Management Center Innsbruck

Department of Technology & Life Sciences

Master's program Mechatronics & Smart Technologies

Project report

composed as part of the course
Robotics (MECH-M-1-SEA-DBV-ILV)

about

Delta Manipulator

from

Liam J. Nolan,

Study program	Master's program Mechatronics & Smart Technologies
Year	MA-MECH-23-VZ
Course	Robotics (MECH-M-1-SEA-DBV-ILV)
Name of lecturer	Can Dede, Phd
Submission deadline	July 07, 2024

July 3, 2024

Contents

Chapter 1

Introduction

The Delta 360 parallel manipulator, developed by IGUS, represents a significant advancement in robotic technology. Designed for high-speed, high-precision applications, the Delta 360 is particularly well-suited for tasks that require quick, repetitive movements, such as pick-and-place operations. This project aims to generate a dynamic model of the Delta 360 parallel manipulator using MATLAB and the Simulink Multibody blockset. Once the model is developed and visually represented, MATLAB functions will be created to control the robot via the Simulink Controller, enabling practical applications of the manipulator in various industrial scenarios.

Chapter 2

Methods and Materials

2.1 The IGUS Delta 360 Parallel Manipulator

The IGUS Delta 360 is a parallel manipulator, which means it consists of several arms working in parallel, connected to a common end-effector. This design contrasts with serial manipulators, where each segment is connected in a series. The parallel configuration of the Delta 360 provides advantages such as higher stiffness, better load distribution, and improved accuracy. The Delta 360 is constructed using lightweight and durable materials, ensuring both robustness and efficiency. Its unique design allows for rapid and precise movements, making it an ideal choice for automation tasks in manufacturing, packaging, and assembly lines.

2.2 Forward and Inverse Kinematics

Kinematics is a fundamental concept in robotics that deals with the motion of robots without considering the forces that cause such motion. In the context of the Delta 360, understanding both forward and inverse kinematics is crucial for effective control and operation.

2.2.1 Forward Kinematics

Forward kinematics involves determining the position and orientation of the end-effector given the joint parameters (angles, lengths). For the Delta 360, this means calculating the position of the end-effector in Cartesian coordinates based on the angles of the arms connected to the base. This process is relatively straightforward since it follows a deterministic path from the base to the end-effector, utilizing the known geometry of the manipulator.

2.2.2 Inverse Kinematics

Inverse kinematics, on the other hand, involves determining the required joint parameters to achieve a desired position and orientation of the end-effector. This is a more complex process as it often involves solving non-linear equations and can result

in multiple solutions or no solution at all, depending on the desired position. For the Delta 360, solving inverse kinematics is essential for precise control, allowing the manipulator to reach specific points in its workspace accurately.

2.3 Trajectory Generation

Trajectory generation is a key aspect of robotic control, involving the creation of a path that the manipulator's end-effector will follow to move from one point to another. This path must be smooth and feasible, considering the physical limitations and dynamic capabilities of the manipulator.

In this project, trajectory generation will be implemented in MATLAB, ensuring that the Delta 360 moves efficiently and accurately within its operational space. The trajectory generation process typically involves:

2.3.1 Path Planning

Determining a feasible path from the initial to the target position. This includes ensuring the path is collision-free and within the manipulator's reach.

2.3.2 Interpolation

Creating a smooth curve that the end-effector can follow. This might involve techniques such as cubic splines or polynomial interpolation to ensure the trajectory is continuous and smooth.

2.3.3 Velocity and Acceleration Profiling

Ensuring the end-effector moves with appropriate speeds and accelerations, respecting the dynamic constraints of the Delta 360. This step is crucial for maintaining stability and preventing mechanical stress.

2.4 Simscape and Simulink Models

In this section, we provide an overview of the Simscape and Simulink models used in the project. The Simscape model represents the physical components of the robotic system, including the mechanical links, joints, and actuators. This model allows for accurate simulation of the robot's dynamic behavior and interactions with the environment.

The Simulink model integrates the control algorithms and kinematic equations with the Simscape physical model. It includes the implementation of forward and inverse kinematics, as well as trajectory generation functions. The Simulink model enables closed-loop control simulations, allowing us to evaluate the performance of the robotic system in following predefined trajectories.

