

# EN.530.646 RDKDC

## *Final Project Report*

Group 7



JOHNS HOPKINS  
UNIVERSITY

Austin Huang • [chuan120@jhu.edu](mailto:chuan120@jhu.edu)

Danny Chang • [ichang9@jhu.edu](mailto:ichang9@jhu.edu)

John Han • [jhan62@jhu.edu](mailto:jhan62@jhu.edu)

Juo-Tung Chen • [jchen396@jhu.edu](mailto:jchen396@jhu.edu)

Yu-Chun Ku • [yku4@jhu.edu](mailto:yku4@jhu.edu)

# ***Table of Contents***

<b>I. Introduction</b>	3
<b>II. Implementation</b>	4
1) System Workflow	4
2) Inverse Kinematics	5
3) Resolved-Rate Control	6
4) Transpose Jacobian based Control	7
5) Other Implementations	9
i. Intermediate Frame Calculation	9
ii. Table Collision	10
iii. Singularity	10
6) Extra Credit	11
<b>III. Results</b>	13
1) Inverse Kinematics	13
2) Resolved-Rate Control	17
3) Transpose Jacobian based Control	21
4) Extra Credit	25
<b>IV. Discussion and Conclusion</b>	27
<b>V. Member Contribution</b>	28
<b>VI. Bibliography</b>	28

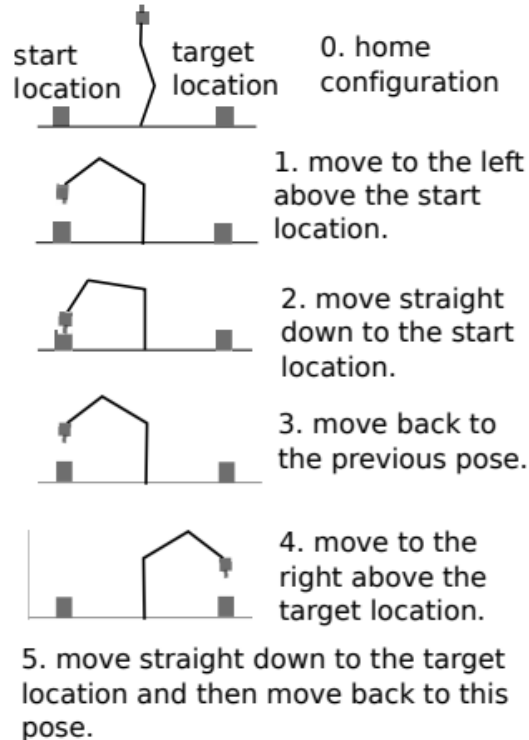
# I. Introduction

This is a report of the Final Project of Robot Devices, Kinematics, Dynamics, and Control for Team 7. Using ROS and MATLAB, we implemented an algorithm that uses various methods for a pick and place task, shown in Figure 1.

In order to achieve our results, we implemented the following algorithms:

- 1) Inverse Kinematics
- 2) Resolved-Rate Control using Differential Kinematics
- 3) Transpose-Jacobian

This report will summarize the algorithm descriptions, system workflow, results, and finally an extra credit task we did for beautiful artwork.



*Figure 1: Overview of the project task*

## II. Implementation

### 1) System Workflow

The main driver of our system is *ur5\_project.m*. Running this script will start a user interface through the command window to run our project, shown in Figure 2. Firstly, it will ask for project termination, which is the default initial state. After typing “N,” it will prompt the user for the “Start” location and the “Target” location as displayed in Figure 3. “R” records the current location of the UR5 robot arm, “T” uses the test case given to us in the project description, and the user can also choose an arbitrary 4 x 4  $SE(3)$  matrix as well.

```
The value of the ROS_MASTER_URI environment variable, http://localhost:11311, will be used to connect to the ROS master.
Initializing global node /matlab_global_node_15708 with NodeURI http://john-ubuntu18:39615/ and MasterURI http://localhost:11311.
----- Terminate ur5_project? [Y/N] ? -----
```

*Figure 2-1: Initial State: Terminate Project?*

The start, start\_above, target, and target\_above frames will then be published onto RVIZ. We also calculate a frame\_im, for an intermediate frame that the robot end-effector (EE) will go in between steps 3 and 4 of Figure 1. Then, the program asks if the extra credit demonstration is desired.

```
Shutting down global node /matlab_global_node_64116 with NodeURI http://john-ubuntu18:41579/ and MasterURI http://localhost:11311
The value of the ROS_MASTER_URI environment variable, http://localhost:11311, will be used to connect to the ROS master.
Initializing global node /matlab_global_node_06364 with NodeURI http://john-ubuntu18:42247/ and MasterURI http://localhost:11311.
----- Terminate ur5_project? [Y/N] ? -----
n
----- Demonstration for extra credit? [Y/N] ? -----
```

*Figure 2-2: Demonstration for Extra Credit Option*

Then, the user will have to input variables options to either teach/record the start/target joint, to use the test start/target location provided, to use the previous g\_start if defined, and to define the user’s own  $SE(3)$  matrix.

```
----- Input desired start location g_start -----
[R] to teach and record the current joint
[T] to use test start location provided
[P] to use previous g_start (if defined)
[-] enter g matrix directly
t

----- Input desired target location g_target -----
[R] to teach and record the current joint
[T] to use test target location provided
[P] to use previous g_target (if defined)
[-] enter g matrix directly
```

