Two-Wheeled Self-Balancing Mobile Robot

Harris Ramos Baroni, Hassan Shahzad, Daniel Helmi[a]

[a]*Department of Computer Science, University College London,*

1. Introduction

In this project, we focus on the design and implementation of a two-wheeled self-balancing robot, leveraging the principles of PID control. The objective is to create a platform that demonstrates fundamental concepts of control theory while emphasising practical electronics design and implementation.

Typically modelled as inverted pendulum on a cart [5], a self-balancing robot represents an intriguing challenge due to the inherent instability of its design [4]. On the surface, such a robot has a limited appeal outside of academics. However, we have seen its real applications such as in the Segway, a two-wheeled, self-balancing electric vehicle [1]. Our project seeks to explore this dynamic equilibrium phenomenon through the lens of electronics and control systems.

We will implement two different modes of operation. One is the teleoperation mode in which the robot can be remotely steered through the use of a gamepad. As with [6], the remote steering is differential driven. The second is the object tracking mode in which the robot uses camera vision to find a marker and maintain a desired pose relative to it.

An important aspect of this project is the implementation of a PID controller to maintain balance of the robot. Originally, the idea was to implement a hybrid analogue-digital PID controller. Kuphaldt discusses fully analogue PID controllers in his textbook [2] which can provide real-time stabilisation. He argues that its faster response speed may be beneficial in applications of motion control. However, we also consider the potential difficulty in maintenance since troubleshooting and debugging a complex analogue circuits can be challenging. Our particular implementation of a PID controller allows us to ease the tuning process through live tuning. The using a fully digital controller also allows us to experiment with more complex control strategies.

We will address various aspects of the design process, including system modeling, theory, electronic design, and testing. By the conclusion of this project, we anticipate achieving a functional two-wheeled self-balancing robot capable of maintaining its upright position in both teleoperation and tracking modes.

2. Theoretical background

2.1. Overall system

The ESP32 is the microcontroller responsible for stabilising the robot through a PID loop and telling the stepper motors what velocity they should be spinning at to stabilise. This is done by utilising an IMU to find how far the robot is from its desired stable position and a Raspberry Pi which feeds the ESP32 which a constant stream of target linear and angular velocity in the form of a twist message.

The Raspberry Pi runs ROS2 nodes to handle communication between the microcontroller, the gamepad and the camera. Ultimately, the nodes are responsible for obtaining inputs from the gamepad and camera and then sending inclination and steering commands to the microcontroller. The gamepad is also used to alter the K_p, K_i and K_d gains for the PID loop used to balance the robot and another value which will be discussed later. We have an OLED display which updates the user of theses values as well as the IMU inclination measurement in real time.

As the stepper motors are driven using 12V, we employ a 3S LiPO battery which directly powers the stepper drivers. This is then connected in parallel to a step down converter capable of outputting 5A max at 5V. This 5V then powers the Raspberry Pi which turns on the ESP32 since both are connected over USB. The OLED display and IMU is then powered by the ESP32.

2.2. Driving the robot

Observe fig. 1, which shows a simplified side profile of the robot. The IMU is calibrated when the robot is fully upright to measure deviations from the upwards vertical. It is unlikely that the centre of mass of the robot lies exactly on the vertical when it is fully upright. Instead, the centre of mass lies at an angle θ_0 from the vertical.

A PID controller is employed to keep the robot balanced as well as drive it forward and backward. In order to drive the forward, we set a target inclination θ_{drive} of the robot from vertical. We seek a suitable error measure

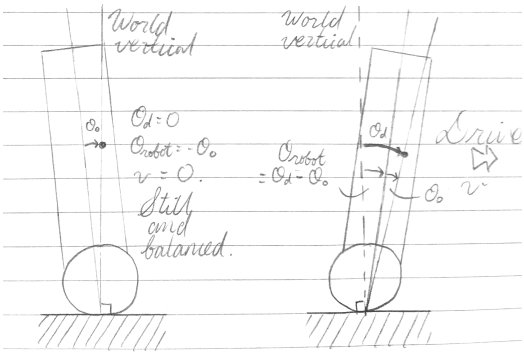


Figure 1: Driving through inclination

ϵ for a PID controller to keep the robot at a desired inclination by outputting command velocities to the wheels. The condition is that the robot be still when we desire the centre of mass of the robot to be upright. Let θ_{robot} be the actual inclination (pitch) that the IMU measures. We can express our condition as follows:

$$v(\epsilon) = K_p\epsilon + K_i \int \epsilon dt + K_d \frac{d\epsilon}{dt}$$

$$\epsilon = v(\epsilon) = 0, \quad \text{if } \theta_{robot} = -\theta_0 \text{ and } \theta_{drive} = 0$$

