} -1 & 0 & 0 & 200 \\ 0 & -1 & 0 & 200 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ r \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 200 \\ 200-r \\ 0 \\ 1 \end{bmatrix} \rightarrow$$

$$T^{robotBase}_{endEff} = \begin{bmatrix} 1 & 0 & 0 & 200 \\ 0 & 1 & 0 & 200-r \\ 0 & 0 & -1 & 80 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We set the z height to be 80 as this allows the end effector to hover over the turnstile as it waits to strike. Using inverse kinematics, the configuration q is calculated and then navigated to using lynx.command(). The z axis orientation is set to be -1 for both red and blue robots to get the elbow up configuration.

Next, the grabBlock function continuously polls the target block's current position and calculates the block's angle from the strikeLineVector. Once the angle is less than 30 degrees, the z coordinate height of the end effector is then set to 60 in $T^{robotBase}_{robotEndEff}$ and inverse kinematics is again used to calculate q and move the end effector to the table surface. Lastly, the function InUrFace is continuously polled to determine whether the block is inside the gripper's hand. When this happens, q(6) is set to be -5, as -15 would grip the block too hard and cause it to slip out if the block wasn't oriented perpendicular to the gripper fingers. From here, the placing function is called to take the block to the goal platform.



FLOWCHART for dynamic function at the end of report – page 11/12

Plan for static blocks

In the competition, static blocks are placed on the platform beforehand and their poses stay unchanged, which makes it easier to be picked up. Our approach for static blocks contains four steps. Firstly, move the robot arm above the target static block with a safety height. We set the safety height, 50mm in this case, to avoid collision from start configuration to above block configuration because all the obstacles (aka platforms) are below the height of 50mm. Secondly, move down the arm and pick the block up, returning to the safety height. Then, move to a target pose above the goal platform. We also set another safety height for the goal platform to avoid collision. Finally, lower the arm to drop the block and move back to safety height. We need to place four static blocks, so the whole process, shown in Figure 1, will be iterated four times.

1.1. Priority for the four static blocks

There are always four static blocks for both teams, and we compare the distance of each block to the robot base to find the closest one to pick up. In case of accident moves during the last picking up process, we call get_object_pose() function every time before picking up.

1.2. Calculate the picked pose

The pose of static blocks is given in a transformation matrix T format. The Z axis gives the orientation of white side and the X, Y axis represents the other sides. All the poses should be calculated in robot frame rather than ground frame. Firstly, we transform the block poses into the robot frame. Our goal is to calculate the pick pose, in another word, the end effector pose corresponding to the block pose. To grasp the block tightly, the fingers should parallel with the side of blocks, shown in Figure 2, which can be achieved by adjusting the angle of joint 5. However, given the strict fifth joint limit, we need to minimize the adjustment.

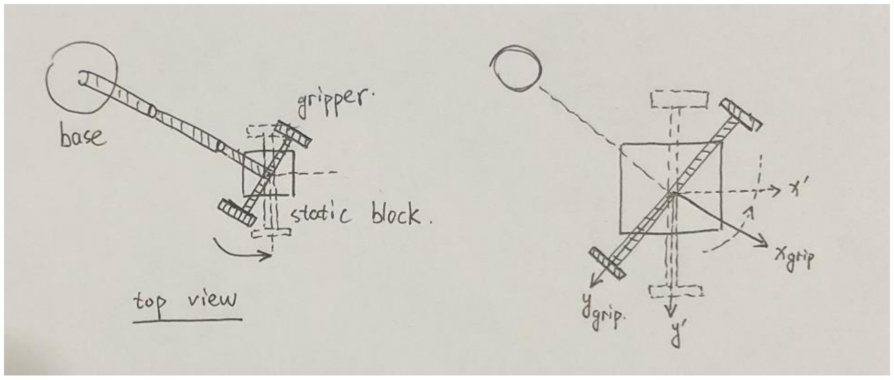


Figure 2. Adjust the end-effector pose

Lynx robot has a special structure, that the first joint controls the rotation in Z axis and joint 2,3,4 control within a plane. Thus, when we assume joint 5 equals 0, the vector from robot base to block gives the 'normal' direction of the end-effector's X axis:

$$X_{grip} = O_{block} - O_{base}$$

Then, to minimize the adjustment of joint 5 is equivalent to finding the closest direction to the end-effector's X axis.

$$X'_{grip} = min(norm(p_{block} - X_{grip}))$$

Above, the vector p_block donates the orientation of the block side. Since the block pose is randomized, p_block can be the X, Y, or Z axis of the block, which represents 6 different directions containing the opposite X, Y and Z axis. In this way, we determine the direction of the target pose's X axis, while the Z axis is fixed since we always pick from top. Finally, the Y axis of picking pose can be calculated by cross product of X and Z axis.

$$Z'_{grip} = [0, 0, -1]; \; Y'_{grip} = -X'_{grip} \times Z'_{grip}$$

We generate the final transformation matrix of picking pose in the robot frame. Note that the picking pose should add the safety height to avoid collision, as mentioned before.

