Project 1-1

Phase 3: Final Report

Group 19

Ion Cristian Bandalac	i6425370	Code Lead
Eglen Vata	i6429690	Recorder
Callum Mc Kechnie	i6424525	Presentation Lead
Vito Martignano	i6402021	Report Lead
Nicon-David Milandru	i6430259	Team Leader
Tudor Ginguta	i6442826	Checker

BCS1300 - Project 1-1

January 2026

Maastricht University

Considerations

Detailed theoretical explanations, short methodological reports, with code snippets, experiment cases, references, and demonstration videos of the implemented algorithms and GUI are available both in the presentation and in the project's GitHub repository: https://github.com/qnicondavid/Self-Driving-Bot.

Methods and Motivation

Code Design

The backend is implemented using an object-oriented architecture, with dedicated classes such as: Microcontroller, Motor, IRSensor, URSensor, Robot or Request Handler.

The GUI is built entirely with JavaFX, following a Maven-style project structure and also an object-oriented design, with classes such as: SensorReader, MechanumVisualizer, ParametersPane or BotDataPane. The user can set parameters, view sensor readings, see the path display, monitor flags and speeds, toggle algorithms, or call actions for the robot to perform.

Directional Control

The robot can be controlled smoothly through the GUI using standard WASD inputs, including diagonal movement via key combinations. Additional controls allow clockwise and counter-clockwise rotation on the spot, with user-adjustable speed for all movement modes.

Data Display

The IR analog and digital sensor readings, the UR distance and emergency-stop flag (disabled when PID is not segmented), the four motor wheel values, and the connection flag are printed continuously in the GUI at a polling interval of 200 ms. This is different from sending requests to the robot: the loop function includes a print-handler that sends the data automatically, without being requested, and the GUI simply receives and displays it. This eliminates delay.

Emergency Stop with Hysteresis

The ultrasonic sensor triggers an emergency stop when obstacles are detected within a threshold distance. This blocks normal key movements. We implement hysteresis with two thresholds (stop at dStop, resume at dStop + Δd) to prevent oscillation when readings fluctuate near a single value. The user is able to change these values in the GUI.

Line Following

PID is a powerful choice for line following because it can be tuned experimentally, without requiring a precise mathematical model of the robot. A Proportional-Integral-Derivative (PID) controller was implemented to enable the robot to follow strips of black electrical tape using its analog infrared sensor readings. The control objective is to minimize the lateral deviation from the tape by continuously adjusting the wheel speeds. The proportional part reacts immediately, the integral part corrects long-term bias, and the derivative part stabilizes sudden changes. Anti-windup protection keeps the integral term from growing uncontrollably. The user can adjust the Kp, Ki, Kd parameters and the maximum speed directly from the GUI, which sends the corresponding requests to the robot. This makes PID tuning straightforward for every type of floor.

Path Display

The four motor speeds continuously sent by the robot are used to update the path display. By integrating these speeds, the GUI computes the new position and orientation of the small robot figure and visualizes its trajectory. The user can calibrate this display for the testing floor by adjusting in the GUI how much the figure rotates, how fast it moves along the drawn path, and the length of the displayed segments.

Reversing in a Straight Line

We implemented a reverse mode that the user can toggle on and off, which performs southward movement at the base speed. We also implemented a timed reverse mode, where the user can set the number of milliseconds for which the robot should move backwards.

Half-Turn & U-Turn

We implemented a turn function that makes small steps on the Y-axis and rotates. At the start of the function, we also have an uninterrupted Y-axis movement. By setting in the GUI the number of steps and the delay of the small Y-axis movements and rotation movements, as well as the time of the initial Y movement, we can perform any type of arc, for example a full U-turn or a circular 90-degree turn. The direction can also be set for each arc: forward or backward, clockwise or counter-clockwise. This idea of using steps came from comparing it to the integral algorithm for calculating curved areas: by splitting the curve into small rectangles, we split curved movements into small straight segments.

Three-Point Turn

The same logic applies for this maneuver, performing three arcs with different X and Y orientations.

Parking in a Box Forward

When this maneuver is called, the robot performs PID line following until a junction is found, then executes a small forward movement to get inside the box. The user can set the duration of this forward movement so the robot can park perfectly inside the box.

Line Following Emergency Stop

We observed that our PID algorithm drifts off the line when we perform continuous checks with the UR sensor. This happens because the ultrasonic waves require time to travel and return, causing delays that block several milliseconds of PID cycles. To fix this, we segmented the PID and emergency stop control. Every 200 ms we pause the PID, request a UR reading, and if the emergency stop logic triggers, we stop the manoeuvre, otherwise, we restart the PID.

Kidnapped Robot Problem

The scenario is that the robot is placed inside a box with tape edges. The line detection is simple: one of the digital IR readings needs to be true.

The first algorithm is the naive solution: the robot goes forward until it eventually reaches an edge, then starts the PID controller for a few seconds.

