Mover 6 Report Part 1

Contents

1. Background:

This report section delineates the steps to develop a robust robot joint controller for Mover6 to manoeuvre the robot end effector from point A to B using various trajectories and trapezoidal velocity profiling. Additionally, the controller implements a joint velocity control algorithm which dynamically adjusts the joint angular velocities to achieve same timing for all the joints, crucial for linear movement.

Achieving accurate end effector position and orientation is critical in numerous industries. Substantial manufacturing processes rely heavily on robot manipulators which must be accurate and precise in their operation to not compromise the production line, which otherwise may have damaging, or even catastrophic consequences. In addition, the application of linear motion, or a movement along a straight line between two points, is extensively used in different aspects of numerous industries such as: vehicle assembly, robotic surgery systems, liquid handling systems etc where linear motion is crucial. An example to illustrate this would be if our Mover6 robot was tasked to move a metallic rod into a hole of same radius. The robot can only guide the rod into the hole without damaging any of the components by following a straight line. On the other hand, trapezoidal velocity profiling of joints is useful to minimize jerk experienced by the robot by adjusting the blending time, which will result in a partial parabolic motion throughout the trajectory.

1.1 Aims and Objectives:

1.1.1 Aims:

Design a robot controller to achieve precise linear, parabolic and spline motion and trapezoidal velocity profiling. Realise the controller as a combination of MATLAB and C++.

1.1.2 Basic Objectives:

1) Send Robot demand signals control over terminal window.
2) Basic MATLAB – ROS communication.

1.1.3 Advanced Objectives:

1) Write the C++ code using modern OOP (object-oriented programming design principles).
    a. Joint Class.
    b. Robot Class.
2) Joints synchronized stopping.
3) Develop a robust and effective joint controller.
4) Achieve linear, spline / parabolic motion.
5) Trapezoidal velocity profiling.
6) Constrain inverse kinematics solver.
7) Validate the controller.

2. Procedure & Motion architecture:

2.1 Joints speed synchronization:

Maximum joint speed (λ_{max}, rad/s) is set for the joint that needs to travel the furthest from current state ($J_{x_{state}}$) to the demand ($J_{x_{demand}}$).Hence maximum time

$$t_{max} = \frac{J_{x_{demand}} - J_{x_{state}}}{\lambda_{max}} \quad (1)$$

∴ Each joint speed: $$\lambda_m = \frac{J_{m_{demand}} - J_{m_{state}}}{t_{max}} \quad , \forall\, m \in \{1,2,\dots,6\}. \quad (2)$$

This ensures that all joints arrive to the destination at the same time while varying velocity.

2.2 Linear motion:

Given a vector $B = [X, Y, Z, A_R, B_R, C_R]^T$ defining the end effector position and orientation in 3-dimensional Euclidean space and a vector $J = [j_{1,init},\ j_{2,init},, j_{3,init}, j_{4,init}, j_{5,init}, j_{6,init}]^T$ defining the initial robot configuration, home or any.

Forward kinematics is computed given the initial joint configuration to calculate the initial end effector pose transformation matrix $I_{init} \in R^{4x4}$. Then $x\ y\ z$ translational coordinates are extracted from I_{init}, to get $A = [X_0, Y_0, Z_0]^T$. A parametric equation of a line is defined between point A & B as

$$\overrightarrow{AB} = t\ \left((X - X_0)\hat{\imath} + (Y - Y_0)\hat{\jmath} + (Z - Z_0)\hat{k}\right), \forall\, t \in [0,1] \quad (3)$$

∴ Any coordinate along the line is defined by the following equations:

$$\begin{aligned} x(t) &= (1-t)X_0 + tX \\ y(t) &= (1-t)Y_0 + tY \\ z(t) &= (1-t)Z_0 + tZ \end{aligned} \quad (4)$$

$$\therefore r(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} \quad (5)$$

Given N samples along the line $r(t)$, a waypoints matrix $P \in R^{3xN}$ is generated:

$$P = \begin{bmatrix} X_0 & \cdots & X \\ Y_0 & \dots & Y \\ Z_0 & \cdots & Z \end{bmatrix} \quad (6)$$

A rotational transformation matrix $R\_T \in R^{4x4}$ is firstly defined which incorporates the Euler rotations $[A_R, B_R, C_R]$. A linear transformation tensor $L \in R^{4\times4\times N}$ is then defined and homogeneous transformation tensor ψ.

$$\psi_{i,j,k} \in R^{4\times4\times N}, \psi_{i,j,k}\ \forall\ i,j \in \{1,2,3,4\}, k \in \{1,2,\dots,N\} \tag{7}$$

$$L_{i,j,k^{th}} = \begin{bmatrix} 1 & 0 & 0 & P(1,k) \\ 0 & 1 & 0 & P(2,k) \\ 0 & 0 & 1 & P(3,k) \\ 0 & 0 & 0 & 1 \end{bmatrix} \forall\ k \in \{1,2,\dots,N\} \tag{8}$$

$$\therefore\ \psi_{i,j,k^{th}} = L_{i,j,k^{th}} \times R\_T_{i,j} \tag{9}$$

The inverse kinematics for each $\psi_{i,j,k^{th}}$ matrix is computed to find the corresponding joint configuration for each k^{th} element in the tensor, hence generating the waypoints joint solutions matrix:

$$\Omega \in R^{Nx6} \tag{10}$$

Finally, Ω is passed for the robot joints controller to execute, thereby achieving linear motion along the trajectory as shown in Figure 1.

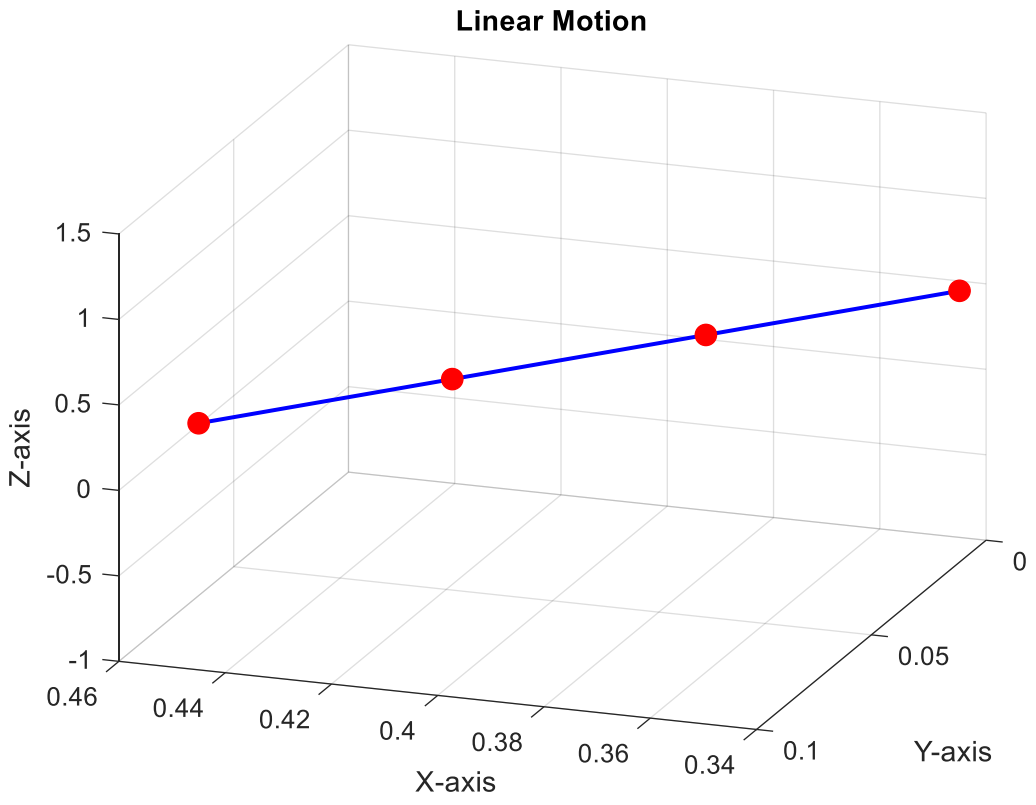

Figure 1. Linear motion profile

2.3 Spline / Parabolic motion:

2.3.1 Parabolic motion:

Given point A $(x_A, y_{A,}, z_A)$ and B (x_B, y_B, z_B) polynomial interpolation is utilized to fit a quadratic trajectory between the initial location and destination of the end effector as follows:

$$r(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} a_x t^2 + b_x t + c_x \\ a_y t^2 + b_y t + c_y \\ a_z t^2 + b_z t + c_z \end{bmatrix} \tag{11}$$

where $t \in [0,1]$ and boundary conditions:

$$\begin{aligned} &\mathrm{r}(0) = \mathrm{A} \\ &r(0.5) = M \\ &r(1) = B \end{aligned} \tag{12}$$

where $M = (x_M, y_{M,}, z_M)$ is the peak mid-point defined between A & B.

