README

This project involves planning and executing a trajectory for the end-effector of the Kuka youBot mobile manipulator. The software handles odometry for the chassis as it moves and implements feedback control to guide the youBot in performing a pick-and-place task. The task includes picking up a block from a specified location, transporting it to a desired location, and placing it accurately.
The Python script outputs a comma-separated values (CSV) file that records:

- Configurations of the chassis and arm.
- Angles of the four wheels.
- State of the gripper (open or closed) over time.
  The resulting CSV file is imported into the **CoppeliaSim** simulator to visualize and validate the performance of the task.

The code is construct by three steps：

1. `Milestone1.py`: Computes the next configuration of the robot
2. `Milestone2.py`: Compute the trajectory for the gripper position and orientation.
3. `Milestone3.py`: Compute the kinematic task-space feedforward plus feedback control law
   Additional files are:
4. `run.py`: Running for the result and plotting.
5. `config.py`: The configuration of the youBot, including the cube initial state and goal state.
6. `additional.py`: The helper function for data storage and plotting.

Milestone 1

1. `NextState(robot_config12, robot_speeds9, dt, w_max)`

- **Purpose**: Computes the next configuration of the robot using:
  - Current configuration (`robot_config12`), speed inputs (`robot_speeds9`), timestep (`dt`), and speed limit (`w_max`).
- **Output**: Updated configuration as a 12-element vector.
- **Key Features**:
  - Uses wheel geometry to compute motion in global coordinates.
  - Updates arm joint and wheel angles with Euler integration.

2. `RunNextState(robot_config12, u, thetad, dt, w_max)`

- **Purpose**: Simulates the robot's motion under constant controls for one second and writes the results to a CSV file.

- **Output**: Robot state at each timestep saved in `project/csv/next_state.csv`.

3. `TestNextState()`

- **Purpose**: Provides test cases to validate the motion control functions.
- **Implementation**:
  - Simulates forward motion, sideways motion, and in-place rotation.
  - Logs results to a CSV file.

Milestone 2

`TrajectoryGenerator(Tse_i, Tsc_i, Tsc_f, Tce_grasp, Tce_standoff, k)`

- **Purpose**: Generates a reference trajectory for the end-effector of the Kuka youBot to perform a pick-and-place task.
  - The trajectory includes the motion of the end-effector and gripper states (open/close) to grasp and place a cube at specified locations.
  - The trajectory is divided into eight distinct motion segments for smooth task execution.
- **Inputs**:
  - `Tse_i`: Initial configuration of the end-effector.
  - `Tsc_i`: Initial configuration of the cube.
  - `Tsc_f`: Desired final configuration of the cube.
  - `Tce_grasp`: End-effector configuration relative to the cube during grasping.
  - `Tce_standoff`: End-effector standoff configuration above the cube.
  - `k`: Number of trajectory points per 0.01 seconds.
- **Output**:
  - A `13 × 1` array representing the concatenated trajectory, saved as a CSV file `project/csv/trajectory.csv`.
- **Key Features**:
  - Uses the **Screw Trajectory Method** for smooth SE(3) transitions.
  - Accounts for gripper state changes during grasp and release.
  - Modular structure allows for easy customization of trajectory resolution and configurations.

Milestone 3

1. `FeedbackControl(X, Xd, Xd_next, Kp, Ki, Xerr_int, dt)`

- **Purpose**: Computes the commanded end-effector twist to follow a desired trajectory using feedback control with proportional-integral (PI) gains.
- **Inputs**:
  - `X`: Current end-effector configuration (SE(3) matrix).
  - `Xd`: Current reference configuration for the end-effector (SE(3) matrix).
  - `Xd_next`: Next reference configuration for the end-effector (SE(3) matrix).
  - `Kp`: Proportional gain matrix.
  - `Ki`: Integral gain matrix.
  - `Xerr_int`: Accumulated error for the integral term.
  - `dt`: Timestep (Δt).
- **Outputs**:
  - `V_new`: Commanded twist for the end-effector.
  - `Xerr`: Current error between the actual and reference configurations.
  - `Xerr_int`: Updated accumulated error for the integral term.
- **Key Features**:
  - Uses the Matrix Logarithm to calculate the error between SE(3) configurations.
  - Computes the commanded twist using proportional and integral error terms.
  - Supports both feedforward and feedback control.

