Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Remote Controlled Segway `ev3dev` Implementations

Autori:
Agnolucci Lorenzo
Baldrati Alberto
Berti Giovanni

Supervisore:
Prof. Michele Basso

Anno Accademico 2019/2020

Index

List of figures

Introduction

A Segway-like two-wheeled robot is a non-linear dynamical system characterized by two equilibrium states, an unstable one and a stable one that is considered not interesting because it corresponds to the horizontal position of the robot. For this reason we will implement a control system that will stabilize the robot in its vertical position, which corresponds to the unstable equilibrium state. The hardware board used to develop the robot is a LEGO® MINDSTORM EV3, which when paired with a MATLAB [1] environment enables quick prototyping of control systems and rapid development iterations. Our control model is based on the linearized model described in [2] which we will cover in section 1.1 on page 1.

`ev3dev` [3] is a Debian Linux-based operating system that runs on several LEGO® MINDSTORM compatible platforms including the LEGO® MINDSTORM EV3 and it represents an alternative to the official proprietary software. `ev3dev` allows to program the robot exploiting the functionalities intrinsically provided by Linux in addition to the ones already available in the LEGO® software. `ev3dev` lets to choose from a great variety of programming languages, for example Java, C++ and Python 3. [4] [5]

In this work we compare and analyze three different implementations of our control system: two preliminary implementation written in Python 3 and a final implementation written in C++. All implementations run on top

of `ev3dev`. We will then further extend the reference model from in-place balancing to allow free movement in all directions. We also developed an Android app with an intuitive GUI to control the robot easily through a joystick-like controller.

Chapter 1

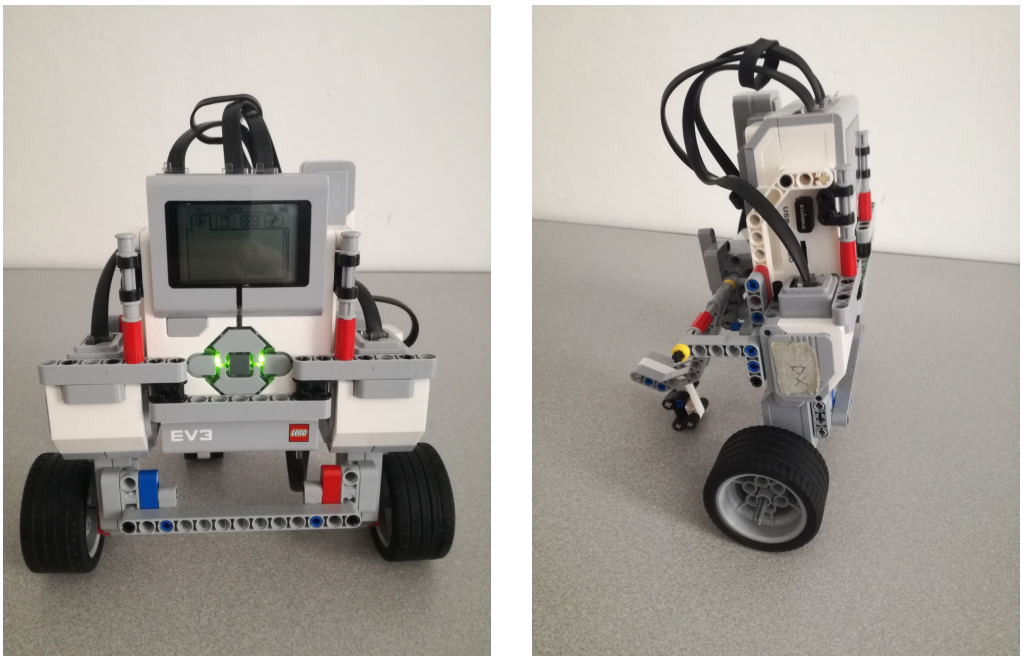
System Model

1.1 System description

The dynamical system associated with the robot (figure 1.1 on the next page) can be approximated by considering the robot body as a material point positioned at a height H above the wheels' axle. The whole system is an inverted pendulum (figure 1.2 on page 3) with wheels and with a kickstand having negligible mass.
The system motion equations use three variables:

- ϕ: the angle that the vector position of the centre of the wheels forms with the x-axis
- ψ: the angle that the straight line that connects the centre of mass of the body and the centre of the wheels forms with the z-axis
- θ: the angle given by the average of the angles of the right and left wheels

The vertical equilibrium position coincides with a ψ angle equal to zero. We use the linearized model obtained in [2] around this equilibrium state.



