Industrobot4.0: Pick and Place PUMA 560 Robot

[1]Kalra, Nitesh; [1]Sanghai, Nikunj, and [1]Wani, Shubham Kiran
Team # 14, [1]Graduate Program in Mechanical Engineering Department, UCLA

***Abstract*—In this work, trajectory tracking control for Unimation PUMA 560 using Computed Torque Control, PD Control with Gravity Compensation and Inverse Dynamics Control methods have been presented. The project mainly focuses on implementing the simulation models and comparing the results achieved in the three strategies for a high-speed pick and place operation. A proof of concept simulation of a conveyor and its object tracking using MATLAB image processing toolbox is also implemented.**

Keywords— Pick-and-Place robots, PUMA 560, Trajectory Generation, Inverse Dynamics Control, PD Control with Gravity Compensation, Computed Torque Control, Error Analysis.

I. INTRODUCTION

The primary motivation for this project stems from the advent of Industry 4.0 and the increasing material handling automation market. The global pick-and-place robot market was valued at USD $148.1 million in 2020 and it is expected to reach USD $2870.13 million by 2026 ("Piece Picking Robots Market | 2022 - 27 | Industry Share, Size, Growth"). The current COVID pandemic has made e-commerce a USD $357 billion industry. Staffing in warehouses is difficult and exacerbated by supply chain issues (Feger, 2022). These can be mitigated by using automated sorting and packing robots which will greatly increase efficiency and reduce time to delivery.

Our work focuses on a high-speed pick-and-place application in the warehouse environment involving a threefold task - Object Interception, Trajectory Tracking and Object Placement.

II. LITERATURE REVIEW

Shuman et. al developed an inverse dynamic controller for a 6-axis robot that could converge the output signal to the reference signal while encountering noise disturbance with minor error for a dynamic model (Shuman 2020). Anderson et. al demonstrated that a simple PID control on PUMA 560 is unable to control performance satisfactorily, provided links were not considered to be decoupled. (Anderson 1988). The work in the past demonstrated the effectiveness of centralized controllers in dealing with dynamically coupled scenarios while highlighting the shortcomings of decentralized controllers under the same boundary conditions. PUMA 560 is a great example where the links are dynamically coupled.

There have been attempts to demonstrate tracking on PUMA 560 and other industrial robots using feedback linearization control supplemented by a Kalman Filter. (Zakeri, Moezi, and Zare 2014). Similarly, there have been implementation/proof of concept studies implementing Sliding Mode Controller or Computed Feedforward controller on PUMA 560 under very controlled conditions (Piltan et al. 2012), (Piltan). Although there has been a comparative study that has demonstrated the performance of controllers for tracking purposes (Akkar and Haddad 2020), the authors could not find a comparative study demonstrating the performance of centralized controllers and decentralized controllers in joint space. The task requires dynamically coupled, high-speed, short cycle time scenarios, where tracking, interception, and repeatability are required. This project aims to demonstrate and compare the performance of three controllers: Computed Torque Feedforward Control, Inverse Dynamics Control, and PD Control with Gravity Compensation. Explicit values for inertia matrix, Coriolis matrix, and other relevant parameters were taken from the Stanford Artificial Intelligence Laboratory study (Armstrong et al.)

III. OBJECTIVES OF THE CONTROLLER DESIGN

For this project, the controller is designed for efficient trajectory tracking application, error tracking, and gradual performance improvement. The project covers the implementation of 3 different control strategies and comparative performance analysis. The desired trajectory and the KPIs are presented below.

Trajectory Generation

The End Effector needs to follow the trajectory in 3D space defined by the three subtasks. The trajectory includes the following phases-

Phase 1: Object Interception (cycle time = 3 sec).

Phase 2: Trajectory Tracking (cycle time = 2 sec).

Phase 3: Object Placement (cycle time = 4 sec).

The time segment for Phase 2 is calculated based on the standard conveyor speed of 0.3 m/s ("Conveyor Speed Calculator & FPM Formula Guide") and the average time taken to complete the wrapping grasp action of 2 sec (J. Falco et al, 2020).

The time segments for Phases 1 and 3 are self-defined, intending to optimize the total cycle time and thus, aiming for a slightly high-speed movement as compared to Phase 2. This was decided well within the joint speed limits of the PUMA 560 manipulator. The location of the object interception and object placement station was decided after factoring in the accessible workspace and to avoid conveyor interference.

The team used the multi-segment, multi-axis trajectory for the defined task. A custom function was defined to generate this trajectory as per the available inputs and required outputs. This function makes use of a predefined function 'mstraj()'

from Peter Corke Robotics Toolbox (Corke 2017). Each segment is linear motion and polynomial blends connect the segments. Figure 1 below represents the desired end-effector trajectory in 3D Cartesian space.

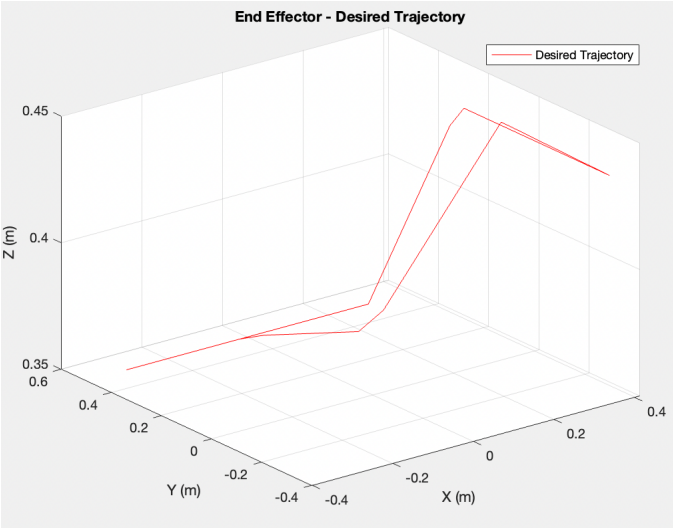


Figure 1: End Effector Desired Trajectory

Key Performance Indicators (KPIs)

The end-to-end time for sorting and placement should be minimized to increase the packages sorted per robot hour. An ambitious target of 9 seconds was selected and the trajectory was generated. Furthermore, this must be achieved by using no more than 75% of the robot's max torque to ensure power efficiency and sufficient factor of safety. The error tolerance was less than 2 cm in the end-effector position.

IV. METHODS

Assumptions

The research focuses on the modeling and effects of the first three joints. The last three axes are considered to be locked and thus their effect is neglected. The elbow-up configuration of the PUMA 560 Robot is considered the home configuration. Conveyor speed is assumed to be constant throughout the simulation.

Kinematics and Dynamics

A common method for constructing the forward kinematics of a robot arm is through Denavit-Hartenberg (DH) parameters. The DH parameters for the configuration (Corke and Armstrong-Helouvry 1994) considered are as follows in Table 1. For kinematics, fkine() and ikine() functions from Peter Corke Robotics Toolbox were employed to compute Forward and Inverse Kinematics respectively. (Corke 2017)

For dynamics, explicit equations were used from the available literature, and functions were defined to compute the dynamic parameters at any given state based on the joint configurations. (Piltan et al. 2012).

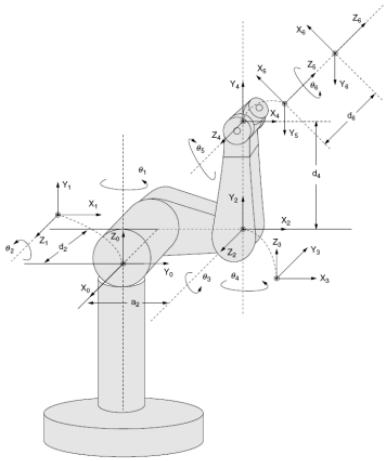


Figure 2: Puma 560 Home Configuration

i	α_{i-1}	θ_i	a_{i-1}	d_i
1	0	q1	0	0
2	-90	q2	0	0.2435
3	0	q3	0.4318	-0.0934
4	90	q4	-0.0203	0.4331
5	-90	q5	0	0
6	90	q6	0	0

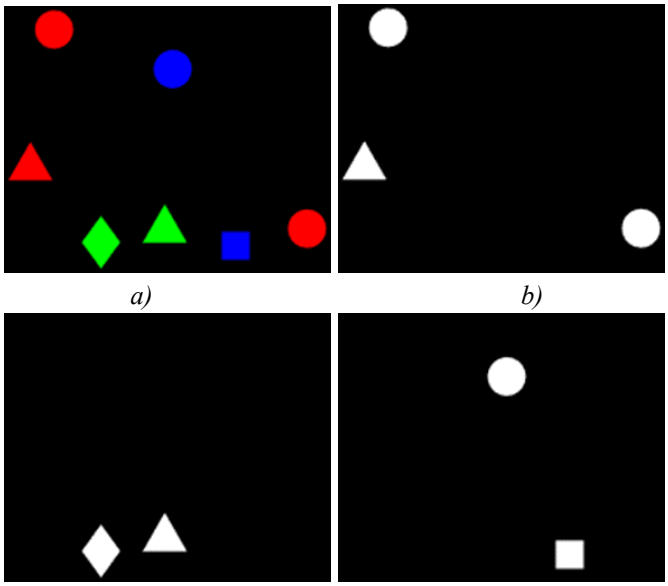
Table 1: DH Parameters

Object Detection and Sorting

The field of computer vision has undergone rapid advancements due to Machine Learning techniques like YOLOv5. However, they are computationally expensive, and hence cannot be done online with limited processing power. A conscious decision was made to use digital image processing algorithms to track where an object is located on a simulated conveyer belt. The task at hand required the identification of blue round objects.

An actual conveyor belt has objects of various shapes, sizes, and colors. To simulate a conveyor belt, objects were generated with various parameters. The rand() function was chosen to generate the y coordinate of the object on the y-axis, with a 20% probability of generation during each program loop. The percentage could be adjusted to increase or decrease the number of shapes generated. The objects had their attributes selected through a random function. Their position was updated based on the number of timesteps elapsed and the preset speed of the conveyor. A frame of the conveyor was used for the detection algorithm, which would be a continuous stream of sampled images for a real-time

application. The image was subjected to RGB channel separation using the imread() and the layers of the matrix were extracted to get the red, green, and blue channels. The simulated conveyor had only 3 colors, so each resultant image was a binary image.



a) *b)* *c)* *d)*

Figure 3: Single frame and RGB channel separation a) Conveyor snapshot b) Red channel C) Green channel d) Blue channel

Shape detection was performed using 2 algorithms, which both differed in the runtime, but gave the location and the radius of the objects. The blue channel image was used in the imfind() function, which identified one circle with its location (x,y) and radius. Similarly, the blob detection algorithm regionprops() identified every shape (blob) and its properties like area, perimeter, circularity, and centroid. If the object is a circle, the circularity will be nearly 1. The difference in location of the object in 2 subsequent frames can be used to estimate the speed of the conveyor and predict its position where it will be intercepted by the robot.

Controllers

Our team used a comparative study method to decide on the type of controller for the defined high-speed task. The following sections cover the implementation and discussions for the three controllers - Computer Torque, PD Control with Gravity Compensation, and Inverse Dynamics Control.

Computed Torque Control

Computed Torque is a decentralized controller where we attempt to reject the effects caused by dynamic couplings of the links as disturbances to the system; which is not suitable if the robot has a high level of dynamic coupling. The controller tries to anticipate this in two ways: one, by applying a feedforward input based on the required velocity and acceleration being anticipated by the trajectory. Secondly, it does so by applying a centralized feedforward input in addition to the decentralized feedforward input. The centralized feedforward input tries to account for the dynamic couplings of the links. This strategy can work when the magnitude of coupling and the speeds at which the robot is working are low. This is the only decentralized controller used in this project. The issue with using a decentralized controller in high-speed cycles where the links are dynamically coupled is that the feedforward input being provided is not enough to compensate for the error generated by the dynamic coupling a major reason for that can be attributed to the open-loop nature of the feedforward signal being provided which only takes in account of the desired trajectory but not the actual trajectory in which the robot is presently situated thus not having a closed-loop response can have a detrimental effect to the accuracy of the controller.

The controller model consists of three main sub-systems - Decentralized Controller, Decentralized Feedforward Action, and Centralized Feedforward Action.

The decentralized Controller subsystem and the decentralized feedforward subsystem can be demonstrated by the schematic:

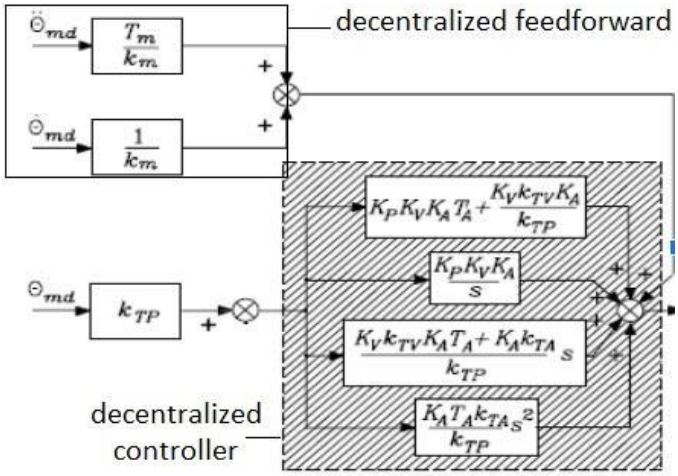


Figure 4: Decentralized Control and Decentralized Feedforward Subsystems of Computed Feedforward Control (Siciliano 2009)

The decentralized feedforward subsystem can be obtained by joint velocities multiplied by $1/k_m$ and joint acceleration multiplied by T_m/k_m. Both terms are with respect to the joint velocities and joint acceleration, derived from the desired trajectory.