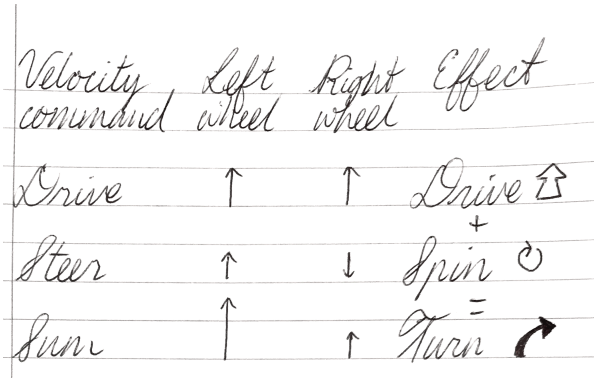
Then $\epsilon = \theta_{drive} - \theta_0 - \theta_{robot}$ satisfies this condition. When $\theta_{drive} = 0$ and $\theta_{robot} = -\theta_0$, the error goes to zero and thus the output velocity v of the PID is also zero, as we wished.

It is important to accurately determine θ_0 so that the our driving and steering commands are realised as we wish. For this reason, we also allow for live editing θ_0 in the EEPROM. This comes with the added benefit of enabling manual compensation for drift in the IMU measurements in real time: simply change θ_0 by the amount of drift.

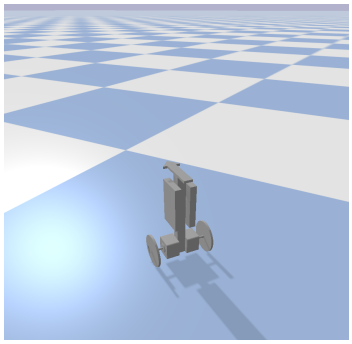
Our balancing PID controller is the most crucial: the robot will not function at all if it is not upright. So excellent tuning will be crucial. For this reason we will enable live tuning of the PID controller. Gamepad inputs are mapped to the desired changes to proportional, derivative and integral gain values and stored in EEPROM external memory so that they can be accessed in later sessions after powering the robot off and on again. We also make the wheels stop altogether once the IMU senses that the robot is past a critical inclination value. This would prevents the wheels from unnecessarily

when the robot is in a state where it will inevitably topple, or perhaps, when the robot is in a toppled state.

2.3. Steering the robot



(a) Turn = Drive + Spin



(b) Successful balancing, driving and steering in PyBullet simulation

Figure 2: Steering the robot

To make the robot turn on the spot, the left and right wheels must go in opposite directions while also respecting the commands to keep the robot upright. In order to turn, the wheels must have a velocity difference which respects the commands to keep the robot at some driving inclination. Figure 2a depicts the way the robot can make turns, that is, driving forward while simultaneously changing the driving direction. To achieve this, it is appropriate to combine the velocity commands for driving and steering with a simple vector sum.

We can verify that this way of combining command velocities is appropriate through simulation. We first constructed a simple URDF model of the robot which also modelled a nonzero θ_0. Using the python physics simulation module called PyBullet, we implemented the aforementioned PID controller which manages the error ϵ by outputting command velocities to drive both wheels forward or backward. Steering and inclination commands were supplied by a Bluetooth gamepad and the robot successfully balanced upright and made turns as desired.

2.4. Filtering IMU data

The stability and robustness of the robot depends on providing accurate data, but IMU sensors are prone to noise and disturbances, which can introduce errors in the measurements. Furthermore, the IMU measures angular

velocity and this data needs to be integrated over time to obtain orientation information. The integration is sensitive to noise and bias errors, which can accumulate over time and lead to drift in the estimation. We filtered the IMU data to reduce noise to provide a smoother and more accurate estimate of the robot's orientation, as well as mitigating integration errors to improve the long-term stability of the estimation.

The dynamics of our robot are more complex than those of a simple pendulum on a cart due to the additional motion of the "cart". This is because the motion is caused by command velocities in the wheels as opposed to command torques. While there already equations of motion that describe the system's behavior when external forces and torques are applied, incorporating velocity commands to the motors adds another layer of complexity.