2. `CalculateJe(robot_config8, Tb0, M0e, Blist)`

- **Purpose**: Computes the combined Jacobian matrix for the robot, which includes both the arm and chassis contributions.
- **Inputs**:
  - `robot_config8`: 8-element configuration vector (chassis position and joint angles).
  - `Tb0`: Transformation from chassis to the arm base.
  - `M0e`: End-effector home configuration.
  - `Blist`: Screw axes in the end-effector frame.
- **Output**:
  - `Je`: Combined Jacobian matrix (`6 × 9`) with contributions from the chassis (4 wheels) and the arm (5 joints).
- **Key Features**:
  - Uses wheel geometry to compute the base Jacobian.
  - Combines the base and arm Jacobians using the adjoint transformation.
  - Outputs a unified Jacobian for controlling both the arm and the chassis.

3. `TestFeedbackControl()`

- **Purpose**: Validates the implementation of feedback control and Jacobian calculations with sample inputs.
- **Implementation**:

1. Defines initial conditions for the end-effector and reference configurations.
2. Computes the commanded twist (`V_new`) using `FeedbackControl`.
3. Calculates the combined Jacobian (`Je`) and uses it to compute joint and wheel velocities (`u_thetad`) using the pseudoinverse.
4. Tests with different `Kp` and `Ki` values to demonstrate the impact of proportional and integral control.

- **Output**:
  - Logs the computed twist and wheel/joint velocities to verify correctness.

Joint Limit Handling

- **Purpose**: Ensures the robot operates safely within its joint limits during trajectory execution.
- **Implementation**:
  - Checks each joint's angle against predefined limits during every timestep using the `testJointLimits` function.
  - If a joint exceeds its limits, the corresponding column in the Jacobian is set to zero, effectively disabling movement in that joint.
  - Recomputes the joint velocities (`u_thetad`) with the modified Jacobian to avoid further violations.
- **Key Features**:
  - Dynamically adjusts motion planning to prevent joint limit violations.
  - Maintains smooth operation without abrupt halts, even when limits are reached.
  - Improves the robustness of the control algorithm for real-world scenarios.

Workflow Summary

1. **Trajectory Generation (Milestone 2)**:
   - Generates a smooth trajectory for the robot to follow, including the gripper's states.
   - Trajectory segments are saved as a reference path.
2. **Feedback Control (Milestone 3)**:
   - Computes the required joint and wheel velocities using the combined Jacobian and feedback errors.
   - Ensures the robot follows the desired trajectory with minimal error.
3. **Joint Limit Handling**:
   - Dynamically detects and handles joint limit violations by modifying the Jacobian matrix during motion.
4. **Output**:
   - Logs the robot’s motion and error data for further analysis or visualization.