*Figure 3-1: User choices for defining start and target locations*

```

----- Frames initiation completed -----

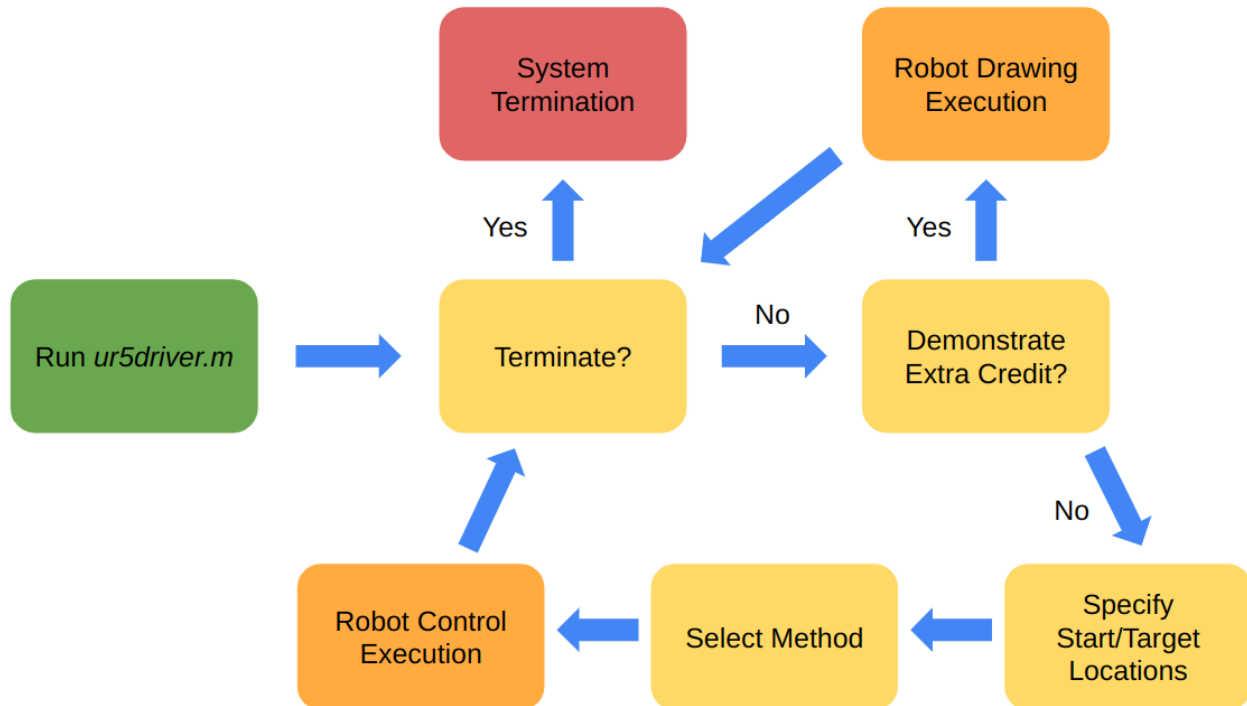
*** [Trajectory Out of URS Workspace]
Trajectory from start to target position may cause URS collision.
Re-route Move-And-Place task using calculated intermediate configuration.

----- Select methods for move-and-place task: -----
[IK]: Inverse Kinematics
[RR]: Resolved-rate Control using Differential Kinematics
[TJ]: Resolved-rate Control using Transpose-Jacobian

```

*Figure 4: Selecting Method for Pick-and-Place*

Afterwards, the system will prompt the user for a final instance to choose which algorithm will be used: “RR” will use Resolved-Rate method, “TJ” will use Transpose Jacobian method, and “IK” will use inverse kinematics, for example shown in Figure 4. Figure 5 displays the overall workflow of our driver. We now cover our implementation of these algorithms.



*Figure 5: Workflow of Algorithm Driver. The yellow boxes are steps that require user input, and the orange boxes are robot processes.*

## 2) Inverse Kinematics

We were given this function, which computes the inverse kinematics by using Denavit-Hartenberg (D-H) parameterization based on the Ryan Keating home position. However, the function computes eight solutions to the inverse kinematics problem. To solve for the most optimal of these

eight, we wrote a script called *minNormIdx.m*, which chooses one of the eight by **choosing the solution with the lowest norm between the current robot joints and the desired robot joints configuration**. For example, call the robot's current joints  $\theta_{ct}$ . Out of the 8 configurations, we choose the optimal  $\theta_{best}$  as:

$$\theta_{best} = \operatorname{argmin} ||\theta - \theta_c||$$

where  $\theta$  is one of the joints given by the solution of *ur5InvKin.m*. Then we use *move\_joints* using  $\theta_{best}$ . The solution to solve the table collision problem will be expanded in the "Other Implementation" section.

### 3) Resolved-Rate Control

First, we set the tolerance values for convergence purposes. Figure 6 shows a pseudo-code of our RR control algorithm.

```
while belowDefinedTolerance()
    Tstep = calculateOptimalTimeStep(Body_Jacobian, Xi)
    checkSingularity()
    qk_next = updateJoints(qk)
    checkTstepSpeedLimit(qk_next, Tstep)
    qk = wrapToPi(qk_next)
    gk = ur5ForwardKinematics(qk)
    checkTableCollision(gk)
    checkdtSpeedLimit(qk_next, Tstep)
    ur5.MoveJoints(gk, dt)
    xi = updateXi(g_desired-1 * gk)
end
err = calculateError(g_desired, gk)
```

Figure 6: Pseudo-code of Resolved-Rate Control

The algorithm operates within a while loop based on convergence. In each time step, we had an adaptive time step that changed based on our current location and the body Jacobian. According to [1], there is a closed form solution to an optimal Time Step. Following this method, we updated our time step as:

$$Tstep = \frac{\xi^T (J_b J_b^T \xi)}{||J_b J_b^T \xi||^2}$$

Next, use *manipulability.m* to make sure that we are not in a singularity. If we are in a singularity, we move from the current configuration. The specifics of Singularity avoidance will be discussed in “Other Implementations.” Otherwise, we update our joint variables  $\theta$  based on the following equation:

$$\theta_{t+1} = \theta_t - K\Delta T J_b^{-1} \xi$$

where  $\theta_t$  is the current joint variables,  $K$  is the control gain,  $\Delta T$  is the time step, and  $\xi$  is the current forward kinematics twist coordinates.

Then, a speed limit control was implemented after the joint variables were implemented. We noticed that sometimes, the next iteration position was too far away, and the time step was not long enough for the robot to get to that position. We calculate a joint *velocity*, which is the difference between the current joint and previous joint space. Then, we compare the maximum value of the joint velocity with the ur5 speed limit. In the case that it is exceeded, we increase Tstep and recalculate the new joints.

Next, we wrapped all the values in our joint variables to be between  $[-\pi, \pi]$ . This causes the trajectory of the robot to “loop” around once it gets near the boundary. We implemented this because without this clamping procedure, the robot would completely miss the target.

Afterwards, we calculated the forward kinematics to calculate the  $SE(3)$  transformation corresponding to the joint variables. Afterwards, we made sure that the robot would not collide with the table by limiting the joint variables.

Finally, we have a *second* speed limit check. At this point, the forward kinematics have been calculated. In the case that the joint variable approaches the minimum or maximum value within  $[-\pi, \pi]$ , the robot has no choice but to do a 360 degree turn. However, the time step is too small if the robot has to turn around to avoid the issue of missing the target. In this speed limit, we simply increase the time step significantly to account for this.

Afterwards, we moved the UR5 robot with the new joint variables, and also updated our twist coordinates. In our test cases and experiments, we generally converge by minimizing the error at each iteration. The thresholds and constants are specified in the Results section.