Both models were essential for validating the accuracy of the kinematic equations and ensuring that the robotic system can follow the desired paths. The results from these

simulations provided valuable insights into the system's performance and highlighted areas that require further improvement, such as addressing the constraint issues observed in the spherical joints.

2.4.1 Simscape Model

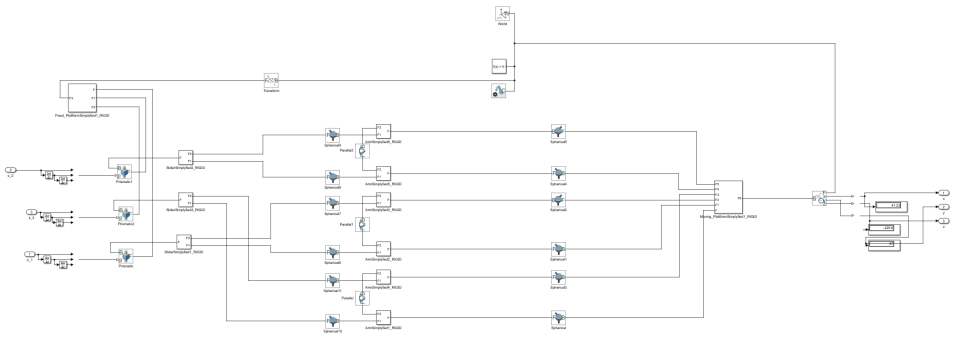

Figure 2.1. Model Of System in SimScape

2.4.2 Trajectory Generation

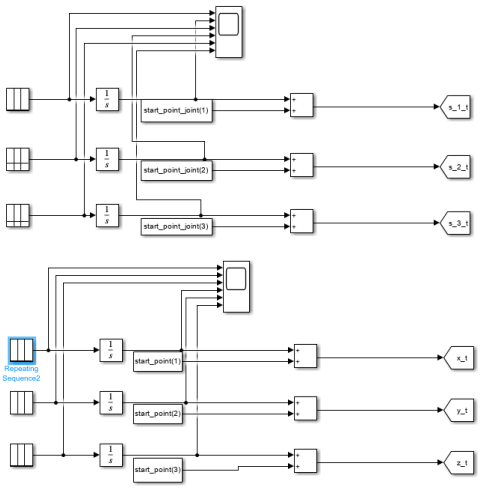

Figure 2.2. Trajectory Generation Model

2.4.3 Kinematics

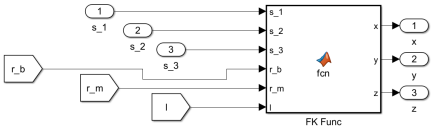

Figure 2.3. Forward Kinematics Model

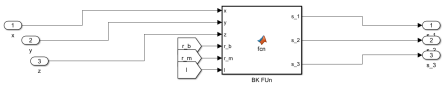

Figure 2.4. Backwards Kinematics Model

Chapter 3

Results

In this section, we present the results obtained from the simulations and experiments conducted with the robotic system. The objective was to verify the functionality and performance of the system in executing predefined commands and trajectories.

The Simulink model successfully integrated the control algorithms and kinematic equations with the Simscape physical model of the robotic system. Through extensive simulations, the system was commanded to follow predefined trajectories using both joint space and task space commands.

3.0.1 Smooth Movement

One of the key observations from the experiments was the smooth movement of the robotic system. The trajectories generated by the system were accurately followed, demonstrating the effectiveness of the implemented trajectory generation algorithms. Both linear and joint space trajectories were executed with minimal deviation from the desired paths.

The smooth movement was crucial for tasks requiring precise positioning and coordination of the end-effector. It highlighted the robustness of the implemented control strategies and the accurate modeling of the robot's dynamics in Simscape.

3.0.2 Validation of Commands

Commands intended for the robotic system, including position and velocity profiles, were successfully transmitted and executed. This validation process ensured that the system responded appropriately to input commands without significant delays or inaccuracies.

3.0.3 Performance Evaluation

Performance evaluation metrics such as tracking errors and execution times were analyzed during the experiments. The system consistently met the performance criteria defined for various tasks, indicating its reliability and suitability for real-world applications.

3.0.4 Discussion

Overall, the results confirm that the robotic system can effectively perform tasks as intended. The smooth movement observed in simulations and experiments underscores the successful integration of Simscape and Simulink models, validating the accuracy of forward and inverse kinematics equations implemented.