$$T_{pick} = \begin{bmatrix} X'_{grip} & Y'_{grip} & Z'_{grip} & O_{block} + \begin{bmatrix} 0 \\ 0 \\ 50 \end{bmatrix} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

After that, the inverse kinematics simply gives the configuration for picking the block.

$$q_{pick} = calculateIK(T_{pick})$$

1.3. Deal with special cases

There are 2 special cases, shown in figure 3,4: sometimes the block is too far away to reach from top; while sometimes two blocks are close and the robot fingers may hit another block when picking. We detect the first case when the inverse kinematics in method 1.1 returns empty configuration. In this case, we pick up the block horizontally rather than vertically, shown in figure 3.

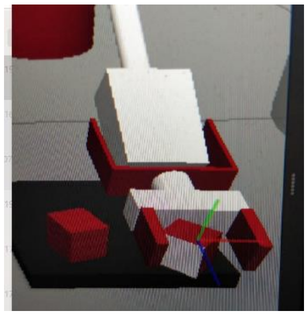


Figure 3. Pick up far away blocks

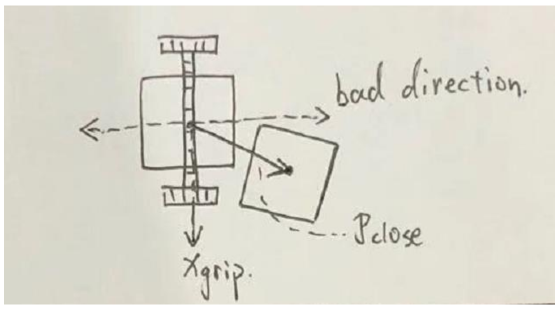


Figure 4. Pick up close blocks

We only adjust the position of the picking pose in this case, and the orientation is fixed.

$$T_{pick} = \begin{bmatrix} 0 & 0 & 1 & X_{block} \\ 0 & -1 & 0 & Y_{block} \\ 1 & 0 & 0 & Z_{block} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We detect the second case when finding another block within a certain distance (30mm in practice). Figure 4 shows that if two blocks are close, the picking up motion in a bad direction will cause collision with another block. In this case, we made some modifications of the picking pose in method 1.1, discarding the bad picking up direction. The bad direction is the closest block direction to the vector connecting 2 blocks.

$$p_{bad} = min(norm(p_{block} - p_{close}))$$

Above, the p_block can be the X, Y, or Z axis of the target block. Note that the opposite vector of the bad direction should also be discarded.

$$X'_{grip} = min(norm(p'_{block} - X_{grip}))$$

Above, the modified p_block only contains 4 different directions, where the 2 bad directions are discarded. The relation between these sets is given below:

$$\{p_{block}\}_6 = \{p'_{block}\}_4 \bigcup \{p_{bad}\}_2$$

The other part of the second case stays the same as method 1.1.

1.4. Move in configuration space

All the motion is achieved in configuration space, and the motion between two configurations mainly uses the provided function lynx.command([goal]) due to the high motion speed and relatively good stability. We design another checking approach, in which we detect the norm difference between current configuration and goal configuration. We keep calling the lynx.command([goal]) function until the norm is less than a tolerance. To avoid infinite loops, the iteration stops over 15 times. All the procedure is done in a function called ‘move()’.

1.5. Stack the static blocks

In the competition, stacking the blocks gives higher scores. We carefully set the height in the goal platform to stack all the static blocks. Take the red team as an example, the goal pose arrangement is shown in Figure 5.