The second algorithm uses continuous circular movements. We implemented a circular movement as steps of small moves: rotate clockwise, then move right. The first circle is a turn on the spot. For every larger circle, we increase the number of steps. After each circle, we give the robot a small nudge forward to increase the circle radius. We keep verifying if the line is detected. The comparison between them is determined solely by the robot's starting position. If the front part of the robot is closer to an edge, the first algorithm will find the line faster, compared to scanning equally in every direction.

Maze Solving

First, we solved the dead ends. We used the segmented PID with emergency control to stop the robot when we encountered an object in a dead end. Then, we move the IR sensors off the line with a small clockwise movement. After that, we perform small clockwise turns on the spot, checking if the line is detected again. When it is found, we start the PID again and the dead end is solved.

The second step was detecting junctions. The intuitive idea was that a junction is detected when both digital IR readings are true. Using that approach, the robot stopped at 90-degree turns (which are not junctions) and sometimes skipped sideways T-junctions, due to continuous PID oscillations. The solution was to count the number of continuous double-digital-true readings while the PID is still adjusting on the line, and stop when the counter reaches a set value. By tuning this threshold, we can stop at any junction: the value needs to be lower to skip a 90-degree turn and higher to stop at a sideways T. After detecting the junction, we give the robot a small nudge forward to center it on the junction.

The first naive idea that came to mind was using a random-choices algorithm, because for something like this we do not need to know which type of junction we have, just stop at a junction. While centered in a junction, we start a clockwise turn on the spot for a random amount of time, then stop and perform the same dead-end recovery: small clockwise steps until the line is found again. When the line is detected, the PID adjusts the robot and a decision has been made. This algorithm solves mazes very slowly, but it is correct.

The next step was moving from one of the simplest algorithms to a faster one. We decided that for choices like left and right, the robot should perform small orientation adjustments and then use the dead-end recovery idea. For forward, we give the robot a small nudge so it exits the junction after being centered in it. By adding a series of choices to a queue, we can define a path. The initial idea was to take a picture of the maze, train an AI to detect the start and stop points, determine an optimal series of choices, and add them to the queue. For now, we add the choices manually in the backend.

Experiments and Results

All of the experiments were performed on a floor with a friction coefficient of 0.5, using black electrical tape. The results are shown in the videos mentioned earlier. The task that took the most time was calibrating the PID coefficients, starting with a 3 cm tape width and adjusting down to the standard 1.5 cm. All experiment cases can be found in the short reports mentioned above.

Findings and Discussion

All the required functions were implemented successfully. We could have finished the AI-based maze-detection algorithm, and also developed a method to identify the type of junction in order to implement other ideas such as Trémaux's algorithm, if we had had more time and clearer communication.

The highlights of our work are the code quality and structure, the GUI design, the coefficient-tuning interface, and, most importantly, the way each problem is tackled using divide and conquer: we split the problem into smaller, easier-to-handle tasks.

References

[1] Parallax Inc. Ultrasonic Distance Sensor Application Note., 2017.
[2] Elecfreaks. HC-SR04 Ultrasonic Ranging Module Datasheet., 2013.
[3] Mayergoyz, I. Mathematical Models of Hysteresis and Their Applications. Academic Press, 2003.
[4] Åström, K.J. & Murray, R.M. Feedback Systems: An Introduction for Scientists and Engineers. Princeton University Press, 2010.
[5] Banzi, M. & Shiloh, M. Getting Started with Arduino (3rd Edition). Maker Media, 2015.
[6] Arduino.cc. pulseIn() Function Reference. Arduino Documentation, 2020.
[7] Arduino.cc. WiFi101 / WiFiNINA HTTP Server Examples. Arduino Documentation, 2022.
[8] Åström, K. J., & Hägglund, T. (2006). PID Controllers: Theory, Design, and Tuning. ISA.
[9] Johnson, C. D. (2005). Process Control Instrumentation Technology. Pearson Prentice Hall.
[10] Arduino. "analogRead() Function." Arduino Documentation.
[11] Pololu Robotics. “QTR-1A and QTR-8A Reflectance Sensor Arrays.”
[12] Siegwart, R., Nourbakhsh, I., & Scaramuzza, D. (2011). Introduction to Autonomous Mobile Robots. MIT Press.
[13] Åström, K. J., & Murray, R. M. (2010). Feedback Systems: An Introduction for Scientists and Engineers. Princeton University Press.
[14] Franklin, G. F., Powell, J. D., & Emami-Naeini, A. (2014). Feedback Control of Dynamic Systems. Pearson.
[15] O. S. Nise (2011). Control Systems Engineering. Wiley.
[16] Arduino. “analogWrite() Function.” Arduino Documentation.
[17] Gfrerrer, A. (2008). Geometry and Kinematics of the Mecanum Wheel. Graz University of Technology.