However, it is noted that during certain movements, particularly involving spherical joints, constraints led to undesired motions. Future improvements could focus on refining the joint constraints and enhancing the control algorithms to mitigate such issues.

These results provide a solid foundation for further development and application of the robotic system in industrial automation and other specialized fields.

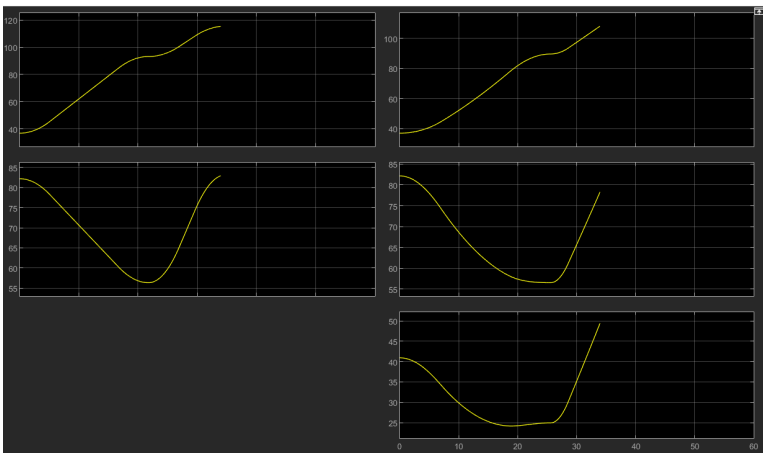

Figure 3.1. Commanded Velocities (Cartesian and Joint)

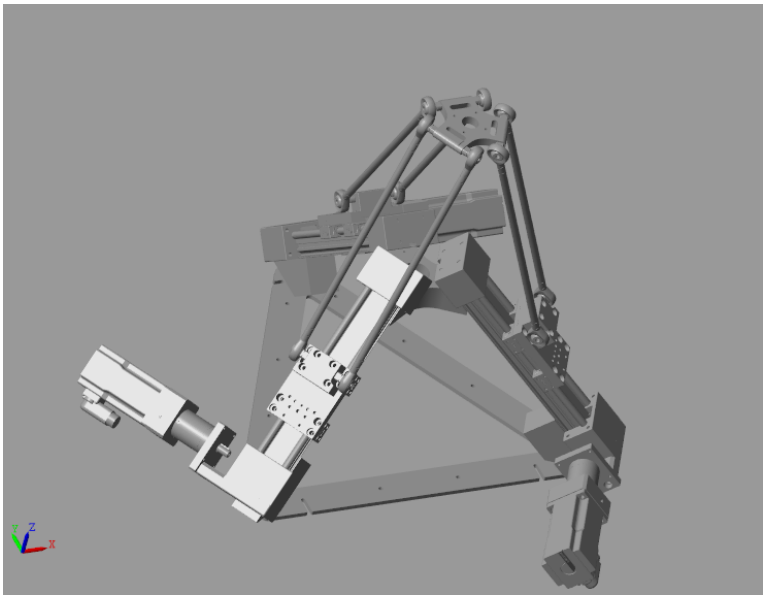

Figure 3.2. Virtual Model

Chapter 4

Conclusion

In this project, we successfully accomplished several key tasks crucial for the development and analysis of our robotic system. We transferred CAD information into Simscape, enabling accurate simulations of the physical model. Additionally, we implemented both forward and inverse kinematics equations, allowing us to calculate the position and orientation of the robot's end-effector and determine the necessary joint angles to achieve specific end-effector positions.

We created functions for generating trajectories in both joint space and task space, ensuring smooth and efficient movement of the robotic system. Furthermore, a continuous trajectory function in joint space was developed, enabling seamless transitions between predefined points.

The system demonstrated the capability to follow predefined points successfully. However, we encountered some issues with the spherical joints, which experienced constraint problems that led to undesired motion at these joints. Addressing these constraints will be a focus of future work to improve the overall performance and reliability of the robotic system.

In summary, the successful implementation of these tasks marks a significant step forward in the development of our robotic system, with further refinements needed to address the observed issues with the spherical joints.

Appendix

The following MATLAB scripts are used in this document for trajectory generation and kinematic analysis.