The centralized part of the controller consists of $R_a K_t^{-1} d_d$ where,

$$d_d = K^{-1}_r \Delta B(q_d) K^{-1}_r \ddot{q}_{md} + K^{-1}r\, C(q_d, \dot{q}_d) K^{-1}_r \dot{q}_{md} + K^{-1}_r\, g(q_d) \quad (1)$$

K^{-1}_{r} is the (n×n) diagonal matrix whose elements represent the gear ratio of the motors, ΔB is the inertia matrix consisting of only the configuration-dependent terms of the inertia matrix, q_{md} is the actual actuator trajectory, $C(q_d, \dot{q}_d)$ is the matrix consisting of Coriolis and Centrifugal terms of the robot with respect to desired joint angles and velocities.

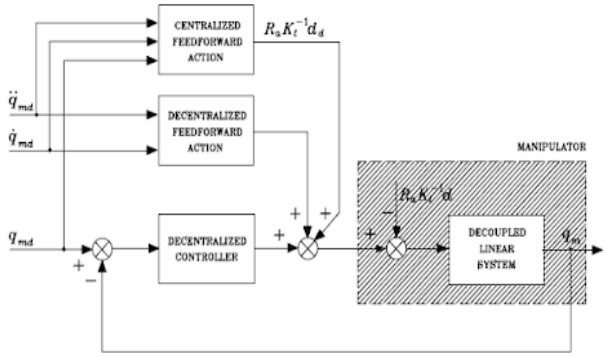


Figure 5: Computed Torque Feedforward Control Diagram

If the centralized part is an open-loop feedforward response, the controller lacks the provision to perform in highly coupled settings. The errors were too high irrespective of the value range of K_D and K_P . After starting with an initial combination of $K_P = 200 * I_{6X6}$ and $K_D = 50 * I_{6X6}$ a series of K_D and K_P values were tested but controller tuning failed to compensate for the high level of couplings.

Testing concluded that the errors noticed at the start- $K_P = 70 * I_{6X6}$; $K_D = 25 * I_{6X6}$, and that at the end- $K_P = 1500 * I_{6X6}$; $K_D = 450 * I_{6X6}$ were very high. The errors reduced for $K_P = 280 * I_{6X6}$ and $K_D = 70 * I_{6X6}$ but were still too high for the required task. The controller had a satisfactory performance as demonstrated by Figure 7.

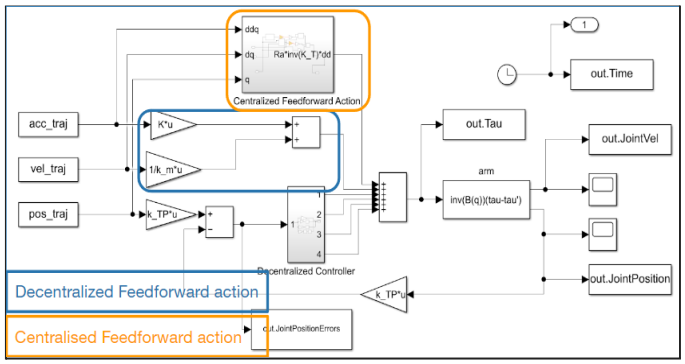


Figure 6: Computed Torque FeedForward Control Simulink Diagram

Controller	K_P	K_D	Comment
Computed Torque Feedforward Control	70	25	High Error, failed to converge
	200	50	High Error plot
	280	70	High Error plot
	1500	450	High Error, failed to converge

Table 2: Controller Gains for Computed Torque Control

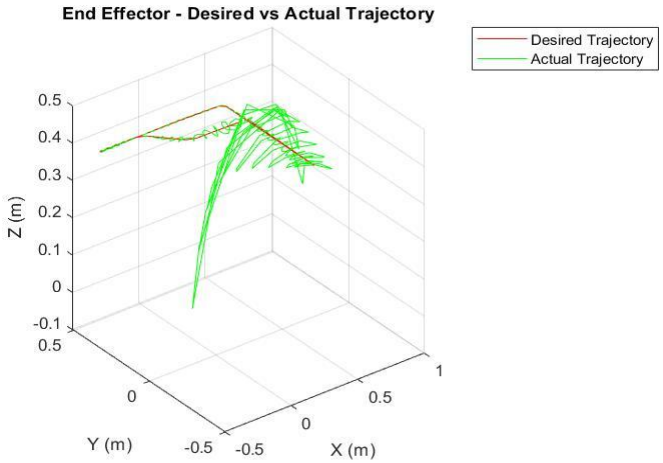


Figure 7: End-Effector Desired vs Actual Trajectory

PD Control with Gravity Compensation

The decentralized controller with computed feedforward was unable to follow the trajectory and led to unstable end effector oscillations. The controller simplifies it into N single-in/single-out (SISO) system, based on the assumptions of highly geared systems with low operational speeds. However, in the application mentioned above, the cycle time is quite low which creates dynamic trajectories, which are tracked very poorly due to the interaction and coupling effects (Siciliano 2009, 327).

To mitigate these effects, we considered the PD Controller with Gravity Compensation, which accounted for the manipulator dynamics and gravity effects. This controller aims to reach the desired posture by exactly compensating for the gravity effects at that point. The governing equation is: (Siciliano et al., 2009, 328)

$$u = g(q) + K_P * \tilde{q} - K_D * \dot{q}, \text{ and } K_D > 0 \qquad (2)$$

The controller calculates the torque to be applied to each joint so that the gravitational torque is balanced at every computation, and the proportional term reduces error in the actual joint angles.

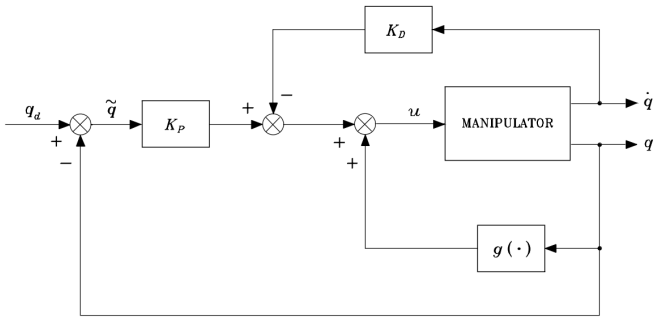


Figure 8: PD Control with Gravity Compensation Simulink Diagram

When $\dot{q}$~0 near the desired position, the proportional and derivative gain reduces to zero, and the robot maintains its position. We can also conclude that there will always be a non-zero tracking error when the $\dot{q}$ is non-zero, which is the case for our trajectory.

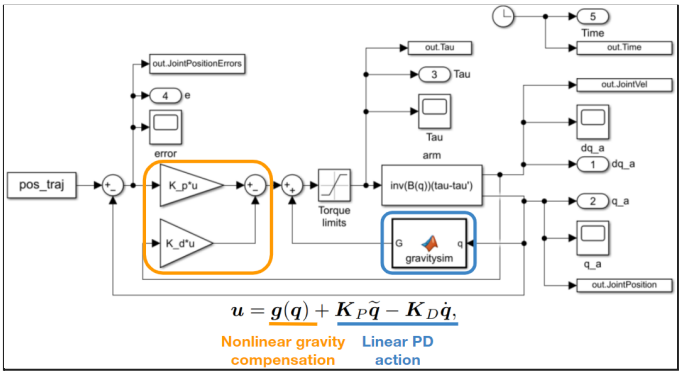


Figure 9: PD Control with Gravity Compensation Simulink Diagram

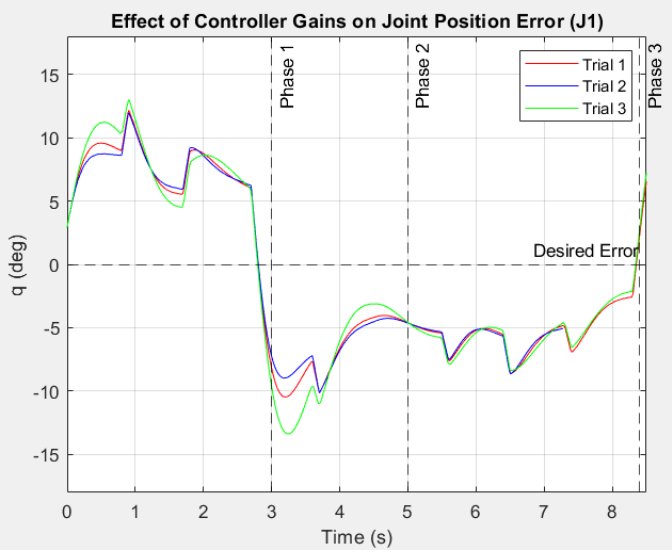


Figure 10: PD Control with Gravity Compensation Simulink Diagram

The equations (inertia, Coriolis, and gravity) in the reference literature were converted into MATLAB functions, which were called to simulate the SIMULINK block diagrams.

The K_P and K_D gains were iterated for a step input on joint 1 until the L2 norm error was minimized. The K_P value was increased until the tracking error was minimized, but the absence of K_D caused oscillations. So, keeping K_P constant, K_D was increased until the response was almost critically damped. This was achieved for $K_P = 150 * I_{6X6}$ and $K_D = 30 * I_{6X6}$.

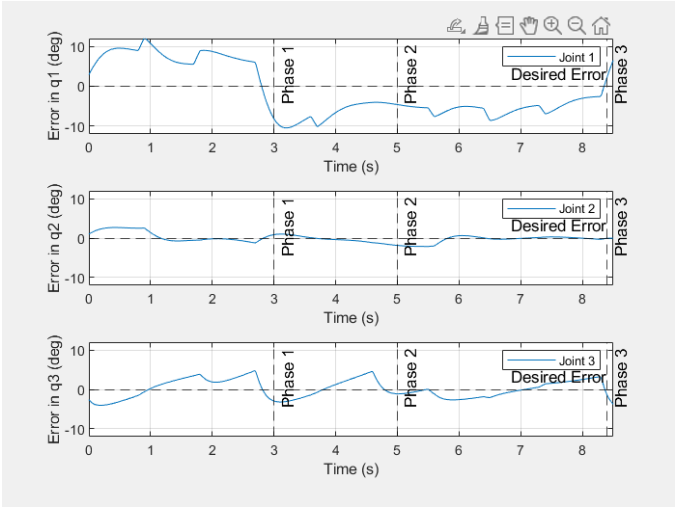


Figure 11: Final Design - Error Time History Plots for Joints 1, 2, and 3

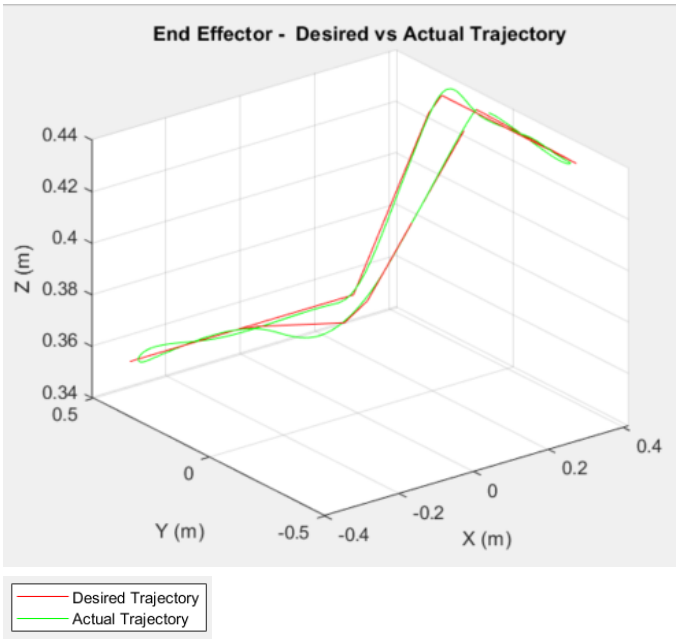


Figure 12: End-Effector Desired vs Actual Trajectory

Inverse Dynamics Control

The goal of this controller is to perform an exact linearization of system dynamics using non-linear state feedback. The controller architecture comprises an outer feedback loop that stabilizes the overall system and an inner feedback loop to get a linear and decoupled input/output relationship based on a double integrator relationship (Siciliano et al., 2009, 330).