(a) (b)

Figure 1.1: Physical system described above. **(a)**: Front view **(b)**: Side view

The system state is described by the vector:

$$x = \begin{bmatrix} \theta & \psi & \dot{\theta} & \dot{\psi} \end{bmatrix}^T$$

The quantities are acquired through the on-board sensors with a 1 degree uncertainty. For the sake of a more streamlined mathematical treatment, all these quantities are then converted to radians before proceeding with control computations. Even though the board engine encoders can both provide their relative angles, for the sake of simplicity we represent them in the model with a single angle θ computed by taking the average of the two actual engine angles.

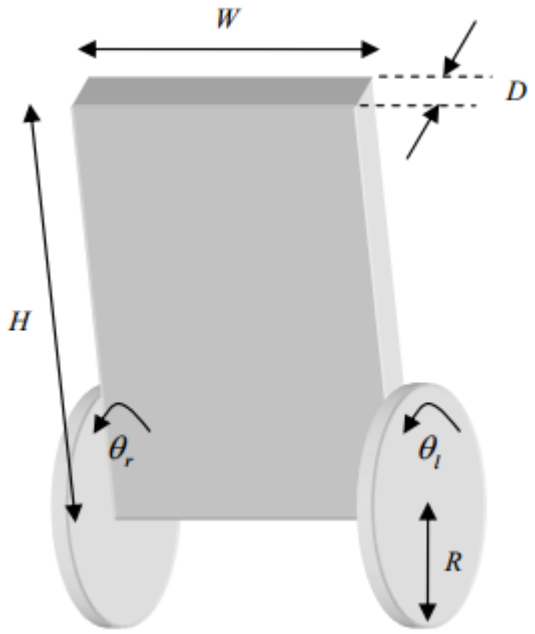


Figure 1.2: System structure

1.2 Control model

The chosen control model is a proportional feedback derived from the theory of optimal control of the type

$$u = -Kx$$

where $u = \begin{bmatrix} V_l & V_r \end{bmatrix}^T$ are the voltages given in input to respectively left and right motors and K is the gain matrix.

In order to make the system more robust we use an augmented state $\tilde{x} = \begin{bmatrix} x(t)^T & z(t) \end{bmatrix}^T$, where $z(t)$ is the integral of the error function $e(t) = x(t) - x_{ref}$ and $e_\theta(t) = \theta - \theta_{ref}$ with $\theta_{ref} = 0$.

Finally we obtain the gain matrix K as described in [2]:

$$K = \begin{bmatrix} 0.8559 & -44.7896 & 0.9936 & -4.6061 & 0.5000 \\ 0.8559 & -44.7896 & 0.9936 & -4.6061 & 0.5000 \end{bmatrix}$$

Chapter 2

Control Model Implementations

2.1 General robot operations

In order to implement the control model stated above we need access to the system's state. `ev3dev` allows us to directly probe the values of engine angles (θ) and speed ($\dot{\theta}$), gyroscope angle (ψ) and angular velocity ($\dot{\psi}$) from the board sensors. Our implementations present a significant difference from the baseline, that is, we have direct access to the angular velocity of the two engines and the gyroscope angle thanks to `ev3dev`, thus numerical integration and derivation are not needed. Empirically, as shown in figure 2.1 on the next page, numerical derivation proved to be more prone to noise. These capabilities reflected in a wider range of sampling frequencies in which the model manages to successfully control the system.

To compute the full augmented state, including the integral action on θ, numerical integration is performed using forward Euler's method as below, with 0 as the initial value:

