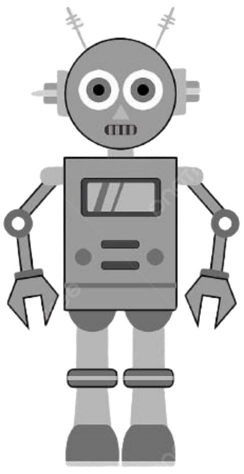
Modern control systems final project report

Self-balancing robot



Team members:

Name	ID
Andrew Botros Ayad	20201322493
Peter Hany Fayez	20221441026
Marina George Zarif	20221443430
Carine Emad Sanad	20221440878
Seveen Samir Wakim	2103117
Sandra Wassim Youssef	20221440830

I. Introduction:

- **Overview:**

  The self-balancing robot project aims to develop a smart robot that can stay upright on its own. It uses advanced sensors like a gyroscope and accelerometer to accurately measure the robot's tilt and acceleration. The robot adjusts its motor speeds in real-time using a control algorithm, a PID controller, to respond dynamically to changes in orientation and maintain stability.

- **Objectives and Goals:**

  The primary objective of this project is to achieve dynamic stability in a self-balancing robot, demonstrating the effective utilization of sensor data and real-time control mechanisms. The specific goals include:

  - Designing and constructing a stable chassis to support the self-balancing mechanism.
  - Integrating sensors, such as a gyroscope and accelerometer, for accurate measurement of tilt and acceleration.
  - Implementing a control algorithm, potentially a PID controller, to dynamically adjust motor speeds based on sensor feedback.
  - Fine-tuning the control algorithm for optimal performance and responsiveness.
  - Selecting and managing a power supply system to balance mobility and runtime efficiently.
  - Optionally incorporating features like obstacle avoidance mechanisms or user interfaces for remote control.

- Conducting thorough documentation and testing to gain comprehensive insights into robotics principles and practical skills in sensor integration, control systems, and autonomous motion.

II. <u>Components used:</u>

- **MPU6050 Sensor:**

  serves as the core of the system, functioning as a 6-axis motion-tracking device. It measures the robot's orientation, acceleration, and angular velocity in real-time.

- **PID Controller (using "PID_v1.h" library):**

  The control system of the robot employs a Proportional-Integral-Derivative (PID) controller. Utilizing the "PID_v1.h" library, the PID controller processes data from the MPU6050 sensor and dynamically adjusts the motor speeds to maintain the robot's balance.

- **DC Motors and PWM Control:**

  The robot's mobility is facilitated by DC motors controlled through Pulse Width Modulation (PWM) signals. The motor drivers interpret these signals, regulating motor speed and direction. Pin connections and motor speed parameters are specified in the code.

- **Kalman Filter:**

  A Kalman filter is implemented in the code to enhance the accuracy of sensor readings, particularly in filtering noise and minimizing deviations in yaw angle measurements. This contributes to improved stability in the robot's balancing behavior.

- **Microcontroller (Arduino Uno):**

  The overall system is governed by a microcontroller, typically an Arduino Uno. The code on the Arduino specifies the setup, configuration, and continuous monitoring of the robot's state. Interrupt-driven programming is employed, enabling the robot to respond promptly to changes in sensor data.

III. <u>Sensor calibration:</u>

- **Calibration Process:**

  - **Power-On Initialization:**

    Start by powering on the self-balancing robot and allowing all components, including the MPU6050 sensor, to stabilize.

  - **Sensor Initialization:**

    Initialize the MPU6050 sensor within the Arduino code to ensure proper communication and data retrieval.

  - **Collect Raw Sensor Data:**

    Begin collecting raw sensor data from the MPU6050, including readings related to acceleration, gyroscope, and orientation.

  - **Stationary Calibration:**

    Place the robot in a stationary position on a level surface. This ensures that the sensor captures readings when the robot is at rest and in an upright position.

- **Offset Identification:**

  Analyze the raw sensor data to identify any offsets or biases present in the readings. These offsets can arise from sensor imperfections or environmental factors.

- **Offset Compensation:**

  Implement offset compensation within the code by subtracting the identified offsets from the raw sensor readings. This adjustment helps correct for any systematic errors in the sensor data.

- **Fine-Tuning Parameters:**

  Fine-tune calibration parameters based on iterative testing. This may involve adjusting the offset values or incorporating scaling factors to align the sensor readings with the actual physical state of the robot.

- **Testing and Validation:**

  Conduct extensive testing to validate the effectiveness of the calibration process. This includes observing the robot's behavior in response to changes in orientation and ensuring that it maintains balance accurately.

o **Offsets or Adjustments Made:**

During the calibration process for the MPU6050 sensor in the self-balancing robot project, the following offsets or adjustments were identified and addressed:

- **Gyroscope Offsets:**

Offset values were determined for the X, Y, and Z axes of the gyroscope readings.

- **Accelerometer Offsets:**

Offsets for the X, Y, and Z axes of the accelerometer readings were identified.

Calibration involved subtracting these offsets from the accelerometer data to improve the accuracy of tilt measurements.

- **Additional Fine-Tuning:**

Iterative testing revealed the need for additional fine-tuning parameters to enhance calibration accuracy.

Small adjustments were made to scaling factors or other calibration parameters to optimize the sensor readings.

IV. Control system design

the self-balancing robot's control system is implemented through a Proportional-Integral-Derivative (PID) controller, designed to continually adjust motor speeds based on the difference between the desired and actual orientation. The three components of the PID controller—Proportional (P), Integral (I), and Derivative (D)—contribute to immediate error correction, elimination of steady-state errors, and damping effects to reduce overshooting, respectively.

The tuning parameters (Kp, Ki, Kd) play a critical role in achieving stability and responsiveness. Kp determines the immediate response strength, Ki addresses accumulated errors over time, and Kd introduces damping effects. The tuning process involves incrementally adjusting these parameters while observing the robot's behavior, aiming for a balance between stability and responsiveness.

In the Arduino code, the final tuned values of Kp, Ki, and Kd are integrated into the PID controller instantiation, providing the necessary parameters for the system to maintain the self-balancing robot's equilibrium effectively. This PID control system ensures the robot's stability with minimal oscillations and responsive correction to disturbances in real-time applications.

V. <u>**Kalman filtering:**</u>

In the self-balancing robot project, the Kalman filter plays a pivotal role in refining sensor data and enhancing the system's overall performance. This advanced filtering mechanism serves to reduce sensor noise, providing cleaner and more reliable data from both the accelerometer and gyroscope. By combining information from these sensors, the Kalman filter ensures improved accuracy in estimating the robot's orientation, effectively smoothing out erratic readings and minimizing deviations in yaw angle measurements. The chosen parameters for the Kalman filter, including process noise (Q), measurement noise (R), and initial estimate (P), have been thoughtfully selected to balance the filter's sensitivity to changes in the underlying process, the accuracy of sensor measurements, and the initial covariance of the estimation. This calibration contributes to the system's stability, precision in yaw angle measurements, adaptability to environmental changes, and the reduction of gyroscope drift, collectively enhancing the self-balancing robot's capability to maintain equilibrium in various conditions.

VI. Motor Control:

The self-balancing robot's motion is facilitated through precise control of its DC motors using the L298 motor driver. The L298 motor driver interprets signals from the PID controller and adjusts both the speed and direction of the motors, ensuring the robot maintains balance and responds effectively to changes in orientation.

- **Motor Speed and Direction Control:**
  The control of DC motor speed in the self-balancing robot relies on Pulse Width Modulation (PWM) signals, a technique that adjusts rotational speed by varying the voltage supplied to the motors. The Arduino code employs the analogWrite function to precisely modulate these PWM signals, enabling accurate control of motor speed. Additionally, direction control is facilitated through the L298 motor driver, which interprets digital signals from the Arduino to determine the movement direction of each motor. The code defines specific pins, such as in1, in2, in3, and in4, for controlling the direction of each motor. By adjusting the digital signals on these pins, the robot can move forward or reverse, providing comprehensive control over its movements.
- **Control Logic:**
  - **Forward Motion:**

    To move the self-balancing robot forward, a PWM signal is applied to both motors. This is achieved by setting one motor to rotate in the forward direction (in1 and in3 HIGH) and the other in the reverse direction (in2 and in4 HIGH).

    The differential speed between the two motors allows the robot to move forward while maintaining balance.

- **Reverse Motion:**

  Reversing the robot is accomplished by applying a PWM signal in the opposite direction. One motor rotates in reverse (in2 and in4 HIGH), while the other moves forward (in1 and in3 HIGH).

  Similar to forward motion, the differential speed ensures the robot moves backward without losing its equilibrium.

- **Stopping the Robot:**

  Bringing the robot to a stop is achieved by setting all motor control pins to LOW (analogWrite to 0).

  This halts the motors and allows the self-balancing robot to remain stationary.

o **Motor Speed Parameters:**

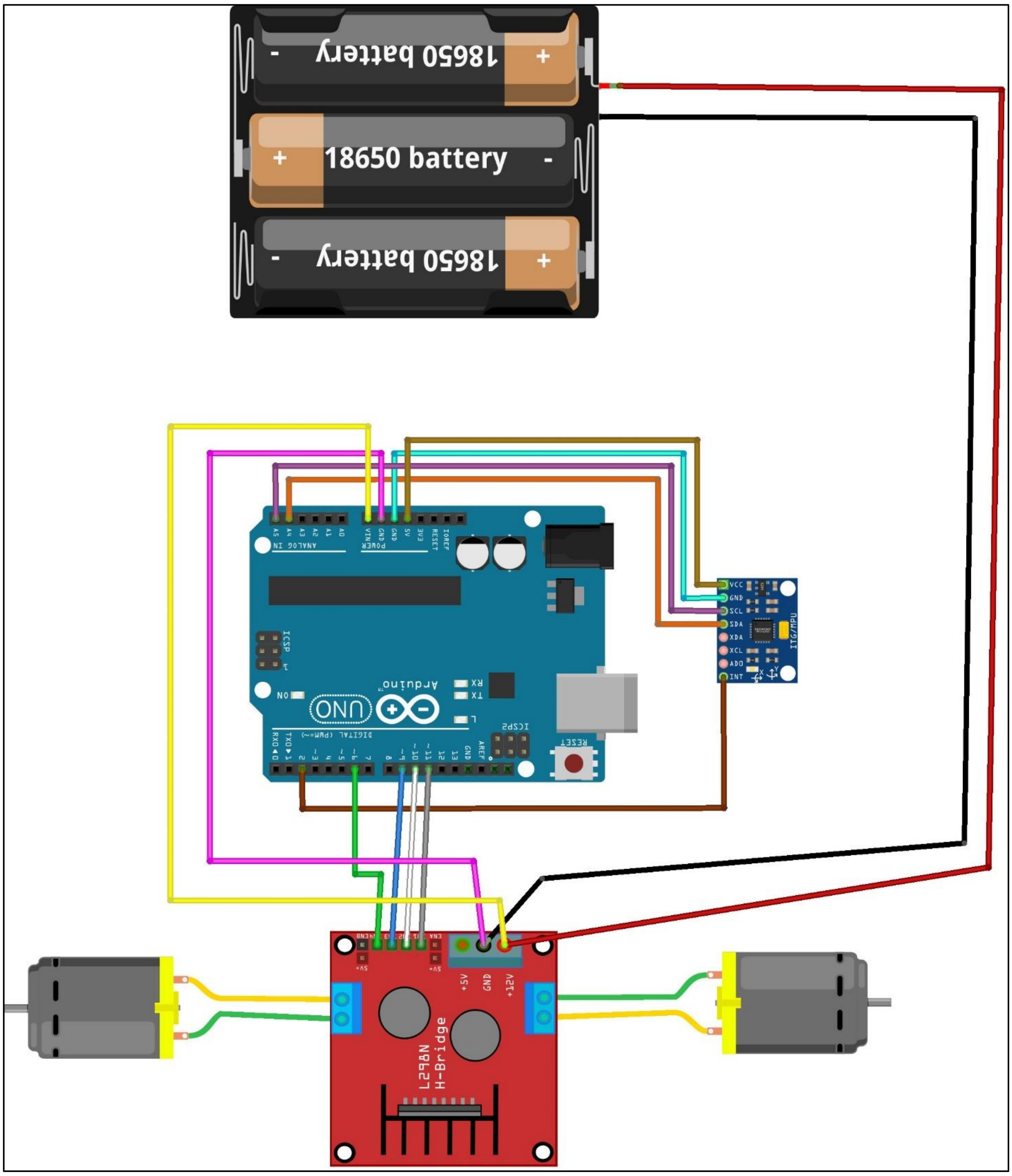
- The specific parameters for motor speed, such as motorSpeed, are defined in the code and can be adjusted based on the robot's dynamics and desired performance.

- The analogWrite function with appropriate speed values sets the PWM signal, influencing the rotational speed of the motors.

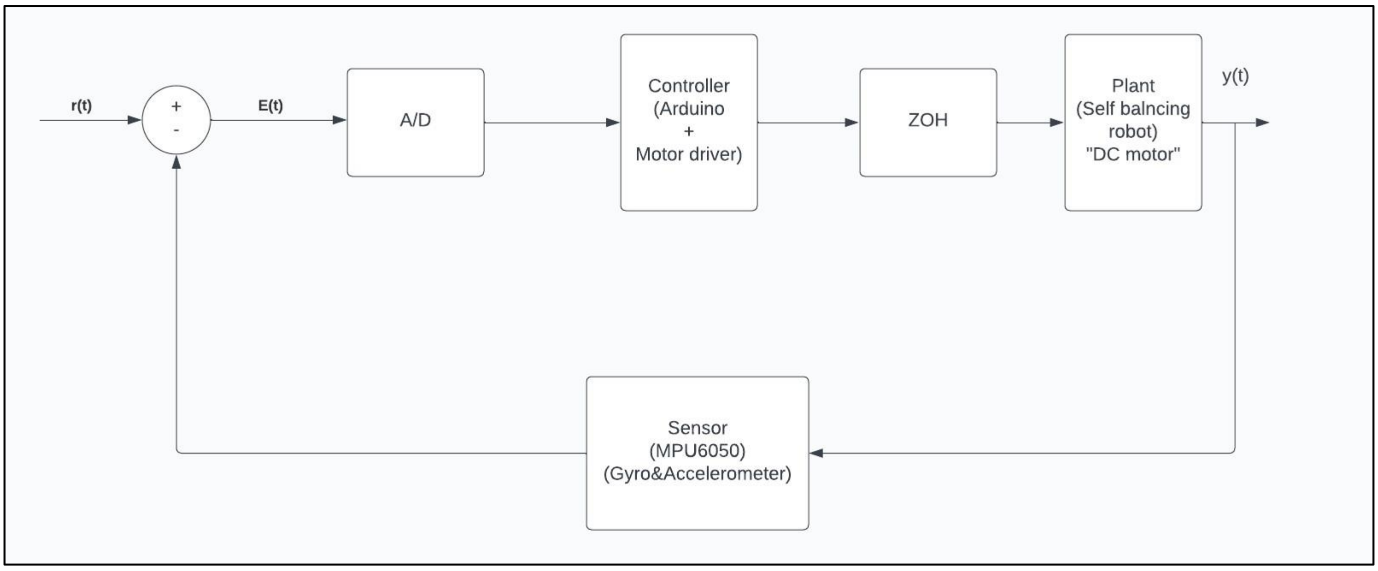
VII. Arduino Code explanation:

The provided code is designed for a self-balancing robot, leveraging an MPU6050 sensor for orientation measurement and implementing a PID controller to regulate motor speeds for balance maintenance. It employs essential libraries such as "I2Cdev.h," "PID_v1.h," and "MPU6050_6Axis_MotionApps20.h" to facilitate sensor communication, PID control, and MPU6050 functionalities. Global variables are defined for pin connections, motor speeds, and various parameters involved in the PID controller and sensor calibration. The setup function initiates serial communication for debugging, initializes the MPU6050 sensor with connection testing and calibration offsets, and activates the Digital Motion Processor (DMP) for orientation tracking. It configures the PID controller with setpoint, Kp, Ki, Kd values, establishes pin modes for motor control, and sets up the overall system. The loop function continuously checks for available sensor data, triggers PID computation when data is ready, and adjusts motor speeds based on the computed PID output, determining whether to move forward, backward, or stop the robot. Sensor data retrieval involves obtaining quaternion, gravity, yaw-pitch-roll angles from the MPU6050, with a Kalman filter applied to enhance accuracy and stability in yaw angle measurements. The motor control functions, namely Forward(), Reverse(), and Stop(), direct the motors by adjusting PWM signals to the motor driver. The overall flow involves initialization, including the setup of communication, sensors, DMP, and PID controller, followed by a loop that ensures continuous sensor data processing and PID-driven motor adjustments for maintaining balance. It's important to note that the code assumes specific motor configurations and may require adjustments based on the robot's hardware setup. Calibration and fine-tuning of PID parameters are recommended for optimal performance.

VIII. Circuit design:



IX. **Block diagram:**


r(t)
+
-
E(t)
A/D
Controller (Arduino + Motor driver)
ZOH
Plant (Self balncing robot) "DC motor"
y(t)
Sensor (MPU6050) (Gyro&Accelerometer)

X. MATLAB code for analysis:

• Code with explanation:

```
% Parameters from Arduino code
Kp = 21;
Ki = 140;
Kd = 0.8;
```

Here, the proportional gain (`Kp`), integral gain (`Ki`), and derivative gain (`Kd`) are defined. These are typically used in a PID (Proportional-Integral-Derivative) controller.

```
% Transfer function
numerator = [Kd, Kp, Ki];
denominator = [1, 0, 0];  % s^2 + 0s + 0
```

This section creates the transfer function of the system using the defined gains. The transfer function is represented as the ratio of two polynomials, where the numerator contains the coefficients of the derivative term (`Kd`), the proportional term (`Kp`), and the integral term (`Ki`). The denominator represents the characteristics of the system.

```
% Create the transfer function
sys = tf(numerator, denominator);
```

The `tf` function is used to create a transfer function (`sys`) with the given numerator and denominator polynomials.

```
% Step response
figure;
step(sys);
title('Step Response');
```

This section plots the step response of the system using the `step` function. The step response shows how the system behaves in response to a unit step input.

```
% Step response with reference tracking
ref_sys = tf(1, [1, 0]);  % Reference input: unit step
closed_loop_sys = feedback(series(sys, ref_sys), 1);
figure;
step(closed_loop_sys);
title('Step Response with Reference Tracking');
```

Here, a reference system (`ref_sys`) is defined to represent a unit step input. The closed-loop system is created by connecting the original system (`sys`) and the reference system in series using `series`, and then applying feedback with `feedback`. The step response of the closed-loop system is then plotted.

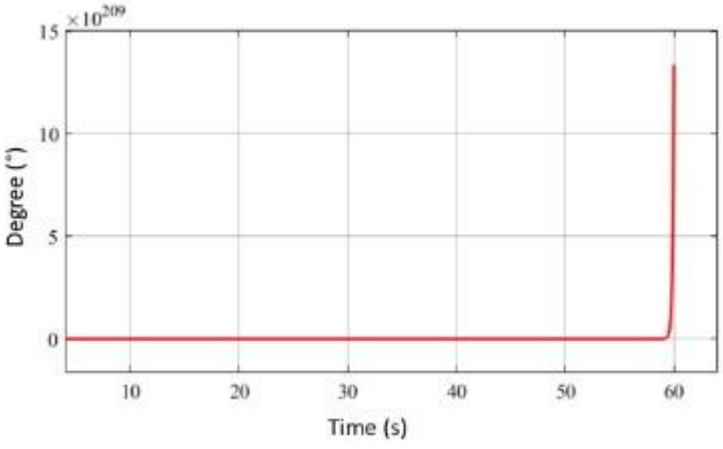
```
% Root locus
figure;
rlocus(sys);
title('Root Locus');
```

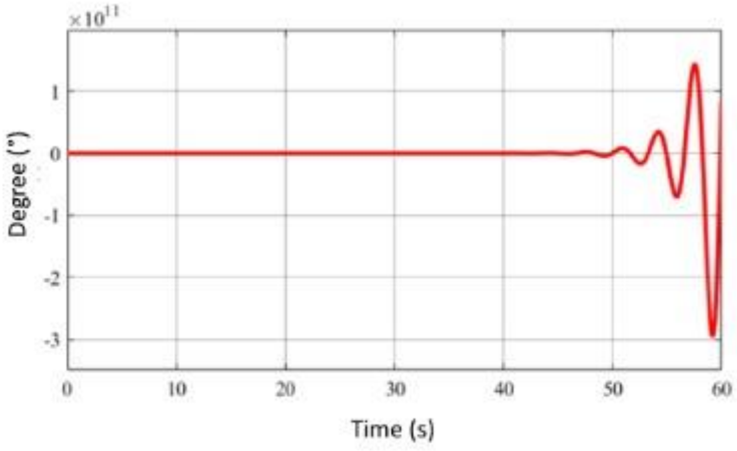
This section plots the root locus of the system using the `rlocus` function. The root locus shows how the poles of the system vary with a parameter, often the proportional gain (`Kp`), providing insights into the system's stability and behavior.

- **<u>Output:</u>**

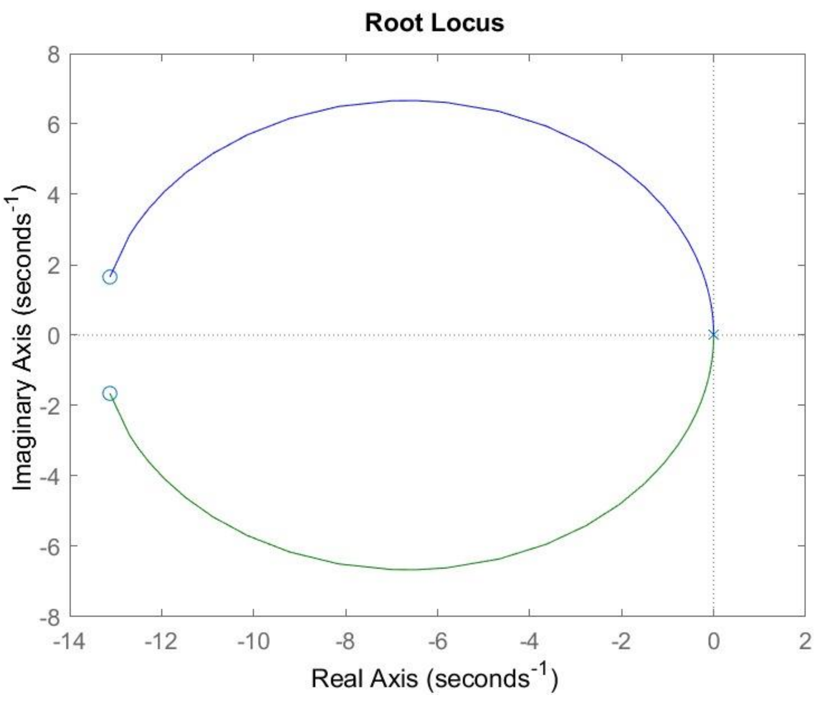
  - **<u>Step response:</u>**

$$\text{Transfer function} = \frac{.398s}{s^3 + .02s^2 - 214.85s - .39}$$





- **<u>Root locus:</u>**



Root Locus
Imaginary Axis (seconds^{-1})
Real Axis (seconds^{-1})
8
6
4
2
0
-2
-4
-6
-8
-14
-12
-10
-8
-6
-4
-2
0
2