How to Run

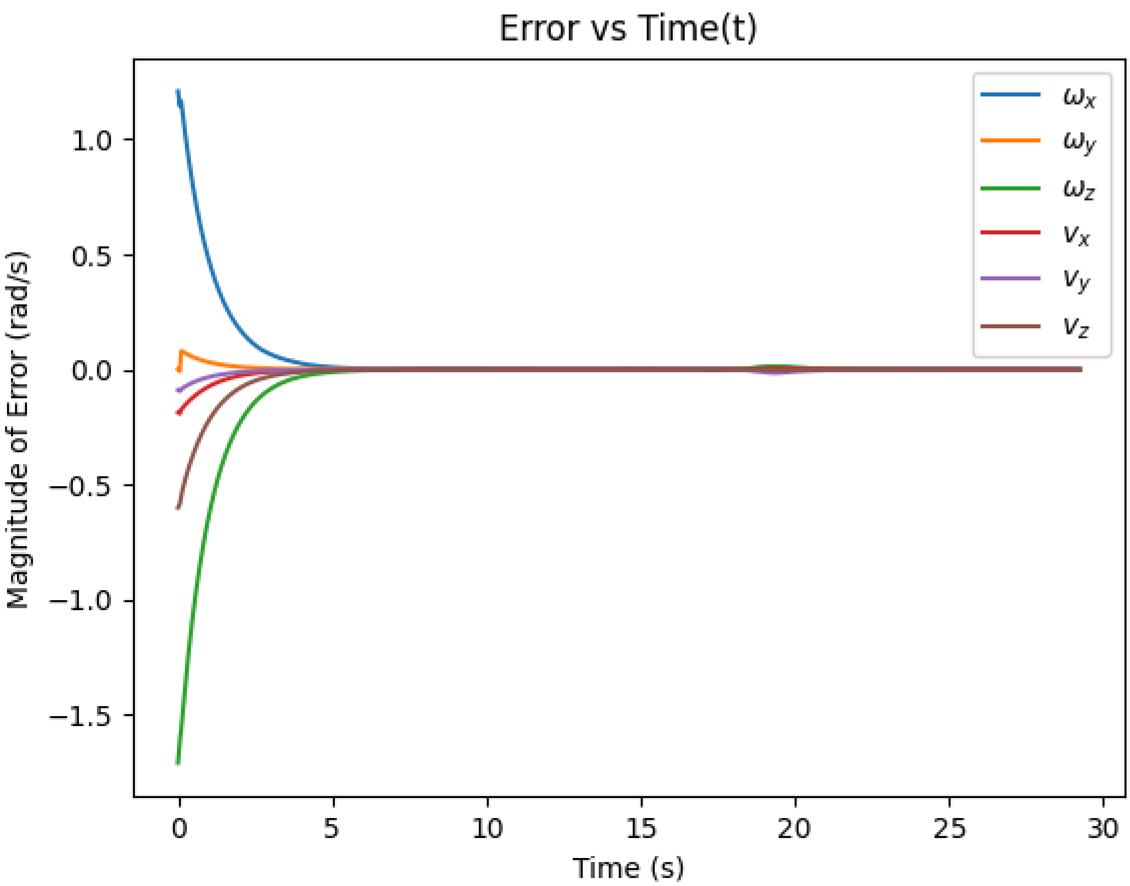
1. **Setup Dependencies**:
    - Ensure `numpy`, `modern_robotics`, `matplotlib`, and other libraries are installed.
    - Place the supporting scripts (`Milestone1`, `Milestone2`, `Milestone3`, `config.py`, and `additional.py`) in the project directory.
2. **Run the Script**:
    - Execute the script: `python3 <script_name>.py`
    - Results will be saved in the `project/csv` folder.
3. **Customize Parameters**:
   - Modify control gains (`Kp`, `Ki`), initial conditions, or trajectory points in the `main()` function to test different scenarios.