$$\theta_{int}(n) = \theta_{int}(n-1) + [t(n) - t(n-1)] \cdot \theta(n-1)$$

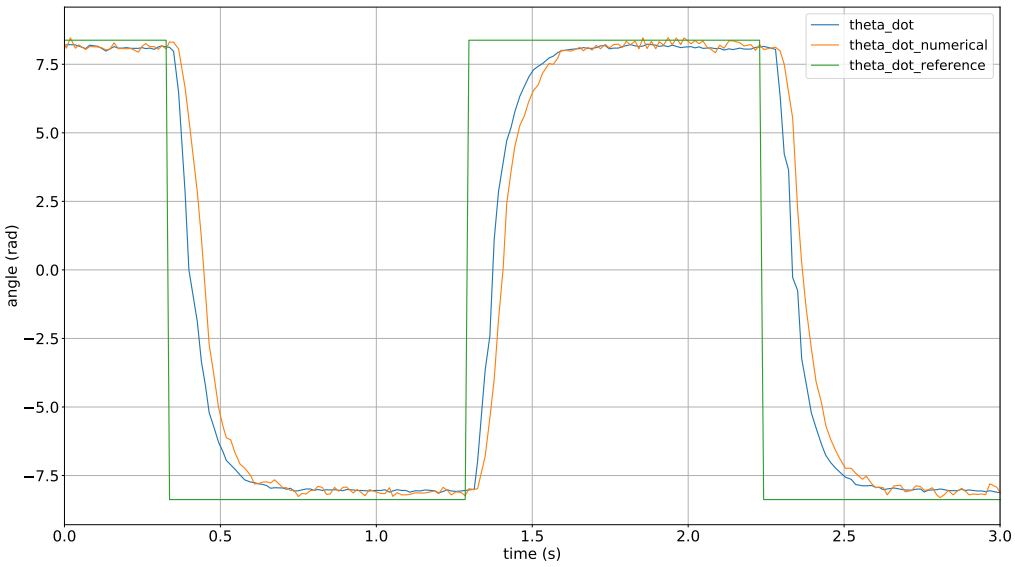


Figure 2.1: Performance of sensor-acquired angular speed (orange) and derived angular speed (green)

When starting the robot it lays upon its kickstand at an angle $\psi = 14°$. Because the initial gyroscope calibration phase resets the internal sensor angle to zero, we account for this discrepancy by subtracting this initial value from each measurement.

2.1.1 General workflow

We can summarize the entire robot lifecycle in the following steps:

1. **Run** the program executable and enter in an **idle** state
2. **Calibrate** the gyroscope for two seconds
3. **Start** the control loop and **lift** the kickstand right after
4. **Keep** the control loop running until requested by the user

5. **Lower** the kickstand, **stop** the control loop and accelerate forward in order to **lay** on the kickstand

6. **Go back** to step 1

2.2 Python implementation

We describe two different Python implementations with difference performance and levels of abstraction. We initially chose Python as the programming language because of its ease of use and expressiveness. Unfortunately its performance limitations proved to be too severe to allow for a correct stabilization from the control model implementation.

2.2.1 High level implementation

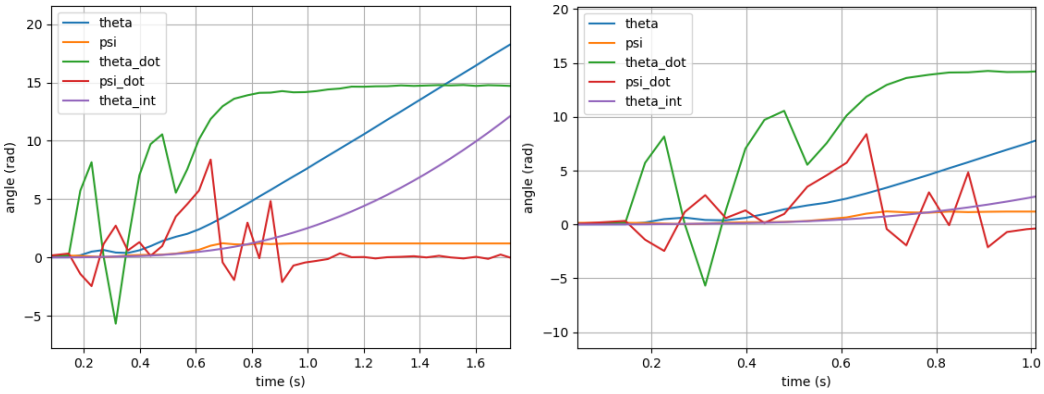
We initially implemented the workflow described above as a finite state machine with the aid of the `python-transitions` [6] library. This strategy was chosen to more closely match the Stateflow abstraction used in the baseline MATLAB implementation [1] [2]. Since the early tests we observed that this implementation could not manage to have control cycles faster than 250ms, much higher than the baseline expected sampling time of 10ms. For this reason we found necessary to come up with a different approach.

2.2.2 Performance oriented implementation

To try to overcome these performance issues we decided to stop using a finite state machine and manually manage states and threads using Python standard library facilities. We used two Python threads to concurrently operate the kickstand engine and handle the control loop. With the aid of a

profiler we noticed that most of the cycle time was spent in the underlying `ev3dev` library calls for files handling. In fact `ev3dev` opens and closes a specific special file each time it has to read a value from a sensor or to set an actuator value. Because of that instead of using `ev3dev` functions to read sensors data and set engines' power we used a faster manual low-level handling of these files.

Even though we strived to keep the cycle times as low as possible, this implementation could not achieve timings below 70ms and proved insufficient to successfully control the robot. As can be seen in figure 2.2, $\dot{\psi}$ (red line) rises abruptly after about 0.5 seconds suggesting the forward fall of the robot. After that moment all the quantities become meaningless as the robot simply accelerates while laying on the ground.