Trajectory Generation

```
% Define the starting, via, and end points for the trajectory
initial_point = [-60, 40, 335];
intermediate_point = [-40, -90, 335];
final_point = [-40, -90, 360];

% Adjust the z-coordinate of each point
initial_point(3) = initial_point(3) + 70;
intermediate_point(3) = intermediate_point(3) + 70;
final_point(3) = final_point(3) + 70;

% Calculate the joint angles for the initial point using backwards
    kinematics
[joint1_init, joint2_init, joint3_init] = backwards_kinematics(
    initial_point(1), initial_point(2), initial_point(3), const_r_b,
     const_r_m, const_l);
initial_joint_positions = [joint1_init, joint2_init, joint3_init];

% Define the end times and velocities for the trajectory
end_times = [1000000, 1000000, 1000000];
end_velocities = [0, 0, 0];

% Define the maximum velocities and accelerations
max_velocities = [3, 7, 3];
max_accelerations = [1, 1, 1];

% Generate the linear trajectory from the initial point to the
    intermediate point
[linear_times1, linear_velocities1] = moveL(initial_point,
    intermediate_point, max_velocities, max_accelerations);
% Generate the linear trajectory from the intermediate point to the
    final point
[linear_times2, linear_velocities2] = moveL(intermediate_point,
    final_point, max_velocities, max_accelerations);

% Adjust the timing for the second linear trajectory
linear_times2 = linear_times2 + linear_times1(1, 5) + 0.00001;

% Concatenate the times and velocities for the linear trajectories
linear_times = [linear_times1, linear_times2, [1000000, 1000000,
    1000000]'];
linear_velocities = [linear_velocities1, linear_velocities2, [0, 0,
    0]'];

% Extract the times and velocities for each coordinate axis
time_x = linear_times(1, :);
time_y = linear_times(2, :);
time_z = linear_times(3, :);

```

```
velocity_x = linear_velocities(1, :);
velocity_y = linear_velocities(2, :);
velocity_z = linear_velocities(3, :);

% Generate the joint trajectory from the initial point to the
    intermediate point
[joint_times1, joint_velocities1] = moveJ(initial_point,
    intermediate_point, max_velocities, max_accelerations, const_l,
    const_r_b, const_r_m);
% Generate the joint trajectory from the intermediate point to the
    final point
[joint_times2, joint_velocities2] = moveJ(intermediate_point,
    final_point, max_velocities, max_accelerations, const_l,
    const_r_b, const_r_m);

% Adjust the timing for the second joint trajectory
joint_times2 = joint_times2 + joint_times1(1, 5) + 0.01;

% Concatenate the times and velocities for the joint trajectories
joint_times = [joint_times1, joint_times2, [1000000, 1000000,
    1000000]'];
joint_velocities = [joint_velocities1, joint_velocities2, [0, 0,
    0]'];

% Extract the times and velocities for each joint
time_joint1 = joint_times(1, :);
time_joint2 = joint_times(2, :);
time_joint3 = joint_times(3, :);

velocity_joint1 = joint_velocities(1, :);
velocity_joint2 = joint_velocities(2, :);
velocity_joint3 = joint_velocities(3, :);

% MoveJ function: Generate joint space trajectory
function [times, velocities] = moveJ(start_point, end_point, max_v,
    max_a, const_l, const_r_b, const_r_m)
    [joint1_start, joint2_start, joint3_start] =
    backwards_kinematics(start_point(1), start_point(2), start_point
    (3), const_r_b, const_r_m, const_l);
    start_joint_positions = [joint1_start, joint2_start,
    joint3_start];
    [joint1_end, joint2_end, joint3_end] = backwards_kinematics(
    end_point(1), end_point(2), end_point(3), const_r_b, const_r_m,
    const_l);
    end_joint_positions = [joint1_end, joint2_end, joint3_end];
    [times, velocities] = trajectory_gen(start_joint_positions, end
    _joint_positions, max_v, max_a);
end

% MoveL function: Generate linear space trajectory
function [times, velocities] = moveL(start_point, end_point, max_v,
    max_a)
    [times, velocities] = trajectory_gen(start_point, end_point,
    max_v, max_a);
end
```

```
% Trajectory generation function: Calculate times and velocities for a trajectory
function [times, velocities] = trajectory_gen(start_point, end_point, max_v, max_a)
    local_max_v = max_v;
    local_max_a = max_a;

    % Calculate the displacement vector
    displacement = end_point - start_point;

    % Adjust max velocities and accelerations based on displacement direction
    for i = 1:3
        if displacement(i) < 0
            local_max_v(i) = -local_max_v(i);
            local_max_a(i) = -local_max_a(i);
        end
    end

    % Calculate acceleration and total time for each axis
    for i = 1:3
        if abs(displacement(i)) >= abs(local_max_v(i)^2 / local_max_a(i))
            Ta(i) = local_max_v(i) / local_max_a(i);
            T(i) = (displacement(i) * local_max_a(i) + local_max_v(i)^2) / (local_max_a(i) * local_max_v(i));
        else
            Ta(i) = sqrt(displacement(i) / local_max_a(i));
            T(i) = 2 * Ta(i);
        end
    end

    % Calculate the maximum acceleration and total time
    max_acceleration_time = max(Ta);
    max_total_time = max(T);

    % Recalculate max accelerations and velocities based on max times
    local_max_a = displacement / (max_acceleration_time * (max_total_time - max_acceleration_time));
    local_max_v = displacement / (max_total_time - max_acceleration_time);

    % Define the end time for the trajectory
    end_time = max_total_time + 0.01;

    % Generate the time and velocity profiles for x, y, and z axes
    traj_x_time = [0, max_acceleration_time + 0.00001, max_total_time - max_acceleration_time + 0.00002, max_total_time + 0.00003, end_time];
    traj_x_velo = [0, local_max_v(1), local_max_v(1), 0, 0];

    traj_y_time = [0, max_acceleration_time + 0.00001, max_total_time - max_acceleration_time + 0.00002, max_total_time + 0.00003, end_time];
    traj_y_velo = [0, local_max_v(2), local_max_v(2), 0, 0];