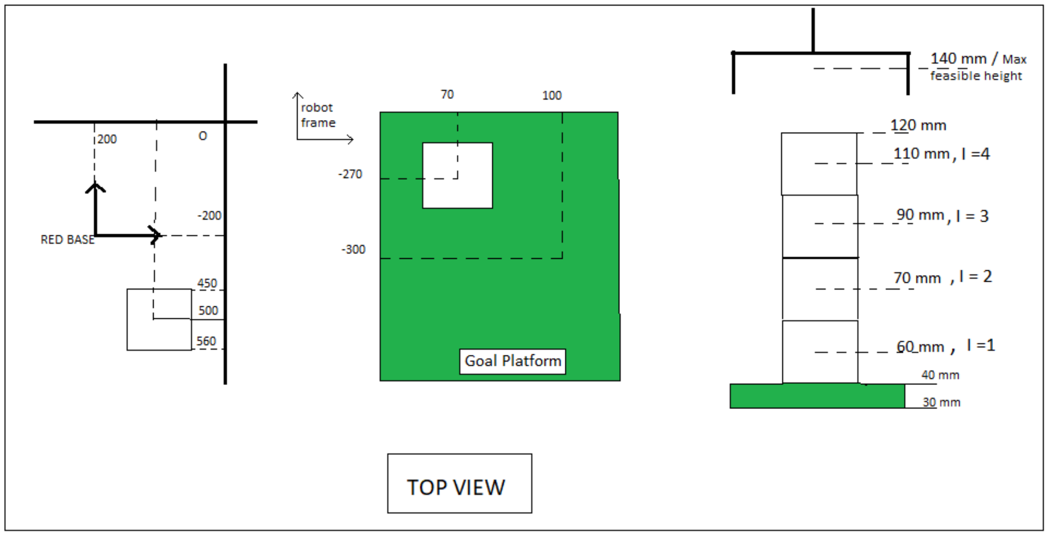


Figure 5. Goal pose arrangement

The horizontal position is [70,-270] in the robot frame mainly because the blocks should not be too close to the edge, which is [50,-250], and not too far away from the robot base, which is [100,-300]. The block is picked and placed one by one, giving the label i equals 1,2,3,4. The maximum height, or safety height, is about 140mm, since z = 150mm returns empty configuration during inverse kinematics in our testing. Our strategy is moving to the safety height every time before dropping, which reduces the chance of collision. Above all, the transformation matrix in robot frame gives:

$$T_{place} = \begin{bmatrix} 0 & -1 & 0 & 70 \\ -1 & 0 & 0 & -270 \\ 0 & 0 & -1 & 140 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad T_{drop_i} = \begin{bmatrix} 0 & -1 & 0 & 70 \\ -1 & 0 & 0 & -270 \\ 0 & 0 & -1 & (30+2i) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The expression '30+2i' , where i = 1,2,3,4, comes from the height analysis in Figure 5 (side view). After that, inverse kinematics gives the placing and dropping configuration.

$$q_{place} = calculateIK(T_{place}); \quad q_{drop} = calculateIK(T_{drop})$$

WhiteSideUp( ) – A function to get side bonus.

Salient points :

1. The basic idea is to align the gripper such that the dot product of Y – axis of gripper frame is parallel to & in same direction as the z axis (z0) of block (z axis corresponds to the white side). This is done by checking for dot product bw the two axis's.
2. The function takes 2 inputs – Tin and pose;
   a. Here T.in or Tin is the pick frame returned by function pickedpose(). This frame is guaranteed to have the gripper squared to the faces of the blocks.
   b. Pose is the Transformation matrix of the i^{th} block returned by get_object_state() in robot frame.
   c. Both inputs are used to compare the White side of the block (z0) with the approach frame of the gripper
3. The function has 2 outputs – Tout and change
   a. Tout is : Tin rotate by 0, $\frac{\pi}{2}$ or $-\frac{\pi}{2}$. The rotation is performed by rotating the gripper frame by the required angle about its z axis. Rotation of gripper happens right before the pick/grab action.

$$minusNinety = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad plusNinety = \begin{bmatrix} \cos(1.5) & -\sin(1.5) & 0 & 0 \\ \sin(1.5) & \cos(1.5) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; joint\ lim\ at\ 1.5$$

i. After the frame is rotated by $+/-\frac{\pi}{2}$, it is checked for feasibility. This is done because if, say, the gripper is approaching the block at an angle of $\frac{\pi}{4} = 0.79$ due to pickedpose(). Then rotating the frame by additional $\frac{\pi}{2} = 1.57$ would not be feasible due to joint limit at 1.5. Hence, in this case, the gripper would be rotated in the opposite direction by the same amount. Now instead of being $0.79 + 1.57 = 2.36\,rad$, the gripper would be at $0.79 - 1.57 = -.78\,rad$, which is the same orientation but in opposite direction.

b. Change – Change signifies the number of theoretical rotation steps required to align the white side up. This variable is used to determine the place-frame of the gripper. It depends on:

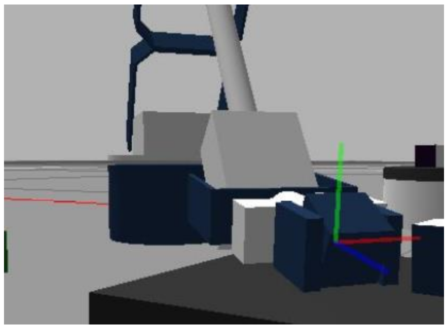
   i. If the block already has white side up – then no aligning is required. Hence, change = 0. For change = 0, the place frame in static() is in vertical orientation at the goal platform

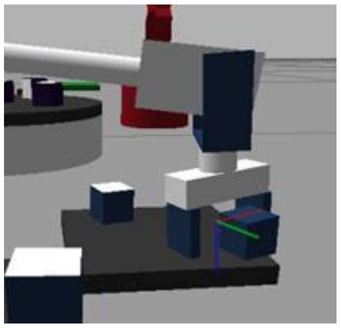
   ii. If the block has white side towards one of the sides, then the block is picked up vertically → Rotated about joint 5 / white side aligned → Moved to goal → Placed horizontally such that white side is facing up. If the first rotated frame deduced is feasible, then change = 1 (i.e. place at q5 = -1.57)

   iii. If in above case, the rotated gripper frame is not feasible (as described above), then the frame is rotated in opposite direction and change = 2; Change =2 is defined in static() function to place horizontally in opposite direction (i.e. q5 = 1.5)- basically an additional rotation step before place.