(a) (b)

Figure 2.2: Plots of the state in performance oriented Python implementation. **(a)**: General view **(b)**: Region of interest

2.3 C++ implementation

Both Python implementations were unable to successfully stabilize the robot due to poor performance and high cycle times. Therefore we decided

to use C++ which is officially supported by `ev3dev` [5] for taking advantage of its notoriously good performance even on embedded systems such as the LEGO® MINDSTORM EV3. However due to the limitations of the hardware of the robot, we could not compile the source code directly on it, but we had to use cross compilation through a Docker image provided by the `ev3dev` project. This has direct implications on the usage of external libraries which is not immediate and straightforward. Furthermore, because of the low-level nature of C++, we leveraged Linux system calls to give a higher priority to both the process and the control loop thread. This higher priority level allows the system to give the program more CPU resources, thus gaining a performance advantage over lower priority processes.

With the C++ implementation we managed to get exceptional performance, with cycle times of at most 5ms and with a 95% quantile of 3ms. To match the timings used in the original MATLAB model we added a sleep instruction in the control loop in order to get to a cycle time of 10ms.

Figure 2.3 on the next page shows that in this implementation the control is able to stabilize the robot after lifting the kickstand. Initially the robot strongly accelerates forward to move from the inclined position held with the kickstand support to the vertical one. After that fluctuations around the unstable equilibrium are minimal. Eventually the robot lays on the kickstand as we can see from the spike of $\dot{\psi}$ which is opposite to the initial one.