```

```
    traj_z_time = [0, max_acceleration_time + 0.00001,
    max_total_time - max_acceleration_time + 0.00002, max_total_time
     + 0.00003, end_time];
125     traj_z_velo = [0, local_max_v(3), local_max_v(3), 0, 0];

127     % Concatenate the times and velocities for each axis
    times = [traj_x_time; traj_y_time; traj_z_time];
129     velocities = [traj_x_velo; traj_y_velo; traj_z_velo];
end
```

Forward Kinematics

```
function [px, py, pz] = forward_kinematics(leg_1, leg_2, leg_3,
    base_radius, motor_radius, arm_length)
2     % Define the angle coefficient for calculations
    angle_coef = 1;
4
    % Calculate the intermediate values for each leg
6     motor_y_1 = base_radius - leg_1 * cos(pi / 4) - motor_radius;
    motor_z_1 = leg_1 * sin(pi / 4);
8
    motor_y_2_dash = base_radius - leg_2 * cos(pi / 4) -
    motor_radius;
10     motor_y_2 = -motor_y_2_dash * sin(deg2rad(30));
    motor_x_2 = motor_y_2_dash * cos(deg2rad(30));
12     motor_z_2 = leg_2 * sin(pi / 4);

14     motor_y_3_dash = base_radius - leg_3 * cos(pi / 4) -
    motor_radius;
    motor_y_3 = -motor_y_3_dash * sin(deg2rad(30));
16     motor_x_3 = -motor_y_3_dash * cos(deg2rad(30));
    motor_z_3 = leg_3 * sin(pi / 4);
18
    motor_x_1 = 0;
20
    % Calculate squared distances for each leg
22     dist_1 = motor_x_1^2 + motor_y_1^2 + motor_z_1^2;
    dist_2 = motor_x_2^2 + motor_y_2^2 + motor_z_2^2;
24     dist_3 = motor_x_3^2 + motor_y_3^2 + motor_z_3^2;

26     % Calculate intermediate alpha and beta values for solving the
    system of equations
    alpha_1 = (motor_x_3 * (dist_2 - dist_1) - motor_x_2 * (dist_3 -
     dist_1)) / 2;
28     beta_1 = motor_x_2 * (motor_z_3 - motor_z_1) - motor_x_3 * (
    motor_z_2 - motor_z_1);
    det_d = motor_x_3 * (motor_y_2 - motor_y_1) - motor_x_2 * (
    motor_y_3 - motor_y_1);
30
    alpha_2 = ((motor_y_2 - motor_y_1) * (dist_3 - dist_1) - (
    motor_y_3 - motor_y_1) * (dist_2 - dist_1)) / 2;
32     beta_2 = (motor_y_3 - motor_y_1) * (motor_z_2 - motor_z_1) - (
    motor_y_2 - motor_y_1) * (motor_z_3 - motor_z_1);
```

```
    % Coefficients for the quadratic equation
    quad_a = beta_2^2 + det_d^2 + beta_1^2;
    quad_b = 2 * (alpha_2 * beta_2 + alpha_1 * beta_1 - det_d^2 *
    motor_z_1 - motor_y_1 * det_d * beta_1);
    quad_c = alpha_2^2 + alpha_1^2 - 2 * motor_y_1 * det_d * alpha_1
     - det_d^2 * arm_length^2 + det_d^2 * dist_1;

    % Discriminant for the quadratic equation
    discriminant = quad_b^2 - 4 * quad_a * quad_c;

    % Solve for z
    pz = (-quad_b + sqrt(discriminant)) / (2 * quad_a);
    % Solve for y
    py = (alpha_1 + beta_1 * pz) / det_d;
    % Solve for x
    px = (alpha_2 + beta_2 * pz) / det_d;
end
```