#### 4) Transpose Jacobian based Control

The Transpose Jacobian was a little trickier. However, the overall concept is similar to RR control. Figure 8 is the Pseudo-Code of the Transpose Jacobian algorithm.

```

while below defined Tolerance
    Tstep = calculateOptimalTimeStep(Body_Jacobian, Xi)
    K = calculateOptimalGain(Tstep)
    checkSingularity()
    qk_next = updateJoints(qk)
    checkTstepSpeedLimit(qk_next, Tstep)
    qk = wrapToPi(qk_next)
    gk = ur5ForwardKinematics(qk)
    checkTstepSpeedLimit2(qk_next, Tstep)
    checkTableCollision(gk)
    ur5.MoveJoints(gk, dt)
    xi = updateXi(g_desired-1 * gk)
end
err = calculateError(g_desired, gk)

```

*Figure 7: Pseudo-code of Transpose Jacobian Control*

Overall, the Transpose Jacobian is relatively similar to resolved rate control. As a result, we will only point out the components of the method that differ from RR. The most distinguishing component of this method in contrast to the previous method is the joint updating step, performed by:

$$\theta_{t+1} = \theta_t - K \Delta T J_b^T \xi$$

We noticed that this method was much more erratic and prone to undesired behavior. Some examples include oscillating at an increasing speed/distance, common singularity problems, and inefficient trajectories towards the goal. As a result, we focused on a few aspects to alter from RR control for stable performance of TJ.

1. Adaptive Gain Control
2. Time Step Regulator

In Resolved Rate, we changed the time step at each iteration in order to account for the distance from the target. In Transpose Jacobian, we also added an update to the gain at every iteration with the following equation:

$$K = G \frac{\omega D}{C \Delta T}$$

where  $G$  is a gain tuning parameter,  $\omega$  is the natural frequency of the system,  $D$  is the damping ratio of the system,  $C$  is a constant proportional to the maximum velocity of robot actuators, and



$\Delta T$  is the current time step at that iteration. If  $G$  is small enough, we found that this had successful results, while also having an inverse relationship with the Time Step, which was ideal since they are multiplied together in the joint variable update step.

Secondly, we added a time step regulator in addition to the speed limit checks done similar to the resolved rate method. We had to clamp the time step as a function of the iteration. For example, in the first ten iterations, the Tstep was 2 seconds at the maximum, whereas it was 0.01 at the maximum in future iterations. This is because initially, TJ tended to oscillate very heavily and often into speed limit or table collision issues.

In our test cases and experiments, we generally converge by minimizing the error at each iteration. The thresholds and constants are specified in the Results section.

## 5) Other Implementations

### i. Intermediate Frame Calculation

In addition to these three tasks, there were other relevant implementations that we incorporated. The first of these is an intermediate frame calculation.

Between steps 3 and 4 in Figure 1, we had to impose a constraint so that the robot would not enter the “dead-zone” or inside the minimum workspace, defined in the UR5 user manual Figure 2.1:

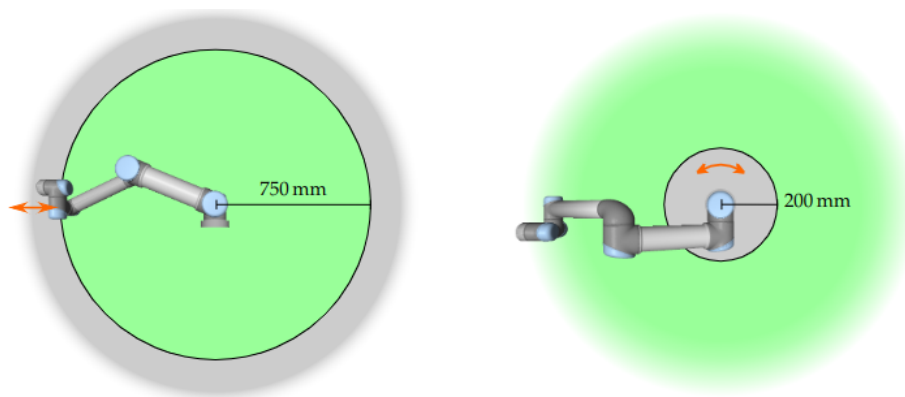


Figure 2.1: Certain areas of the workspace should receive attention regarding pinching hazards, due to the physical properties of the robot arm. One area is defined for radial motions, when the wrist 1 joint is at a distance of at least 750 mm from the base of the robot. The other area is within 200 mm of the base of the robot, when moving in the tangential direction.

In order to avoid entering the minimum workspace (ie 20 cm within the base) we calculated an “intermediate frame” between the start and target goals. After checking to make sure the inputted

start and end goals were within the workspace, we calculated the intermediate frame in the following manner.

First, we calculated the midpoint of the positions of *start* and *target*. There is a high probability that this midpoint will be within the minimum workspace. In this case, we moved the x and y coordinates of the frame by 25 cm and moved the frame up by 15 cm in the z direction. These values allowed consistent results for a smooth transition from start to target locations.

## ii. Table Collision

The following pseudo code is an overview of our solution to the table collision problem.

```
z_3 = calculateJoint3Height(qk)
z_EE = calculateEEHeight(qk)
if z_3 <= 0 or Joint2 >= 0 or Joint2 <= -pi
    qk(2) = -pi/2
end
if z_EE <= 0
    qk(3) = -sign(qk(3)) * pi/4
    qk(2) = -pi/2
end
```

*Figure 8: Pseudo code of table collision solution*

To summarize, we calculated the forward kinematics of joint3, which is the main perpetrator for table collision. To do this, we changed *ur5FwdKin.m* to accept an optional argument to calculate the forward kinematics of an intermediate joint instead of just the end effector. In addition, we checked the value of joint 2, which is responsible for the angle of the robot and the table.  $Z_3$  is the height of the third joint, and we checked if that value is less than or equal to 0 or if joint 2 is far enough to collide with the table. In addition, we checked the height of the end-effector, which will give us information of how high it is from the table. In this case, we change the third joint to be  $\pi/4$  or  $-\pi/4$  in order to remove it from the table while maintaining a decent trajectory.