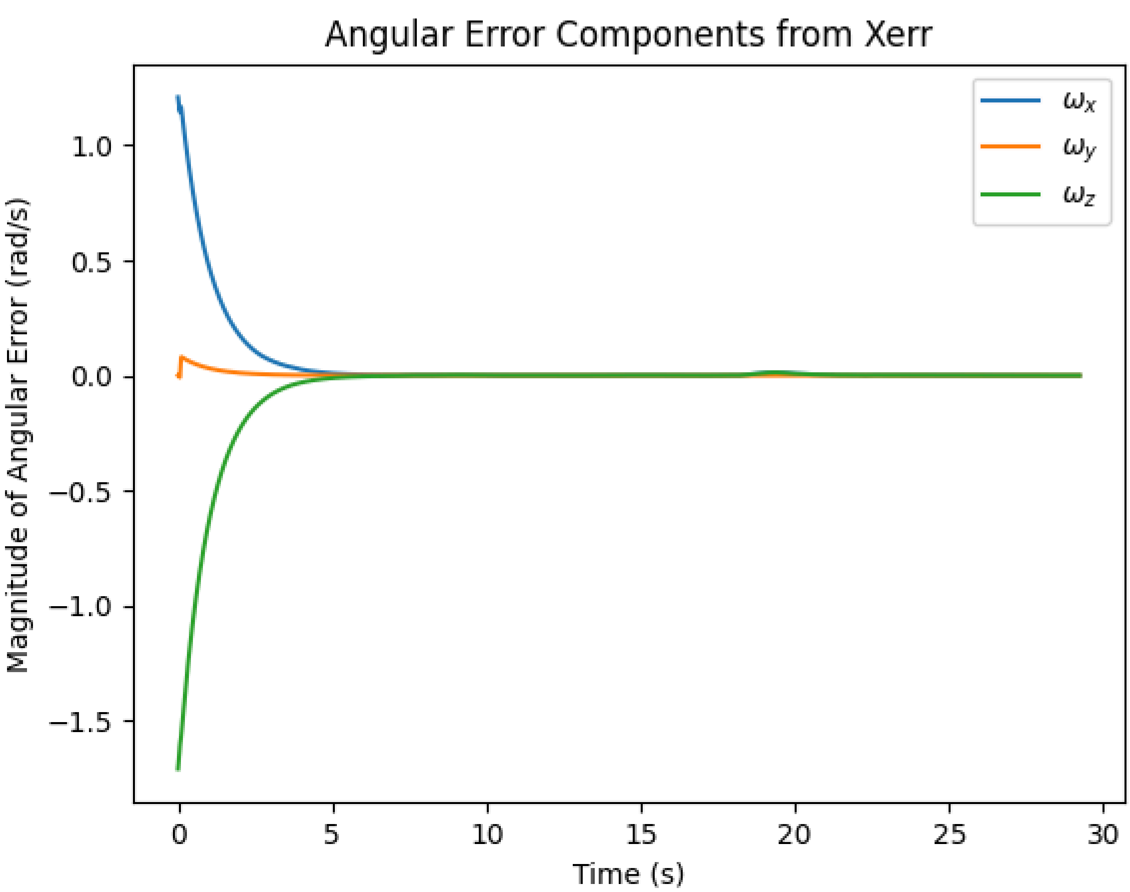
   Log for the running result

```
(446) (base) ericchen@EricdeMacBook-Pro ME449 % python -u "/Users/ericchen/ME449/project/code/run.py"
TrajectoryGenerator():
100%|████████████████████████████████████████| 8/8 [00:00<00:00, 42.21it/s]

Main():
100%|████████████████████████████████████████| 2923/2923 [00:03<00:00, 856.90it/s]

Data of 13-vector generated.


Error generated.

Results written to /csv folder.
```