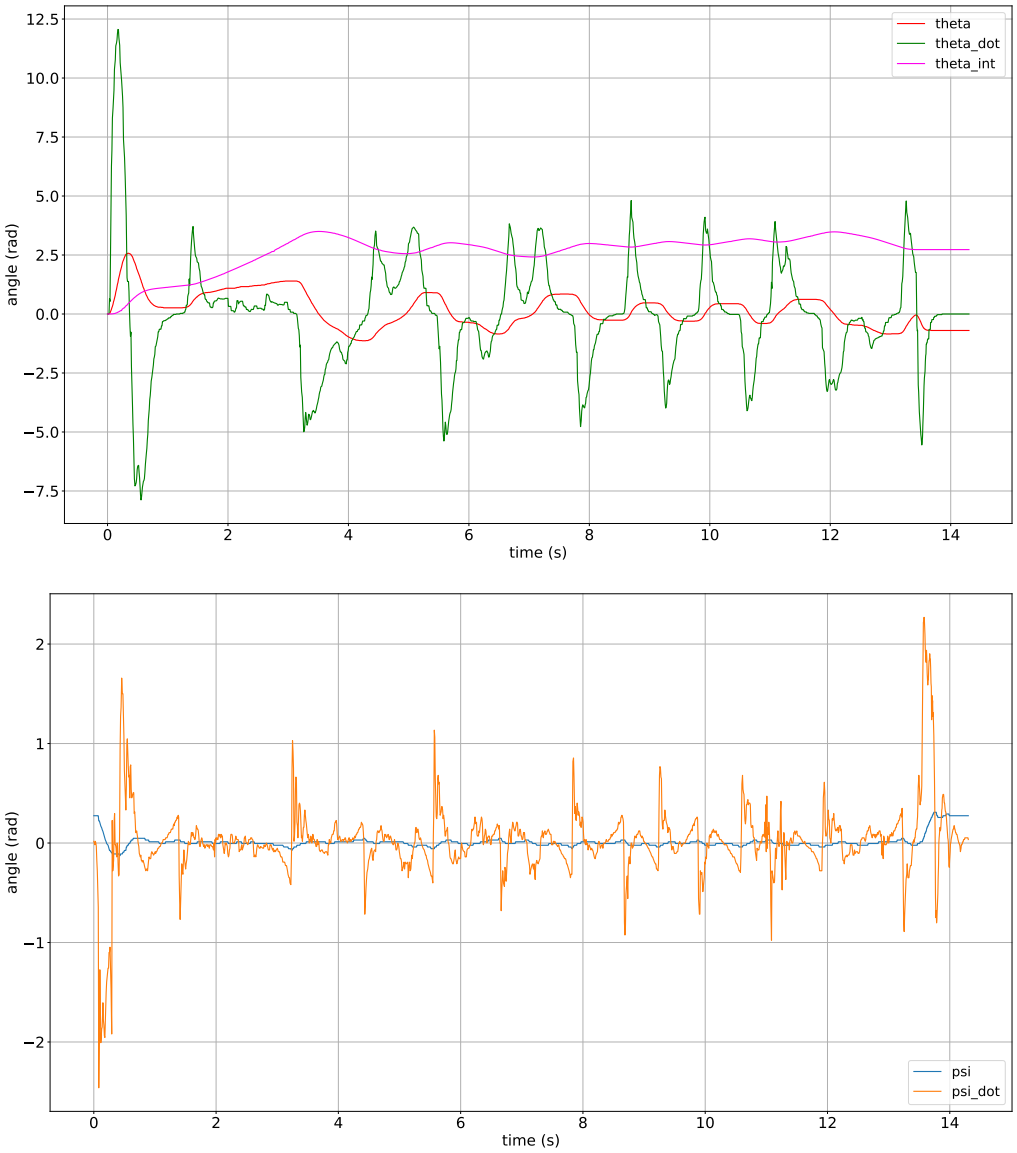


Figure 2.3: Plots of the state in C++ implementation

Chapter 3

Robot Movement

3.1 Model extension

The model described in chapter 1 can be easily extended in order to deal with variable reference wheel angular velocity $\dot{\theta}$. In the original model the quantities θ, $\dot{\theta}$ and θ_{int} also represent the discrepancy from the reference, in fact when the robot stays still all the references are simply equal to zero. In the extended model we conceptually do the same operations with variable references not necessarily equal to zero, so the state of the model becomes:

$$\tilde{x}(n) = \begin{bmatrix} \theta_{err}(n) & \psi(n) & \dot{\theta}_{err}(n) & \dot{\psi}(n) & \tilde{\theta}_{int}(n) \end{bmatrix}^T$$

where:

- $\dot{\theta}_{err}(n) = \dot{\theta}(n) - \dot{\theta}_{ref}(n)$
- $\theta_{err}(n) = \theta(n) - \theta_{ref}(n)$, with
  - $\theta_{ref}(n) = \theta_{ref}(n-1) + [t(n) - t(n-1)] \cdot \dot{\theta}_{ref}(n)$
- $\tilde{\theta}_{int}(n) = \tilde{\theta}_{int}(n-1) + [t(n) - t(n-1)] \cdot \theta_{err}(n-1)$

Obviously even in this extended model we want to keep the robot in the vertical unstable equilibrium state, so the references for ψ and $\dot{\psi}$ are still zero.

To support a wider range of motion we also add the possibility of controlling the robot steering. This can be easily done by applying a slightly different power to the right and left motors. Specifically, we define a steering component s_{ref} and we add it to a motor and subtract it from the other, so that we have:

$$u = \begin{bmatrix} V_l \mp s_{ref} & V_r \pm s_{ref} \end{bmatrix}^T$$

With these changes we get a fully controllable robot capable of moving in all directions in a two-dimensional plane while maintaining the unstable vertical position.

In this model a critical situation arises when the reference speed change abruptly, in fact this case can lead to instability. Because of that we apply a variable number of cycles of stabilization, that is, we gradually adjust the reference speed to gently reach the desired one.

As can be seen in figure 3.1 on the following page, $\dot{\theta}$ (blue line) follows the evolution of its reference $\dot{\theta}_{ref}$ (orange line) but fluctuates around it because the robot has to keep its balance even while moving. While $\dot{\theta}_{ref}$ (and the corresponding θ_{ref}) is equal to zero θ_{err} (green line) coincides with θ, otherwise it represents the discrepancy between the reference and the actual value. When the reference speed changes we can notice the stabilization cycles cited above, in fact $\dot{\theta}_{ref}$ is gradually increased and reduced.