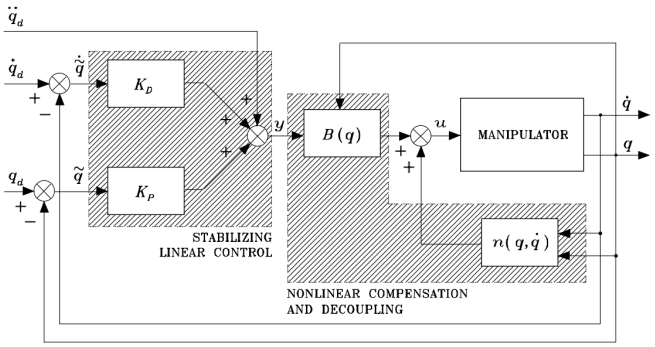


Figure 13: Inverse Dynamics Control Block Diagram

The governing equation for the inner loop is given by -

$$u = B(q)y + n(q, \dot{q}) \quad (3)$$

where, $n(q, \dot{q}) = C(q, \dot{q})\dot{q} + F\dot{q} + g(q)$ (4)

The stabilizing control law, y is given by -

$$y = \ddot{q}_d + K_D \dot{q}_d + K_P q_d \quad (5)$$

From an implementation standpoint, this makes it a little challenging as it largely depends on the accuracy of the parameters and equations of the system dynamic model. This is because this method assumes perfect linearization of nonlinear dynamic terms. Furthermore, as the technique is based on the nonlinear feedback of the current state of the system, computation needs to be performed online, which makes it hardware and computationally intensive (Siciliano 2009). Figure 14 below is a high-level snapshot of the Simulink block diagram. A scaled picture for the same can be found in the Appendix.

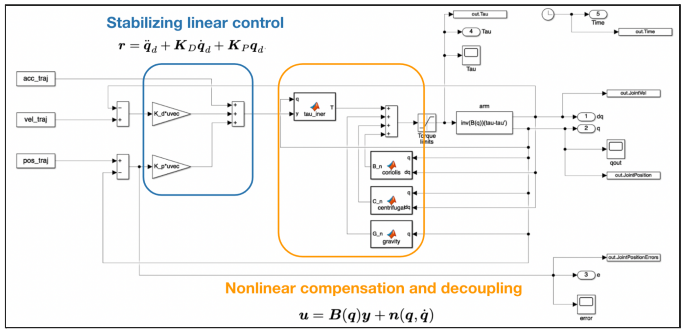


Figure 14: Inverse Dynamics Control Simulink Diagram

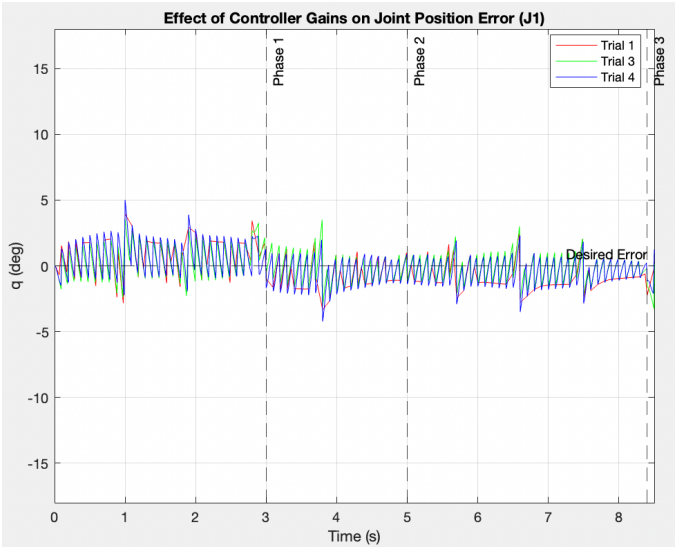
Based on the available data sheets and literature, MATLAB scripts were coded to compute the Inertia matrix, non-linearities including Coriolis, Centrifugal, and Gravity compensation matrices. The outputs of these codes were then converted into individual functions to be used by the controller as custom MATLAB function blocks (MATLAB).

Table 3 below presents some of the versions of controller gains iterated. The controller was incrementally tuned. The proportional gain was gradually increased till the error plot showed an oscillatory response at about $K_P = 100 * I_{6X6}$. The system was then tested for higher values in conjunction with derivative gain, but due to extremely high computational times, higher errors in joint space, and no significant improvements in the Cartesian space, the design was set at $K_P = 200 * I_{6X6}$.

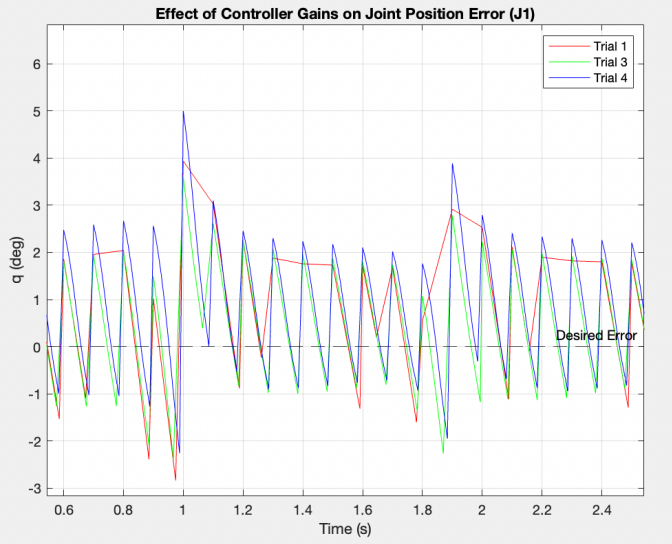
Controller	K_P	K_D	Max Error	Comment
Inverse Dynamics Control	100	15	~3.9° (J1)	High oscillation period, high error
	150	25	~3.7° (J1)	High error
	200	40	~3.5° (J1)	Final Design
	1200	100	~5° (J1)	High computation time, high error

Table 3: Controller Gains for Inverse Dynamics Control

The derivative gain was then tuned to reduce the overshoot and improve system stability. The values were incrementally changed but no significant change was observed in the damping effect. The value was finally set $K_D = 40 * I_{6X6}$ to maintain low noise sensitivity. The plots presented in Figure 15 show the effect of different controller gains on position errors in joint space. The graphs have only been plotted for Joint 1 and Trials 1, 3, and 4 from Table 3 for the sake of clarity.



a)



b)

Figure 15: Performance Analysis of Controller Gains - a) Time History Assessment; b) Zoomed in view from 0.6s to 2.6s

The final gains were chosen with the goal to reduce transient and steady-state errors, while at the same time maintaining minimal computational time and noise sensitivity.

The error time history plots for the final design are presented in Figure 16. A representative plot showing the comparison of desired and actual trajectories in Cartesian space is shown in Figure 17.

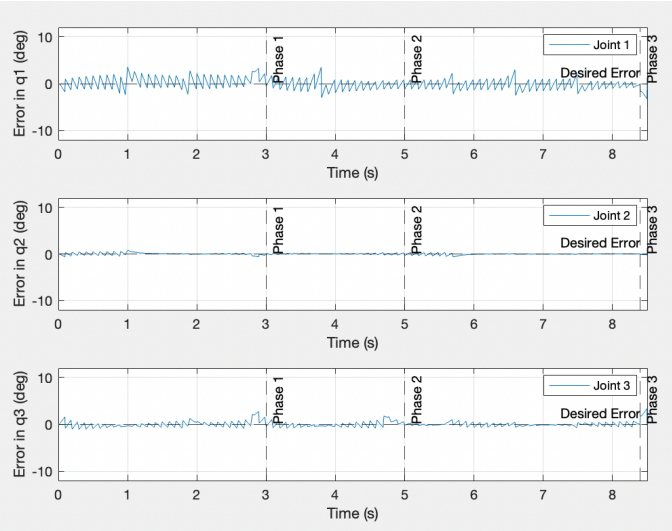


Figure 16: Final Design - Error Time History Plots for Joints 1, 2, and 3

V. RESULTS

Figure 18 is a series of snapshots of the robot's response to the defined trajectory tracking task with Inverse Dynamics Control. The snapshots were taken at the end of each phase. As evident from the snapshots, the manipulator accurately intercepts the object, tracks the trajectory while grasping the object (grasping action not implemented), and places it on the designated place station.

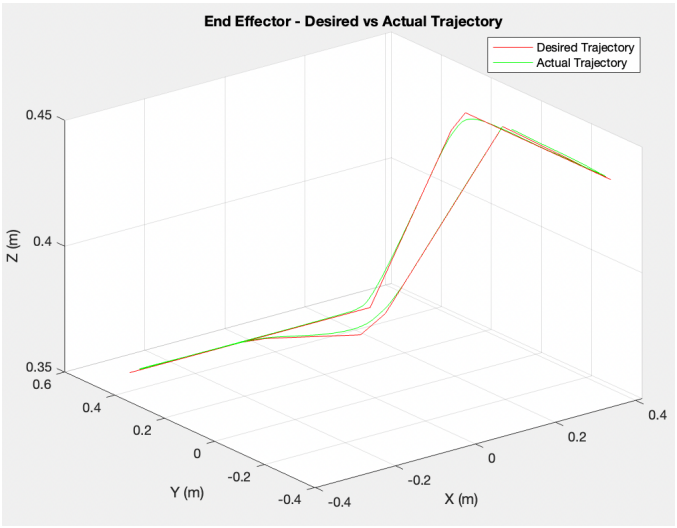
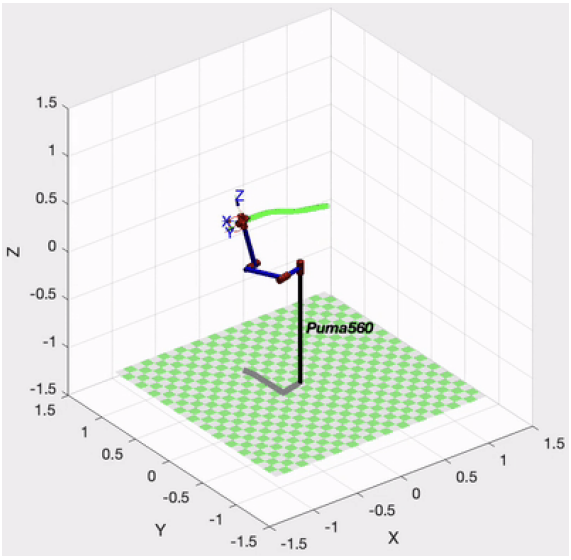


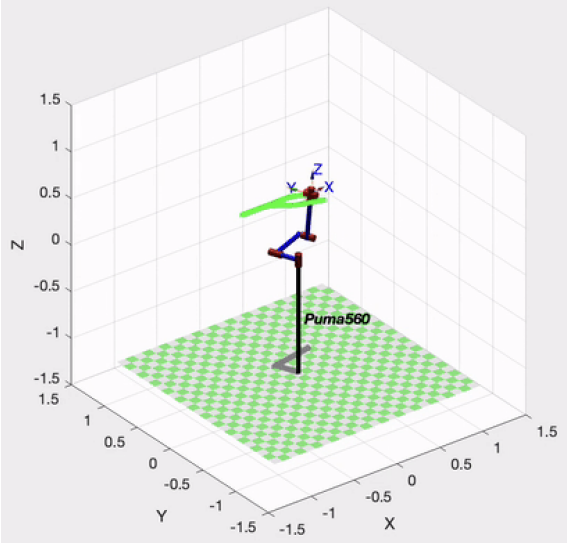
Figure 17: End-Effector Desired vs Actual Trajectory

Based on the error assessment presented in previous sections, it can be concluded that Computed Torque Control is not a suitable control technique for the proposed application. Figure 19 and Figure 20 represent and compare the trajectory tracking performance in X, Y, and Z directions for PD Control with Gravity Compensation and Inverse Dynamic Control strategies.

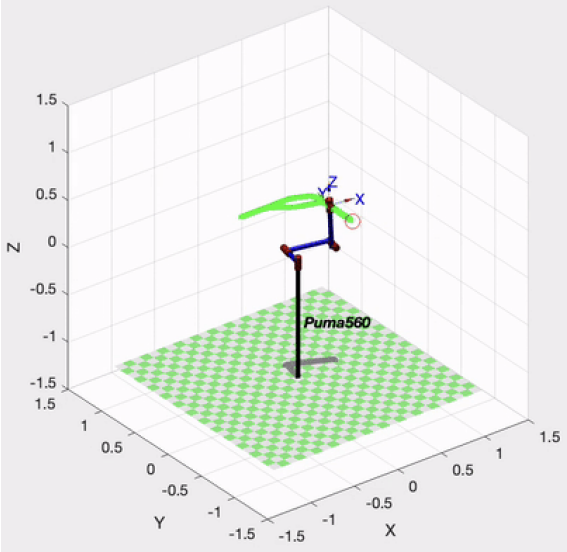
The plots clearly show that the model with Inverse Dynamic Control very closely follows the desired trajectory with minimal error. However, based on the required accuracy levels, one might consider the former if desired speeds are relatively low and there are computational constraints.



a)



b)



c)

Figure 18: Snapshots of the Robot's Response to the defined Trajectory Tracking task - a) End of Phase 1; b) End of Phase 2; c) End of Phase 3.

Figure 21 and Figure 22 also compare the Euclidean errors in the End-Effector workspace to present the data from an application standpoint. PD Control with Gravity Compensation yields a maximum error of ~ 0.1 m, while Inverse Dynamics Control yields a maximum error of ~ 0.02 m, thus, highlighting its superior performance for these applications.

Error Comparison

Trajectory Comparison in X, Y, Z

Euclidean Error

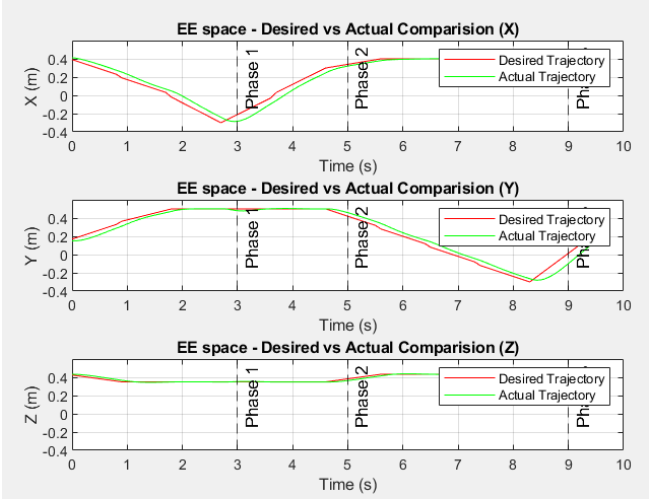


Figure 19 - PDGC - Trajectory Comparison in X, Y, and Z directions

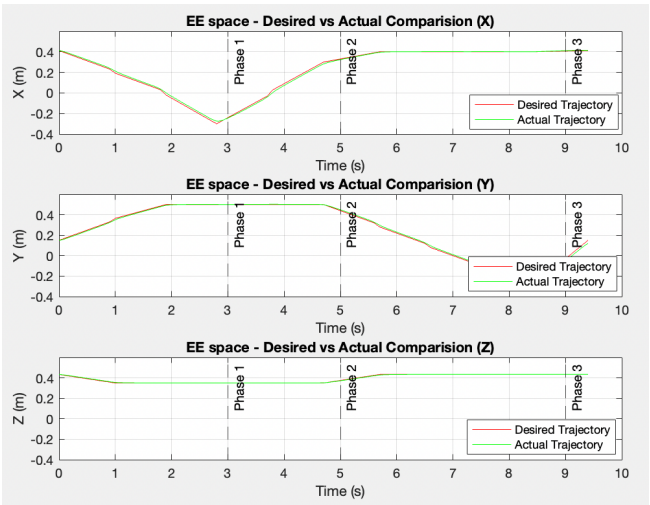


Figure 20 - Inverse Dynamics Control - Trajectory Comparison in X, Y, and Z directions

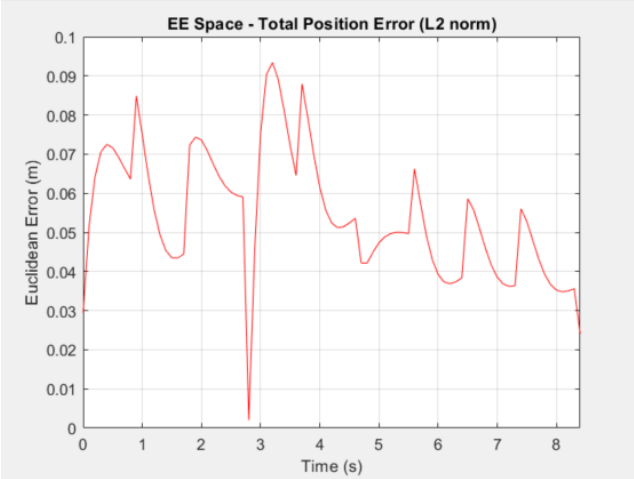


Figure 21 - PD Control with Gravity Compensation - Euclidean Error

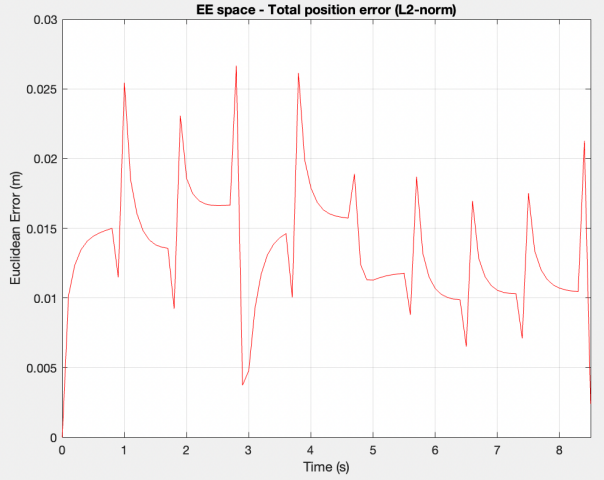


Figure 22 - Inverse Dynamics Control - Euclidean Error

VI. DISCUSSION

Anderson et.al has already demonstrated that PID control was not able to deliver satisfactory performance in dynamically coupled conditions of PUMA 560 links (Anderson 1988). With recent research demonstrating that computed torque control could deliver satisfactory performance in a controlled setting three controllers were compared in a high dynamic coupling, high-speed cycle scenario to ascertain the difference between centralized controllers and decentralized controllers. The Computed Torque Feedforward Controller failed to converge to solutions for the defined task. The reason can be attributed to the controller's inability to take into account geometric nonlinearities and dynamic couplings in the closed-loop response. This highlights that there are limitations to which a feedforward compensation can handle disturbances introduced within the system due to dynamic couplings. Both centralized controllers: PD control with Gravity Compensation and Inverse Dynamics Control were able to track the trajectory. Inverse Dynamic Control performed the best among the two controllers yielding an error five times less in magnitude than PD control with Gravity Compensation.

VII. FUTURE DIRECTIONS

The controller was capable and satisfied all the KPIs defined for the task. The error could be further reduced by increasing the cycle time. The parameters defined in this paper have uncertainties, which could be updated online using adaptive control.

The DIP algorithm could be improved to enhance robustness to identify any objects with Machine Learning frameworks like YOLOv5 and Tensorflow.

VIII. ACKNOWLEDGEMENT

This project was the result of mutual collaboration and the effort of all team members. The team would like to thank the course professor, Dr. Veronica Santos, and teaching assistant, Jonathan Bopp, for their guidance on this project throughout the quarter.

REFERENCES

[1] Armstrong, Brian, Oussama Khatib, and Joel Burdick. 10.1109/ROBOT.1986.1087644. "The Explicit Dynamic Model and Inertial Parameters of the PUMA 566 Arm." *Proceedings. 1986 IEEE International Conference on Robotics and Automation, 1986*, 510-518.

[2] "Ch. 3 - Basic Pick and Place." n.d. Robotic Manipulation. Accessed June 10, 2022. https://manipulation.csail.mit.edu/pick.html.

[3] "Cobot cycle times — DoF." 2019. DoF. https://dof.robotiq.com/discussion/1677/cobot-cycle-times.

[4] "Conveyor Speed Calculator & FPM Formula Guide." n.d. Cisco-Eagle. Accessed June 10, 2022. https://www.cisco-eagle.com/category/3363/calculating-conveyor-speed.

[5] Corke, Peter. 2017. *Robotics, Vision and Control: Fundamental Algorithms In MATLAB® Second, Completely Revised, Extended And Updated Edition*. N.p.: Springer International Publishing.

[6] Corke, P. I., and B. Armstrong-Helouvry. 1994. "A search for consensus among model parameters reported for the PUMA 560 robot." *Proceedings of the 1994 IEEE International Conference on Robotics and Automation* 2:1608-1613. 10.1109/ROBOT.1994.351360.

[7] J. Falco et al. 2020. "Benchmarking Protocols for Evaluating Grasp Strength, Grasp Cycle Time, Finger Strength, and Finger Repeatability of Robot End-Effectors,." *IEEE Robotics and Automation Letters* 5, no. 2 (April): 644-651. 10.1109/LRA.2020.2964164.

[8] MATLAB. n.d. *MATLAB*. R2020a ed. Natick, Massachusetts: The MathWorks Inc.

[9] "Piece Picking Robots Market | 2022 - 27 | Industry Share, Size, Growth." n.d. Mordor Intelligence. Accessed June 10, 2022. https://www.mordorintelligence.com/industry-reports/piece-picking-robots-market.

[10] Piltan, Farzin, Sara Emamzadeh, Zahra Hivand, and Forouzan Shahriyari. 2012. "PUMA-560 Robot Manipulator Position Sliding Mode Control Methods Using MATLAB/SIMULINK." *International Journal of Robotic and Automation* 6 (3).

[11] Siciliano, Bruno. 2009. *Robotics: Modelling, Planning and Control*. Edited by Luigi Villani, Lorenzo Sciavicco, and Giuseppe Oriolo. N.p.: Springer.

APPENDIX

A.1 Motor and Drive Parameters

Parameter	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6
Gear Ratio	62.61	107.36	53.09	76.01	71.91	76.73
Maximum Torque (N-m)	97.6	186.4	89.4	24.2	20.1	21.3
Break Away Torque (N-m)	6.3	5.5	2.6	1.3	1.0	1.2

Table 4: Motor and Drive Parameters

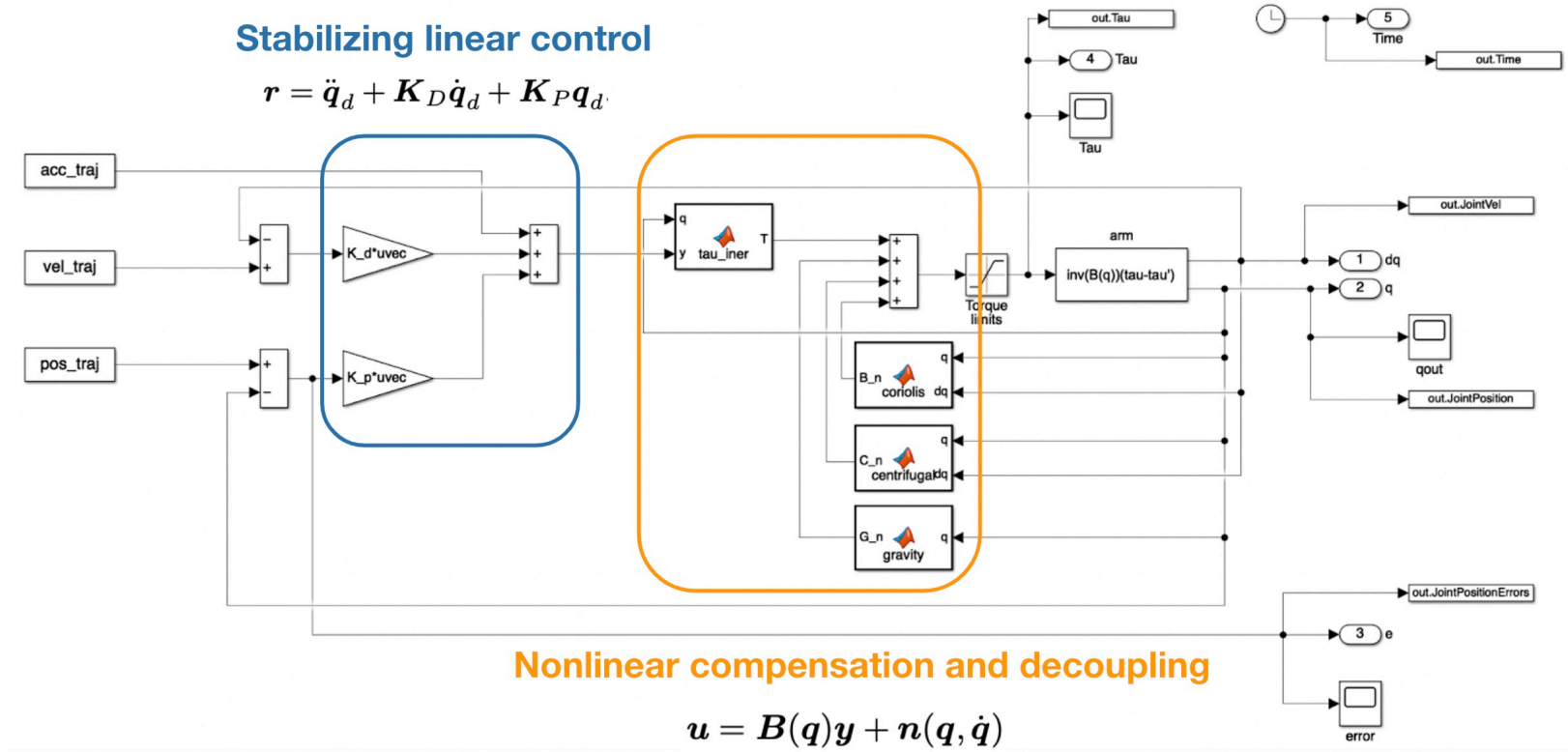
A.2 Team Members and Contributions

Team Members	**Contributions**
Kalra, Nitesh	Trajectory planning and generation, Custom Manipulator Dynamics and non-linear disturbance blocks, Inverse Dynamics Control Design and Simulation, Testing, Gain tuning and parametric study, error analysis, Write-up (Presentation and Report)
Sanghai, Nikunj	Custom Computed Torque feedforward blocks, Computed Torque Controller design, Testing, and parametric study of PUMA 560. Literature Review, Discussion and Presentation and Report Writing
Wani, Shubham Kiran	Object Detection and Tracking using Segmentation, PD controller with gravity compensation simulation, Parametric study and gain tuning, error analysis, Video generation, Presentation and Report Writing

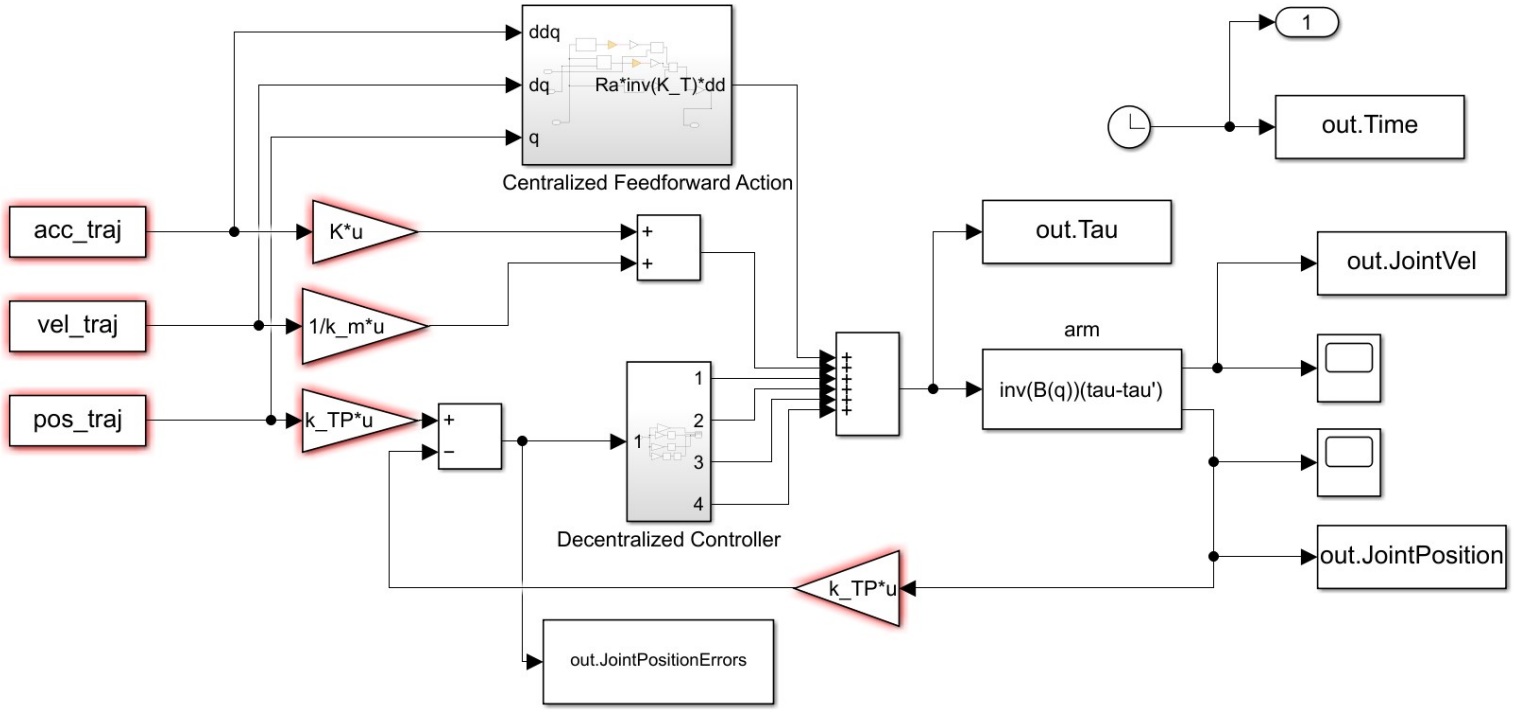
Table 5: Team Contributions

A.3 Simulink Diagrams and MATLAB Scripts

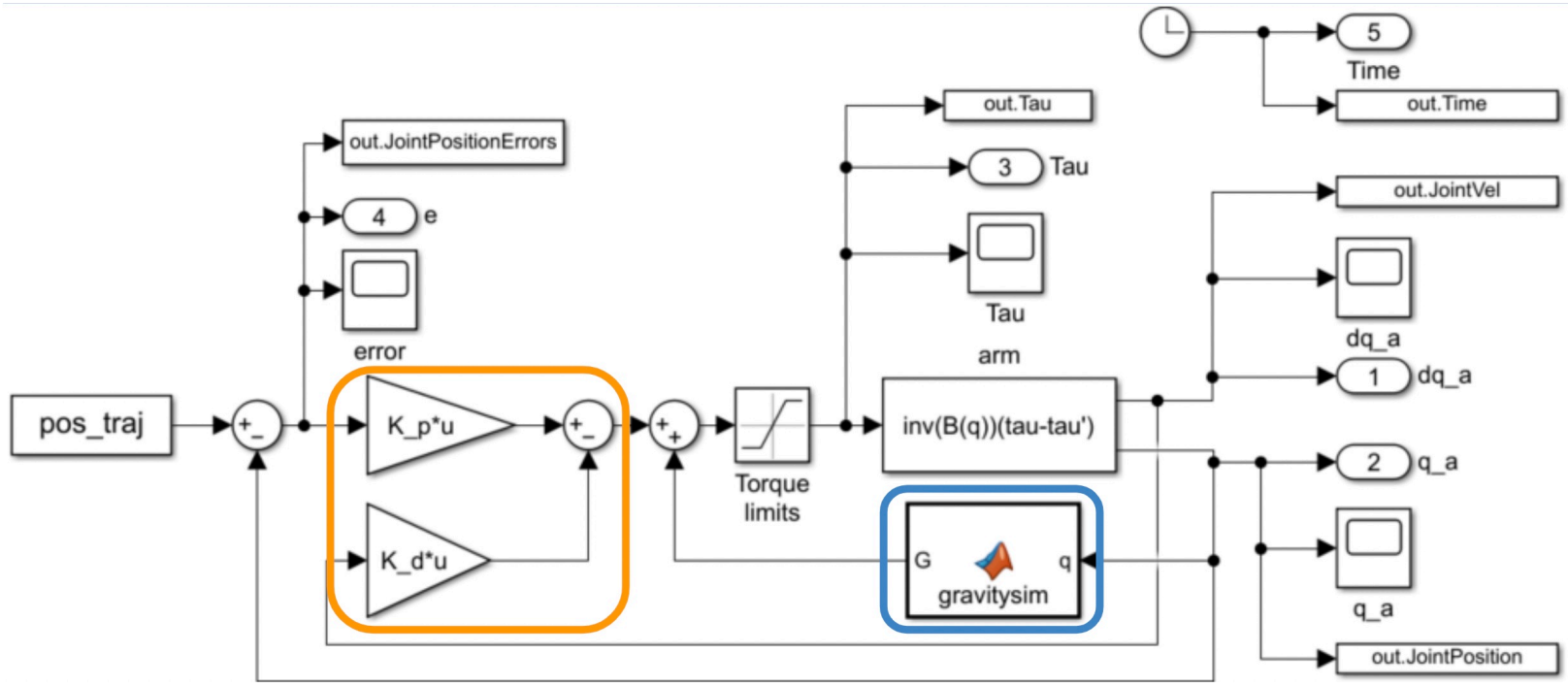
Inverse Dynamics Control



Computed Torque Control



PD Control with Gravity Compensation



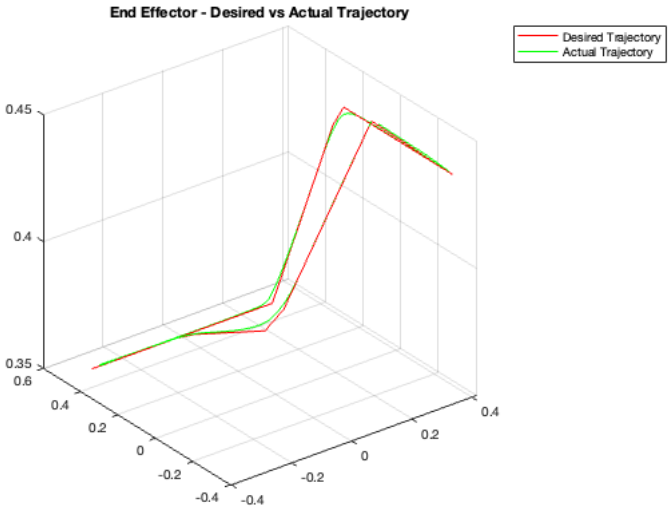
MATLAB Scripts

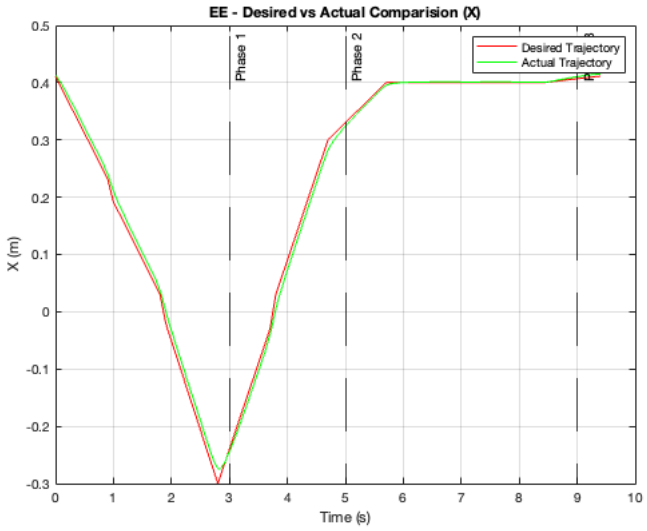
```
clc; clear all;

% Global varibales
global a2 a3 a6 d2 d3 d4 I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11 I12 I13 I14 I15 I16 I17 I18 I19 I20 I21 I22 I23 Im g1 g2 g3 g4 g5

% DH params for robots
a2 = 0.4318; a3 = -0.0203; a6 = 0.1;
d2 = 0.2435; d3 = -0.0934; d4 = 0.4331;

% Torque limits
max_torques = 0.75*[97.6 186.4 89.4 24.2 20.1 21.3]';
min_torques = -0.75*[97.6 186.4 89.4 24.2 20.1 21.3]';

% Friction Matrix
F_v = zeros(6,6);
% F_v = diag([0.05 0.05 0.05 0.05 0.05 0.05]);

% Inertia constants in kg.m^2
I1 = 1.43;
I2 = 1.75;
I3 = 1.38;
I4 = 0.69;
I5 = 0.372;
I6 = 0.333;
I7 = 0.298;
I8 = -0.134;
I9 = 0.0238;
I10 = -0.0213;
I11 = -0.0142;
I12 = -0.011;
I13 = -0.00379;
I14 = 0.00164;
I15 = 0.00125;
I16 = 0.00124;
I17 = 0.000642;
I18 = 0.000431;
I19 = 0.0003;
I20 = -0.000202;
I21 = -0.0001;
I22 = -0.000058;
I23 = 0.00004;

Im = [ 1.14 4.71 0.827 0.2 0.179 0.193 ];

% Gravitational constants (N.m)
g1 = -37.2;
g2 = -8.44;
g3 = 1.02;
g4 = 0.249;
g5 = -0.0282;

% Gain Matrices
K_p = diag([200 200 200 200 200 200]);
K_d = diag([40 40 40 40 40 40]);

% Trajectory inputs
dt = 0.1; % time step
x_0 = 0.4115; y_0 = 0.1501; z_0 = 0.4331;

x0=[x_0,y_0,z_0]; % home position
T0 = [1 0 0 x_0;
    0 1 0 y_0;
    0 0 1 z_0;
    0 0 0 1];
q0 = (ik(T0))';

% Conveyor - Grasp start
x_s = -0.3; y_s = 0.5; z_s = 0.35;
xs=[x_s,y_s,z_s];

% Phase 1 - Home to conveyor trajectory
tf_i = 3; % total time
wp_i=[x_0-0.2,y_0+0.2,z_s;
    x_0-0.4,y_s,z_s;
    x_s,y_s,z_s];

tseg_i = [1,1,1]; % time per segment

[t_i,xi_d,dxi_d,ddxi_d] = traj(x0,wp_i,dt,tf_i,tseg_i);

t1 = t_i; % time stamps

% Conveyor - Grasp end
x_e = 0.3; y_e = 0.5; z_e = 0.35;
xe=[x_e,y_e,z_e];

% Phase 2 - Conveyor pickup trajectory
tf_c = 2; % total time
```

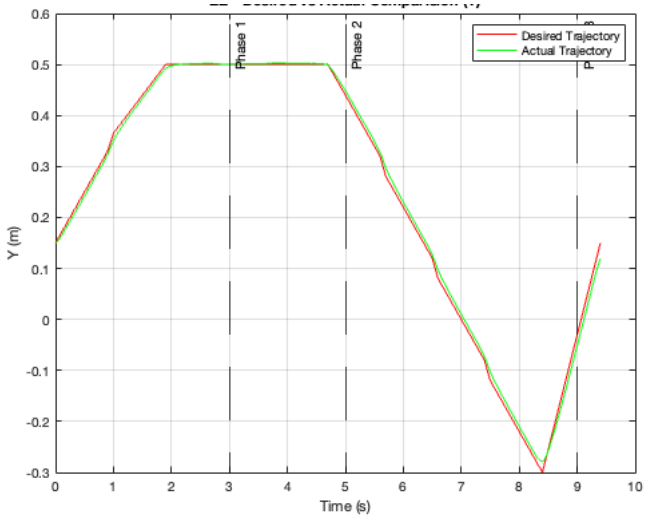
```
    x_e,y_e,z_e];
tseg_c = [1,1]; % time per segment

[t_c,xc_d,dxc_d,ddxc_d] = traj(xs,wp_c,dt,tf_c,tseg_c);

t2 = tf_i + t_c; % time stamps

% Place point
x_f = 0.4; y_f = -0.3; z_f = z_0;
xf=[x_f,y_f,z_f];

% Phase 3 - Conveyor to station placing trajectory
tf_p = 4; % total time
wp_p=[x_f,y_e-0.2,z_f;
    x_f,y_e-0.4,z_f;
    x_f,y_e-0.6,z_f;
    x_f,y_f,z_f];

tseg_p = [1,1,1,1]; % time per segment

[t_p,xp_d,dxp_d,ddxp_d] = traj(xe,wp_p,dt,tf_p,tseg_p);

t3 = tf_i + tf_c + t_p; % time stamps

% Phase 4 - Place station to home trajectory
tf_f = 1; % total time
wp_f=[x_0,y_0,z_0];
tseg_f = [1]; % time per segment

[t_f,xf_d,dxf_d,ddxf_d] = traj(xf,wp_f,dt,tf_f,tseg_f);

t4 = tf_i + tf_c + tf_p + t_f; % time stamps

% Combined trajectory data
t_temp = [t1, t2, t3, t4];

% Desired trajectory data points
x_d = [x0; xi_d; xc_d; xp_d; xf_d];

t = t_temp(1:95);

tf = (tf_i + tf_c + tf_p +tf_f) - 0.6; %simulating till 9.4 s

% Desired Joint Configurations
for i = 1:length(t)
    T = [1 0 0 x_d(i,1);
        0 1 0 x_d(i,2);
        0 0 1 x_d(i,3);
        0 0 0 1];
    q_d(i,:) = ik(T);
    %     J = p560.jacob0(q_d(i,:));
    %     v = (q_d(i,:))';
    %     test = (inv(J)*(v))';
end

for i = 1:length(t)-1
    dq_d(i,:) = (q_d(i+1,:)-q_d(i,:))/dt;
end

dq_d(length(t),:) = [0 0 0 0 0 0];

for i = 1:length(t)-1
    ddq_d(i,:) = (dq_d(i+1,:)-dq_d(i,:))/dt;
end

ddq_d(length(t),:) = [0 0 0 0 0 0];

% Simulink model inputs
pos_traj = [t', q_d];
vel_traj = [t', dq_d];
acc_traj = [t', ddq_d];

% Controller Simulation
out = sim("InverseDynamicsController",tf);

% Output Data
q_a = [out.JointPosition.Data(1,:); out.JointPosition.Data(2,:); out.JointPosition.Data(3,:); out.JointPosition.Data(4,:); out.JointPosition.Data(5,:)
Q_a = (q_a)'; % for trajectory with robot

% Actual trajectory data points
for i = 1:length(q_a)
    T_a = fk(q_a(:,i));
    [R_traj,P] = tr2rt(T_a);
    P_a(:,i) = P;
end

% Trajectory plots - 3D space
figure('Name','EE trajectories');
plot3(x_d(:,1),x_d(:,2),x_d(:,3),'Color',[1 0 0]); % desired trajectory
```

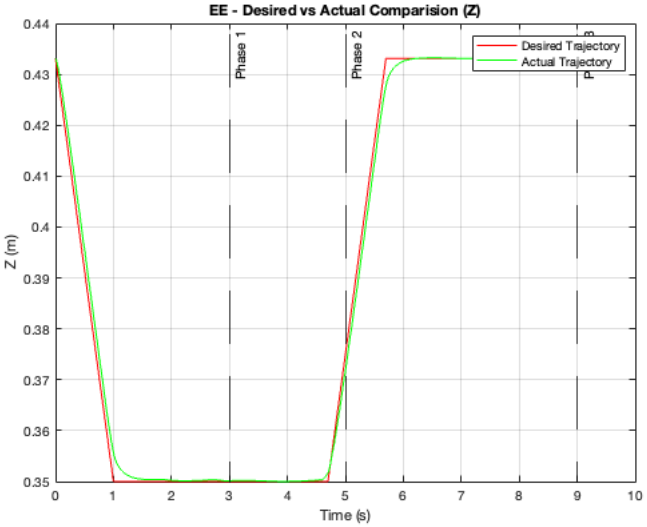
```
grid on;
plot3(P_a(1,:),P_a(2,:),P_a(3,:),'Color',[0 1 0]); % actual trajectory
title('End Effector - Desired vs Actual Trajectory');
legend('Desired Trajectory', 'Actual Trajectory');

% EE space Error Analysis
% X-Direction
figure('Name','EE - Desired vs Actual Comparision (X)');
plot(t(1,:),x_d(:,1),'Color',[1 0 0]);
hold on;
grid on;
plot(out.Time.Data(:,1),P_a(1,:),'Color',[0 1 0]);
xline(3,'--k','Phase 1');
xline(5,'--k','Phase 2');
xline(9,'--k','Phase 3');
title('EE - Desired vs Actual Comparision (X)');
legend('Desired Trajectory', 'Actual Trajectory');
xlabel('Time (s)');
ylabel('X (m)');

% Y-Direction
figure('Name','EE - Desired vs Actual Comparision (Y)');
plot(t(1,:),x_d(:,2),'Color',[1 0 0]);
hold on;
grid on;
plot(out.Time.Data(:,1),P_a(2,:),'Color',[0 1 0]);
xline(3,'--k','Phase 1');
xline(5,'--k','Phase 2');
xline(9,'--k','Phase 3');
title('EE - Desired vs Actual Comparision (Y)');
legend('Desired Trajectory', 'Actual Trajectory');
xlabel('Time (s)');
ylabel('Y (m)');

% Z-Direction
figure('Name','EE - Desired vs Actual Comparision (Z)');
plot(t(1,:),x_d(:,3),'Color',[1 0 0]);
hold on;
grid on;
plot(out.Time.Data(:,1),P_a(3,:),'Color',[0 1 0]);
xline(3,'--k','Phase 1');
xline(5,'--k','Phase 2');
xline(9,'--k','Phase 3');
title('EE - Desired vs Actual Comparision (Z)');
legend('Desired Trajectory', 'Actual Trajectory');
xlabel('Time (s)');
ylabel('Z (m)');
hold off;

z_block=0.35;y_block= 0.5; speed_conveyer = 0.3; dt = 0.1;
x_block_i= -0.3; x_block_f= 0.3;

% Trajectory with Robot
% Robot Definition
L1 = Link('revolute','d', 0, 'a', 0,'alpha', 0, 'modified', 'qlim',[-2*pi,2*pi]);
L2 = Link('revolute','d', d2, 'a', 0,'alpha', -pi/2, 'modified', 'qlim',[-2*pi,2*pi]);
L3 = Link('revolute','d', d3, 'a', a2,'alpha', 0, 'modified', 'qlim',[-2*pi,2*pi]);
L4 = Link('revolute','d', d4, 'a', a3,'alpha', pi/2, 'modified', 'qlim',[-2*pi,2*pi]);
L5 = Link('revolute','d', 0, 'a', 0,'alpha', -pi/2, 'modified', 'qlim',[-2*pi,2*pi]);
L6 = Link('revolute','d', 0, 'a', 0,'alpha', pi/2, 'modified', 'qlim',[-2*pi,2*pi]);

Puma560 = SerialLink([L1 L2 L3 L4 L5 L6],'name','Puma560');

% Initialize video
RobotTrajectory_Video = VideoWriter('RobotTrajectory_Video','MPEG-4');
RobotTrajectory_Video.FrameRate = 10;
open(RobotTrajectory_Video)

figure('Name','Trajectory with robot');
for i = 1 : length(Q_a)
    Puma560.plot(Q_a(i,:),'scale',0.5);
    xlim([-1.5,1.5]); ylim([-1.5,1.5]); zlim([-1.5,1.5]);
    T_temp = Puma560.fkine(Q_a(i,:));
    [R_traj, P_traj] = tr2rt(T_temp);
    hold on;
    plot3(P_traj(1),P_traj(2),P_traj(3),'g*','MarkerSize',2);
    x_block_pos = x_block_i + speed_conveyer*(out.tout(i)-3);
    if (out.tout(i)>5)
        if (out.tout(i)<8.5)
            o=plot3(P_traj(1),P_traj(2),P_traj(3),'ro');
        else
            o=plot3(0.4,-0.3,0.4331,'ro');
        end
    else
        o=plot3(x_block_pos,y_block,z_block,'ro');
    end
    o.MarkerSize=10;
    drawnow
    hold on;
```

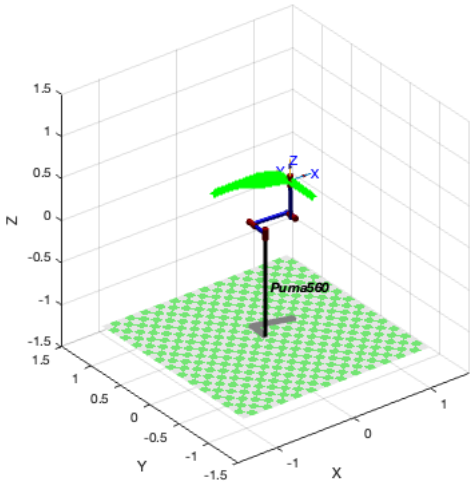
```
    grid on;
    frame = getframe(gcf); %get frame
    writeVideo(RobotTrajectory_Video, frame);
    set(o,'Visible','off')
    hold off;
end
close(RobotTrajectory_Video);
```