## iii. Singularity

We used *detjac* in order to detect a singularity. We realized that running into singularity is a natural part of robot control. We set the manipulability threshold constant to be  $10^{-4}$ . When it is detected, we check which joint is causing the singularity, since we know that it is only joints 3 and 5 that will cause the singularity case. The following is the pseudo code for our singularity handling.

```

if Singularity
    if joint3 < pi/180 or joint3 - pi < pi/180
        joint3 = sign(joint3) * pi/4
    else if joint5 < pi/180 or joint5 - pi < pi/180
        joint5 = sign(joint5) * pi/4
    end
end
end

```

*Figure 9: Pseudo code of handling singularity*

To summarize, we check if joint3 and joint5 are close to a singularity, which is 0 or  $\pi$ . In that case we add  $\pi/4$  to the direction that it was originally facing. We incorporated the sign in order to encourage the robot to stay relatively closer to the trajectory.

## 6) Extra Credit

For our extra credit, we attempted to draw the Johns Hopkins University logo (shown in Figure ) with the UR5 robot.



*Figure 10: JHU logo that the UR5 Robot will draw*

We integrated image processing techniques in order to achieve this. First, we calculated the boundary by using the MATLAB image processing toolbox. Then, we converted the boundary into keypoints that the robot can use to draw the logo. During this step, we also scaled the keypoints so that the logo could be large enough to capture all the details.

The keypoints were organized into data structures that the robot could use as waypoints. Specifically, they were grouped up by region so that the robot could draw smoothly on the flat surface. Since these are 3D points, we used inverse kinematics to calculate the joint angles for each

waypoint. Since they are grouped by regions, the robot can draw each region at a time to complete the picture.

### III. Results

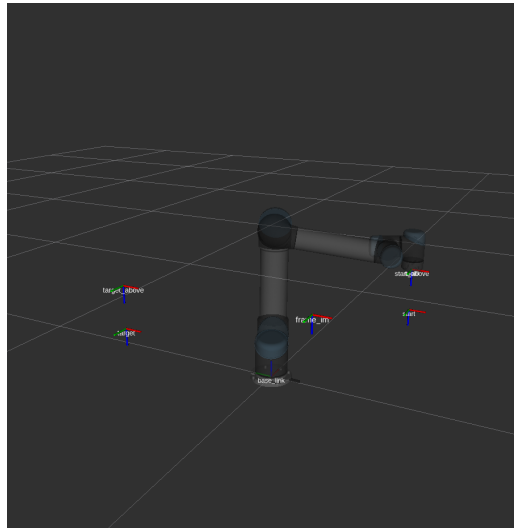
After the robot has converged to the final destination based on the tolerance value that we set, we calculate the error based on the equations given to us in the project description, shown in Figure 11.

$$d_{SO(3)} = \sqrt{\text{Tr}((R - R_d)(R - R_d)^T)}$$
$$d_{\mathbb{R}^3} = \|\mathbf{r} - \mathbf{r}_d\|.$$

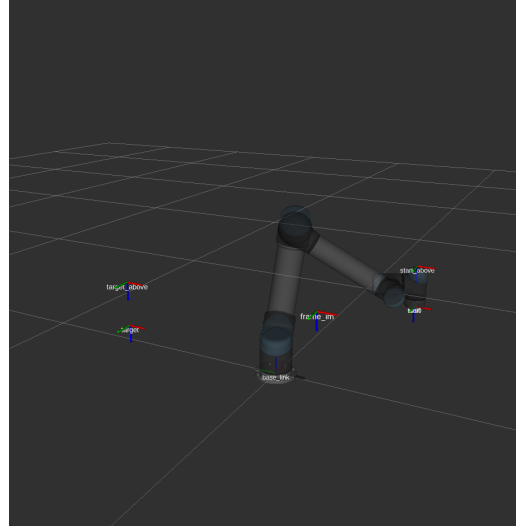
Figure 11: Error Calculation given in the Project Description

#### 1) Inverse Kinematics

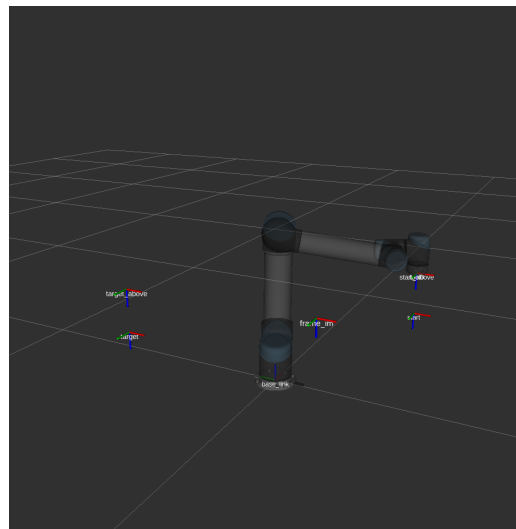
Table of Test Sequence:

Description	Image
<i>start_above</i>	

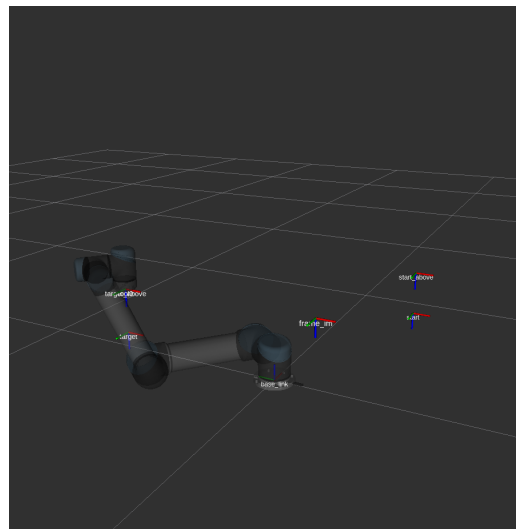
*start*



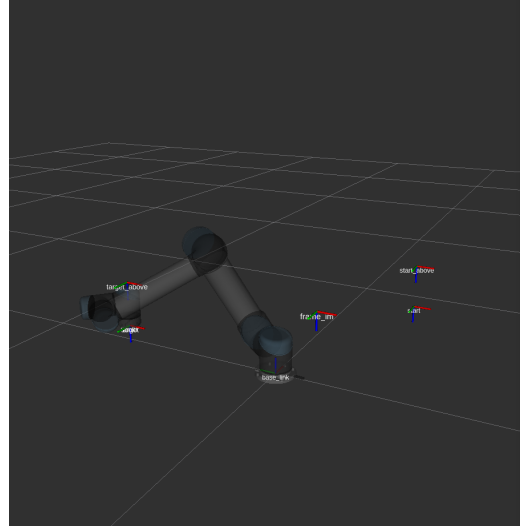
*Back to start\_above*



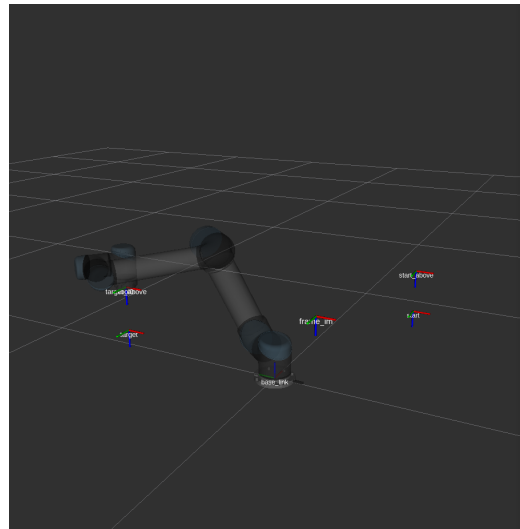
*target\_above*



*target*



*target\_above*



## Command Window:

```
----- Select methods for move-and-place task: -----
[IK]: Inverse Kinematics
[RR]: Resolved-rate Control using Differential Kinematics
[TJ]: Resolved-rate Control using Transpose-Jacobian
ik

-> URS started moving to intermediate home configuration.
<- URS moved to intermediate home configuration.

-> URS started moving to g_start_above.
<- URS moved to g_start_above.

-> URS started moving to g_start.
<- URS moved to g_start.

-> URS started moving to g_start_above.
<- URS moved to g_start_above.

-> URS started moving to g_target_above.
<- URS moved to g_target_above.

-> URS started moving to g_target.
<- URS moved to g_target.

-> URS started moving to g_target_above.
<- URS moved to g_target_above.

-> URS started moving to intermediate home configuration.
<- URS moved to intermediate home configuration.

----- Errors for Start Configuration and Target Configuration -----
error_start_pos = 0.0007 [cm]
error_start_rot = 0.0011 [deg]
error_target_pos = 0.0011 [cm]
error_target_rot = 0.0021 [deg]

----- Duration of execution -----
duration = 36.4205 [s]
```

*Figure 12: RVIZ and Command window results for Inverse Kinematics*

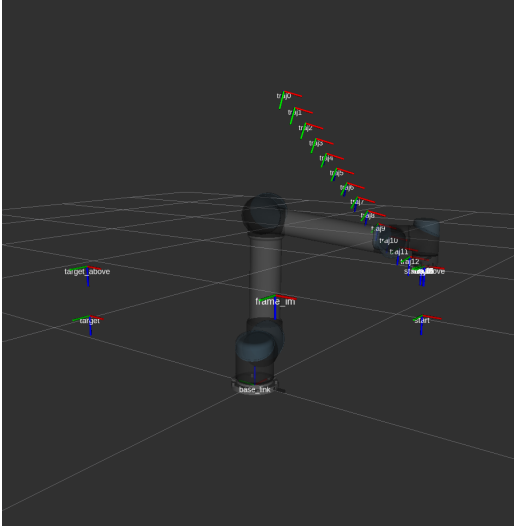
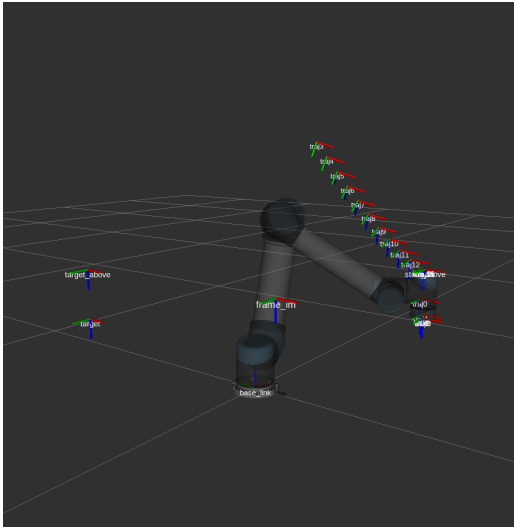
## Summary of results:

<b>Start Position Error</b>	0.0007 cm
<b>Start Rotation Error</b>	0.001 deg
<b>Target Position Error</b>	0.0011 cm
<b>Target Rotation Error</b>	0.0021 deg
<b>Execution Time</b>	36.4205 s

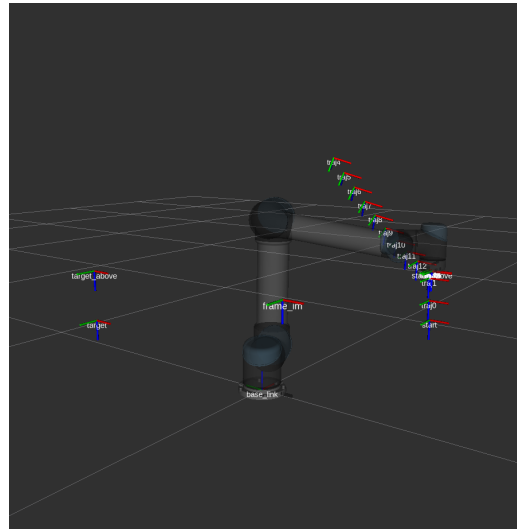


## 2) Resolved-Rate Control

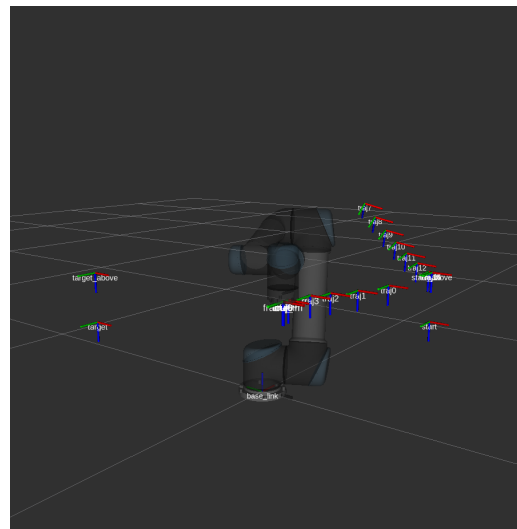
Table of Test Sequence:

Description	Image
<i>start_above</i>	 A 3D simulation of a robotic arm in a dark environment with a grid floor. The arm is in a 'start_above' pose, with its base labeled 'base' and its end effector labeled 'target'. A series of red arrows labeled 'target' points from the end effector towards a target point labeled 'target' on the floor. The target point is also labeled 'target' and 'target_above'.
<i>start</i>	 A 3D simulation of a robotic arm in a dark environment with a grid floor. The arm is in a 'start' pose, with its base labeled 'base' and its end effector labeled 'target'. A series of red arrows labeled 'target' points from the end effector towards a target point labeled 'target' on the floor. The target point is also labeled 'target' and 'target_above'.