Given 6 unknowns and 6 coordinates a unique 2nd order polynomial can be found that fits the trajectory by solving a system of equations:

For $x(t)$:

$$\begin{bmatrix} a_x \\ b_x \\ c_x \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0.25 & 0.5 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_A \\ x_B \\ x_M \end{bmatrix} \tag{13}$$

For $y(t)$:

$$\begin{bmatrix} a_y \\ b_y \\ c_y \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0.25 & 0.5 & 1 \end{bmatrix}^{-1} \begin{bmatrix} y_A \\ y_B \\ y_M \end{bmatrix} \tag{14}$$

For $z(t)$:

$$\begin{bmatrix} a_z \\ b_z \\ c_z \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0.25 & 0.5 & 1 \end{bmatrix}^{-1} \begin{bmatrix} z_A \\ z_B \\ z_M \end{bmatrix} \tag{15}$$

The matrix in (13,14 and 15) is defined from (11,12) and is invertible, thereby a unique solution is guaranteed.

Example:

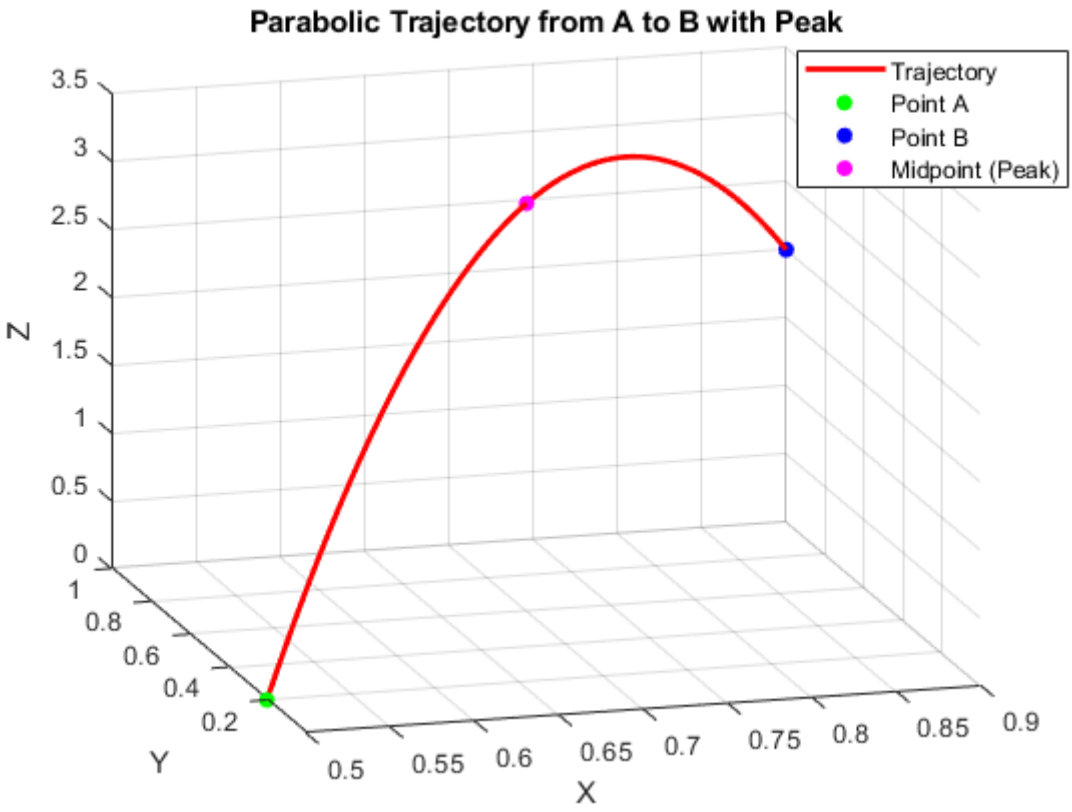

Figure 2. Parabolic motion trajectory example [6]

Finally, steps (5) to (10) are repeated, achieving a parabolic trajectory.

2.3.2 Spline motion:

Given points A and B the equations for any n^{th} order polynomial (i.e. equation 11) can be defined and repeat the above steps to execute the motion hence achieving a generalized solution. An example of spline motion is shown in Figure 3.

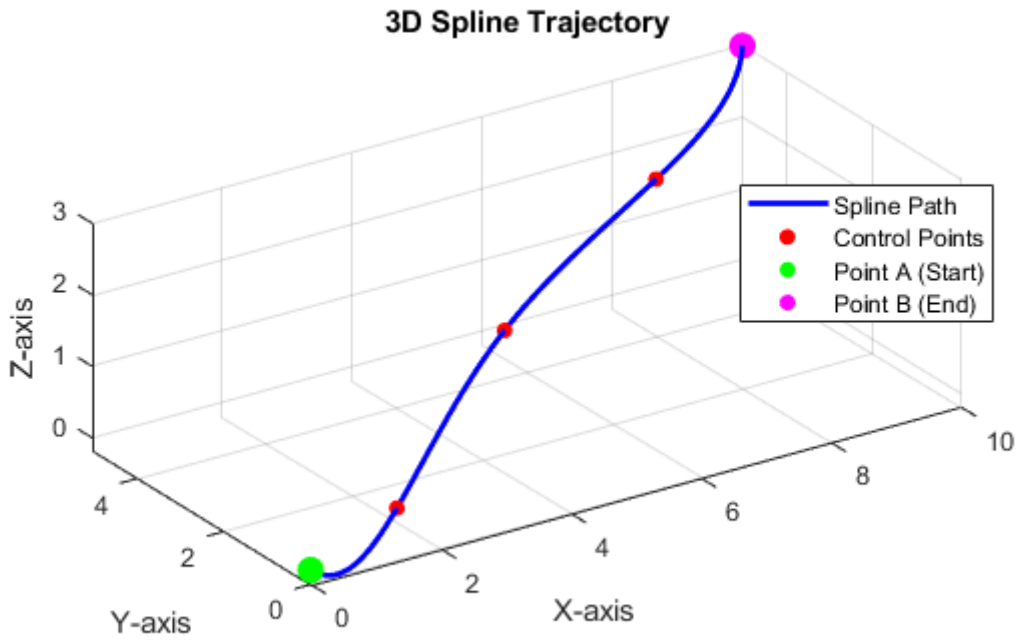

Figure 3. Spline trajectory between initial position and final position.

Keynote:

Using the methods developed above, any type of motion is achievable, however the algorithm must ensure a valid solution constrained within the reachable workspace. A method to address this problem is outlined in section 6.

2.2.4 Trapezoidal velocity profiling:

A method has been developed to execute a trapezoidal velocity profile for each joint as follows. Given a point A (initial position) and B (final position) as defined above, the inverse kinematics is computed to obtain the joint configurations at both locations. The maximum time for the motion profile t_{max} and the blending time t_{blend} (for this experiment $t_{blend} = \frac{t_{max}}{4}$) are defined. Note for a symmetric trapezoid t_{blend} must satisfy the following constraint $t_{blend} < \frac{t_{max}}{2}$. The maximum velocity V_{max} is calculated (unique for each joint to ensure correct final displacement) as:

$$V_{max} = \frac{|J_{demand} - J_{initial}|}{t_{max} - t_{blend}} \tag{16}$$

And the angular velocity for each joint as:

$$\therefore v(t) = \begin{cases} (\frac{J_{demand} - J_{current}}{|J_{demand} - J_{current}|}) \times (\frac{V_{max}}{t_{blend}})t, \ 0 \leq t < t_{blend} \\ (\frac{J_{demand} - J_{current}}{|J_{demand} - J_{current}|}) \times V_{max}, \ t_{blend} \leq t < \ t_{max} - t_{blend} \\ (\frac{J_{demand} - J_{current}}{|J_{demand} - J_{current}|}) \times V_{max} \times (1 - \frac{(t-(t_{max}-t_{blend}))}{t_{blend}}), t_{max} - t_{blend} \leq \ t < t_{max} \end{cases} \quad (18)$$

where $J_{initial}, J_{demand}, J_{current},$ are the initial joint position, joint demand and current joint position respectively; angular displacement as:

$$s(t) = \int_{t_0}^{t_{max}} v(t)\, dt \text{ , as } t \to t_{\max} \text{ , } s(t) \to (J_{demand} - J_{initial}) \quad (19)$$

and angular acceleration as:

$$\alpha(t) = \frac{d(v(t))}{dt} \quad (20)$$

Below is an example that has been tested and verified on the physical robot, for which simulation matched the actual robot joint profile with the exact V_{max} and final position.