Kalman filters can provide high estimation accuracy by explicitly modeling system dynamics and sensor noise [3]. As discussed, obtaining an analytical solution for the robots equations of motion can be challenging. For this reason, we used a complementary filter instead as it was simpler to implement provided satisfactory results. See the appendix for the algorithm.

Line 11 is given with respect to the frame our IMU is calibrated to. We tuned the value $\alpha = 0.996$ after some simple experimentation. The intuition behind the value is that we trust the gyroscope data more in the short term. We also want to give the acceleration data less weight because we found that the slight jittering of the stepper motors caused the final angle estimate to drift even when the robot was completely stationary.

2.5. Hand tracking

In the tracking mode, the robot uses its camera to find hands within the frame. Then it adjusts its pose with respect to a hand in the frame. In particular, it attempts to centre the hand within the frame by facing it directly and it also attempts to maintain a desired distance from the hand.

We used the MediaPipe library for Python to recognise hands from image data. We can obtain x and y coordinates of a point of a hand within the frame. A simple PID controller is employed to make the x coordinate of the hand go to 0 through steering commands.

To maintaining distance another simple PID controller was implemented, this time taking an estimate for the distance of the hand from the camera and outputting an inclination command to drive the robot. To estimate the distance d_{real}, we chose a open palm hand pose with fingers together for the camera to recognise. Then we obtained the (x,y) coordinates of

two chosen finger joints and found their separation s in the frame. This separation is inversely proportional to the hand's distance from the camera: $d_{\text{real}} = m(\frac{1}{s}) + b$. The constants m, b are specific to the camera. A simple experiment was used to determine them: hold the hand at several distances d_{real}, note down the value s, and make a plot of d_{real} against $\frac{1}{s}$. Obtaining m and s from the graph is straightforward.

2.6. Parallel programming with an RTOS

We encountered issues when attempting to perform multiple tasks within the same main loop on the ESP32 microcontroller. Specifically, we observed jitters and slow movements in our stepper motors when driving them alongside other operations, such as spinning the microROS node or reading data from the IMU. Recognising the need for a more efficient approach, we adopted a strategy to parallelise our system's tasks.

To address these issues, we leveraged the dual-core architecture of the ESP32, dedicating core 1 to handle tasks related to spinning the micro-ROS node as well as reading and filtering data from the IMU. Meanwhile, core 0 was utilised to drive the stepper motors. This separation of tasks helped mitigate interference between critical motor control operations and other non-time-critical tasks. To facilitate this parallel execution, we utilised FreeRTOS, an open-source real-time operating system for embedded systems. FreeRTOS provided the necessary features for task scheduling, priority management, and synchronisation between cores, enabling us to effectively partition our system's workload.

However, during our implementation, we encountered a challenge where gamepad inputs managed by the ROS2-microROS system on core 1 were not effectively passing through to the task running on core 0. To address this issue, we adjusted the priority of the task on core 0. By lowering its priority, we allowed the higher-priority tasks (like spinning the node) execute without interruption. While this adjustment resolved the jittery movement issue, it came at the expense of slightly reduced maximum motor speeds. This balance of task priorities ensured smooth and responsive operation of the overall system.

3. Electronic composition

Our system contains:

- NEMA 17 stepper motors x2

- Raspberry Pi 4 4GB
- ESP32 Microcontroller
- Stepper motor driver x2 (A4988)
- IMU (MPU6050)
- Bluetooth gamepad
- Camera
- EEPROM (1KB)
- 12V 3S LiPO battery

4. Electrical and Electronics Design

4.1. Battery circuitry

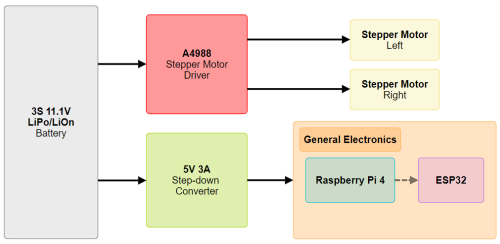


Figure 3: Battery circuitry

See fig. 3. We used a LiPo batter to power the stepper motors. The UBEC is a step down converter with an inductor to prevent current surge to the electronics when the motors spin suddenly.

4.2. SBC-MCU communication architecture with ROS2 and MicroROS

Explanation: See fig. 4 which depicts the communication architecture to communicate from the Raspberry Pi 4 to the ESP32. The Raspberry Pi is running ROS2 humble nodes and the ESP32 is running a microROS node.