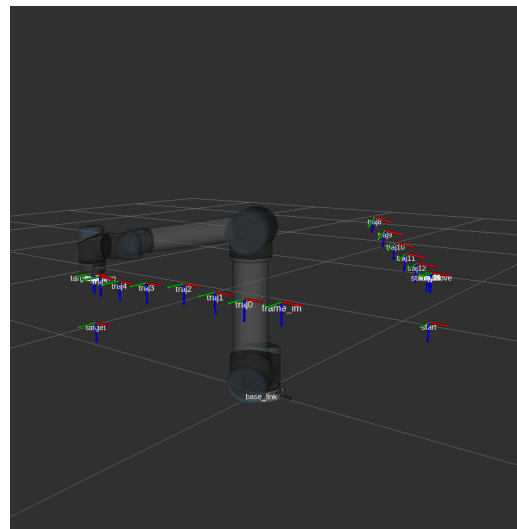
*Back to start\_above*



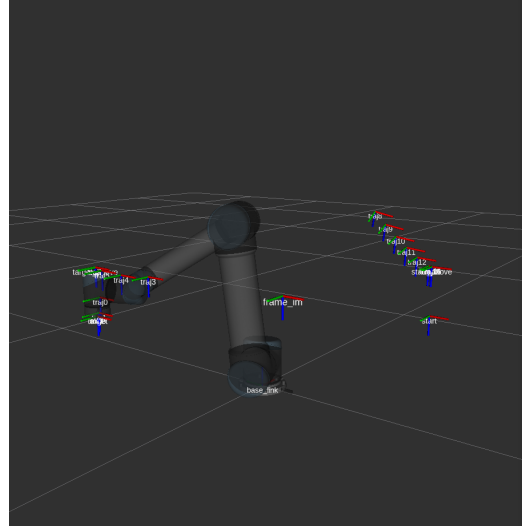
*intermediate frame*



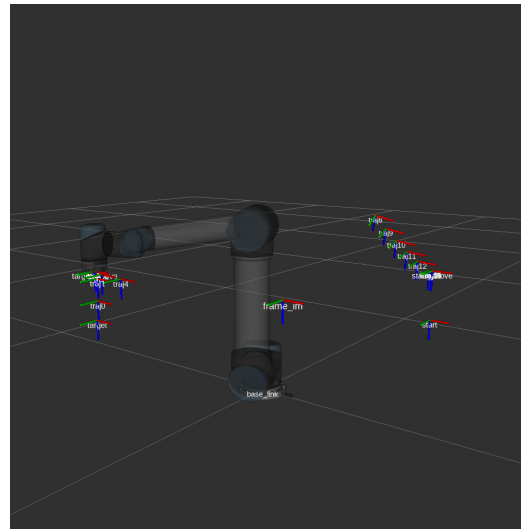
*target\_above*



*target*



*Back to target\_above*



## Command Window:

```

----- Select methods for move-and-place task: -----
[IK]: Inverse Kinematics
[RR]: Resolved-rate Control using Differential Kinematics
[TJ]: Resolved-rate Control using Transpose-Jacobian
rr

-> URS started moving to intermediate home configuration.
<- URS moved to intermediate home configuration.

-> URS started moving to g_start_above.
<- URS moved to g_start_above.

-> URS started moving to g_start.
<- URS moved to g_start.

-> URS started moving to g_start_above.
<- URS moved to g_start_above.

-> URS started moving to g_im.
<- URS moved to g_im.

-> URS started moving to g_target_above.
<- URS moved to g_target_above.

-> URS started moving to g_target.
<- URS moved to g_target.

-> URS started moving to g_target_above.
<- URS moved to g_target_above.

-> URS started moving to intermediate home configuration.
<- URS moved to intermediate home configuration.

----- Errors for Start Configuration and Target Configuration -----
error_start_pos = 0.0630 [cm]
error_start_rot = 0.0000 [deg]
error_target_pos = 0.0614 [cm]
error_target_rot = 0.0000 [deg]

----- Duration of execution -----
duration = 47.4940 [s]

```

*Figure 13: RVIZ and Command window results for Resolved Rate Control*

## Summary of parameters and thresholds:

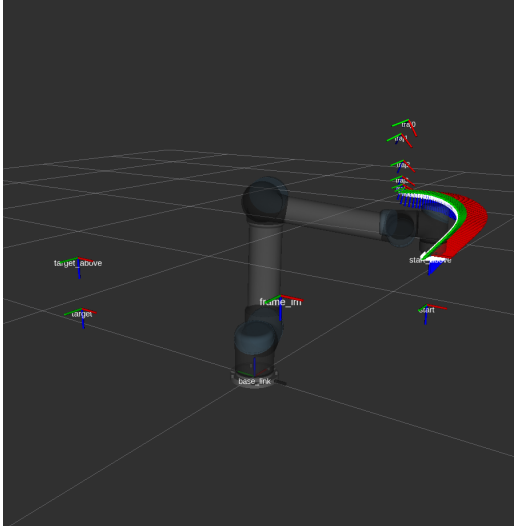
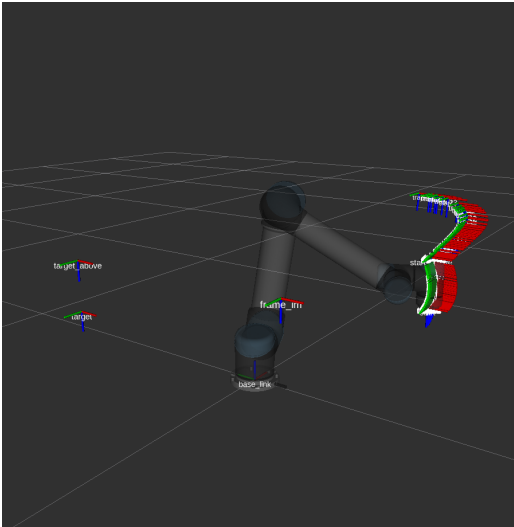
<b>vk threshold</b>	$10^{-3}$
<b>wk_threshold</b>	$\text{deg2rad}(0.1)$
<b>manipulability_threshold</b>	$10^{-3}$