4. Alignment is attempted only when a vertical oriented pick is feasible. This is because, as long as block position is feasible for such a configuration, vertical orientation of wrist would give us a chance of picking up the block with its faces squared to the gripper. This is due to the roll motion of the wrist at Joint 5.
   Also, a horizontal pick has following issues -

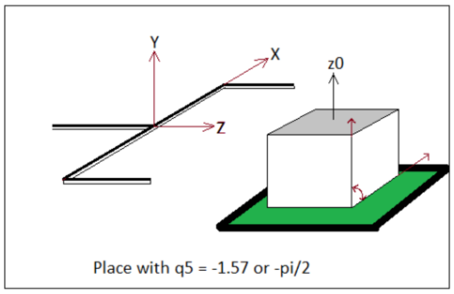
   a. Block nature of wrist – on attempting a horizontal pick, the accuracy of the pick was seen to be an issue as the block shaped joint 5 was often found to be hitting the platform.

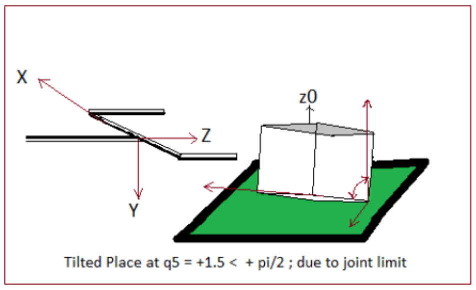

Horizontal Pick - faces not square w/gripper


Vertical pick - faces squared w/gripper

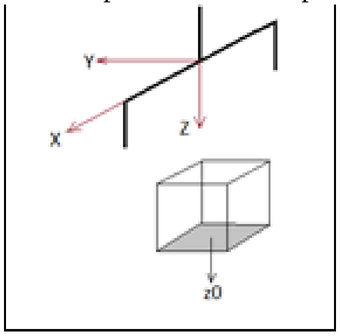
   As can be seen in the picture attached, such a pick would decrease the stacking accuracy over the goal platform as the gripper may or may not have a good grip of the block. This may cause the stacked blocks to topple down while placing.

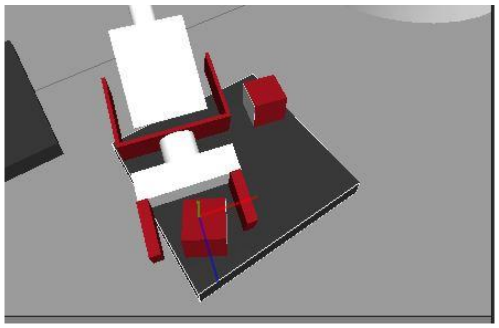
   b. All blocks that required aligning would be placed at q5 $= -\frac{\pi}{2}$ instead of q5 = $+1.5 < +\frac{\pi}{2}$. This is because the joint limit for joint 5 is from +1.5 to -2.0. To perform accurate placements, we would need an angle of $\frac{\pi}{2}$ or $-\frac{\pi}{2}$ (which is +1.57 to -1.57 rads). This way the down face of the block would be flat against the goal platform. Hence placing at an angle of q5 = $+1.5 < \frac{\pi}{2}$ could cause issues while building up stacks of blocks and may cause the block to fall off if not placed properly.





5. Only single step alignments attempted – this means that any case where a multiple pick-drop sequence was required to align the white side up, would not be attempted. This was done keeping in mind that we had a 60 sec time limit. We aimed to prioritizing stacking all blocks instead of spending more time to align a single block and miss out on multiple others. Example of this configuration is the case where white side was facing downwards:





Another case where aligning was not possible was when the blocks position was infeasible for a vertical pick. As shown in screenshot above, even picking such blocks horizontally did not instill much confidence in these orientations as the block faces were not always squared to the gripper. Thus, we did not attempt to orient them as such picks were observed to be risky to align and the robot would often drop the block due to lack of grip.