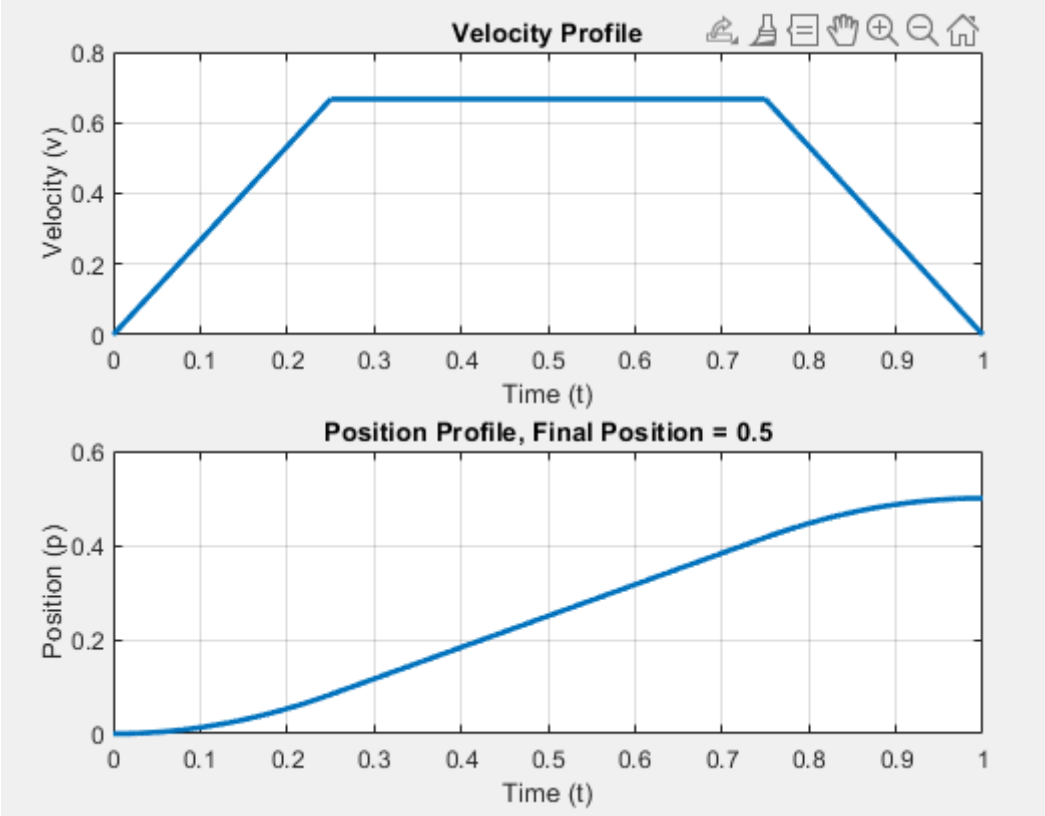

Figure 4. Trapezoidal velocity profile of a joint

Therefore, by tuning t_{blend} we can minimize jerk on the robot e.g. increasing t_{blend} reduces jerk.

3. Customized Inverse Kinematics Solver:

The inverse kinematics solver utilized from MATLAB by invoking '*ik*' command has proven accurate and satisfactory in the majority of the testing. However, the solver is not capable of handling joint angle constraints thus it is probable, the solution that the solver provides can result in rotations of the joints outside their allowable range which for the robot is defined $-\frac{\pi}{2} \leq \theta_i \leq \frac{\pi}{2}$ for angle joint θ_i. The inverse kinematics problem is casted as a constrained optimization problem with linear inequality constraints on the joint angles. For every homogenous transformation $\psi_{i,j,k^{th}}$ defined in (7), a set of joint angles is found that can achieve this position and orientation. The cost function is defined as follows:

$$J(u) = \sum_{i=1}^{6} w_i * (D_{i,k} - P(u)_{i,k})^2 \tag{21}$$

where u denotes the joint angles vector, w is the weights vector, D_k is $1x6$ vector that represents the desired position and orientation of end effector on the k^{th} waypoint and $P(u)_k$ is $1x6$ vector that represents the position and orientation of the end effector that results from u. In addition, the constraints are defined as:

$$\frac{-\pi}{2} \leq u \leq \frac{\pi}{2} \tag{22}$$

For a desired position and orientation $X = 0.1$, $Y = 0.1$, $Z = 0.2$, $A = 0$, $B = 1.5708$ and $C = 0$ the joint angles across the entire motion from '*ik*' and from solution of (21,22) are displayed below. As it can be seen from Figure 5 joint 3 exceeds the limit of $\frac{\pi}{2}$, whereas solution from our optimization does not violate any constraints and the end effector position is $X = 0.1029$, $Y = 0.1060$, $Z = 0.1998$, $A = -0.0154$, $B = 1.5686$ and $C = 0.0152$. However, the solution from (21,22) tends to come at the cost of increased computation and non-smooth transition from waypoint to waypoint as indicated in Figure 6.

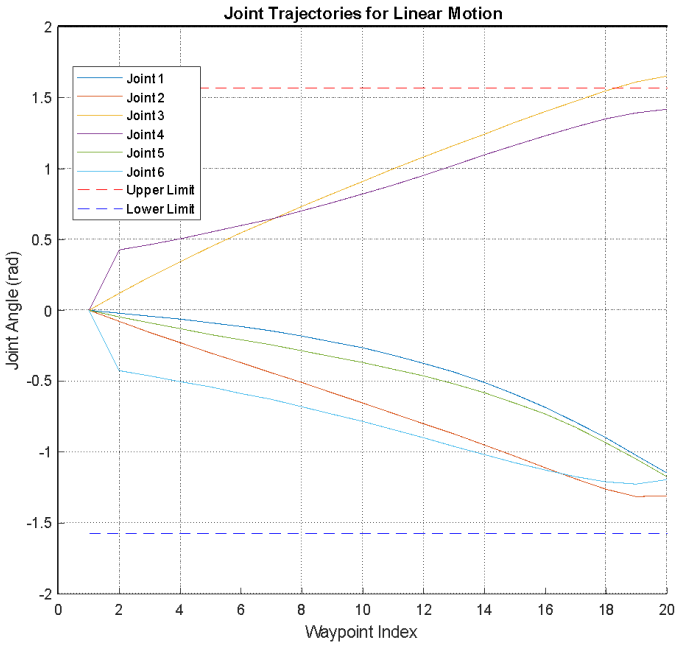

Figure 5. IK solver joint angle trajectory

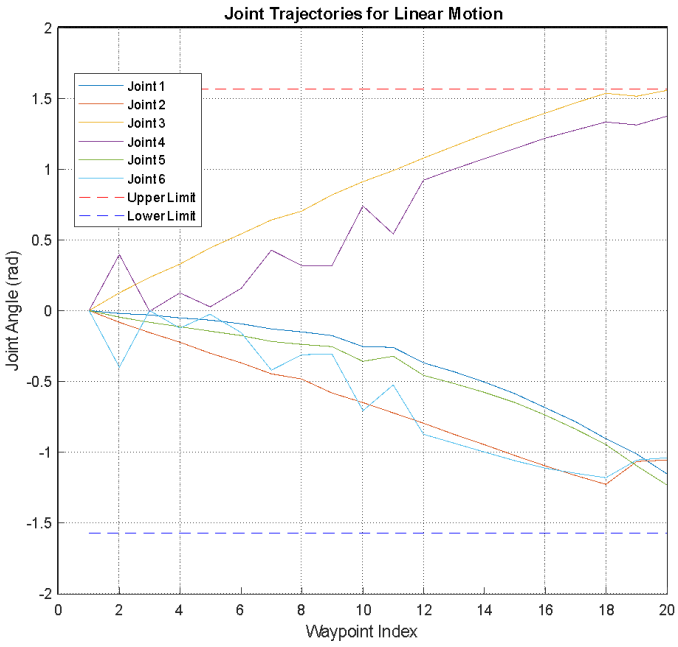

Figure 6. Custom optimization problem joint angle trajectory

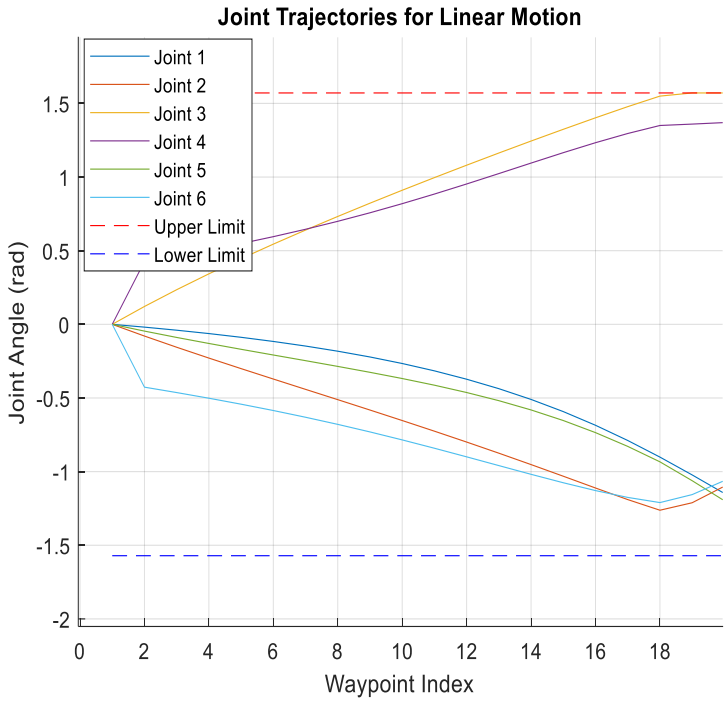

Figure 7. IK solver joint angle trajectory for adjusted URDF file

4. Forward kinematics:

$$H_1^0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & l1 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos(-\theta_1) & -\sin(-\theta_1) & 0 & 0 \\ sin(-\theta_1) & \cos(-\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{23}$$