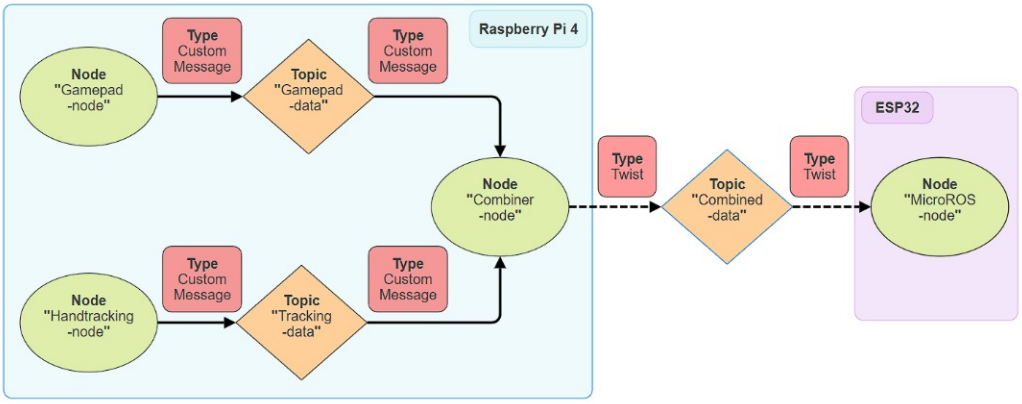


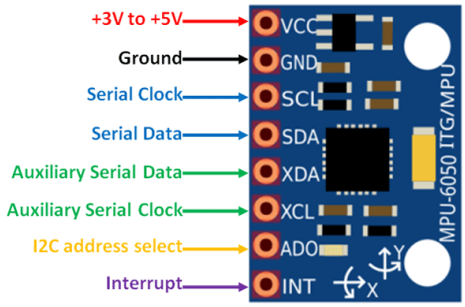
Figure 4: Flowchart: Raspberry Pi and ESP32 interaction

- (Gamepad Node) Responsible for handling gamepad inputs. This node receives gamepad input data over Bluetooth and maps those to inclination and steering commands. The y axis of the joystick determines the inclination command and the x axis of the joystick is mapped to steering command velocities. Certain button presses are mapped to commands to change the PID gains for balancing and another important value. One of the buttons of the gamepad is used to toggle to hand tracking node on I'm and off. The tracking mode is off by default. It publishes on a topic called "gamepad_data".

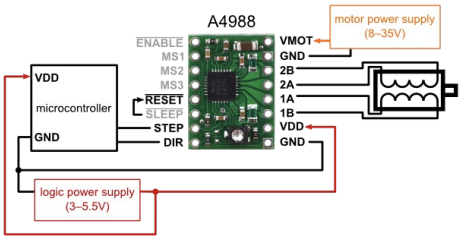
- (Hand tracking node) Uses camera data and processed with mediapipe library. It determines commands to centre the hand in the view frame using a PID loop as well as to maintain a desired distance from the hand. Publishes data to "tracking_data".

- (Combiner node) Handles the mode toggling process, ensuring that only one complete set of commands sent to the microcontroller. Subscribes to both "gamepad_data" and "tracking_data" and publishes on a topic called "combined_data".

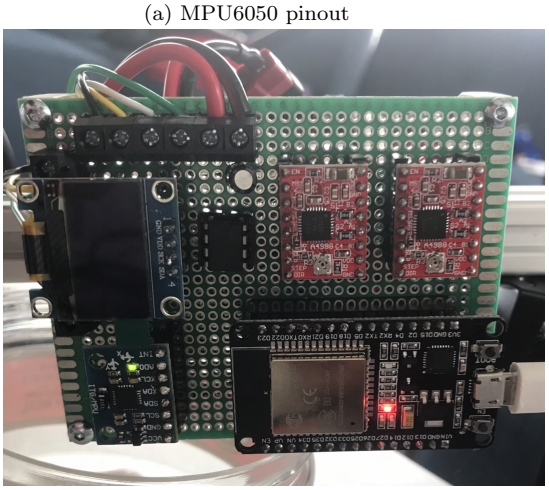
- (MicroROS node) Subscribes to the "combined_data" topic to obtain the commands from the gamepad and camera. These are used to then drive the motors as desired.