## Summary of results:

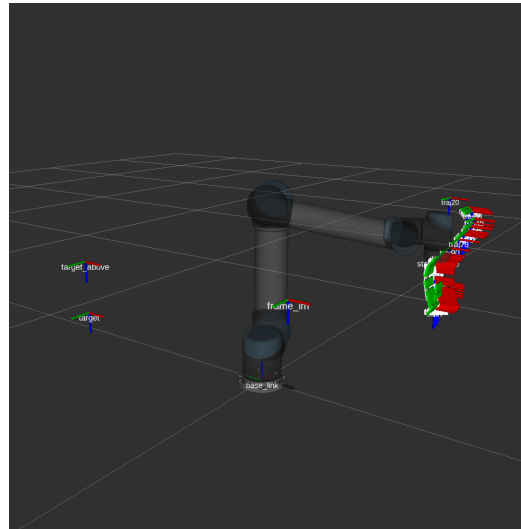
<b>Start Position Error</b>	0.0630 cm
<b>Start Rotation Error</b>	0.0000 deg
<b>Target Position Error</b>	0 0614 cm
<b>Target Rotation Error</b>	0.0000 deg
<b>Execution Time</b>	47.4940 s

### 3) Transpose Jacobian based Control

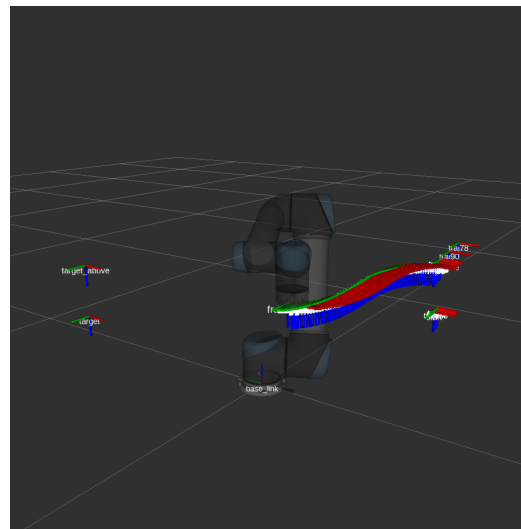
Table of Test Sequence:

Description	Image
<i>start_above</i>	 A 3D simulation of a robotic arm in a dark environment with a grid floor. The arm is in a 'start_above' pose, with its end effector positioned above a target point. A red and green trajectory line is visible, indicating the path of the end effector. Labels include 'target_above', 'origin', 'frame_in', 'frame_out', and 'start'.
<i>start</i>	 A 3D simulation of a robotic arm in a dark environment with a grid floor. The arm is in a 'start' pose, with its end effector positioned above a target point. A red and green trajectory line is visible, indicating the path of the end effector. Labels include 'target_above', 'origin', 'frame_in', 'frame_out', and 'start'.

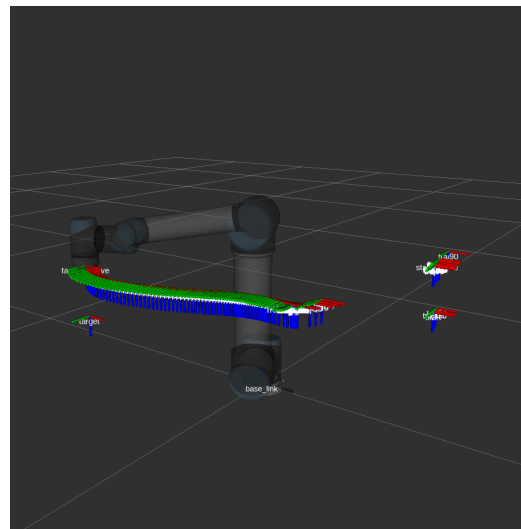
*Back to start\_above*



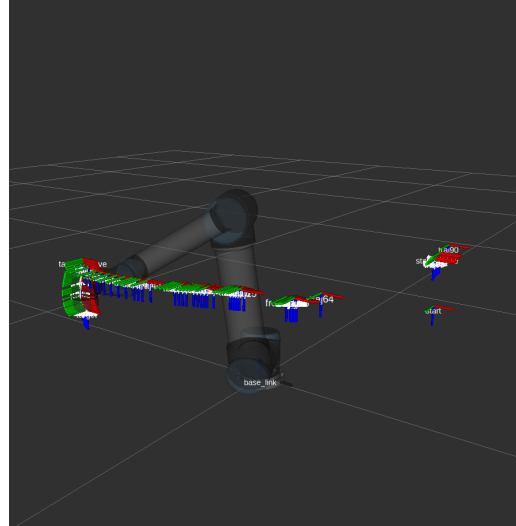
*intermediate frame*



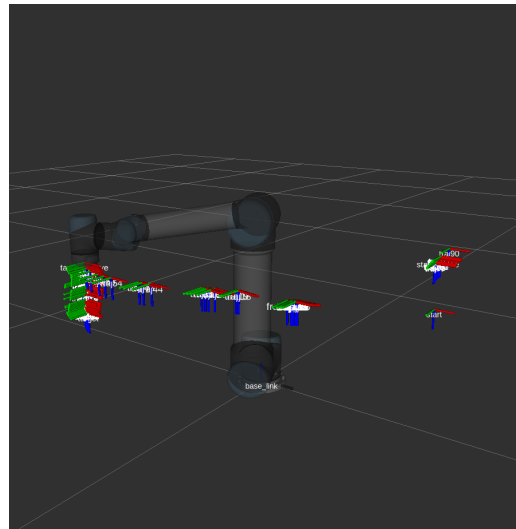
*target\_above*



*target*



*Back to target\_above*



## Command Window:

```

----- Select methods for move-and-place task: -----
[IK]: Inverse Kinematics
[RR]: Resolved-rate Control using Differential Kinematics
[TJ]: Resolved-rate Control using Transpose-Jacobian
tj

-> URS started moving to intermediate home configuration.
<- URS moved to intermediate home configuration.

-> URS started moving to g_start_above.
<- URS moved to g_start_above.

-> URS started moving to g_start.
<- URS moved to g_start.

-> URS started moving to g_start_above.
<- URS moved to g_start_above.

-> URS started moving to g_im.
<- URS moved to g_im.

-> URS started moving to g_target_above.
<- URS moved to g_target_above.

-> URS started moving to g_target.
<- URS moved to g_target.

-> URS started moving to g_target_above.
<- URS moved to g_target_above.

-> URS started moving to intermediate home configuration.
<- URS moved to intermediate home configuration.

----- Errors for Start Configuration and Target Configuration -----
error_start_pos = 0.5509 [cm]
error_start_rot = 0.1399 [deg]
error_target_pos = 0.5349 [cm]
error_target_rot = 0.1409 [deg]

----- Duration of execution -----
duration = 433.1358 [s]