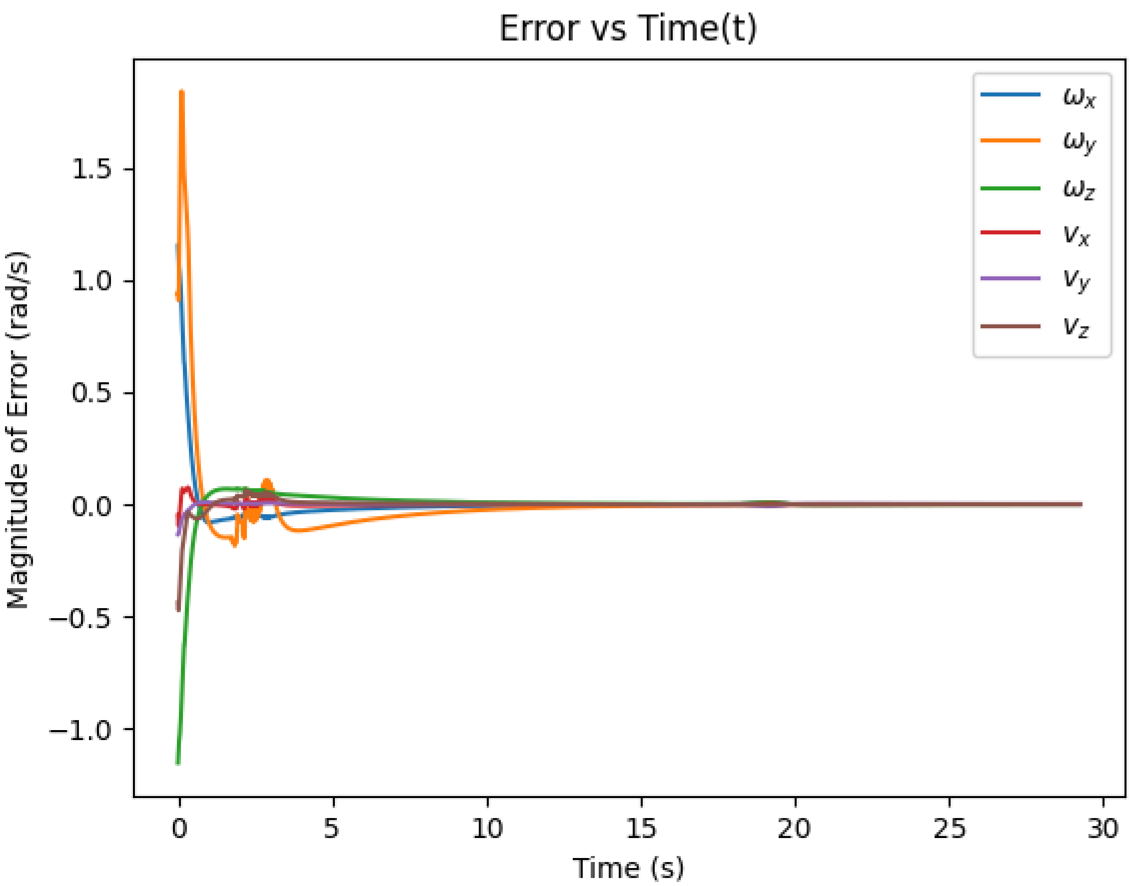
Result

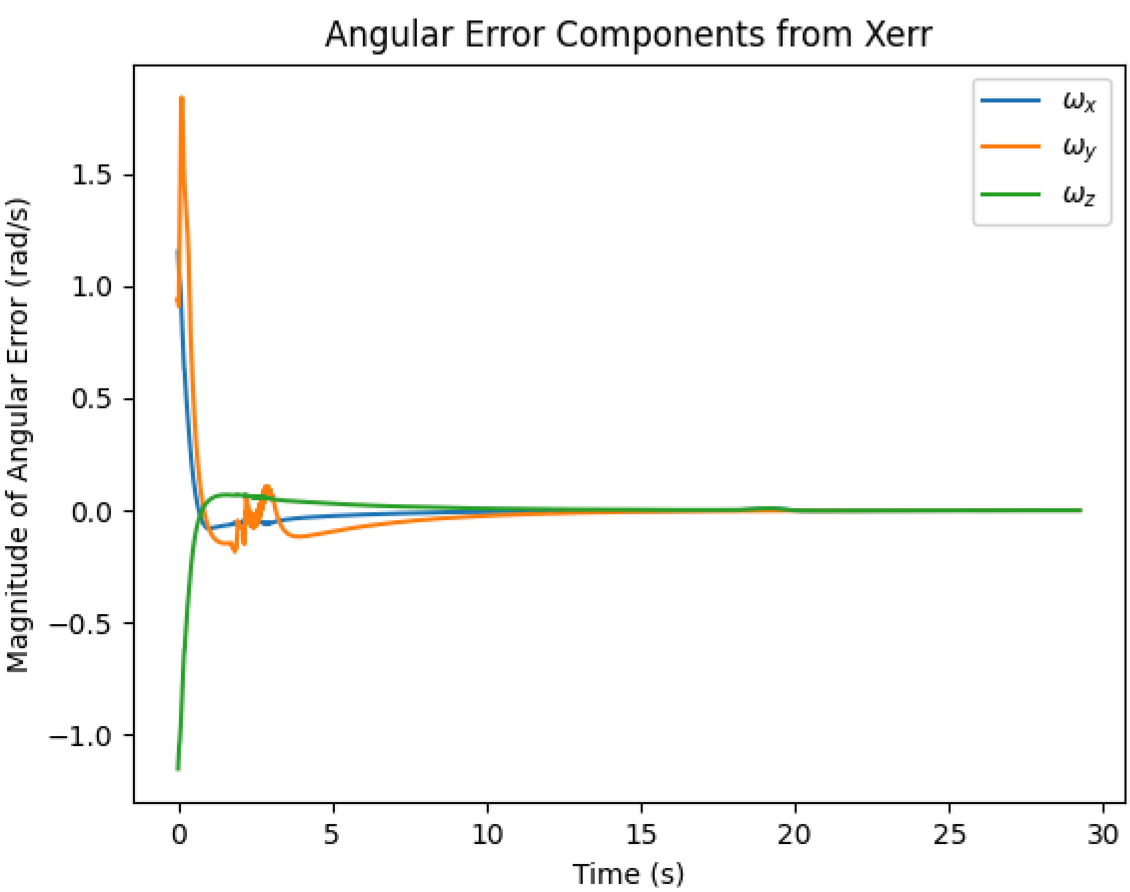
1. Best:


Error vs Time(t)
ω_x
ω_y
ω_z
v_x
v_y
v_z
Magnitude of Error (rad/s)
Time (s)
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
5
10
15
20
25
30

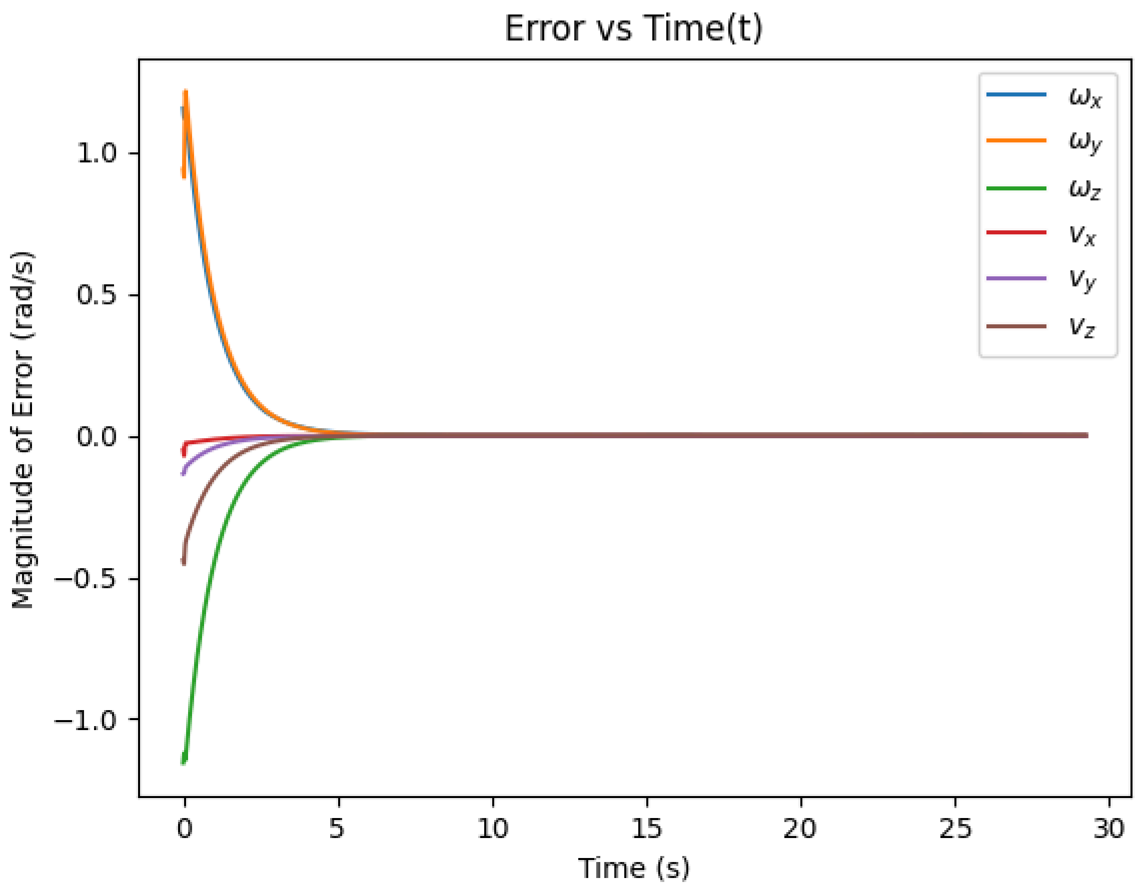

Angular Error Components from Xerr
ω_x
ω_y
ω_z
Magnitude of Angular Error (rad/s)
Time (s)
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
5
10
15
20
25
30