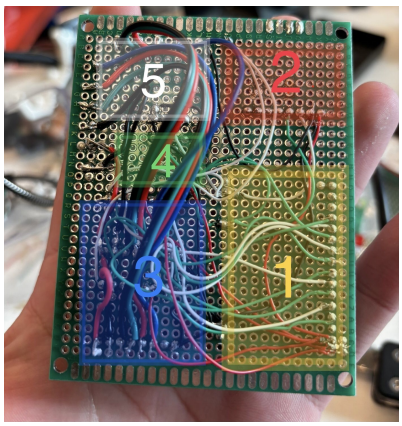
(a) MPU6050 pinout



(b) A4988 pinout



(c) Components arranged



(d) Electronics wiring

Figure 5: Component pinouts and assembly

4.3. Pinouts and Communication protocols

To enable the smallest step resolution (one 16^{th} of a step), we set the MS1, MS2 and MS3 pins in the A4988 to high. Show table. This ensures the smoothest experience.

The ESP32 can read and write to an external EEPROM using the I^2C protocol. This protocol is also used for the ESP32 to communicate to the OLED display and the MPU6050. The OLED display is interfaced with I^2C communication as it does not need to be high speed and using fewer pins is desirable.

The benefit of running the PID on the ESP32 instead of the Raspberry Pi is that latency in transmitting and receiving the data between both computers makes the system naturally more unstable since in control theory, the

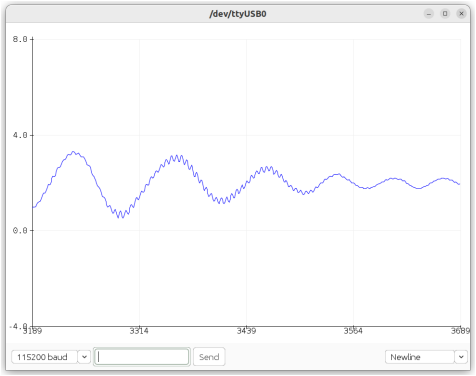
root locus poles get closer to positive.

Figure 5d depicts the wiring of (1) the ESP32, (2) the OLED display, (3) the two A4988 stepper drivers, (4) the external EEPROM and (5) the MPU6050.

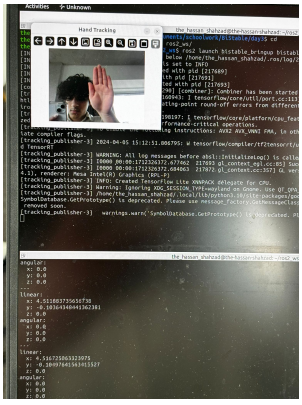
5. Experimental Evaluation

5.1. The robot in action

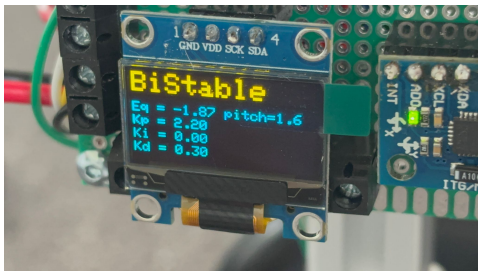
(a) Spinning robot with no support



(b) Inclination angle restoring to stability



(c) Hand tracking using MediaPipe



(d) OLED displaying values stored in memory

5.2. Debugging, testing and tuning

We employed a few measures to make debugging, testing and tuning processes more efficient.

Firstly, we enabled live tuning of the PID gains for balancing and the angle offset θ_0 as discussed in section 2.2, facilitated by displaying the values on the OLED display.

Secondly, a switch is hooked up to the enable pin of stepper drivers (fig. 5b). This way, we could disable the motors whenever we needed. As we already discussed, we found that the jitter in the stepper motors caused slight drift in the filtered IMU readings. We found that tuning the value of θ_0 stored in the EEPROM was most successful when the motors were turned off.

We also used secure shell (SSH) to remotely interface with the Raspberry Pi's Ubuntu terminal. This was important for experimentation as we could remotely start nodes from a laptop connected onto the same network. The alternative would be to connect the Pi to a monitor through HDMI to then interface with the OS directly and finally disconnect it so that the robot can freely move.

6. Discussion and Conclusion

We were very content with the successful ROS-MicroROS architecture. This made our programs modular and scalable. Suppose that our hand tracking had bugs. Then we could always stop publishing from the hand tracking node and the rest of the communications are unaffected. Suppose we wanted to control the robot with a keyboard, then we could simply add another node to handle keyboard inputs. The use of parallel programming to synchronise the ROS process and motor driving process was very successful.