$$H_2^1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & l2 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos(\theta_2) & 0 & sin(\theta_2) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta_2) & 0 & \cos(\theta_2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{24}$$

$$H_3^2 = \begin{bmatrix} \cos\left(\frac{\pi}{2}\right) & 0 & sin\left(\frac{\pi}{2}\right) & 0 \\ 0 & 1 & 0 & 0 \\ -sin\left(\frac{\pi}{2}\right) & 0 & \cos\left(\frac{\pi}{2}\right) & l3 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos(\theta_3) & 0 & sin(\theta_3) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta_3) & 0 & \cos(\theta_3) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{25}$$

$$H_4^3 = \begin{bmatrix} 1 & 0 & 0 & l4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 & 0 \\ sin(\theta_4) & \cos(\theta_4) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{26}$$

$$H_5^4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & l5 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos(\theta_5) & 0 & sin(\theta_5) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta_5) & 0 & \cos(\theta_5) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{27}$$

$$H_6^5 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & l6 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos(\theta_6) & -\sin(\theta_6) & 0 & 0 \\ sin(\theta_6) & \cos(\theta_6) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (28)$$

$$\therefore\ H_6^0 = H_1^0 * H_2^1 * H_3^2 * H_4^3 * H_5^4 * H_6^5 \quad (29)$$

$$Note\ , -\frac{\pi}{2} \leq \theta_i \leq \frac{\pi}{2}, \quad \forall\, i \in \{1,2,3,4,5,6\}. \quad (30)$$

The algebraic model of the arm and its comparison to the URDF [8] were verified, confirming that the analytically derived forward kinematics matched the actual solution from the URDF file.

5. System architecture:

5.1 High level System Architecture:

Steps:

1) The desired end effector pose is received from the user in MATLAB.
2) The current robot configuration is retrieved, forward kinematics are applied, and the initial pose, denoted as Pose A, is calculated.
3) The specific motion profile to be executed by the robot is selected.
4) The required calculations are then performed as defined in Section 2.
5) A validation check is carried out, as outlined in Section 6.
6) If the trajectory is validated, the joint configurations are sent to the C++ joint controller. If the trajectory is not approved, a warning is issued, the program is terminated, and the user is prompted to provide a valid location within the reachable workspace.
7) The motion is performed on the physical robot.

Figure 8 illustrates a visual representation of the architecture.

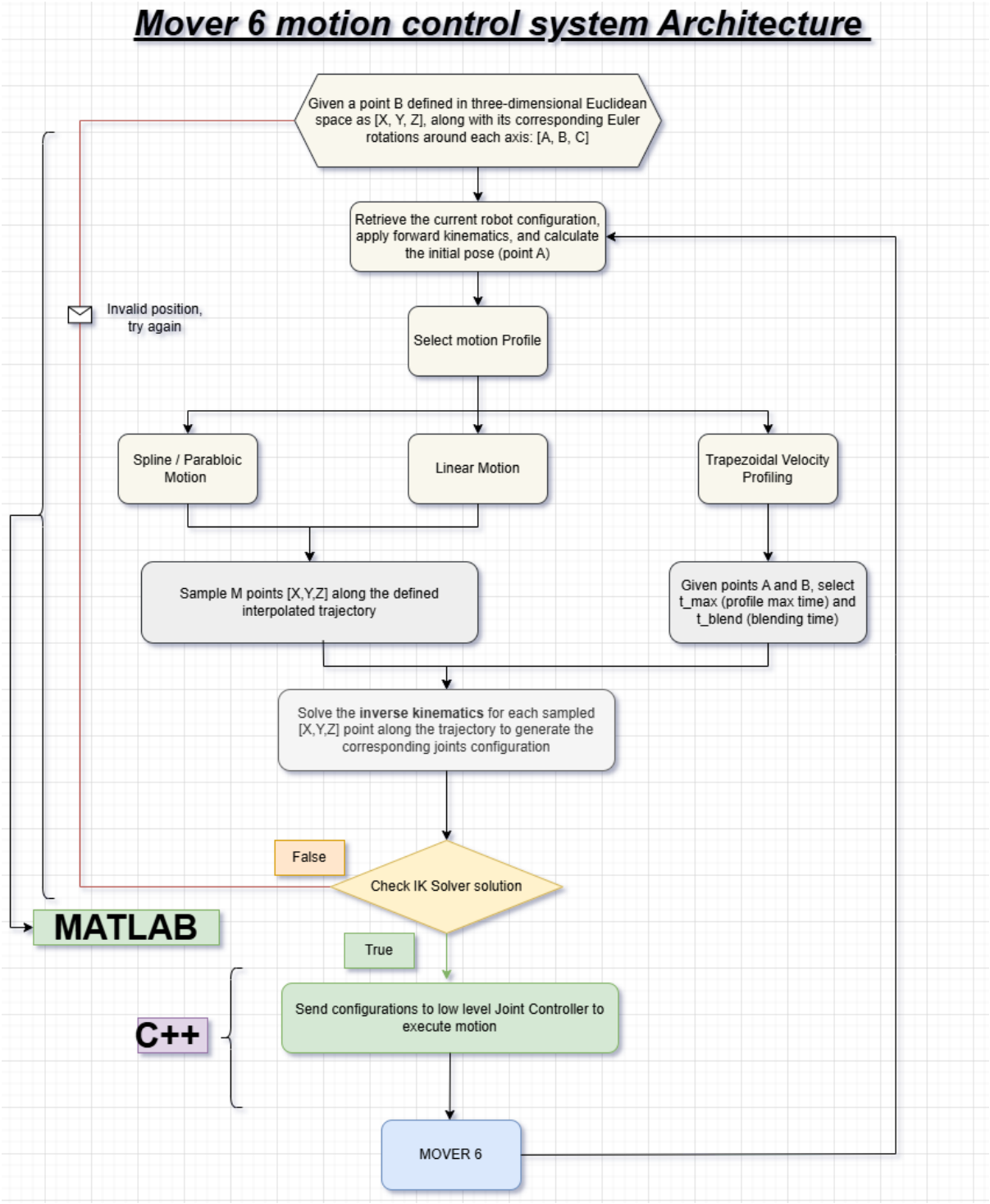

Figure 8. System Architecture

5.2 Software Architecture:

5.2.1 MATLAB:

As illustrated in Figure 8, the following steps are carried out:

MATLAB handles steps 1 to 6 and sends the joint configurations of the trajectory to the C++ joint controller as a 32-bit float vector of size N by 6. The C++ code then reconstructs the original matrix from this vector.

A critical design decision was made regarding whether to send each Nth joint configuration, wait for the motion to complete, and then sequentially send the configurations to the C++ controller, or to directly send an N by 6 vector and reconstruct the matrix in C++. It was determined that sending the complete N by 6 vector was more efficient and optimal. Although the /joint_state topic in MATLAB was subscribed to in order to obtain the initial position, this approach performed poorly during the execution of the full motion. Reading the joint states to ensure all joints converged to the desired solution before sending the next configuration proved slow in MATLAB, often resulting in an empty array when reading quickly and introducing latency. As a result, the decision was made to send the entire N by 6 vector to the C++ controller, where the matrix was reconstructed, allowing the entire motion to be executed there. This approach was significantly faster and did not encounter errors when reading states.

5.2.2 C++:

The system employs modern software architectures and best practices by breaking it into modular blocks with common features. Object-oriented programming (OOP) principles in C++ are utilized to optimize the codebase, as illustrated in [3]. Custom C++ classes were developed, which could be further modularized into libraries. These include a *Joints* class and a *Robot* class, each with their respective attributes and methods, such as joint states, demand, relative velocity, and trapezoidal velocity. Additionally, Finite State Machine (FSM) architectures were implemented to manage transitions between different modes, significantly enhancing the program's efficiency and optimization. For instance, defining a robot with 1000 joints requires modifying only a single line of code, ensuring that the system continues to operate correctly.

As a result, **ZERO GLOBAL VARIABLES** were utilized.

5.2.3 ROS architecture:

The system publishes to the /joint_demands topic using a standard Float32MultiArray message and subscribes to the /joint_states topic using a JointState sensor message. Additionally, a control message is utilized to publish to the mover6 /JointJog topic in order to actuate the robot.

The system also employs **boost::bind** to bind the robot object to the subscriber callback functions, allowing direct access and modification of the robot's state. The **boost::bind** function is utilized during the initialization of the **ros::Subscriber**, enabling both the message and the robot object (passed by reference) to be forwarded to the callback functions (**jointsCallback** and **Multi_Array_CB**). This approach ensures that the subscriber functions receive both the sensor data and the relevant robot instance for processing, allowing the joint states and demands to be updated within the Robot object as messages are received from the ROS topics. By adopting this method, the need for global variables is eliminated, as all data is encapsulated within the Robot object and explicitly passed to the callback functions, thereby maintaining a clean and modular structure for the code.

6. Joint controller validation and testing:

Validation:

Multiple MATLAB simulations were conducted, and the results aligned with expectations. Additionally, the actual joint states were plotted alongside the predicted trajectory, and the deviation consistently remained within the specified tolerance.

All errors are handled appropriately, with points outside the reachable workspace being rejected as follows:

Based on the ROS URDF wiki [2], the URDF file was updated as described in [10] by redefining the joint types from continuous to revolute. The required joint upper and lower limits were then added as per (30) .In reality the joint limits differ—for instance, Joint 1 ranges from -129.8° to 129.8°. However, for the purposes of this project, all joints were constrained within the limits mentioned above.

This turned out to be a pivotal step, as it ensured that the MATLAB inverse kinematics solver now respects the joint limits. As a result, the solver always outputs a constrained and valid solution within the reachable workspace, adhering to the limits defined in the updated URDF file.

If a point is outside the reachable workspace, a warning is triggered, and the program is terminated. A message is then displayed to the user, prompting them to provide a valid position within the reachable workspace.

As illustrated in Figure 5, this step is fundamentally critical. In this example, the inverse kinematics (IK) solver provided an invalid solution that violated the joint limits, even though the point was reachable within the workspace. By constraining the IK solver as described above, we ensured an optimal solution with no uncertainties, as shown in Figure 7.

Testing:

The system was deployed on the physical robot, and the real joint values were read in MATLAB. These values corresponded with the predicted motion trajectory, including linear, parabolic, and spline profiles.

For the trapezoidal velocity profile, we also verified that the actual maximum velocity V_{max} sent to the robot matched the predicted value. Additionally, the integration of the velocity profile resulted in the robot accurately reaching the desired joint location.

7. Conclusion:

Robust linear motion, parabolic, spline, and trapezoidal velocity profiling of the joints were successfully achieved using the software architecture defined above. Furthermore, methods were developed that generalize to any nonlinear trajectory defined in a three-dimensional Euclidean space. The inverse kinematics solutions were ensured to be valid and within the reachable workspace, as illustrated in Section 6 and exemplified in Figure 7. This approach resulted in a robust and generalizable robot joint controller.

Link to the videos: Robotics Assignment - Google Drive.

8. References:

[1] Linear Motion | NSK Automation

[2] ROS URDF wiki, https://wiki.ros.org/urdf/XML/joint.

[3] C++ Code.

[4] MATLAB Code Linear motion.

[5] MATLAB code parabolic profile example.

[6] Forward kinematics analytical solution verification.

[7] MATLAB code spline trajectory.

[8] Constrained URDF file.

9 Appendix

9.1 C++ Code:

```
/*
Title: MOVER6 Joint Controller
Date: 10/17/2024
Authors: Elion Selko & Ali Mohamed
This C++ code implements a motion controller for the MOVER6 robot.
It accepts an N x 6 vector of waypoint joint configurations, provided either via a terminal window or
MATLAB.
The code reconstructs the N x 6 waypoint configuration matrix, sequentially calculates the required joint
velocities, and publishes them to the joint jog topic to actuate the robot.
The implementation follows an Object-Oriented Programming (OOP) architecture with distinct `Robot` and
`Joint` classes, each encapsulating their respective attributes and methods.
Various operational flags can be toggled, such as selecting between relative or trapezoidal velocity motion
profiles.
*/
#include <ros/ros.h>
#include "std_msgs/String.h"
#include "control_msgs/JointJog.h"
#include "std_msgs/MultiArrayLayout.h"
#include "sensor_msgs/JointState.h"
#include <std_msgs/Float32MultiArray.h>
#include <boost/bind.hpp>
#include <algorithm>
#include <chrono>
#include <string>
#include <sstream>
#include <iostream>
#include <vector>



using namespace std;

// JOINT CLASS
class Joint {
public:
    string Joint_Name;
    float Joint_C_Value = 0; // Joint current value
    float Joint_Demand = 0;
    float init_state = 0; // Joint initial state
    float Velocity = 0;

    float Sign() const
    {
        return (Joint_Demand - Joint_C_Value) / abs(Joint_Demand - Joint_C_Value);
    }

    // Trapezoidal Velocity Profiling
    float Trapezoidal_Velocity(std::chrono::time_point<std::chrono::steady_clock> t_start, float t_max,
float t_blend) const
    {
        float current_time = (std::chrono::duration<float>(std::chrono::steady_clock::now() -
t_start)).count();
        // Get current time
        float v_max = abs(Joint_Demand - init_state) / (t_max - t_blend);
        // cout << "Current Time: " << current_time << endl;
        ROS_INFO("Joint: %s, Current_T: %s, V_Max = %s,", Joint_Name.c_str(),
to_string(current_time).c_str(),       to_string(v_max).c_str());

        float t_acc = t_blend;  // Time for acceleration/deceleration

        // If current time is within the acceleration phase
        if ((current_time < t_acc)) {
            return Sign() * (v_max / t_acc) * current_time;  // Accelerate
        }
        // If current time is in the constant velocity phase
        else if (current_time < t_max - t_acc) {
            return Sign() * v_max;  // Constant velocity
        }
        // If current time is in the deceleration phase
        else if (current_time < t_max) {
            return Sign() * v_max * (1 - (current_time - (t_max - t_acc)) / t_acc);  // Decelerate
        }
        // After t_max, stop
        else {
            return 0;
        }
    }
};
```

```
// ROBOT CLASS
class Robot {
public:
    const int Num_of_Joints; // Number of Joints
    vector<Joint> Robot_Joints; // Vector of ROBOT JOINTS
    control_msgs::JointJog msg_start;
    std::stringstream ss;
    bool know_states = false;
    bool know_demands = false;
    float max_distance = 0;
    float lambda = 2; // Maximum velocity unit for normalization
    float t_max = 0; // time for furthest joint movement - for RELATIVE motion
    float Maximum_Time_T = 1; // Maximum time - for trapezodial velocity motion
    float T_Blend = Maximum_Time_T / 4; // Blending time
    bool Vel_P = true; // true for trapezoidal velocity profile and false for relative velocity
    std::chrono::time_point<std::chrono::steady_clock> Start_Time; // clock
    vector<vector<float>> joint_matrix; // Joints configuration matrix


    // Robot constructor
    Robot(int num_of_joints) : Num_of_Joints(num_of_joints), Robot_Joints(num_of_joints) {
        for (int i = 0; i < Num_of_Joints; ++i) {
            Robot_Joints[i].Joint_Name = "joint" + to_string(i + 1);
        }
        msg_start.duration = 5; // Duration default to function unless otherwise specified.
    }

    /*
    The Max_Time() function determines the maximum time needed for the joint that must travel the greatest
distance,
    based on the parameter lambda (as defined above).
    */
    float Max_Time() {
        max_distance = 0;
        for (const auto& joint : Robot_Joints) {
            float distance = abs(joint.Joint_Demand - joint.Joint_C_Value);
            if (distance > max_distance) { // Compare manually to find the max
                max_distance = distance;
            }
        }
        t_max = max_distance / lambda;
        ROS_INFO("Max time %s", to_string(t_max).c_str());
        return t_max; // Return the maximum distance
    }

    // Starts clock timer
    void Start_Timer() {
        Start_Time = std::chrono::steady_clock::now();
    }
    // Prints time
    void Print_Start_Time() {
        // Convert Start_Time to duration since epoch and print it as a float
        auto duration_since_epoch = Start_Time.time_since_epoch();
        float seconds = std::chrono::duration<float>(duration_since_epoch).count();
        cout << "Start time (seconds since epoch): " << seconds << " seconds" << endl;
    }

    // Method to print the joint demand matrix (joint_matrix)
    void PrintJointMatrix() const {
        ROS_INFO("Joint Demand Matrix:\n");
        for (const auto& row : joint_matrix) {
            for (float value : row) {
                ROS_INFO("%s", to_string(value).c_str());
            }
            cout << endl;
        }
    }


};

// Joints states callback function, updates the robot joint states.
void jointsCallback(const sensor_msgs::JointState::ConstPtr& msg, Robot* robot) {

    int i = 0;
    string joint_info = "";

    for (const auto& position : msg->position)
    {
        if (i < robot->Num_of_Joints)
        {
            robot->Robot_Joints[i].Joint_C_Value = position;
            joint_info += std::to_string(robot->Robot_Joints[i].Joint_C_Value);
            joint_info += "\t";
        }
        i++;
    }
```

```
    robot->know_states = true;
    ROS_INFO("Received State %s", joint_info.c_str());
}

/*
The Multi_Array_CB callback function processes the N x 6 joint configuration vector,
reconstructs the robot joint matrix for sequential processing,
and acknowledges the received demand values to initiate execution.
*/
void Multi_Array_CB(const std_msgs::Float32MultiArray::ConstPtr& msg, Robot* robot) {
    int i = 0;  // Index for traversing the flattened demand array
    int set_count = 0; // Track how many sets of 6 values we've processed
    string joint_info = "";
    /*  The following commented for loop can be utilized when MATLAB provides individual joint
configurations instead of an     N x 6 array.*/
    /*for (const auto& demand : msg->data)
    {
        if (i < robot->Num_of_Joints)
        {
            robot->Robot_Joints[i].Joint_Demand = demand;
            joint_info += std::to_string(robot->Robot_Joints[i].Joint_Demand);
            joint_info += "\t";
        }
        i++;
    }
    */

    // Iterate through the received flattened data
    for (size_t j = 0; j < msg->data.size(); ++j) {
        // Calculate the row (configuration) index and the column (joint) index
        int row = j / robot->Num_of_Joints;  // Each row corresponds to a joint configuration
        int col = j % robot->Num_of_Joints;  // Each column corresponds to a specific joint

        // If we have not yet reached a new row, initialize it
        if (row == robot->joint_matrix.size()) { // matrix size is zero originally
            robot->joint_matrix.push_back(vector<float>(robot->Num_of_Joints, 0.0f));  // Initialize new
row with 6 joints
        }

        // Store the demand in the appropriate joint of the current configuration
        robot->joint_matrix[row][col] = msg->data[j];

        // Update joint information for logging
        joint_info += std::to_string(msg->data[j]) + "\t";

        // If we've processed a full set (6 values), increment the set count
        if (col == robot->Num_of_Joints - 1) {
            set_count++;
        }
    }

    robot->know_demands = true; // Demands acknowledgment
    ROS_INFO("Received Demands %s", joint_info.c_str());
}


int main(int argc, char** argv) {

    Robot Mover6(6);  // Creates a Robot with 6 joints

    // ROS Initialization
    ros::init(argc, argv, "goal_movement_example");
    ros::NodeHandle n;

    /* Create publisher to attach to JointJog */
    ros::Publisher chatter_pub = n.advertise<control_msgs::JointJog>("/JointJog", 1);

    // Subscribe to joint states and joint demands, passing the Mover6 object to the callbacks
    ros::Subscriber chatter_sub = n.subscribe<sensor_msgs::JointState>("/joint_states", 1000,
boost::bind(&jointsCallback, _1, &Mover6));

    ros::Subscriber chatter_sub_2 = n.subscribe<std_msgs::Float32MultiArray>("/joint_demands",
1000,   boost::bind(&Multi_Array_CB, _1, &Mover6));

    ros::Rate loop_rate(10);

    ros::Duration(2.0).sleep();
    // Mover6.Max_Time(); // computes maximum time

    while (ros::ok()) {

        if (Mover6.know_states && Mover6.know_demands) {

            Mover6.Start_Timer();
            // Mover6.PrintJointMatrix();
            // ROS_INFO("Matrix Size: %s", to_string(Mover6.joint_matrix.size()).c_str());
            // sleep(10);
            for (int N = 0; N < Mover6.joint_matrix.size(); N++)
```

```
            {
                for (int p = 0; p < 6; p++)
                {
                    Mover6.Robot_Joints[p].init_state = Mover6.Robot_Joints[p].Joint_C_Value;
                }
                Mover6.Max_Time();
                bool all_within_tolerance = false; // All joints within tolerance check
                while (!all_within_tolerance)
                {
                    int l = 0;  // Counter
                    all_within_tolerance = true;
                    for (int j = 0; j < 4; j++)                                         // Control the
first 4 joints
                    {
                        Mover6.Robot_Joints[j].Joint_Demand = Mover6.joint_matrix[N][j];
                        const auto& joint = Mover6.Robot_Joints[j];
                        Mover6.msg_start.joint_names.clear();
                        Mover6.msg_start.velocities.clear();
                        ROS_INFO("Setting message");
                        Mover6.msg_start.joint_names.push_back(joint.Joint_Name);
                        float V = 0;
                        if (Mover6.Vel_P) // Trapezoid
                        {
                            V = joint.Trapezoidal_Velocity(Mover6.Start_Time, Mover6.Maximum_Time_T,
Mover6.T_Blend);
                        }
                        else // Relative motion
                        {
                            V = joint.Sign() * (abs(joint.Joint_Demand - joint.Joint_C_Value) /
Mover6.t_max);
                        }
                        ROS_INFO("Joint: %s, Joint_D: %s, Joint_S: %s, Velocity: %s, l:%s",
joint.Joint_Name.c_str(), to_string(joint.Joint_Demand).c_str(), to_string(joint.Joint_C_Value).c_str(),
to_string(V).c_str(), to_string(l).c_str());
                        // sleep(3);
                        Mover6.msg_start.velocities.push_back((abs(joint.Joint_Demand -
joint.Joint_C_Value) > 0.05) ? V : 0);
                        ROS_INFO("Sending message");
                        chatter_pub.publish(Mover6.msg_start);

                        if (abs(joint.Joint_C_Value - joint.Joint_Demand) < 0.05) {
                            l++;  // Increment the counter if the joint is within tolerance
                        }
                        else {
                            all_within_tolerance = false;  // If any joint is out of tolerance, set the
flag to false
                        }
                    }
                    // If all joints are within tolerance, break out of the loop
                    if (l >= 4) {
                        all_within_tolerance = true;  // All joints are within tolerance, set flag to true
                    }
                    ros::spinOnce();
                    loop_rate.sleep();
                }
            }
            Mover6.know_demands = false;
            Mover6.joint_matrix.clear();
        }
        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}
```

9.2 MATLAB Code:

Linear motion, inverse kinematics and communication with ROS:

```
% Date 17/12/2024
% Authors: Ali Mohamed and Elion Selko
% Code Description: This script takes a user defined end effector translation and
% rotation defined in X, Y, Z, A, B, C which determines the pose of end effector frame with
% respect to robot base frame.
% The end effector moves linearly between the initial point and the
% final point.
% The URDF file is equipped with joint limits, which ik solver needs to satisfy
% whenever an Inverse Kinematics problem is solved, otherwise the code
% breaks and throws an error.
% The algorithm is also equipped with a customized inverse kinematics
% solver which the user can use if they uncomment lines  [90-94] and [100-106]
% Additionally the user may establish a communication with ROS topics and
% publish joint conifgurations by uncommenting lines [40-46] and [129-135]
clear
```

```
clc
close all
lastwarn('');
rng default
% Position of end effector in meters
X = 0.445;
Y = 0;
Z = 0.4425;
% Pose of end effector in radians
A = 0;
B = 0;
C = 0;
N = 20; % Number of samples/ waypoints
%% Linear motion controller
% Establish connection with the desired topics
% rosinit
% pub = rospublisher('/joint_demands','std_msgs/Float32MultiArray');
% sub = rossubscriber("/joint_states", "sensor_msgs/JointState");
% Joint_Received = receive(sub,10);
% display(Joint_Received.Position);
% Initial_Joint_Config= struct('JointName', {"joint1", "joint2", "joint3", "joint4", "joint5", "joint6"},
'JointPosition', num2cell(Joint_Received.Position'));
% initial_Pos = getTransform(mover6, Initial_Joint_Config , 'link6');
mover6 = importrobot('CPMOVER6.urdf');
initial_Pos = getTransform(mover6,  mover6.homeConfiguration, 'link6');
% Define an inverse kinematics object and the optimization weighting
% parameters
ik = inverseKinematics('RigidBodyTree', mover6);
weights = [0.25 0.25 0.25 1 1 1];
x_int = round(initial_Pos(1, 4),4);
y_int = round(initial_Pos(2, 4),4);
z_int = round(initial_Pos(3, 4),4);
t = linspace(0, 1, N); % parameter t for interpolation
% Define points along a straight line between initial and final position
x_co = (1-t) * x_int + t*X;
y_co = (1-t) * y_int + t*Y;
z_co = (1-t) * z_int + t*Z;
points_matrix = [x_co; y_co; z_co]; % Waypoint matrix 3 X N
% Extract current rotation transformation matrix of the end effector and
% apply the rotation transformation matrix from rotation A,B,C around Z, Y
% and X axis respectively
Rotation_TF = rotm2tform(tform2rotm(initial_Pos)*eul2rotm([A,B,C]));
% Initiatize and preallocate memory for Linear Transformation Matrix and HTM
Linear_TF = zeros(4, 4, N);
H_T_M = zeros(4, 4, N);
for i = 1:N
Linear_TF(:,:,i) = [1 0 0 points_matrix(1, i);
                    0 1 0 points_matrix(2, i);
                    0 0 1 points_matrix(3, i);
                    0 0 0 1];
H_T_M(:, :, i) = Linear_TF(:,:,i) * Rotation_TF;  % homogeneous transformation matrix
end
initialguess = mover6.homeConfiguration;
% initialguess=Initial_Joint_Config;
Waypoints_Joints_Sol_M = zeros(N, 6); % Waypoints, joints solutions matrix
EndEffector_Realized_WayPoint=zeros(N,3); % Realized positions
% Set the opitmization terminal conditions for the solver fmincon as well as
% intiating a Multistart object for multiple solution searches
% options=optimoptions(@fmincon,'MaxFunctionEvaluations',1e6,'StepTolerance',...
%     1e-6,'ConstraintTolerance',1e-2);
% ms=MultiStart("FunctionTolerance",1e-5);
% u=[initialguess.JointPosition];
% weightsMS=[50 50 50 1 1 1];
for i = 1:N
  current_HTM = H_T_M(:, :, i);
  [config, solnInfo] = ik('link6', current_HTM, weights, initialguess); % Solve IK
  Waypoints_Joints_Sol_M(i,:) = [config.JointPosition];
  % CostF=@(u)Cost(u,current_HTM,weightsMS,mover6);
  % problem=createOptimProblem('fmincon','objective',CostF,'x0',u,...
  % 'options',options,...
  % 'ub',pi/2*ones(size([initialguess.JointPosition])),...
  % 'lb',-pi/2*ones(size([initialguess.JointPosition])));
  % [u,cost]=run(ms,problem,4);
  % Waypoints_Joints_Sol_M(i,:) = u;
  for j = 1:6
     initialguess(j).JointPosition =  Waypoints_Joints_Sol_M(i,j);
  end
 Tr(:,:,i) = getTransform(mover6,config,'link6');
 EndEffector_Realized_WayPoint(i,:)=round(Tr(1:3,end,i)',4);
end
Angles = tform2eul(Tr);
[message, warningId] = lastwarn;
Difference=(Tr(1:3,end,end)-H_T_M(1:3,end,end));
if ~isempty(message)
  error('Invalid pose: %s\nPlease Try another position in space !\n', message);
elseif (norm(Difference,2)>.05)
   fprintf("Error in reached X: %.4f meters\nError in reached Y: %.4f meters\nError in " + ...
       "reached Z: %.4f meters\n",Difference(1),Difference(2),Difference(3));
```

```
   error('Invalid pose: %s\nPlease Try another position in space !\n', message);
end
fprintf("Error in reached X: %.4f meters\nError in reached Y: %.4f meters\nError in " + ...
    "reached Z: %.4f meters\n",Difference(1),Difference(2),Difference(3));
% Publish all waypoints' robot joint angles as a single vector to
% '\joint_demands'
% Data=[];
% for n=1:N
%     Data=[Data,Waypoints_Joints_Sol_M(n,:)];
% end
% msg = rosmessage(pub);
% msg.Data = Data;
% send(pub,msg);
%% Plotting
% Joint angle trajectory
figure;
hold on;
colours=[{'r'},{'g'},{'b'},{'y'},{'c'},{'k'}];
DisplayName=[];
jointsAm=6;
for jointIdx = 1:jointsAm
   plot(Waypoints_Joints_Sol_M(:, jointIdx),[colours{jointIdx}], 'DisplayName', sprintf('Desired Joint %d',
jointIdx));
end
hold on
plot([1 N],[1.571 1.571],'r--','DisplayName','Joint Upper Limit')
plot([1 N],[-1.571 -1.571],'r--','DisplayName','Joint Lower Limit')
xticks(1:N)
hold off;
legend('show','Location','northwest');
xlabel('Waypoint Index');
ylabel('Joint Angle (rad)');
title('Joint Trajectories for Linear Motion');
grid on;
% The predicted linear trajectory and the realized trajectory from solving
% inverse kinematics
figure;
plot3([x_int, X], [y_int, Y], [z_int, Z], 'b-', 'LineWidth', 1.5); % Line
hold on;
plot3(x_co, y_co, z_co, 'ro', 'MarkerSize', 8, 'MarkerFaceColor', 'r'); % Endpoints
plot3(EndEffector_Realized_WayPoint(:,1),EndEffector_Realized_WayPoint(:,2),EndEffector_Realized_WayPoint(:
,3),'go', 'MarkerSize', 8, 'MarkerFaceColor', 'g'); % Endpoints
grid on;
legend("Line","Desired Waypoints","Realized Waypoints")
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
%% Define the objective function
function [J]=Cost(u,endEffector,weights,mover6)
Joint_State=struct('JointName', {"joint1", "joint2", "joint3", "joint4", "joint5", "joint6"},
'JointPosition', num2cell(u));
Transform=getTransform(mover6,Joint_State,'link6');
% J=InverseKinematics_mex(Transform,endEffector,weights); % User may choose
% to run an object file, building an object file can be done using MATLAB Coder
J=InverseKinematics(Transform,endEffector,weights);
end
function [J,c,ceq]=InverseKinematics(Transform,endEffector,weights)
   AnglesEnd=tform2eul(endEffector);
   TrEnd=tform2trvec(endEffector);
   AnglesTransf=tform2eul(Transform);
   TrTransf=tform2trvec(Transform);
   J=sum(([TrEnd,AnglesEnd]-[TrTransf, AnglesTransf]).^2.*weights);
   ceq=[];
   A=[reshape(AnglesEnd,[],1)-pi/2;reshape(-AnglesEnd,[],1)-pi/2];
   c = A;
end
```

Parabolic profile:

```
%%%
% PARABOLIC MOTION PROFILE EXAMPLE
%%%


% Define the points A and B
A = [0.5, 0.2, 0];  % Starting point A (x_A, y_A, z_A)
B = [0.9, 1, 2.0];  % Ending point B (x_B, y_B, z_B)

% Compute the midpoint M between A and B
M = (A + B) / 2;

% Define a peak offset to lift the midpoint above the line
z_peak_offset = 2;  % You can adjust this value to increase or decrease the peak height

% Adjust the z-coordinate of the midpoint to add the peak
M(3) = M(3) + z_peak_offset;
```

```
% Find the coefficients for the parabolas in x, y, and z
[a, b, c] = find_parabola_coefficients(A, B, M);

% Parameter t from 0 to 1
t = linspace(0, 1, 100);  % Interpolation parameter t

% Compute the parabolic trajectory for each axis
x_trajectory = a(1) * t.^2 + b(1) * t + c(1);
y_trajectory = a(2) * t.^2 + b(2) * t + c(2);
z_trajectory = a(3) * t.^2 + b(3) * t + c(3);

% Plot the result
figure;
plot3(x_trajectory, y_trajectory, z_trajectory, 'r-', 'LineWidth', 2); hold on;
scatter3(A(1), A(2), A(3), 'go', 'filled'); % Starting point
scatter3(B(1), B(2), B(3), 'bo', 'filled'); % Ending point
scatter3(M(1), M(2), M(3), 'mo', 'filled'); % Midpoint (peak)
grid on;
xlabel('X'); ylabel('Y'); zlabel('Z');
title('Parabolic Trajectory from A to B with Peak');
legend('Trajectory', 'Point A', 'Point B', 'Midpoint (Peak)');

% Function to compute the coefficients of the parabola for each axis
function [a, b, c] = find_parabola_coefficients(A, B, M)
    % A = [x_A, y_A, z_A], B = [x_B, y_B, z_B], M = [x_M, y_M, z_M]
    % Equation system: Ax = [a, b, c], solve Ax = b

    % Solve the system for each coordinate axis
    % Form the matrix and right-hand side
    A_matrix = [0, 0, 1; 1, 1, 1; 0.25, 0.5, 1];  % For x, y, and z axis
    b_vector = [A(1); B(1); M(1)];  % Using x-coordinate values
    coeff_x = A_matrix \ b_vector;

    b_vector = [A(2); B(2); M(2)];  % Using y-coordinate values
    coeff_y = A_matrix \ b_vector;

    b_vector = [A(3); B(3); M(3)];  % Using z-coordinate values
    coeff_z = A_matrix \ b_vector;

    % Output coefficients
    a = [coeff_x(1), coeff_y(1), coeff_z(1)];
    b = [coeff_x(2), coeff_y(2), coeff_z(2)];
    c = [coeff_x(3), coeff_y(3), coeff_z(3)];
end
```

Forward kinematics Analytical solution verification:

```
% Load the URDF file for validation
clc; clear; close all;

% Analytical Forward Kinematics
syms theta1 theta2 theta3 theta4 theta5 theta6
syms l1 l2 l3 l4 l5 l6

% Analytical Transformation Matrices
H01 = [cos(-theta1), -sin(-theta1), 0, 0;
       sin(-theta1),  cos(-theta1), 0, 0;
       0,             0,            1, l1;
       0,             0,            0, 1];

H12 = [cos(theta2), 0, sin(theta2), 0;
       0, 1, 0, 0;
       -sin(theta2), 0,  cos(theta2), l2;
       0, 0, 0, 1];

H23 = [cos(pi/2), 0, sin(pi/2), 0;
       0,         1, 0,         0;
       -sin(pi/2), 0, cos(pi/2), l3;
       0,         0, 0,         1] * ...
      [cos(theta3), 0, sin(theta3), 0;
      0,  1, 0, 0;
       -sin(theta3),0, cos(theta3), 0;
       0,            0,           0, 1];

H34 = [cos(theta4), -sin(theta4), 0, l4;
       sin(theta4),  cos(theta4), 0, 0;
       0,             0,            1, 0;
       0,             0,            0, 1];

H45 = [cos(theta5), 0, sin(theta5), 0;
      0,  1, 0, 0;
       -sin(theta5),0, cos(theta5), l5;
       0,            0,           0, 1];
```

```
H56 = [cos(theta6), -sin(theta6), 0, 0;
       sin(theta6),  cos(theta6), 0, 0;
       0,             0,          1, l6;
       0,             0,          0, 1];

% Overall Analytical Transformation
H06_analytical = simplify(H01 * H12 * H23 * H34 * H45 * H56);

disp('Analytical Forward Kinematics (H06):');
disp(H06_analytical);

% ------------------------------------------------------------------
% Load URDF and Compute Forward Kinematics
disp('Loading URDF...');
robot = importrobot('CPMOVER6.urdf');
showdetails(robot);

% Define Joint Configuration for Testing
jointAngles = [0.6, -pi/6, pi/8, -pi/6, -pi/4, pi/3]; % Example angles

% Compute Forward Kinematics using URDF
config = homeConfiguration(robot);
for i = 1:length(config)
    config(i).JointPosition = jointAngles(i);
end

% Get the Forward Kinematics Transform from Base to End Effector
T_URDF = getTransform(robot, config, 'link6', 'base_link');

disp('Forward Kinematics from URDF:');
disp(T_URDF);

% ------------------------------------------------------------------
% Validate Both Approaches
% Substitute Values into Analytical Solution
H06_analytical_eval = double(subs(H06_analytical, ...
    [theta1, theta2, theta3, theta4, theta5, theta6, l1, l2, l3, l4, l5, l6], ...
    [jointAngles, 0.130, 0.0625, 0.190, -0.06, 0.290, 0.055]));

% disp('Analytical Forward Kinematics for Joint Angles:');
% disp(H06_analytical_eval);

% Compare Results
difference = T_URDF - H06_analytical_eval;
disp('Difference Between URDF and Analytical Solutions:');
disp(difference);
```

Spline trajectory:

```
% Define points A and B in 3D space
A = [0, 0, 0]; % Example: A at (0, 0, 0)
B = [10, 5, 3]; % Example: B at (10, 5, 3)

% Define intermediate points for the spline (optional, for shaping the path)
x = [A(1), 2, 5, 8, B(1)];
y = [A(2), 1, 3, 4, B(2)];
z = [A(3), 0, 1, 2, B(3)];

% Generate fine points for the smooth curve using spline interpolation
t = 1:length(x); % Parameter for each point
t_fine = linspace(1, length(x), 100); % Fine parameterization for smoothness

% Interpolate x, y, and z separately
x_fine = spline(t, x, t_fine);
y_fine = spline(t, y, t_fine);
z_fine = spline(t, z, t_fine);

% Plot the spline path
figure;
plot3(x_fine, y_fine, z_fine, 'b-', 'LineWidth', 2); % Spline curve
hold on;
scatter3(x, y, z, 'ro', 'filled'); % Control points (A, B, and intermediates)
scatter3(A(1), A(2), A(3), 100, 'g', 'filled'); % Start point A
scatter3(B(1), B(2), B(3), 100, 'm', 'filled'); % End point B

% Add labels and grid
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
title('3D Spline Trajectory');
grid on;
axis equal;
legend('Spline Path', 'Control Points', 'Point A (Start)', 'Point B (End)');
view(3); % Set 3D view
```

Constrained URDF file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- ================================================================================== -->
<!-- |    This document was autogenerated by xacro from catkin_ws/src/cpr_robot/robots/CPRMover6.urdf.xacro
| -->
<!-- |    EDITING THIS FILE BY HAND IS NOT RECOMMENDED                                  | -->
<!-- ================================================================================== -->
<robot name="CPRMover6" xmlns:xacro="http://ros.org/wiki/xacro">
  <!-- URDF file for the Commonplace Robotics Mover4 robot arm -->
  <!-- Version 1.1 from Oct. 04th, 2016. -->
  <link name="base_link">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint0.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint0Coll.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </collision>
  </link>
  <link name="link1">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint1.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint1Coll.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </collision>
  </link>
  <link name="link2">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint2.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint2Coll.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </collision>
  </link>
  <link name="link3">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint3.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint3Coll.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </collision>
  </link>
  <link name="link4">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint4.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint4Coll.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </collision>
  </link>
  <link name="link5">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint5.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </visual>
```

```
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint5Coll.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </collision>
  </link>
  <link name="link6">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint6.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://cpr_robot/robots/CPRMover6/Joint6Coll.dae" scale="0.001 0.001 0.001"/>
      </geometry>
    </collision>
  </link>
  <joint name="joint1" type="revolute">
    <axis xyz="0 0 -1"/>
    <parent link="base_link"/>
    <child link="link1"/>
    <origin rpy="0 0 0" xyz="0 0 0.130"/>
    <limit effort="100" velocity="30" lower="-1.571" upper="1.571" />
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
  <joint name="joint2" type="revolute">
    <axis xyz="0 1 0"/>
    <parent link="link1"/>
    <child link="link2"/>
    <origin rpy="0 0 0" xyz="0 0 0.0625"/>
    <limit effort="100" velocity="30" lower="-1.571" upper="1.571" />
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
  <joint name="joint3" type="revolute">
    <axis xyz="0 1 0"/>
    <parent link="link2"/>
    <child link="link3"/>
    <origin rpy="0 1.57079632679 0" xyz="0 0 0.190"/>
    <limit effort="100" velocity="30" lower="-1.571" upper="1.571" />
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
  <joint name="joint4" type="revolute">
    <axis xyz="0 0 1"/>
    <parent link="link3"/>
    <child link="link4"/>
    <origin rpy="0 0 0" xyz="-0.06 0 0"/>
    <limit effort="100" velocity="30" lower="-1.571" upper="1.571"/>
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
  <joint name="joint5" type="revolute">
    <axis xyz="0 1 0"/>
    <parent link="link4"/>
    <child link="link5"/>
    <origin rpy="0 0 0" xyz="0 0 0.290"/>
    <limit effort="100" velocity="30" lower="-1.571" upper="1.571"/>
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
  <joint name="joint6" type="revolute">
    <axis xyz="0 0 1"/>
    <parent link="link5"/>
    <child link="link6"/>
    <origin rpy="0 0 0" xyz="0 0 0.055"/>
    <limit effort="100" velocity="30" lower="-1.571" upper="1.571"/>
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
</robot>
```