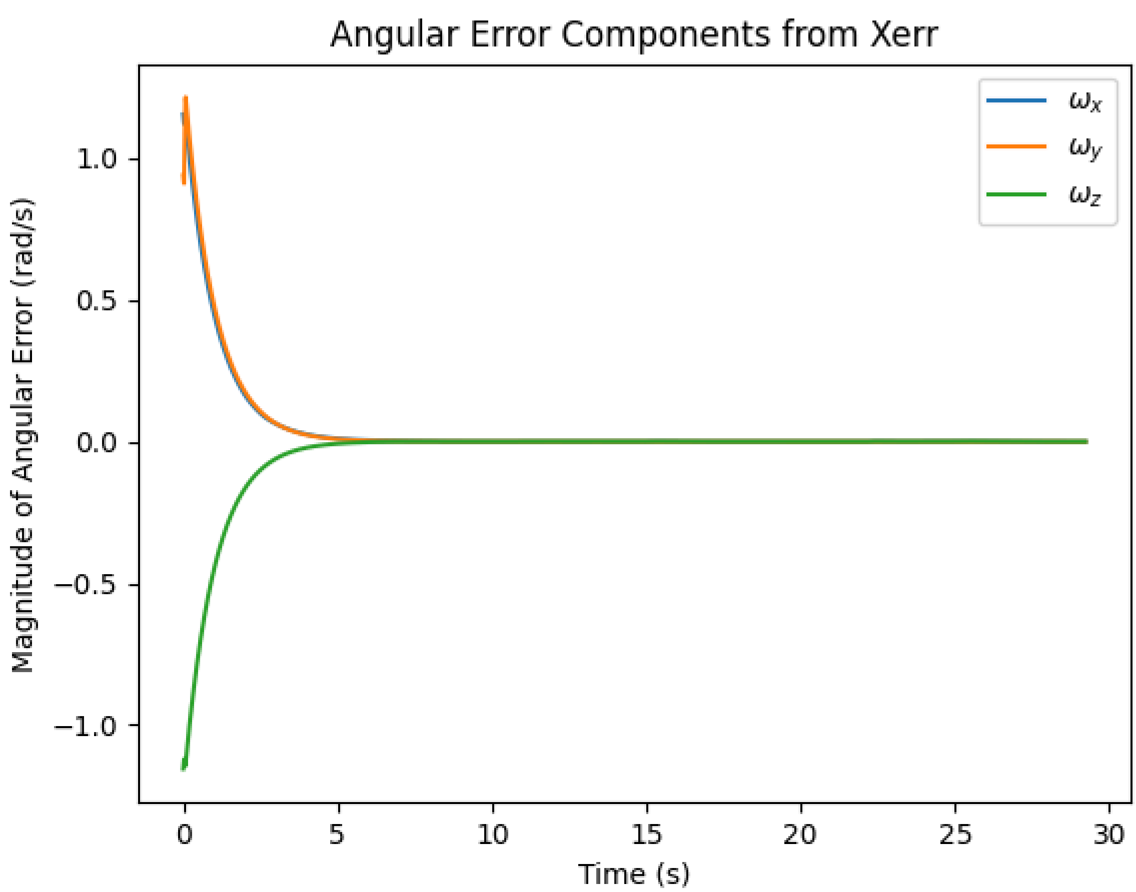
2. Overshoot:


Error vs Time(t)
ω_x
ω_y
ω_z
v_x
v_y
v_z
1.5
1.0
0.5
0.0
−0.5
−1.0
Magnitude of Error (rad/s)
0
5
10
15
20
25
30
Time (s)