There are some aspects that we would like to improve. One example is using a layout where the effect of the heating of the LiPO battery on other components (such as the Raspberry Pi) is mitigated. We did not particularly find this troubling but it could be a potential issue if the battery gets too hot.

Regarding the balance, it would be better if the IMU were interfaced using SPI as it is crucial that the IMU quickly updates the balancing program on the inclination error to optimise the time taken to correct the robot's pose. We also feel that the system would have benefited greatly from highly accurate gyroscope data. We empirically determined that the steering method

works in simulation, but when translated to real life, the drift in the inclination estimate would deteriorate any stability in the robot after it leaned past any larger angles or after a long time of running. We still discovered a few ways to mitigate drift in the inclination estimate, such as having the motors stop when the robot leaned at large angles. We would like to experiment with other filtering and data fusion techniques in the future, perhaps using a simplified Kalman filter (even if the dynamics of the system are complex) of a combination of Kalman and complementary filters. We did try a Kalman filter library and the estimated angle had very low drift, but it flickered a lot. Perhaps we could reduce the resolution of the estimate to avoid jittering in the motors due to a flickering inclination angle estimate.

We found that the camera data was rather slow and had a delay of about 1500 to 2000 milliseconds. This does not affect the balance of the robot as it is a secondary ability that runs separately, but the tracking response time is heavily affected.

Overall, we were pleased with many aspects of the project but we wish to continue its development to achieve even better performance later.

7. Appendix: Algorithms and Program Code

Algorithm 1 Complementary Filter Algorithm for IMU Data Fusion

Initialise variables:
$\theta_{\text{gyro}} \leftarrow 0$ ▷ Initial orientation estimate from gyroscope
$\theta_{\text{acc}} \leftarrow 0$ ▷ Initial orientation estimate from accelerometer
$\alpha \leftarrow 0.996$ ▷ Complementary filter coefficient ($0 < \alpha < 1$)
previous_time $\leftarrow 0$

procedure ComplementaryFilter($\omega_{\text{gyro}}, \theta_{\text{acc}}$, current_time)
    ω ▷ Read raw gyroscope data
    $[a_x, a_y, a_z]$ ▷ Read raw accelerometer data
    $\Delta t \leftarrow$ current_time − previous_time
    $\theta_{\text{acc}} \leftarrow \arctan(a_y, a_x)$ ▷ Compute accelerometer angle
    $\theta_{\text{gyro}} \leftarrow \theta_{\text{gyro}} + \omega_{\text{gyro}} \cdot \Delta t$ ▷ Integrate gyro data
    $\theta_{\text{gyro}} \leftarrow \alpha \cdot \theta_{\text{gyro}} + (1 - \alpha) \cdot \theta_{\text{acc}}$ ▷ Apply compl. filt.
    return θ_{gyro} ▷ Filtered orientation estimate
end procedure

Our system depends on several programs. See our GitHub repositories for the full ROS2 packages as well as the PyBullet simulation. Packages: BiStable_bringup, BiStable_arduino, BiStable_description, BiStable_scripts, bistable_interfaces. Simulation: bistable. It is likely that you do not have a Bluetooth gamepad at hand to try the simulation. To control the robot in the simulation using the arrow keys on you keyboard, switch to the "keyboard_inputs" branch of the "bistable" repository instead of main.

References

[1] Segway Inc. Segway homepage, 2006. http://www.segway. com/.

[2] Control systems (A volume in the online textbook "Lessons in Industrial Automation"). Tony R. Kuphaldt. url: https://control.com/textbook/. Released under the Creative Commons Attribution 4.0 International Public License: https://creativecommons.org/licenses/by/4.0/.

[3] Pengfei Gui, Liqiong Tang, and Subhas Mukhopadhyay. "MEMS based IMU for tilting measurement: Comparison of complementary and kalman filter based data fusion". In: *2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*. 2015, pp. 2004–2009. DOI: 10.1109/ICIEA.2015.7334442.

[4] Hanna Hellman and Henrik Sunnerman. *Two-Wheeled Self-Balancing Robot*. 2015.

[5] Mahmoud Khaled et al. *Balancing a Two Wheeled Robot*. 2009.

[6] Jingtao Li et al. "Controller design of a two-wheeled inverted pendulum mobile robot". In: *2008 IEEE International Conference on Mechatronics and Automation*. 2008, pp. 7–12. DOI: 10.1109/ICMA.2008.4798717.