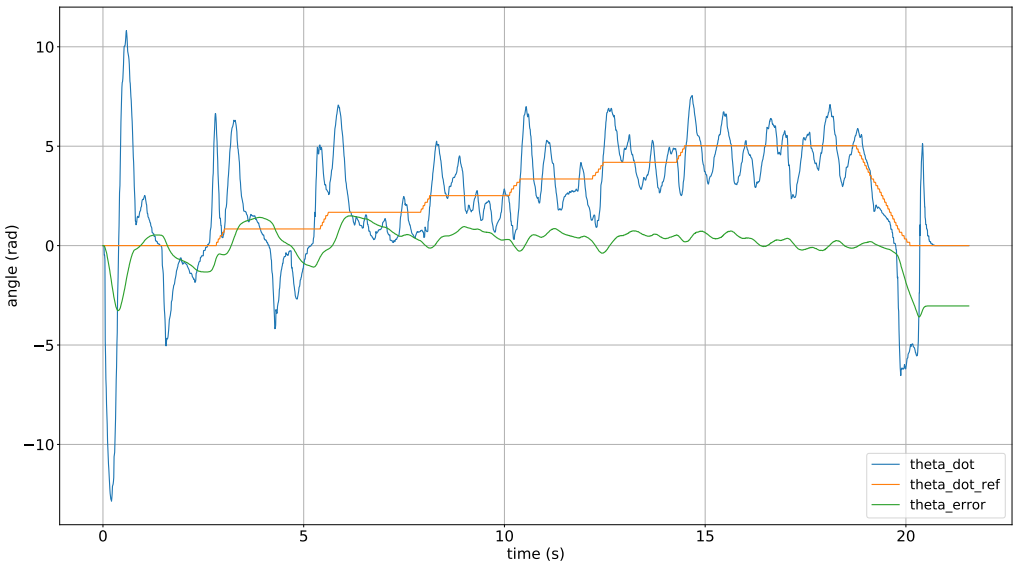


Figure 3.1: Plot of θ_{err} and of $\dot{\theta}_{ref}$ in comparison with the actual $\dot{\theta}$

3.2 Remote control with SSH and EasyControl

We further extended the C++ implementation according to the model described in the previous section by letting the user to dynamically change speed and steer references via a command line interface connected to the robot through SSH. The directives available to control the robot are defined in terms of strings, such as `speed_ref=10`, `steer_ref=3`, `start`. In figure 3.2 on the next page is displayed a screenshot of the CLI during SSH connection.

To make sure that the control commands given by the user don't threaten the robot's stability, we put an upper and lower bound on the values that the speed reference and steer reference can take. These values are defined in terms of the engines' duty cycle. We empirically determined the bounds to be ±30 for the speed reference and ±15 for the steer reference.

Later we developed an Android app that connects a smartphone to the

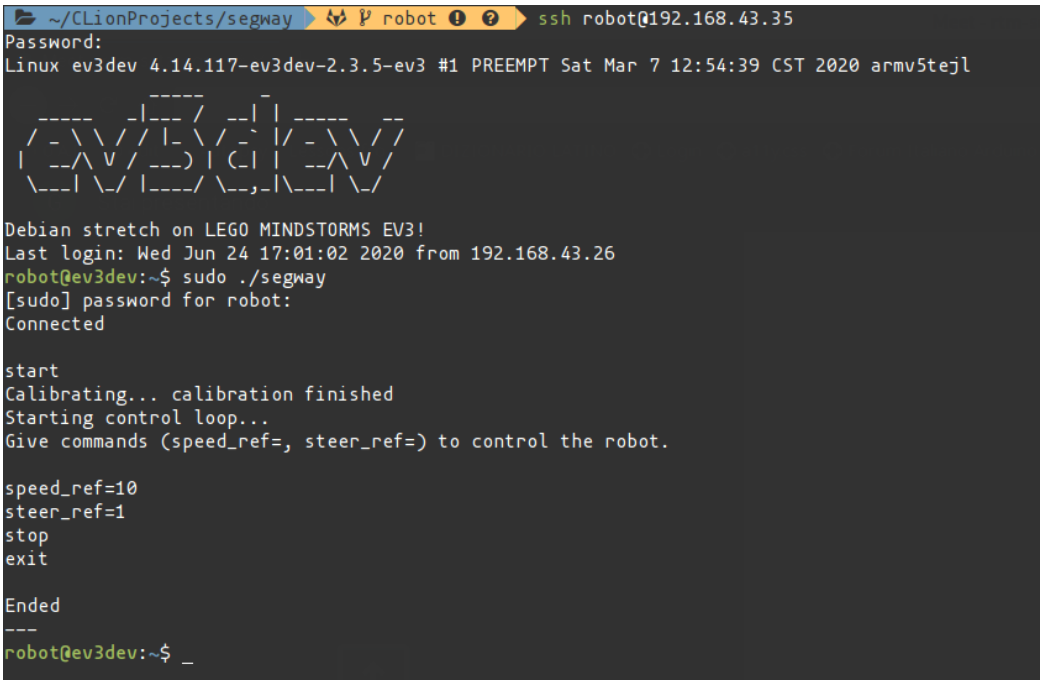


Figure 3.2: Screenshot of the CLI during SSH connection

robot via SSH when they are connected to the same Wi-Fi network to enable a more user-friendly remote control environment. Even in this case the textual nature of the control is preserved, in fact the commands given by the user via the app are converted into strings and sent to the robot through SSH. The commands are sent by the app at 60Hz (or equivalently every 16ms) and the average latency is about 60ms. *EasyControl* app was developed with the Kotlin programming language and Android Studio IDE.