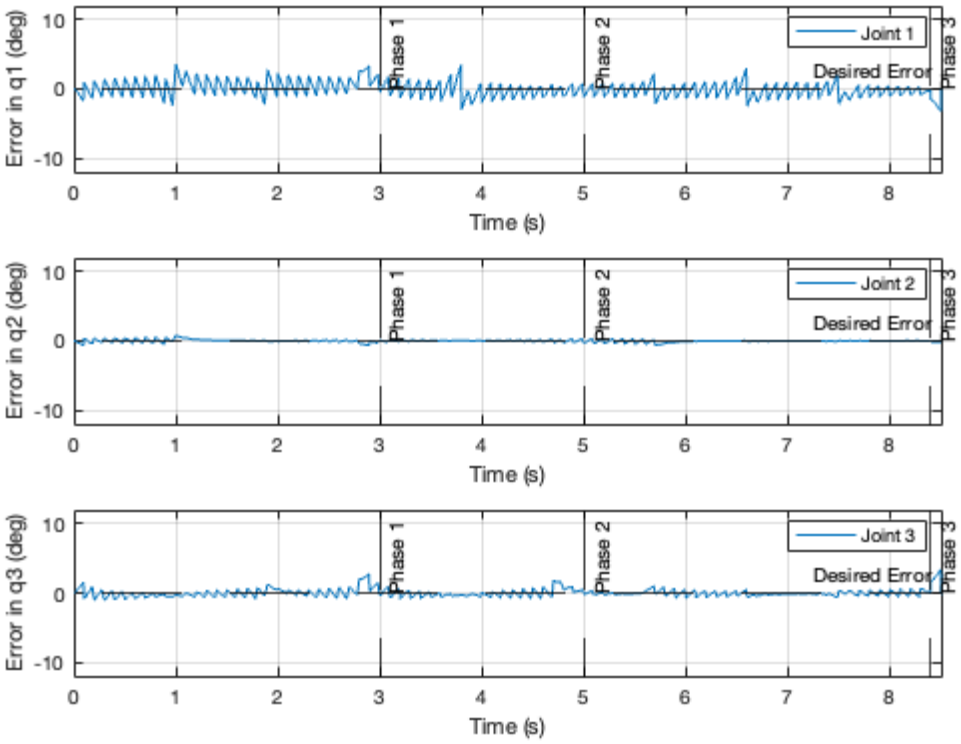






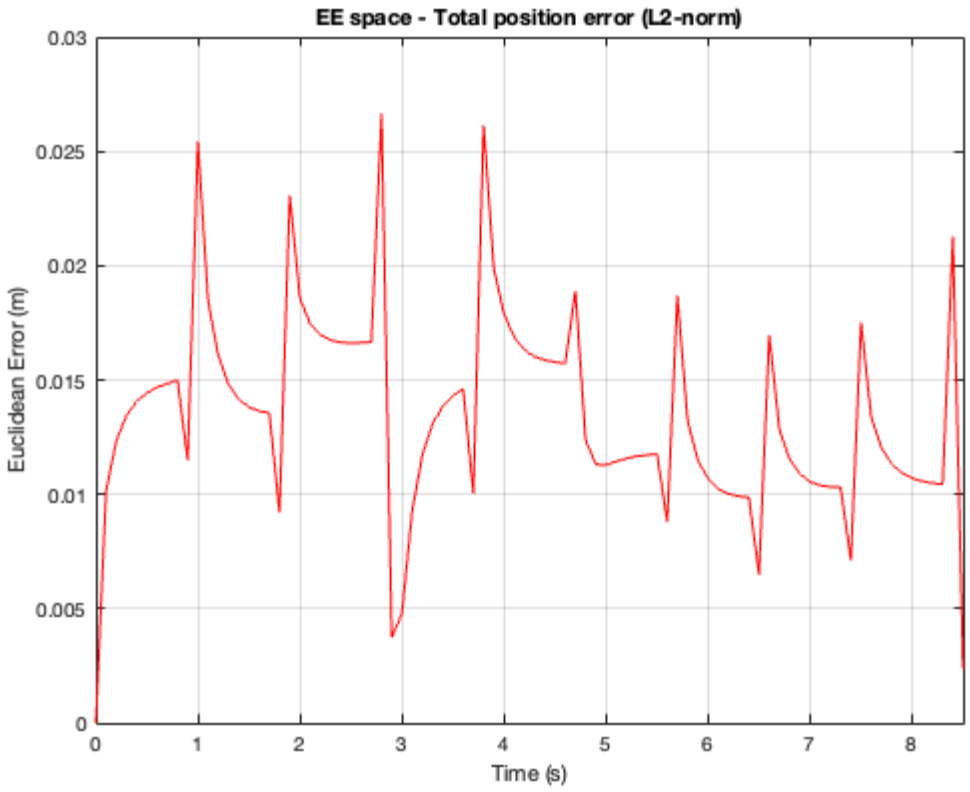
Published with MATLAB® R2022a

```
close all;
figure('Name','Joint Position Errors');
subplot(3,1,1)
plot(out.Time.Data(:,1),out.JointPositionErrors.Data(1,:)*(180/pi));
grid on;
yline(0,'--k','Desired Error');
hold on;
xlim([0,8.5]); % for param_10
xline(3,'--k','Phase 1');
xline(5,'--k','Phase 2');
xline(8.4,'--k','Phase 3');
ylim([-12,12]);
legend('Joint 1');
xlabel('Time (s)');
ylabel('Error in q1 (deg)');

subplot(3,1,2)
plot(out.Time.Data(:,1),out.JointPositionErrors.Data(2,:)*(180/pi));
grid on;
yline(0,'--k','Desired Error');
hold on;
xlim([0,8.5]); % for param_10
xline(3,'--k','Phase 1');
xline(5,'--k','Phase 2');
xline(8.4,'--k','Phase 3');
ylim([-12,12]);
legend('Joint 2');
xlabel('Time (s)');
ylabel('Error in q2 (deg)');

subplot(3,1,3)
plot(out.Time.Data(:,1),out.JointPositionErrors.Data(3,:)*(180/pi));
grid on;
yline(0,'--k','Desired Error');
hold on;
xlim([0,8.5]); % for param_10
xline(3,'--k','Phase 1');
xline(5,'--k','Phase 2');
xline(8.4,'--k','Phase 3');
ylim([-12,12]);
legend('Joint 3');
xlabel('Time (s)');
ylabel('Error in q3 (deg)');
```



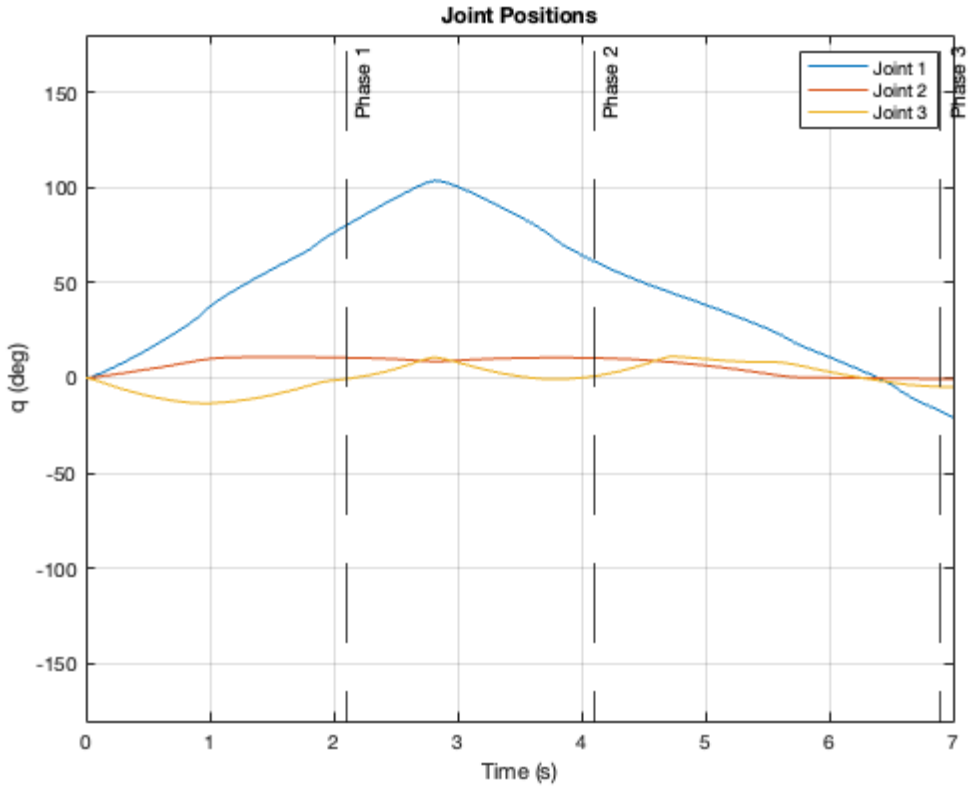
Published with MATLAB® R2022a

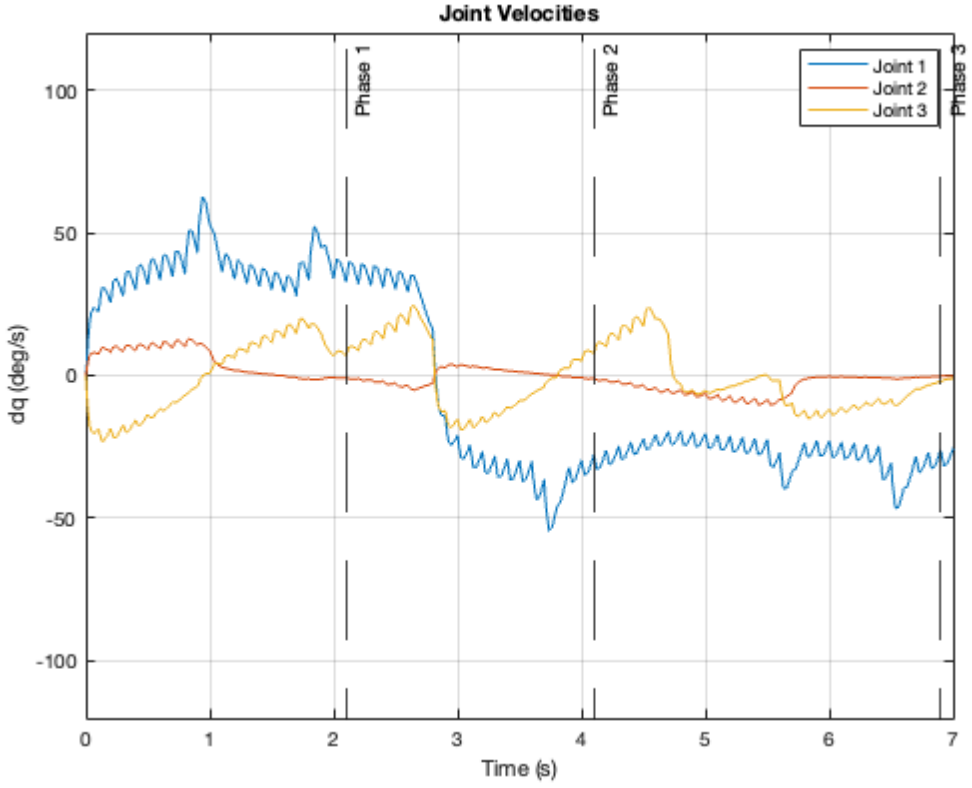
```
vqx = interp1(out.Time.Data(:,1),P_a(1,:),t(1,:),"spline")';
vqy = interp1(out.Time.Data(:,1),P_a(2,:),t(1,:),"spline")';
vqz = interp1(out.Time.Data(:,1),P_a(3,:),t(1,:),"spline")';
vqnorm = sqrt((x_d(:,1)-vqx).^2+(x_d(:,2)-vqy).^2+(x_d(:,3)-vqz).^2);
figure('Name','EE - Desired vs Actual Comparision (L2 norm)');
plot(t(1,:),vqnorm,'Color',[1 0 0]);
hold on;
grid on;
title('EE space - Total position error (L2-norm)');
xlim([0,8.5]);
xlabel('Time (s)');
ylabel('Euclidean Error (m)');
```



Published with MATLAB® R2022a

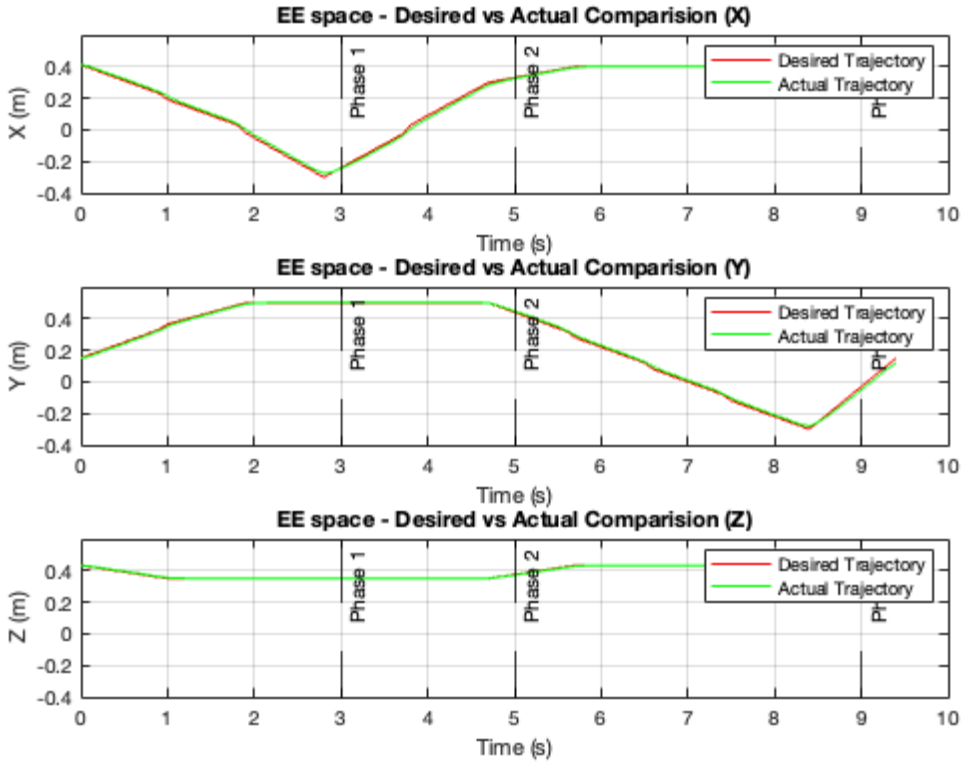
```
figure('Name','Joint Positions');
plot(out.Time.Data(:,1),out.JointPosition.Data(1,:)*(180/pi));
hold on;
plot(out.Time.Data(:,1),out.JointPosition.Data(2,:)*(180/pi));
hold on;
plot(out.Time.Data(:,1),out.JointPosition.Data(3,:)*(180/pi));
grid on;
title('Joint Positions');

% xlim([0,9.4]); % for param_10
% xline(3,'--k','Phase 1');
% xline(5,'--k','Phase 2');
% xline(9,'--k','Phase 3');

xlim([0,7]); % for param_7
xline(2.1,'--k','Phase 1');
xline(4.1,'--k','Phase 2');
xline(6.9,'--k','Phase 3');

% ylim([-pi,pi]);
ylim([-180,180]);
legend('Joint 1','Joint 2','Joint 3');
xlabel('Time (s)');
ylabel('q (deg)');

figure('Name','Joint velocities');
plot(out.Time.Data(:,1),out.JointVel.Data(1,:)*(180/pi));
hold on;
plot(out.Time.Data(:,1),out.JointVel.Data(2,:)*(180/pi));
hold on;
plot(out.Time.Data(:,1),out.JointVel.Data(3,:)*(180/pi));
grid on;
title('Joint Velocities');

% xlim([0,9.4]); % for param_10
% xline(3,'--k','Phase 1');
% xline(5,'--k','Phase 2');
% xline(9,'--k','Phase 3');

xlim([0,7]); % for param_7
xline(2.1,'--k','Phase 1');
xline(4.1,'--k','Phase 2');
xline(6.9,'--k','Phase 3');

% ylim([-pi,pi]);
ylim([-120,120]);
legend('Joint 1','Joint 2','Joint 3');
xlabel('Time (s)');
ylabel('dq (deg/s)');
```





Published with MATLAB® R2022a

```
% EE space Error Analysis
% X-Direction
figure('Name','EE space - Desired vs Actual Comparision');
subplot(3,1,1)
plot(t(1,:),x_d(:,1),'Color',[1 0 0]);
hold on;
grid on;
plot(out.Time.Data(:,1),P_a(1,:),'Color',[0 1 0]);
xline(3,'--k','Phase 1');
xline(5,'--k','Phase 2');
xline(9,'--k','Phase 3');
title('EE space - Desired vs Actual Comparision (X)');
ylim([-0.4,0.6]);
legend('Desired Trajectory', 'Actual Trajectory');
xlabel('Time (s)');
ylabel('X (m)');

% Y-Direction
subplot(3,1,2)
plot(t(1,:),x_d(:,2),'Color',[1 0 0]);
hold on;
grid on;
plot(out.Time.Data(:,1),P_a(2,:),'Color',[0 1 0]);
xline(3,'--k','Phase 1');
xline(5,'--k','Phase 2');
xline(9,'--k','Phase 3');
title('EE space - Desired vs Actual Comparision (Y)');
ylim([-0.4,0.6]);
legend('Desired Trajectory', 'Actual Trajectory');
xlabel('Time (s)');
ylabel('Y (m)');

% Z-Direction
subplot(3,1,3)
plot(t(1,:),x_d(:,3),'Color',[1 0 0]);
hold on;
grid on;
plot(out.Time.Data(:,1),P_a(3,:),'Color',[0 1 0]);
xline(3,'--k','Phase 1');
xline(5,'--k','Phase 2');
xline(9,'--k','Phase 3');
title('EE space - Desired vs Actual Comparision (Z)');
ylim([-0.4,0.6]);
legend('Desired Trajectory', 'Actual Trajectory');
xlabel('Time (s)');
ylabel('Z (m)');
hold off;
```



Published with MATLAB® R2022a

Contents


```
function M = inertia(q)
```

```
% Extracting from input matrix
q1 = q(1);
q2 = q(2);
q3 = q(3);
q4 = q(4);
q5 = q(5);
q6 = q(6);

% Intertial constant reference (Kg.m^2)
%Inertia constants in kg.m^2
I1 = 1.43;
I2 = 1.75;
I3 = 1.38;
I4 = 0.69;
I5 = 0.372;
I6 = 0.333;
I7 = 0.298;
I8 = -0.134;
I9 = 0.0238;
I10 = -0.0213;
I11 = -0.0142;
I12 = -0.011;
I13 = -0.00379;
I14 = 0.00164;
I15 = 0.00125;
I16 = 0.00124;
I17 = 0.000642;
I18 = 0.000431;
I19 = 0.0003;
I20 = -0.000202;
I21 = -0.0001;
I22 = -0.000058;
I23 = 0.00004;

Im = [ 1.14 4.71 0.827 0.2 0.179 0.193 ];

% Inertia matrix elements
M11 = Im(1)+I1+(I3*cos(q2)*cos(q2))+(I7*sin(q2+q3)*sin(q2+q3))+(I10*sin(q2+q3)*cos(q2+q3))+(I11*sin(q2)*cos(q2))+(I21*sin(q2+q3)*sin(q2+q3))+2+(I5*cos

M12 = (I4*sin(q2))+(I8*cos(q2+q3))+(I9*cos(q2))+(I13*sin(q2+q3))-(I18*cos(q2+q3));

M13 = (I8*cos(q2+q3))+(I13*sin(q2+q3))-(I18*cos(q2+q3));

M22 = Im(2)+I2+I6+2*(I5*sin(q3)+I12*cos(q2)+I15+I16*sin(q3));

M23 = (I5*sin(q3))+I6+(I12*cos(q3))+(I16*sin(q3))+(2*I15);

M33 = Im(3)+I6+(2*I15);

M35 = I15+I17;

M44 = Im(4)+I14;

M55 = Im(5)+I17;

M66 = Im(6)+I23;

M21 = M12;

M31 = M13;

M32 = M23;
```

```
Not enough input arguments.

Error in inertia (line 4)
q1 = q(1);
```

M Matrix with coupling interactions- Centralized Control

```
M = [M11 M12 M13 0 0 0;
    M21 M22 M23 0 0 0;
    M31 M32 M33 0 M35 0;
    0 0 0 M44 0 0;
    0 0 0 0 M55 0;
    0 0 0 0 0 M66];
```

Published with MATLAB® R2022a

```
function B_n = coriolis(q,dq)

% Extracting from input matrices
q1 = q(1);
q2 = q(2);
q3 = q(3);
q4 = q(4);
q5 = q(5);
q6 = q(6);

dq1 = dq(1);
dq2 = dq(2);
dq3 = dq(3);
dq4 = dq(4);
dq5 = dq(5);
dq6 = dq(6);

%Inertia constants in kg.m^2
I1 = 1.43;
I2 = 1.75;
I3 = 1.38;
I4 = 0.69;
I5 = 0.372;
I6 = 0.333;
I7 = 0.298;
I8 = -0.134;
I9 = 0.0238;
I10 = -0.0213;
I11 = -0.0142;
I12 = -0.011;
I13 = -0.00379;
I14 = 0.00164;
I15 = 0.00125;
I16 = 0.00124;
I17 = 0.000642;
I18 = 0.000431;
I19 = 0.0003;
I20 = -0.000202;
I21 = -0.0001;
I22 = -0.000058;
I23 = 0.00004;

Im = [ 1.14 4.71 0.827 0.2 0.179 0.193 ];

% Coriolis matrix elements
b112=2*(-I3*sin(q2)*cos(q2)+I5*cos(q2+q2+q3) +I7*sin(q2+q3)*cos(q2+q3) -I12*sin(q2+q2+q3)-I15*2*sin(q2+q3)*cos(q2+q3)+I16*cos(q2+q2+q3)+I21*sin(q2+q3)

b113=2*(I5*cos(q2)*cos(q2+q3)+I7*sin(q2+q3)*cos(q2+q3)-I12*cos(q2)*sin(q2+q2)+ I15*2*sin(q2+q3)*cos(q2+q3)+I16*cos(q2)*cos(q2+q3)+I21*sin(q2+q3)*cos(q

b115=2*(-sin(q2+q3)*cos(q2+q3)+I15*2*sin(q2+q3)*cos(q2+q3)+I16*cos(q2)*cos(q2+q3)+I22*cos(q2+q3)*cos(q2+q3));

b123=2*(-I8*sin(q2+q3)+I13*cos(q2+q3)+I18*sin(q2+q3));

b214=I14*sin(q2+q3)+I19*sin(q2+q3)+2*I20*sin(q2+q3)*(1-0.5);

b223=2*(-I12*sin(q3)+I5*cos(q3)+I16*cos(q3));

b225=2*(I16*cos(q3)+I21);

b235=2*(I16*cos(q3)+I22);

b314=2*(I20*sin(q2+q3)*(1-0.5))+I14*sin(q2+q3)+I19*sin(q2+q3);

b412=-1*(I14*sin(q2+q3)+I19*sin(q2+q3)+2*I20*sin(q2+q3)*(1-0.5));

b412=b214;

b413=-1*(I14*sin(q2+q3)+I19*sin(q2+q3)+2*I20*sin(q2+q3)*(1-0.5));

b413=-1*b314;

b415=-I20*sin(q2+q3)-I17*sin(q2+q3);

b514=I20*sin(q2+q3)+I17*sin(q2+q3);

% Coriolis Matrix B
% B = zeros(15,6);
%
% B(1,1) = b112;
% B(1,2) = b113;
% B(1,4) = b115;
% B(1,6) = b123;
%
% B(2,3) = b214;
% B(2,6) = b223;
% B(2,8) = b225;
% B(2,11) = b235;
%
```

```
%
% B(4,1) = b412;
% B(4,2) = b412;
% B(4,4) = b415;
%
% B(5,3) = b514;

% B_n = B.*[qd1*qd2; qd1*qd3; qd2*qd3; 0; 0; 0];

% B_test = [B(1,1) B(1,2) B(1,3);
%     B(2,1) B(2,2) B(2,3);
%     B(3,1) B(3,2) B(3,3)];
%
% B_n = B_test*[dq1*dq2; dq1*dq3; dq2*dq3];

B_n = [b112*dq1*dq2 + b113*dq1*dq3 + b123*dq2*dq3;
    b223*dq2*dq3;
    0;
    b412*dq1*dq2 + b413*dq1*dq3;
    0;
    0];


end
```

```
Not enough input arguments.

Error in coriolis (line 4)
q1 = q(1);
```

Published with MATLAB® R2022a

```
function C_n = centrifugal(q,dq)

% Extracting from input matrices
q1 = q(1);
q2 = q(2);
q3 = q(3);
q4 = q(4);
q5 = q(5);
q6 = q(6);

dq1 = dq(1);
dq2 = dq(2);
dq3 = dq(3);
dq4 = dq(4);
dq5 = dq(5);
dq6 = dq(6);

%Inertia constants in kg.m^2
I1 = 1.43;
I2 = 1.75;
I3 = 1.38;
I4 = 0.69;
I5 = 0.372;
I6 = 0.333;
I7 = 0.298;
I8 = -0.134;
I9 = 0.0238;
I10 = -0.0213;
I11 = -0.0142;
I12 = -0.011;
I13 = -0.00379;
I14 = 0.00164;
I15 = 0.00125;
I16 = 0.00124;
I17 = 0.000642;
I18 = 0.000431;
I19 = 0.0003;
I20 = -0.000202;
I21 = -0.0001;
I22 = -0.000058;
I23 = 0.00004;

Im = [ 1.14 4.71 0.827 0.2 0.179 0.193 ];

% Coriolis matrix elements
b112=2*(-I3*sin(q2)*cos(q2)+I5*cos(q2+q2+q3) +I7*sin(q2+q3)*cos(q2+q3) -I12*sin(q2+q2+q3)-I15*2*sin(q2+q3)*cos(q2+q3)+I16*cos(q2+q2+q3)+I21*sin(q2+q3)

b113=2*(I5*cos(q2)*cos(q2+q3)+I7*sin(q2+q3)*cos(q2+q3)-I12*cos(q2)*sin(q2+q2)+ I15*2*sin(q2+q3)*cos(q2+q3)+I16*cos(q2)*cos(q2+q3)+I21*sin(q2+q3)*cos(q

b115=2*(-sin(q2+q3)*cos(q2+q3)+I15*2*sin(q2+q3)*cos(q2+q3)+I16*cos(q2)*cos(q2+q3)+I22*cos(q2+q3)*cos(q2+q3));

b123=2*(-I8*sin(q2+q3)+I13*cos(q2+q3)+I18*sin(q2+q3));

b214=I14*sin(q2+q3)+I19*sin(q2+q3)+2*I20*sin(q2+q3)*(1-0.5);

b223=2*(-I12*sin(q3)+I5*cos(q3)+I16*cos(q3));

b225=2*(I16*cos(q3)+I21);

b235=2*(I16*cos(q3)+I22);

b314=2*(I20*sin(q2+q3)*(1-0.5))+I14*sin(q2+q3)+I19*sin(q2+q3);

b412=-1*(I14*sin(q2+q3)+I19*sin(q2+q3)+2*I20*sin(q2+q3)*(1-0.5));

b412=b214;

b413=-1*(I14*sin(q2+q3)+I19*sin(q2+q3)+2*I20*sin(q2+q3)*(1-0.5));

b413=-1*b314;

b415=-I20*sin(q2+q3)-I17*sin(q2+q3);

b514=I20*sin(q2+q3)+I17*sin(q2+q3);


% Centrifugal matrix elements
c12 = I4*cos(q2)-I8*sin(q2+q3)-I9*sin(q2)+I13*cos(q2+q3)+I18*sin(q2+q3);
c13 = 0.5*b123;
c21 = -0.5*b112;
c23 = 0.5*b223;
c31 = -0.5*b113;
c32 = -1*c23;

c51 = -0.5*b115;
c52 = -0.5*b225;

% %Centrifugal matrix C
```

```
%
% C(1,2) = c12;
% C(1,3) = c13;
%
% C(2,1) = c21;
% C(2,3) = c23;
%
% C(3,1) = c31;
% C(3,2) = c32;
%
% C(5,1) = c51;
% C(5,2) = c52;

% C_n = C.*[qd1^2; qd2^2; qd3^2; 0; 0; 0];

% C_test = [C(1,1), C(1,2), C(1,3);
%     C(2,1), C(2,2), C(2,3);
%     C(3,1), C(3,2), C(3,3)];
%
% C_n = C_test*[dq1^2; dq2^2; dq3^2];

C_n = [c12*dq2^2 + c13*dq3^2;
    c21*dq1^2 + c23*dq3^2;
    c13*dq1^2 + c32*dq2^2;
    0;
    c51*dq1^2 + c52*dq2^2;
    0];


end
```

```
Not enough input arguments.

Error in centrifugal (line 4)
q1 = q(1);
```

Published with MATLAB® R2022a

```
function G_n = gravity(q)

% Extracting from input matrices
q1 = q(1);
q2 = q(2);
q3 = q(3);
q4 = q(4);
q5 = q(5);
q6 = q(6);

% Gravitational constants (N.m)
g1 = -37.2;
g2 = -8.44;
g3 = 1.02;
g4 = 0.249;
g5 = -0.0282;

% Gravity matrix elements
G2 = (g1*cos(q1))+(g2*sin(q2+q3))+(g3*sin(q2))+(g4*cos(q2+q3))+(g5*sin(q2+q3));
G3 = (g2*sin(q2+q3))+(g4*cos(q2+q3))+(g5*sin(q2+q3));
G5 = g5*sin(q2+q3);

% Gravity matrix G

G_n = [0; G2; G3; 0; G5; 0];

end
```

```
Not enough input arguments.

Error in gravity (line 4)
q1 = q(1);
```

Published with MATLAB® R2022a