Toss() function – Toppling opponents stack

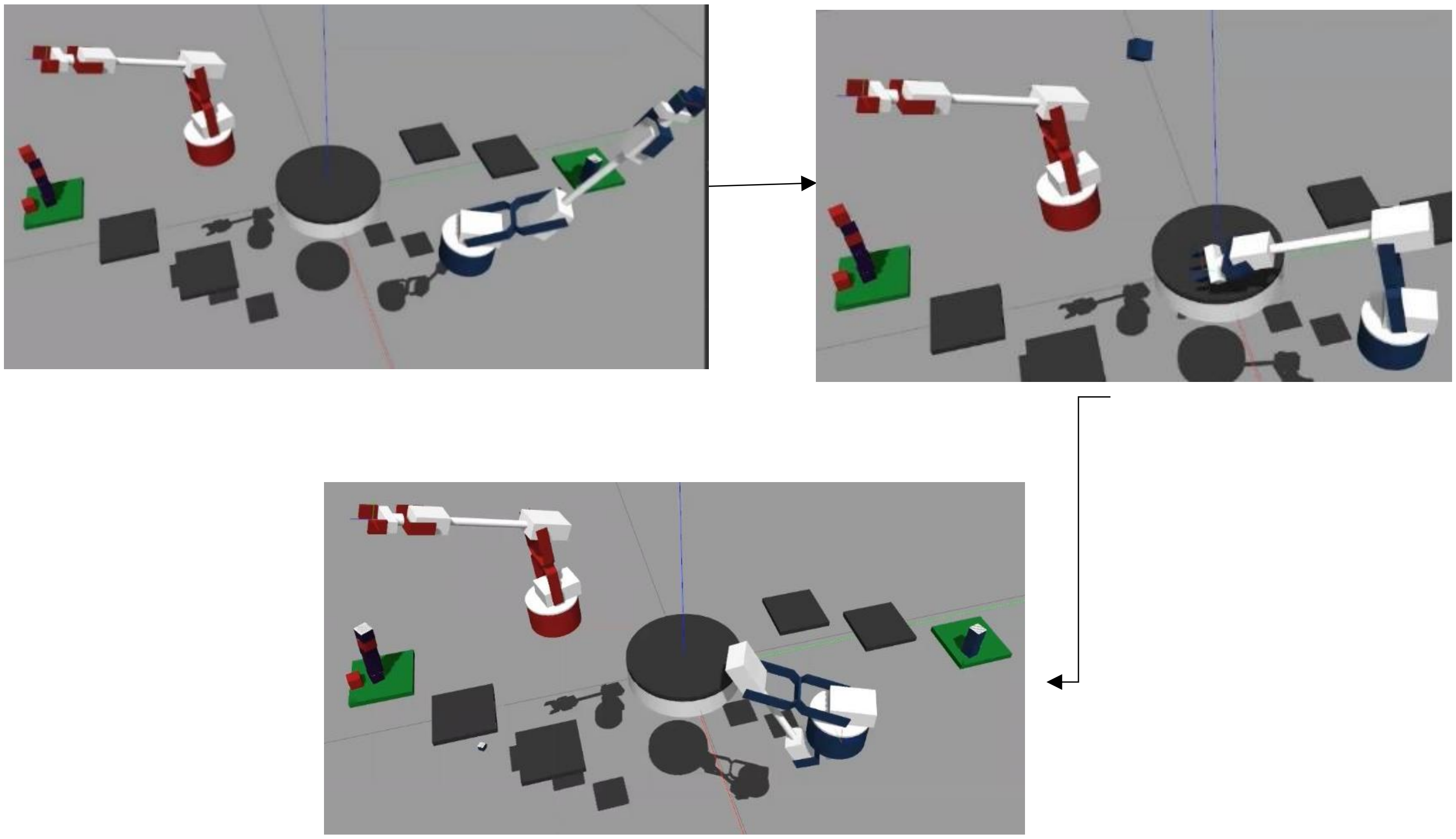
Summary → This function is aimed at toppling the opponents stacked tower by sacrificing the last static block

1. This function is called inside static() after the pick for the last static block has been completed.
2. It returns a single value 'stop'. If stop = 1 → it would execute a toss function as opponents stack/block tower detected. Otherwise, If stop = 0 then toss() breaks out and static() continues to stack the last block.
3. It checks the number of blocks + height of the highest block at opponent's goal position. The goal position is hardcoded for each team's opponent with a tolerance of + / - 50 mm in each direction (to account for goal platform size).

Steps –1. Finds the highest block on opponent's platform, stores the highest block position and height using get_object_state()

1. On detecting the highest block on opponents platform, uses basic geometry to find an appropriate 'firing Angle' for joint 1 . It then goes to an intermediate position at firing angle using calculateIK( )
2. Once firing angle has been reached, it then uses Velocity Kinematic to accelerate and release block in order to topple the opponent's tower.
   --Screengrabs from the day of tournament showing the intermediate positon, throwing action and end result respectively---

FLOWCHART for WhiteSideUp() at the end of report - page 12/12



Analysis –

Type	No of iterations	No. of blocks	Avg Blocks picked	No. of infeasibly oriented blocks	Blocks placed	Blocks stacked	Blocks Aligned	Points	Comments
Static	5	4	4	0	4	100%	100%		
	5	4	3.4	2	3.4	100%	50%		
	5	4	4	0	4	100%	100%		
	5	4	3.4	1	3.4	100%	75%		
	5	4	4	0	4	100%	100%		
	5	4	4	0	4	100%	100%		
Dynamic	2	5	2	-	2	100%	40%		
	2	5	0	-	0	-	0%		
	2	5	1	-	1	100%	20%		
	2	5	1	-	1	100%	20%		
R Robin	1	9	4	0	4	100%	44%	30	1D + 3Static
R Robin	1	9	4	0	4	100%	44%	30	1D + 3Static
R Robin	1	9	4	0	4	100%	44%	24	4 Static
Finals	1	9	4	2	4	50%	11%	11	First block infeasible
Trial run after	1	9	4	1	3	75%	44%		TEST for Toss function

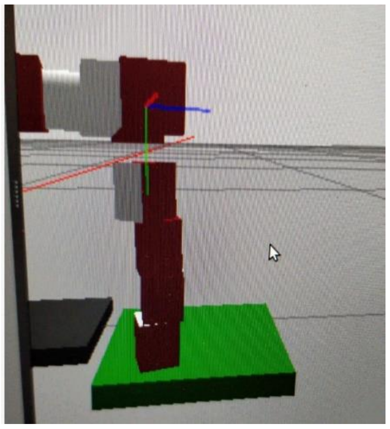
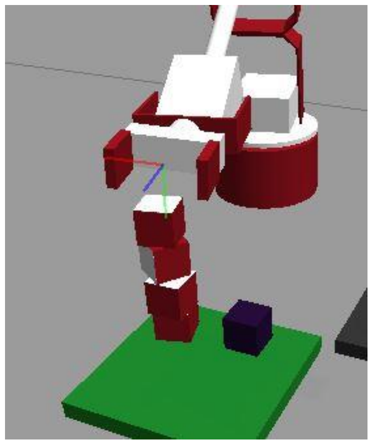
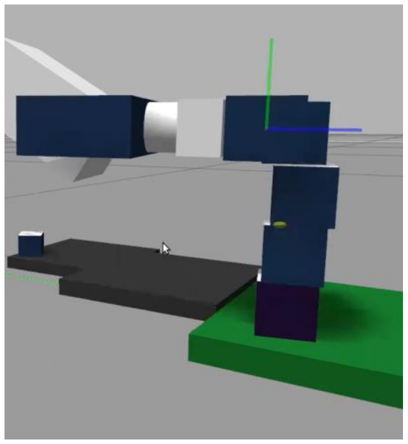
Basis the analysis table –

1. Average Success overall – 53 % (out of 9 blocks)
2. Average Success (Statics only) – 90%
3. Average Success (Dynamics only) - 20 %
4. Average Success (Tournament) – 40% (out of 9 blocks)

$$** \ success = \frac{blocks\ stacked\ \boldsymbol{with}\ side\ bonus}{blocks\ available\ at\ start}$$

Evaluation –

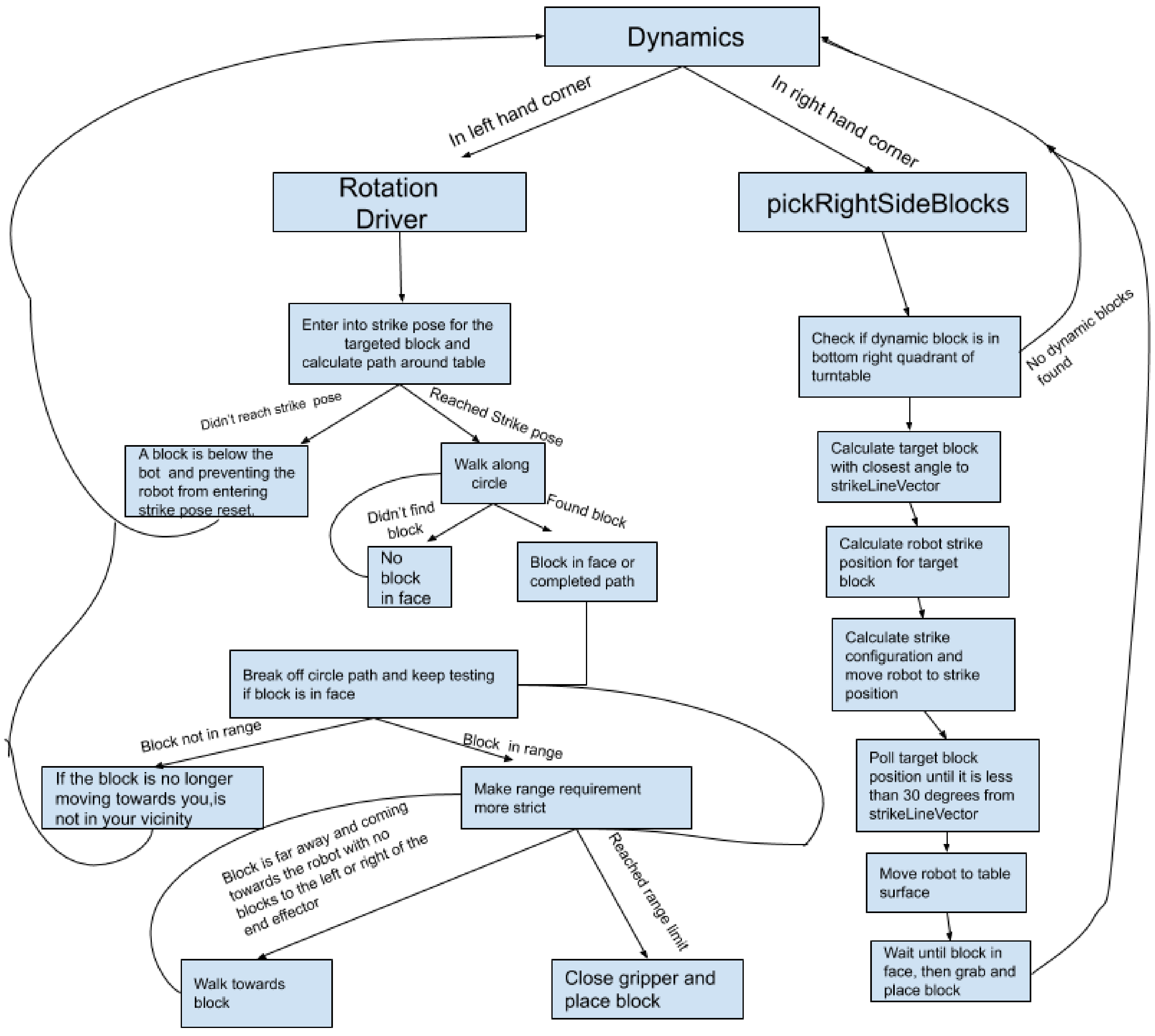
1. The static function worked quite well. As can be seen in the table above – it was able to always pick up all the four static blocks. In addition, except for blocks in infeasible orientations, it would stack all other blocks with their white side up.
2. Average success rate for static only function was 90%. During finals, it failed to stack blocks as the first picked block was in an infeasible orientation which prevented stacking of the multiple blocks. As a result, all blocks were on the floor of the ground platform.
3. The Achilles heel of our project proved to be the dynamic function. Although we had improved over time – we ran out of time to further refine it. The issue seemed to be the picking up of the dynamic block. The robot would find the block and track it easily but would not be able to grip it half the time.
4. The Toss function was our last resort before finals. It was written, tested and refined over just a couple of hours. Although, we did not get to use it during the final match – a trial run after the final tournament proved that the function worked as it was able to knock of a block from the opponent's stack. Given some more time and testing, it could have proven to be a lethal tool in the tournament.
5. For close blocks, it changed the direction of holding these blocks and avoided collisions with other blocks with a 100% success rate. For far away (infeasible) blocks, it grasped the blocks and placed them to the goal successfully in all the tests, though sometimes we failed to stack the infeasible blocks with others because of the inaccurate orientation. Note that the infeasible orientation for some blocks only affects the stacking slightly in many cases, shown in the figures below, where the blocks are not aligned but we got a 4-block-tower.

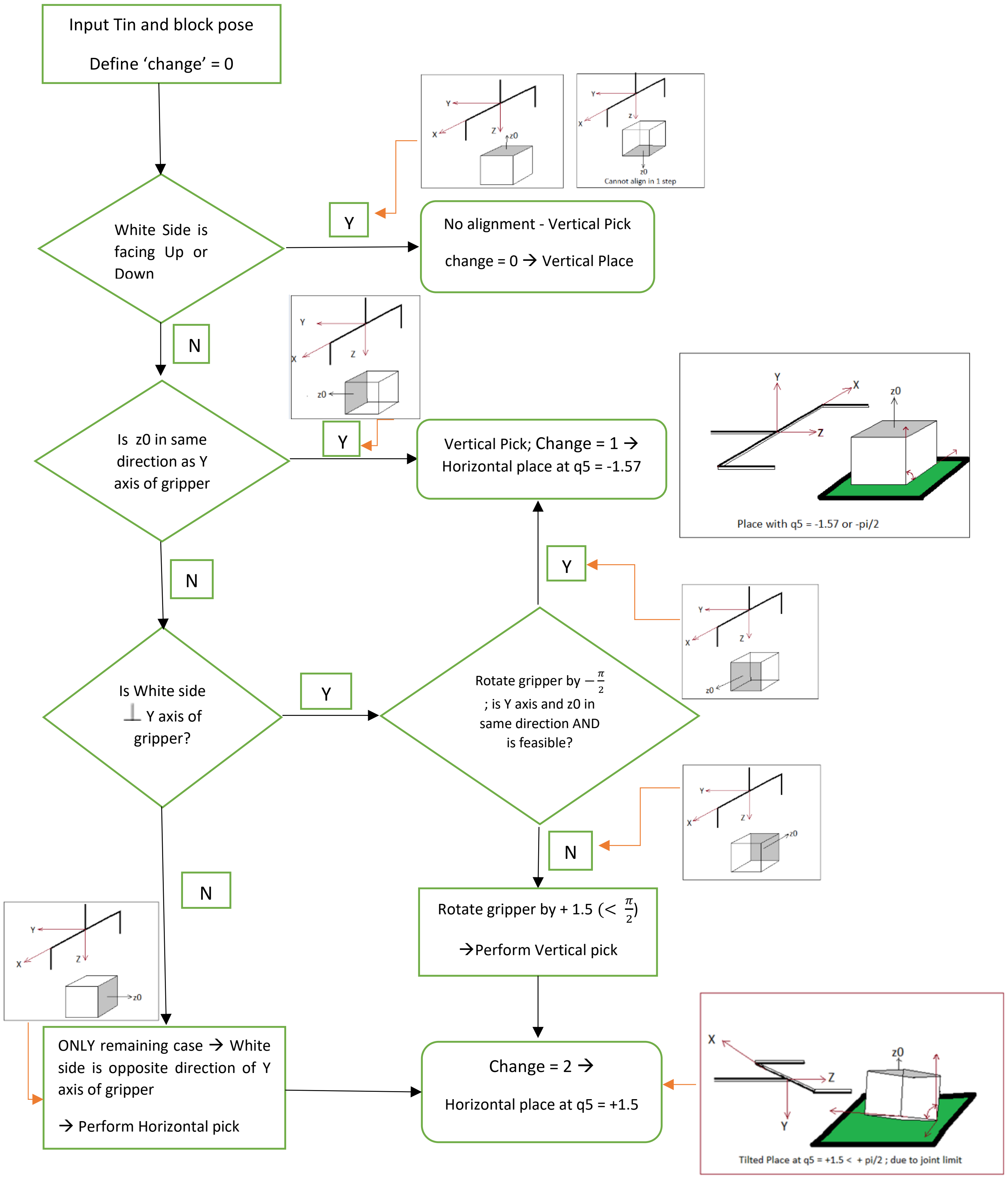
Improvement –

Barring the time limit of 1 minute, we would work on aligning an infeasibly oriented block using the white side up function. This would be a two-step process where the concept would be essentially the same as the current blocks. All that would be needed is an extra loop which would align it after pick, drop it, pick it up and align again before placing at the goal platform.

Dynamic blocks were extremely difficult to acquire and stack. This was due to the fact that the blocks were moving extremely fast relative to the movement of the lynx robot. As a result, any movement towards a block had to be recalculated to account for block movement in the transient time between finding a block and moving towards it. We developed two strategies that attempted to accomplish this. The first method used something like a POMDP, where we funneled the desired block into a desired position and grabbed it once the funnel was complete. The second strategy involved location tracking of a block. However, due to time constraints we were unable to refine either method to consistently grab 3 or more dynamic blocks. The static blocks, although a more consistent source of points, tended to knock over blocks. The cause for this was twofold; firstly, we required the robot to move extremely fast, and this came at the cost of precision. The problem was amplified by the absence of rigorous collision avoidance, which we were unable to implement due to time constraints. All in all, our planner worked remarkably, and we all became quite familiar with the challenge that is robotics.


Dynamics
In left hand corner
In right hand corner
Rotation Driver
pickRightSideBlocks
Enter into strike pose for the targeted block and calculate path around table
Didn't reach strike pose
Reached Strike pose
A block is below the bot and preventing the robot from entering strike pose reset.
Walk along circle
Didn't find block
Found block
No block in face
Block in face or completed path
Break off circle path and keep testing if block is in face
Block not in range
Block in range
If the block is no longer moving towards you,is not in your vicinity
Make range requirement more strict
Block is far away and coming towards the robot with no blocks to the left or right of the end effector
Reached range limit
Walk towards block
Close gripper and place block
Check if dynamic block is in bottom right quadrant of turntable
No dynamic blocks found
Calculate target block with closest angle to strikeLineVector
Calculate robot strike position for target block
Calculate strike configuration and move robot to strike position
Poll target block position until it is less than 30 degrees from strikeLineVector
Move robot to table surface
Wait until block in face, then grab and place block

WhiteSideUp( ) Flow chart :


Input Tin and block pose
Define ‘change’ = 0
White Side is facing Up or Down
Y
No alignment - Vertical Pick
change = 0 → Vertical Place
Cannot align in 1 step
N
Is z0 in same direction as Y axis of gripper
Y
Vertical Pick; Change = 1 →
Horizontal place at q5 = -1.57
Place with q5 = -1.57 or -pi/2
N
Is White side ⊥ Y axis of gripper?
Y
Rotate gripper by $-\frac{\pi}{2}$; is Y axis and z0 in same direction AND is feasible?
Y
N
Rotate gripper by + 1.5 ($< \frac{\pi}{2}$)
→Perform Vertical pick
N
ONLY remaining case → White side is opposite direction of Y axis of gripper
→ Perform Horizontal pick
Change = 2 →
Horizontal place at q5 = +1.5
Tilted Place at q5 = +1.5 < + pi/2 ; due to joint limit