The functionalities of the app are pretty straightforward and they are shown in figure 3.2 on page 16:

(a) App splash screen with app logo.

(b) Main app screen. The `Connect` button can be used to connect the application to the robot. The smartphone and the robot need to be on the same network

(c) Settings screen. The user can set the robot IP address used for connection

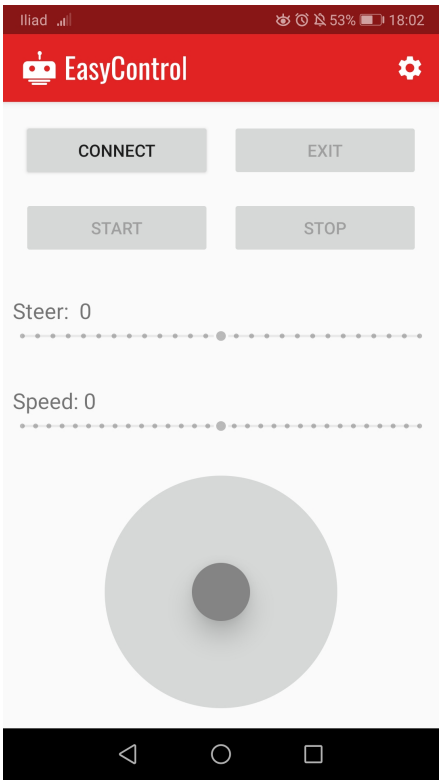
(d) Connection error dialog in case of connection issues

(e) Screen state after successfully connecting to the robot. The `Start` button begins program execution, the `Stop` button terminates it and `Exit` disconnects the app from the robot

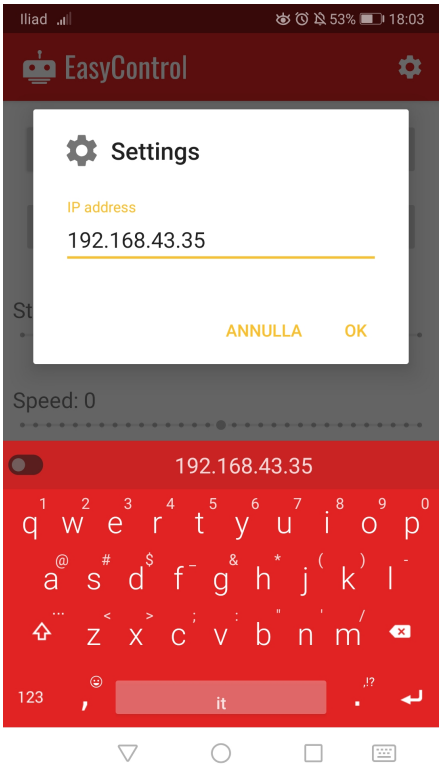
(f) Screen state after starting the control program. The two sliders reflect the reference values input through the joystick positioned in the lower section of the screen but cannot be used as input themselves. The user can control the robot through the joystick and can move it in all directions. Specifically the vertical axis of the joystick is relative to the speed value while the horizontal axis controls the steering. The joystick itself outputs values between zero and one, however we need values in the range that keeps the robot stable ($[-30, 30]$ for the speed and $[-15, 15]$ for the steer), therefore we apply a scaling of the raw values produced by the joystick.


Iliad 74% 12:20
EasyControl

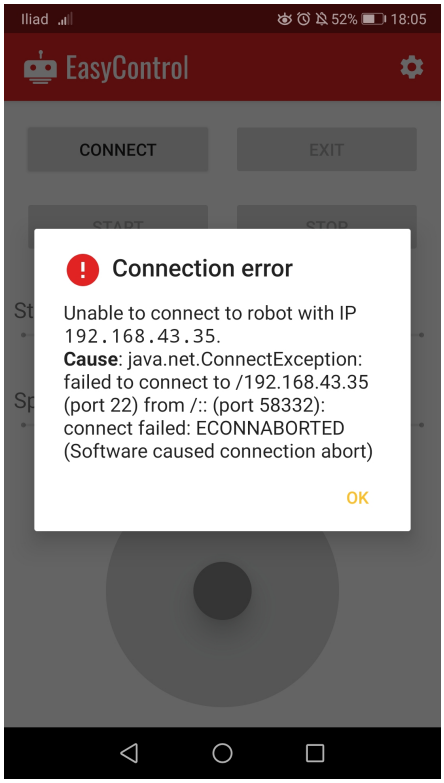
(a)


Iliad 53% 18:02
EasyControl
CONNECT
EXIT
START
STOP
Steer: 0
Speed: 0

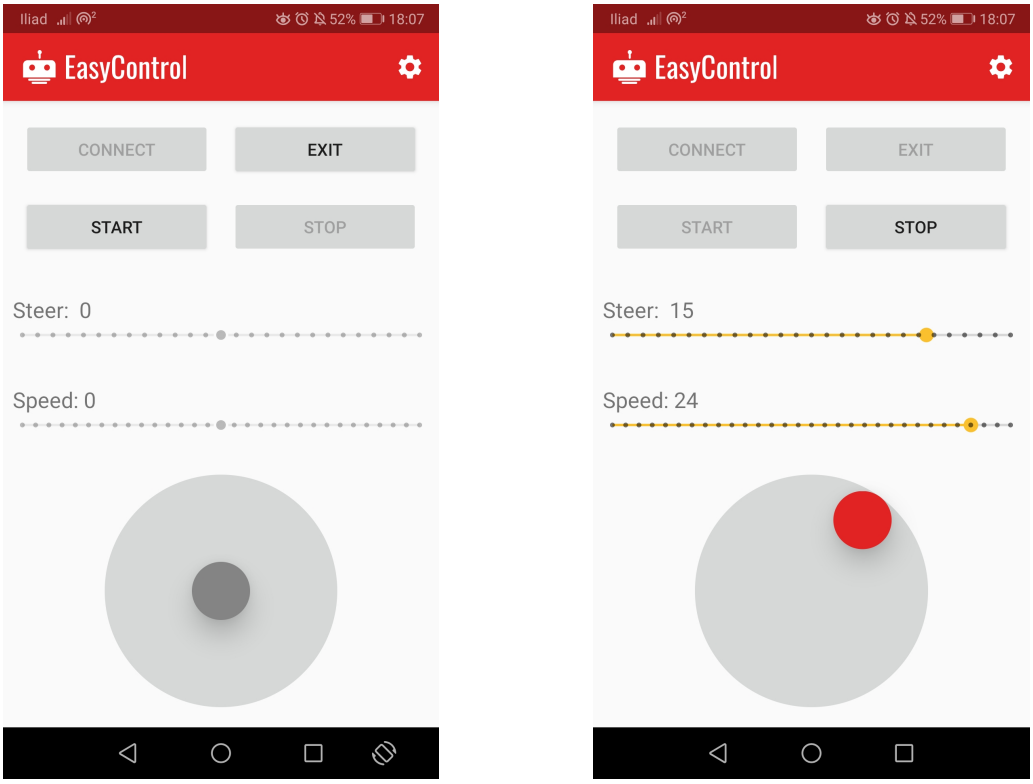
(b)


Iliad 53% 18:03
EasyControl
Settings
IP address
192.168.43.35
ANNULLA
OK
Speed: 0
192.168.43.35

(c)


Iliad 52% 18:05
EasyControl
CONNECT
EXIT
Connection error
Unable to connect to robot with IP 192.168.43.35.
Cause: java.net.ConnectException: failed to connect to /192.168.43.35 (port 22) from /:: (port 58332): connect failed: ECONNABORTED (Software caused connection abort)
OK

(d)



(e) (f)

Figure 3.3: App screenshots

Conclusions

In this work are described various implementations of a control system for a Segway-like two-wheeled robot using the open source Debian Linux-based `ev3dev` operating system.

The biggest challenge we faced was not being able to reach the required performance for such a low-latency task as the stabilization of the robot in its unstable equilibrium state. Unfortunately both our Python-based implementations did not fulfill such real time requirements. This made it necessary to rewrite the entire control software in C++ while favouring a more low-level approach to the software implementation strategy. The facilities offered by the Linux system equipped on the LEGO® MINDSTORM EV3 board allowed us to easily extend the model to include a variable speed reference and pair it with a remote control Android application.

As a future work it would be interesting to equip the robot with a camera or a proximity sensor to allow the use of an obstacle detection and avoidance mechanism.

Code of all implementations and Android app available at `https://gitlab.com/turboillegali/segway`

Bibliography

[1] The MathWorks Inc. Matlab r2020a, 2020.

[2] Sergio Carleo Davide Martini. Control of a lego mindstorms ev/3 two-wheeled robot. 2020.

[3] ev3dev. https://www.ev3dev.org/.

[4] ev3dev. ev3dev-lang-python, 2020. https://github.com/ev3dev/ev3dev-lang-python.

[5] Denis Demidov. ev3dev-lang-cpp, 2020. https://github.com/ddemidov/ev3dev-lang-cpp.

[6] transitions. https://github.com/pytransitions/transitions.