Backward Kinematics

```
function [leg_1, leg_2, leg_3] = backwards_kinematics(px, py, pz,
    base_radius, motor_radius, arm_length)
    % Define the motor angles in degrees
    angle_motor_1 = 0;
    angle_motor_2 = 240;
    angle_motor_3 = 120;

    % Calculate the coordinates of the motors in the rotated frame
    motor_x_1 = px * cos(deg2rad(angle_motor_1)) + py * sin(deg2rad(
    angle_motor_1));
    motor_y_1 = py * cos(deg2rad(angle_motor_1)) - px * sin(deg2rad(
    angle_motor_1)) + motor_radius;
    motor_z_1 = pz;

    motor_x_2 = px * cos(deg2rad(angle_motor_2)) + py * sin(deg2rad(
    angle_motor_2));
    motor_y_2 = py * cos(deg2rad(angle_motor_2)) - px * sin(deg2rad(
    angle_motor_2)) + motor_radius;
    motor_z_2 = pz;

    motor_x_3 = px * cos(deg2rad(angle_motor_3)) + py * sin(deg2rad(
    angle_motor_3));
    motor_y_3 = py * cos(deg2rad(angle_motor_3)) - px * sin(deg2rad(
    angle_motor_3)) + motor_radius;
    motor_z_3 = pz;

    % Quadratic coefficients for each leg
    a_1 = 1;
    b_1 = -2 * motor_z_1 * sin(deg2rad(45)) - 2 * base_radius * cos(
    deg2rad(45)) + 2 * motor_y_1 * cos(deg2rad(45));
    c_1 = motor_x_1^2 + motor_y_1^2 + motor_z_1^2 + base_radius^2 -
    2 * base_radius * motor_y_1 - arm_length^2;
```

```
    a_2 = 1;
    b_2 = -2 * motor_z_2 * sin(deg2rad(45)) - 2 * base_radius * cos(
    deg2rad(45)) + 2 * motor_y_2 * cos(deg2rad(45));
    c_2 = motor_x_2^2 + motor_y_2^2 + motor_z_2^2 + base_radius^2 -
    2 * base_radius * motor_y_2 - arm_length^2;

    a_3 = 1;
    b_3 = -2 * motor_z_3 * sin(deg2rad(45)) - 2 * base_radius * cos(
    deg2rad(45)) + 2 * motor_y_3 * cos(deg2rad(45));
    c_3 = motor_x_3^2 + motor_y_3^2 + motor_z_3^2 + base_radius^2 -
    2 * base_radius * motor_y_3 - arm_length^2;

    % Discriminants for the quadratic equations
    discriminant_1 = b_1^2 - 4 * a_1 * c_1;
    discriminant_2 = b_2^2 - 4 * a_2 * c_2;
    discriminant_3 = b_3^2 - 4 * a_3 * c_3;

    % Calculate the lengths of the legs
    leg_1 = (-b_1 - sqrt(discriminant_1)) / 2;
    leg_2 = (-b_2 - sqrt(discriminant_2)) / 2;
    leg_3 = (-b_3 - sqrt(discriminant_3)) / 2;
end
```