```

Figure 14: RVIZ and Command window results for Transpose Jacobian

## Summary of parameters and thresholds:

<b>vk threshold</b>	$10^{-2}$
<b>wk_threshold</b>	$\text{deg2rad}(0.1)$
<b>manipulability_threshold</b>	$10^{-4}$
<b>Gain Tune</b>	0.2618

## Summary of results:

<b>Start Position Error</b>	0.5509 cm
<b>Start Rotation Error</b>	0.1399 deg
<b>Target Position Error</b>	0.5349 cm
<b>Target Rotation Error</b>	0.1409 deg
<b>Execution Time</b>	433.1358 s

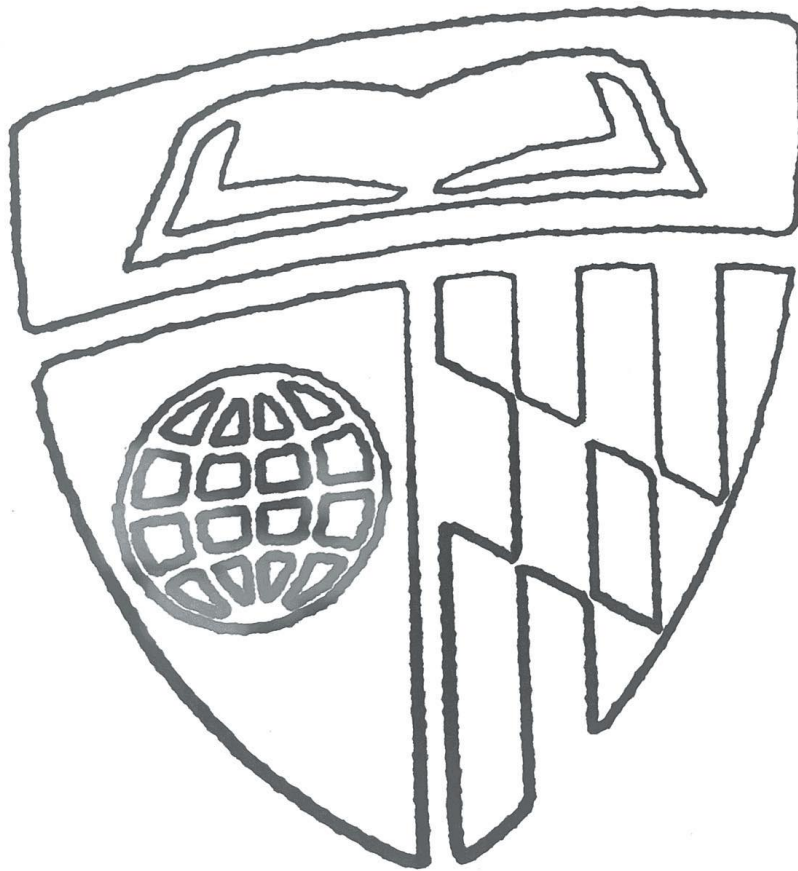


#### 4) Extra Credit

In simulation, we drew the JHU logo using one of the axes of the updated frames.



In simulation, the logo looks perfect. We ran the UR5 robot to draw on a piece of paper by attaching a marker to the end effector.



which also looks very decently. The YouTube link that shows the UR5 drawing our logo can be found at <https://youtube.com/shorts/koWIBAzIVos?feature=share>.

## IV. Discussion and Conclusion

In conclusion, we implemented a simple pick and place task with the UR5 by implementing three tasks: Inverse Kinematics, Resolved Rate control, and Transpose Jacobian. All three methods were largely successful. IK was the fastest at around 36 seconds, RR was second fastest at around 47.5 seconds, and TJ was by far the longest at around 7 minutes. This is because TJ tends to be highly erratic and unstable in behavior, so we had to take it very slow. In terms of accuracy, IK was the most accurate, and TJ was the least accurate. This is also because we had to raise the tolerance of TJ in order to achieve convergence faster. However, we still see it as a huge accomplishment; our first iteration of the TJ method was around 24 minutes per trajectory (would have been around 3 hours to run in total!).

There are some reasons why our methods were largely successful. First, we were able to take into account the majority of the safety and edge cases. We accounted for improper start/target location based on the UR5 workspace, table collisions, joint limit, speed limits, singularities and avoiding the minimum workspace for all three tasks. We did this by implementing an adaptive gain and time step, calculating various forward kinematics, checking the joint variables, and clamping values accordingly to ensure good performance at every iteration. This meticulous approach allowed us to cover for many edge/failure cases while optimizing the speed and performance of the algorithm.

We worked very hard for this project, and enjoyed learning how to use MATLAB and ROS in order to control the robot efficiently and accurately.

## V. Member Contribution

Name	Contribution
Austin Huang	<ul style="list-style-type: none"><li>▪ Created all draft codes.</li><li>▪ Fine tuned and finalized RRcontrol, TJcontrol.</li><li>▪ Fine tuned and finalized extra credit draft codes and functions.</li><li>▪ Fine tuned and finalized safety checks and handle cases codes.</li></ul>
Danny Chang	<ul style="list-style-type: none"><li>▪ Created all draft codes.</li><li>▪ Created all function codes.</li><li>▪ Fine tuned and finalized RRcontrol, TJcontrol, IKcontrol.</li><li>▪ Created and finalized main driver code.</li><li>▪ Fine tuned and finalized safety checks and handle cases codes.</li></ul>
John Han	<ul style="list-style-type: none"><li>▪ Final Report.</li><li>▪ Fine tuned and finalized TJcontrol.</li><li>▪ Assisted with finalizing safety checks and handle cases codes.</li></ul>
Juo-Tung Chen	<ul style="list-style-type: none"><li>▪ Drafted and tested IKcontrol parent function code.</li><li>▪ Tested all basic function codes.</li><li>▪ Assisted with safety checks and handle cases codes.</li></ul>
Yu-Chun Ku	<ul style="list-style-type: none"><li>▪ Drafted and tested IKcontrol parent function code.</li><li>▪ Tested all basic function codes.</li><li>▪ Assisted with safety checks and handle cases codes.</li></ul>

## VI. Bibliography

[1] Buss, Samuel. *Introduction to Inverse Kinematics with Jacobian Transpose, PseudoInverse and Damped Least Square Methods*. October, 2009. <http://graphics.cs.cmu.edu/nsp/course/15-464/Spring11/handouts/iksurvey.pdf>