Angular Error Components from Xerr
ω_x
ω_y
ω_z
1.5
1.0
0.5
0.0
−0.5
−1.0
Magnitude of Angular Error (rad/s)
0
5
10
15
20
25
30
Time (s)

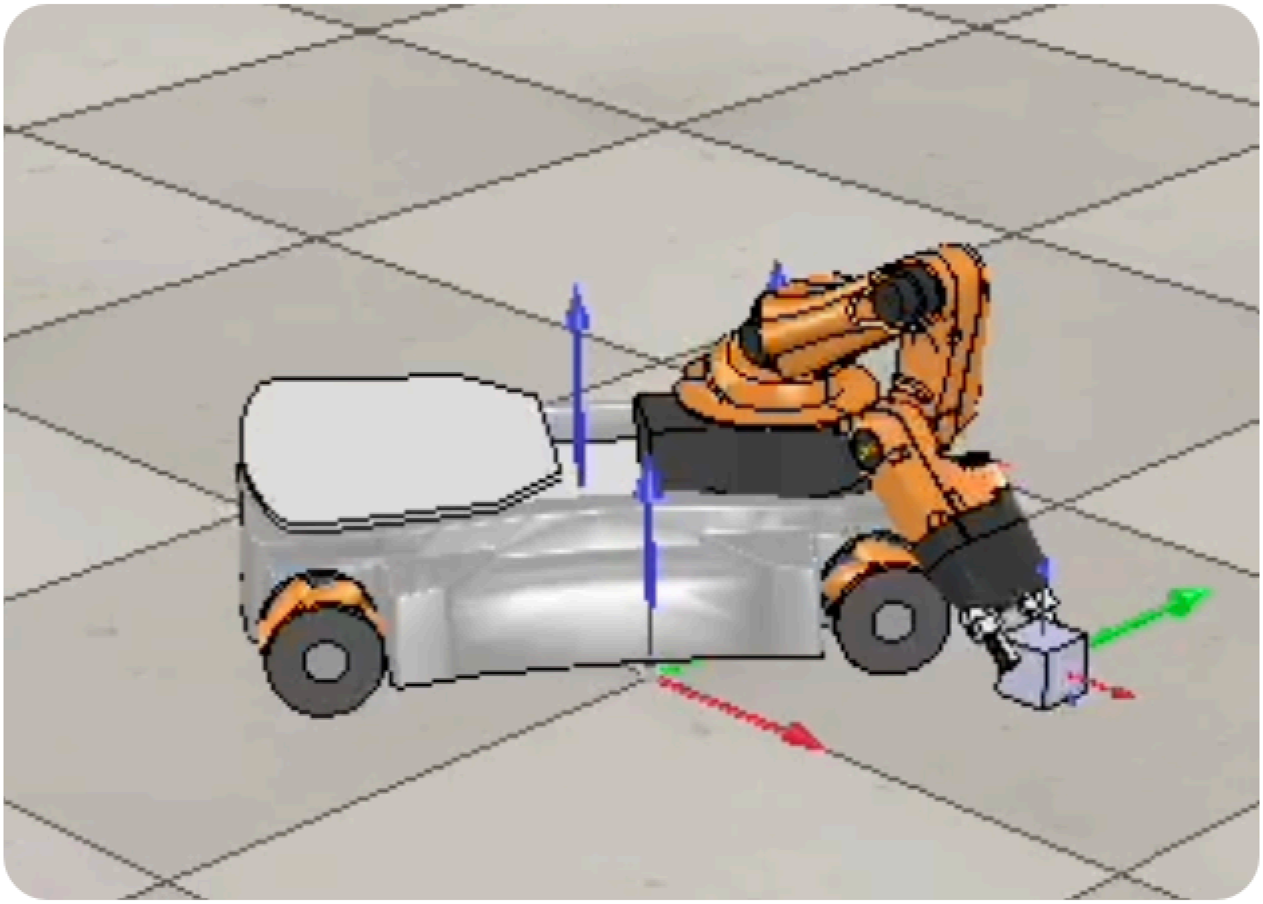
3. Newtask:





4. Jointlimit:

With joint limit:



Without joint limit: