FINAL PROJECT

DESING OF EMBEDDED SYSTEMS

MSC IN EMBEDDED COMPUTING SYSTEMS

SCUOLA SUPERIORE SANT'ANNA

Ball Track

Author:
Leonardo Lai

Email:
leonardo.lai@sssup.it

Date: February 17, 2019

1 Description

Project *BallTrack* is a combination of automatic control and machine learning. The objective is to drive a ball through a tortuous path on a 2DOF platform, and reach the target position, trying to avoid collision with obstacles as much as possible. A microcontroller monitors system parameters and tilts the plate accordingly. No user interaction is needed, except to reset the ball position; thanks to reinforcement learning, the controller always know the best action to perform in order to fulfill the task.

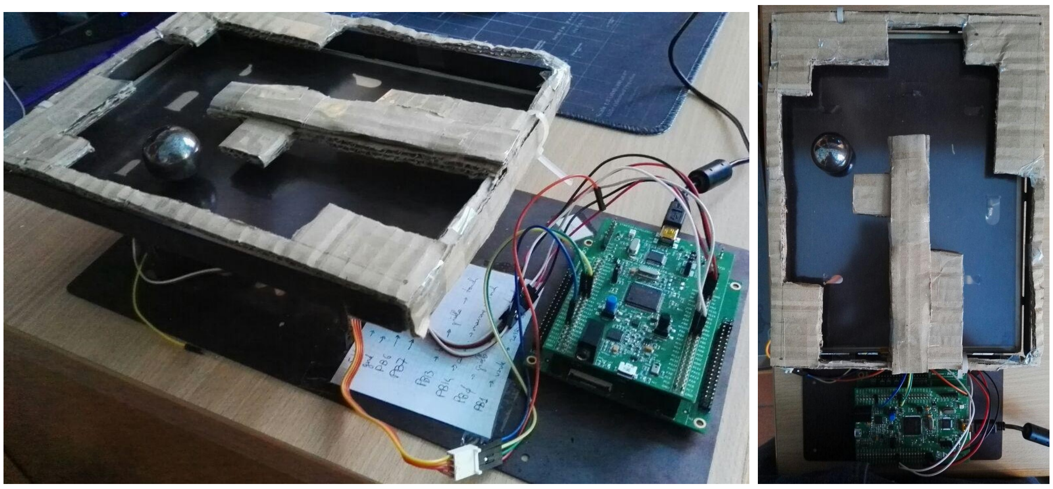

Figure 1: Ball and plate

2 Physical architecture

2.1 Ball and plate system

The ball-and-plate system is composed of a **12” touchscreen** lying on a metal support, two **servomotors** allowing 2DOF to the plate (pitch/roll), and four springs in the corners to dampen undesired oscillations, and reject spurious yaw rotations too. The overall platform architecture is represented in Figure 2.

Touchscreen
The touchscreen is resistive and features a 4-pin interface, which is described in [5]. The four pins, all of different colors, are identified according to Table 1.

Color	Touch PIN
Yellow	`TPIN0`
Orange	`TPIN1`
Red	`TPIN2`
Brown	`TPIN3`

Table 1: Touchscreen pins

It allows reading the current position as analog value, one coordinate at a time. To read x, by convention the coordinate associated to the short side, connect `TPIN0` to `Vref` (3.3 V) and `TPIN2` to `gnd`, then read `TPIN1`. Viceversa, to read y, connect `TPIN3` to `Vref` and `TPIN1` to `gnd`, then read `TPIN2`.

Servomotors

The servomotors are two identical ACE RC S1903 [3]. They are arranged orthogonally, so that the platform can pitch and roll within a range. The one mounted on the long side is denoted by `Servo0`, the other by `Servo1`.
Every motor has 3 pins, whose purpose is deducted by its color according to Table 2.

Color	Servo PIN
Black	5 V
Red	PWM
White	gnd

Table 2: Servomotor pins

Both servos work at PWM frequency of 50 Hz. The duty cycle operational range of each motor is affected by structural constraints of the platform: the pulse to achieve rest position or maximum tilt are determined empirically and listed in Table 3.

Parameter	Pulse (ms)
`Servo0` rest	1.31
`Servo0` min	0.95
`Servo0` max	1.60
`Servo1` rest	1.55
`Servo1` min	1.20
`Servo1` max	1.85

Table 3: Servomotor limits

Field mask

A cardboard mask is firmly joined to the metal support, so that only a fraction of the touchscreen surface can be trodden by the ball. This is to reproduce the presence of obstacles in the field. Although tightly coupled, the mask does not interfere with the touchscreen, nor affects its position measurements. The actual mask is visible in Figure 1.

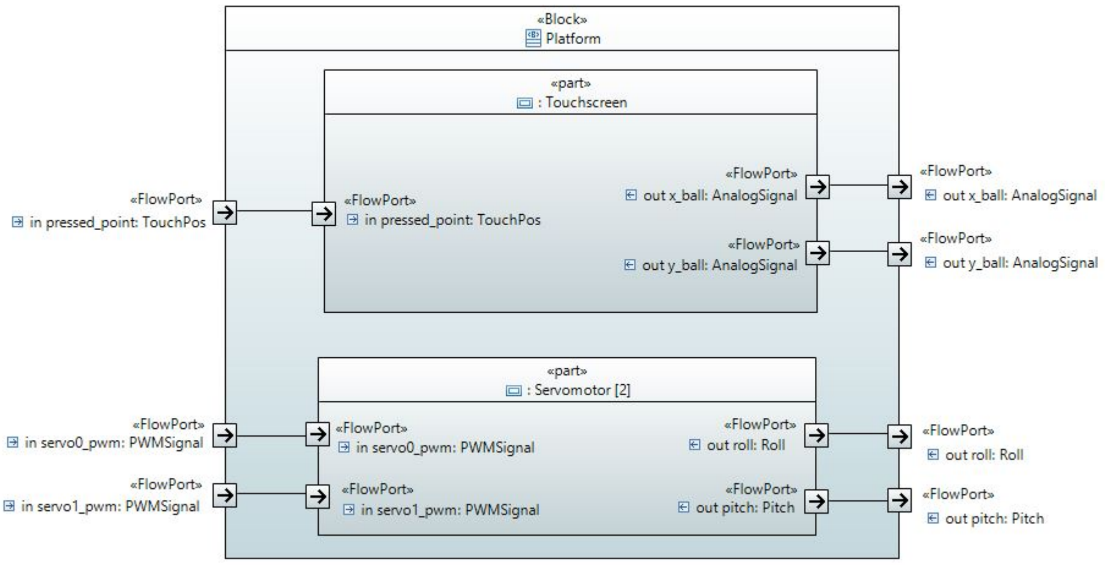

Figure 2: Physical internal representation of the platform subsystem

2.2 Hardware

The microcontroller is a **STM32F4 Discovery** board (Fig. 3), featuring 32-bit Cortex M4 processor (168 MHz clock), 1 MB ROM and 192 kB RAM [2].
Power is supplied through the micro-AB USB port (5 V DC).

GPIO lines are used to physically connect to the platform: the actual mapping between Discovery pins and sensors/actuators can be found in Fig. 4.
When polling the touchscreen for coordinates, the analog values are read by one of the integrated **analog-to-digital converter** of the board. Specifically, this project uses `ADC1`, which belongs to `APB2` (*Advanced Peripheral Bus*). `ADC1` pins are accessible through GPIO: channel 8 is mapped to `PB0`, channel 9 to `PB1`.
Servomotors PWM is regulated by timer `TIM4`, located on `APB1`: its channels 1 and 2 are connected to `PB6` and `PB7`, respectively.
Figure 5 provides an overview of the internal components and signals of the Discovery board; Fig. 6, instead, illustrates the exchange of signals with the platform.

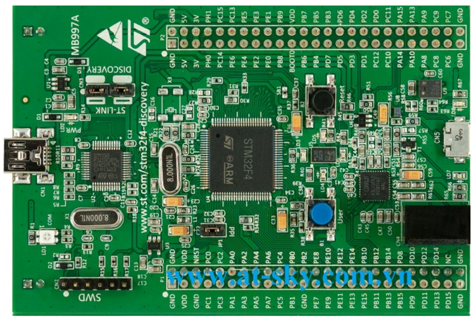

Figure 3: STM32F4 Discovery board

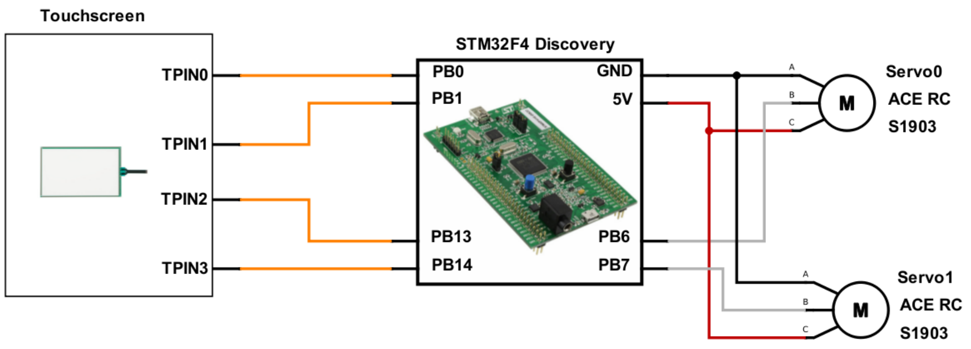

Figure 4: Connections between Discovery and peripherals

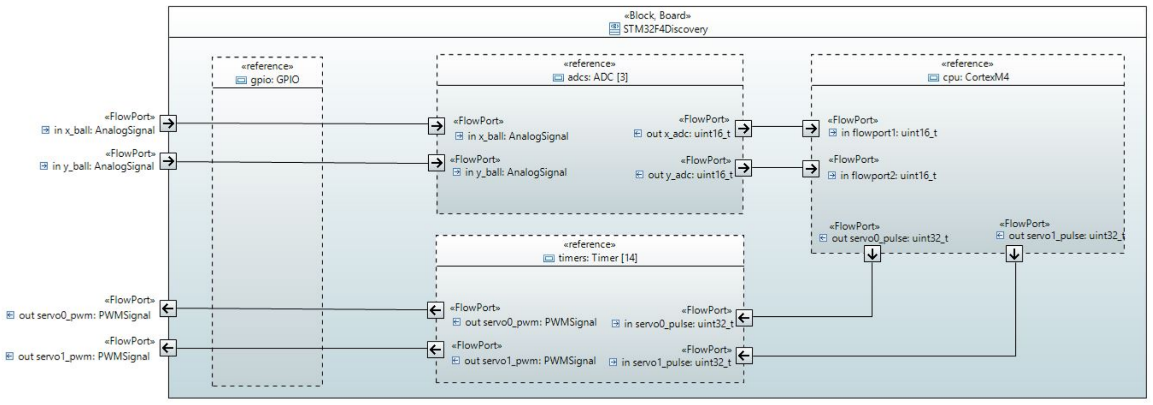
«Block, Board»
STM32F4Discovery
«reference»
gpio: GPIO
«reference»
adcs: ADC [3]
«reference»
cpu: CortexM4
«reference»
timers: Timer [14]
«FlowPort» in x_ball: AnalogSignal
«FlowPort» in y_ball: AnalogSignal
«FlowPort» out servo0_pwm: PWMSignal
«FlowPort» out servo1_pwm: PWMSignal
«FlowPort» in x_ball: AnalogSignal
«FlowPort» in y_ball: AnalogSignal
«FlowPort» out x_adc: uint16_t
«FlowPort» out y_adc: uint16_t
«FlowPort» in flowport1: uint16_t
«FlowPort» in flowport2: uint16_t
«FlowPort» out servo0_pulse: uint32_t
«FlowPort» out servo1_pulse: uint32_t
«FlowPort» out servo0_pwm: PWMSignal
«FlowPort» out servo1_pwm: PWMSignal
«FlowPort» in servo0_pulse: uint32_t
«FlowPort» in servo1_pulse: uint32_t

Figure 5: Internal representation of the board: all the signals coming from/to the board need to pass through GPIO

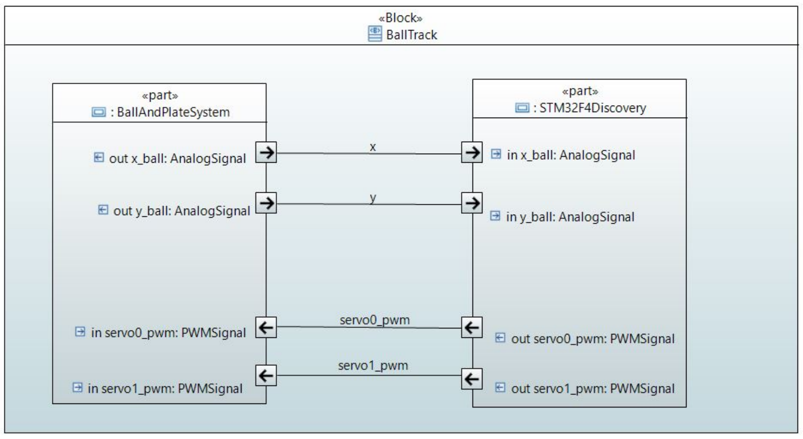

Figure 6: Physical communication between board and platform

The reinforcement learning model is trained on an external, more powerful desktop machine (Intel i7-7700 2.8 GHz); the final weights are then exported and statically hardcoded into the microcontroller control code (see Section 3).

2.3 Functional architecture

From a functional perspective, it is possible to distinguish between two distinct subsystems inside *BallTrack*: the *BallAndPlateSystem* and the *Controller*. The former takes as input a two-dimensional actuation command and returns as output the coordinates of the ball; viceversa, the latter reads these coordinates and generates the actuation command accordingly. This is represented in Figure 7.

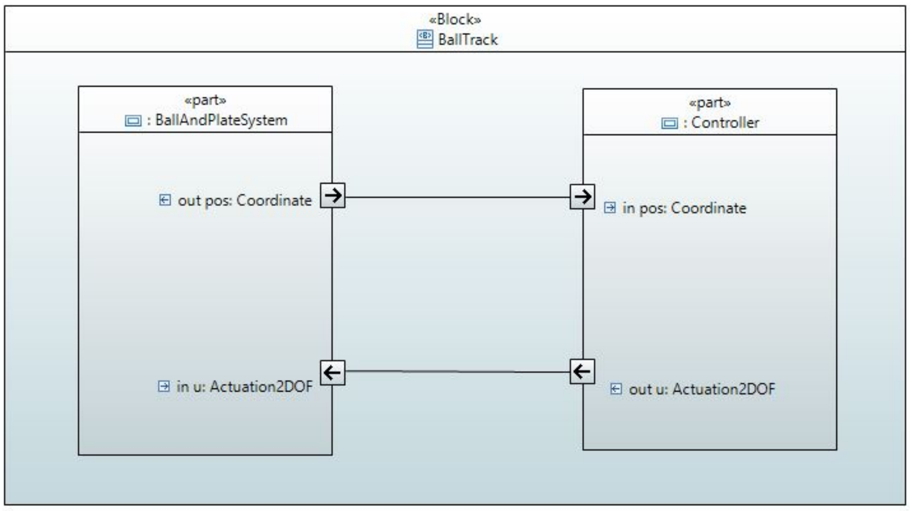

Figure 7: Functional representation of the whole system

Ball and plate system
BallAndPlateSystem offers an interface to read the current position from the touchscreen and regulate the angle of each individual servomotor, labeled with an ID. *Touchscreen* uses internally the interface offered by the *ADC* subsystem. Since both ADC and touchscreen can be faulty and return wrong readings from time to time, a function *ValidateRead* added: by comparing the current value with previous ones, it is often able to detect such issues; if so, the sample is simply ignored and no control action taken at all.

Controller
The *Controller* itself can be split into two parts: *Pilot*, which is responsible for determining, at high level, the optimal position to take as reference, and *PositionRegulator*, that is the automatic control model which ultimately computes the actuation value given current position and reference one. This is shown in Figures 9 and 10.
PositionRegulator implements proportional, derivative and integrative control actions, yet it is not a *PID*. The actuation u evolves according to the rule here below,

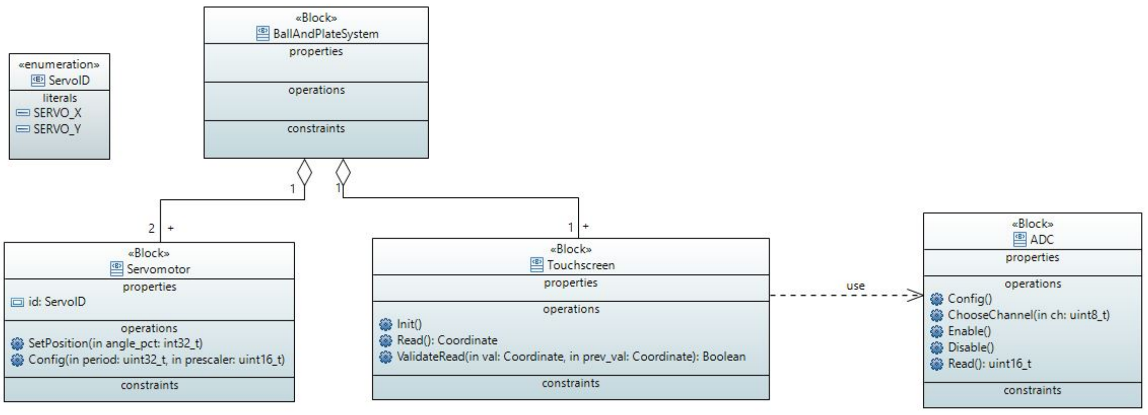

Figure 8: Representation of the ball and plate subsystem

where k denotes the k_{th} sample, e is the error and ι the cumulative error. K_p, K_d, K_a and K_i are parameters. Sampling is done at frequency 20 Hz.

$$\begin{aligned}
&u_k = K_p e_k + K_d(e_k - e_{k-1}) + K_a\left[(e_k - e_{k-1}) - (e_{k-1} - e_{k-2})\right] + K_i \iota_k \\
&\iota = \iota_{k-1} + e_k
\end{aligned}$$

The cumulative error, which helps eliminating the steady-state error, is actually limited to prevent the integrative action from getting too large and potentially slow to discharge. Being this component non-linear, the controller can not be expressed in transfer function form. However, ignoring it, the result would be the following TF:

$$\begin{aligned}
u_k &= K_p e_k + K_d(e_k - e_{k-1}) + K_a\left((e_k - e_{k-1}) - (e_{k-1} - e_{k-2})\right) \\
K(z) &= \frac{U(z)}{E(z)} = \frac{(K_p + K_d + K_a)z^2 - (K_d + 2K_a)z + K_a}{z^2}
\end{aligned}$$

The adopted parameter values are listed in Table 4.

Parameter	Value
K_p	0.035
K_d	0.5
K_a	0.05
K_i	0.0065

Table 4: Controller parameters

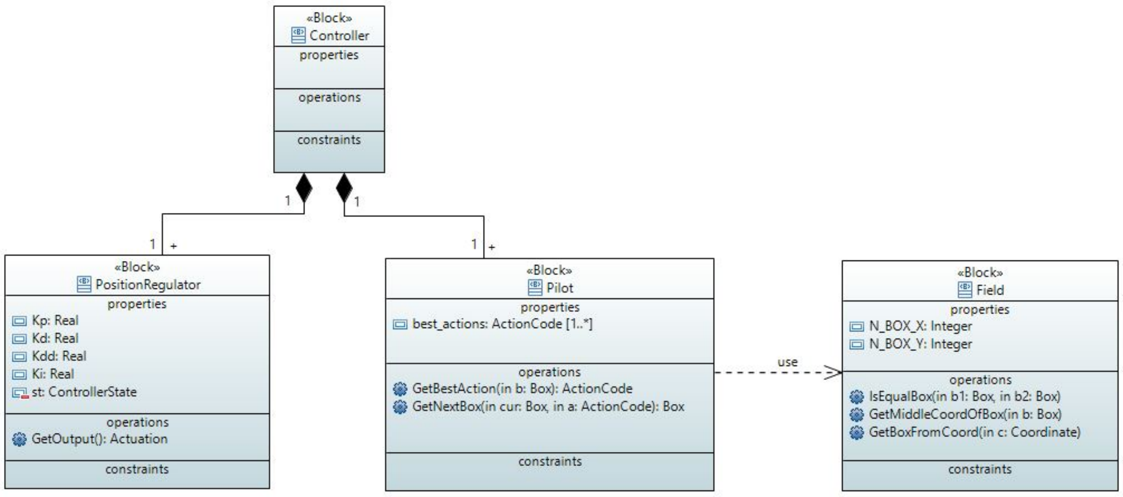

Figure 9: Representation of the controller

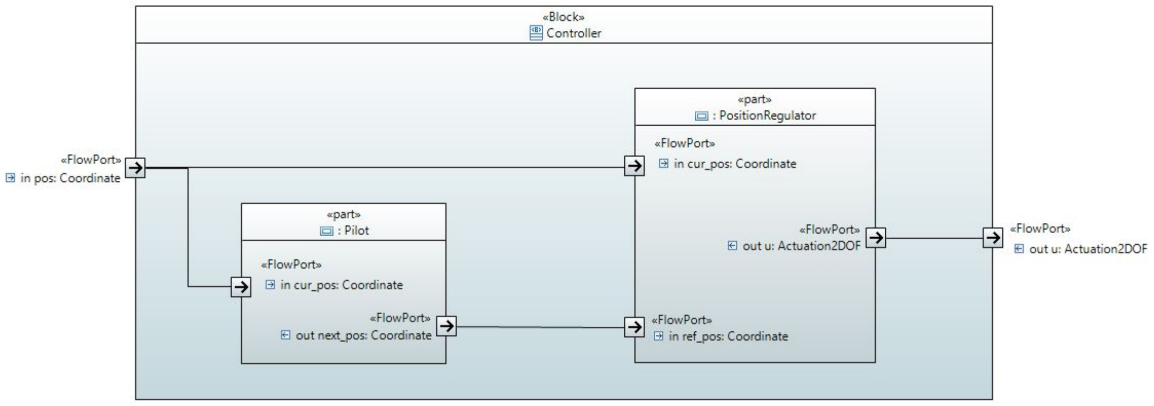

Figure 10: Internal representation of the controller

The Pilot subsystem constitutes a layer between the low-level automatic control performed by *PositionRegulator* and the reinforcement learning algorithm trained on a separate machine. Given the indices of the current state-box, it is able to return the code associated to the best control action to perform, as well as the indices of the next box according to that action. The conversion between state-boxes and coordinates, and viceversa, is delegated to the class $Field$, which represents the state-box spatial model of the AI.

3 AI

The ML core is based on **Q-learning**, a popular reinforcement learning algorithm [7]. Q-learning associates a quality value $Q(s,a)$ to every pair state-action (s,a); during the learning phase, this value is updated according to the Bellman equation:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a')\right]$$

where r is the immediate reward associated to that transition. α, γ and ϵ are hyper-parameters. The learning phase consists in iterating several times over state-action pairs: the state s is picked randomly every time, while the action to be performed is selected according to ϵ-greedy policy, where ϵ is called *exploration factor*:

$$\begin{aligned} n &= rand[0,1) \\ a &= \begin{cases} \text{random action} & \text{for } 0 \le n < \epsilon \\ \arg\max_a Q(s,a) & \text{for } \epsilon \le n < 1 \end{cases} \end{aligned}$$

Finally, the mapping between state-actions and rewards/punishments has to be defined *a priori*, and there are multiple ways to do so. In this application the adopted policy is simple: actions which directly lead to the target state are assigned a large reward, instead a punishment is returned when the ball takes a step (larger penalty if the new position is dangerous), hits the wall or any obstacle. Hyper-parameters, rewards and punishments together affect what the network learns and how fast.
For this specific context, states are generated discretizing the spatial coordinates with a sufficiently thin granularity. In particular, the longest side is partitioned into 27 segments, whereas the shortest one is divided in 21, for a total of 567 possible states. It should be also observed that the ball diameter is not negligible in practice (about 2 cm), therefore choosing a box size much smaller than ball radius would make little sense. The possible actions are just 8, and correspond to rolling one box UP/DOWN/ LEFT/RIGHT, or diagonally. Remaining still is not allowed.
With respect to other ML approaches, Q-learning has a very low time- and space-complexity (no layers, no matrix multiplications), a robust theoretical background [6] and is quite easy to implement. The hardest part is the design of states, actions and rewards, which is non-trivial and depends on the specific problem.

3.1 Rewards

Rewards and punishments values are lister in Table 5.
The adopted policy is defined so that positive rewards are rare and occur only at the end of an episode. While “halfway” rewards might speed up the localization of good paths, it’s very difficult to design them in a way that doesn’t result in the vehicle cycling through a loop of states to take the same reward again and again. Simply forbidding the ball to take the same reward twice would make the Q-matrix update procedure not time-independent (now depends on past actions too), thus less deterministic and more complex to handle.

Event	Reward
Final state reached	+500
Collision with object	-200
Step to dangerous box	-40
Step near dangerous box	-10
Simple diagonal step	-3
Simple axial step	-2

Table 5: Rewards and punishments

3.2 Hyper-parameters

The hyper-parameter α is called *learning rate* and determines how much the previous Q value affects the new one: a value of 0 prevents the network from learning anything, while 1 means that it only considers the newest information. γ, on the other hand, is called *discount factor* and weights the importance of future rewards: a value of 0 makes it consider only the current reward, 1 favors long-term rewards. The chosen values for *BallTrack* are listed in the Table 6.

Parameter	Name	Range	Used
α	Learning rate	[0,1]	1
γ	Discount factor	[0,1]	1
ϵ	Exploration factor	[0,1]	0.3

Table 6: Hyper-parameters

3.3 Training

The training phase ends after a fixed number of iterations, in this case 10000, which is more than ten times the product of number of states by actions. In this way, each pair state-action is visited and updated multiple times, allowing the network to converge to the optimal values. On a decent desktop machine, the neural network requires between $10\,\mathrm{s}$ and $1\,\mathrm{min}$ to fully train.
At the end of the process, it is possible to visualize what it learned by displaying the best action (according to the trained model) from each box, as shown in Figure 11. White areas represent obstacles (it corresponds indeed to the shape of the cardboard mask), instead arrows point to the next box to reach. The target box, in the bottom left corner, is represented by a circle.

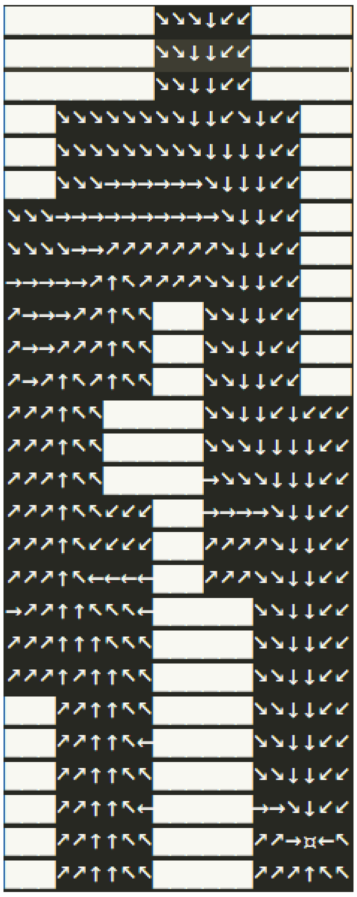

Figure 11: Trained network: best action for each state box

4 Software overview

4.1 Programming language

The microcontroller code is entirely written in **C**, interfaced with the OSEK compliant RTOS **Erika OS** by Evidence [1]. The machine learning script used to generate weights (best actions), which runs outside of the embedded environment, is instead written in **Python** for convenience.

4.2 Naming conventions

In the C code, the following naming conventions are adopted.

- *Variables*: all variables shall be declared using `snake_case`, typical of C.
- *Functions*: all functions whose scope is not limited to the current file shall be declared using `lowerCamelCase`, in order to keep consistent with the other Erika and CMSIS functions. Also, they should include as prefix the logical name of the module they belong to, followed by underscore (for instance, *Servo_* is the prefix for servomotors related functions). Instead, other auxiliary functions shall adopt `snake_case`, without additional constraints.

– *Files*: all files shall be named in compliance with `snake_case` style.

4.3 Code organization

The code is logically distributed over multiple files, with descriptive names often corresponding to those seen in the previous sections. File `conf.oil` is the configuration file for Erika OS, containing information about the used hardware, scheduling and task declarations. The purpose of other source files is summarized in Table 7.

File	Content
`adc`	manipulate the integrated analog-to-digital converter
`controller`	compute the actuation value given reference position
`field`	convert coordinates to state-box and viceversa
`main`	initialize board, OS, resources and tasks
`pilot`	find the best action from current position
`serial`	communicate via USART for debugging
`servo`	regulate the angular position of servomotors
`touch`	read the ball position from the touchscreen

Table 7: Source files

4.4 Threads

The application is designed to run on a single-thread, because all control steps are equally important (i.e., no priority) and, most importantly, must be performed one after the other in a precise order. Since this control task is only minimally parallelizable, multi-threading would be of no benefit in this case, let alone on a single-core platform. This only task is activated periodically, every 50 ms; the period value was chosen taking into account the expected characteristics of the system (1 s settling time as order of magnitude), and ensuring that the required computation time is always less than the period. A way to measure the computation time is, for instance, by the means of an oscilloscope, observing the output signal of a GPIO pin that is set at the beginning of the task and reset at the end.

5 Tests

Multiple parts of the final implementation were tested, both hardware and software, including:

- Collision avoidance: experiments show that the ball almost never hits the obstacles in a hard way. Soft collisions occur sometimes, mostly because of hardware failures or prototype inaccuracies.
- Speed: the ball travels quite fast, generally managing to reach the destination from any starting point in less than 15 seconds.

- Servomotor limits: a boundary-value analysis, including robustness tests, has been performed on the functions to regulate servomotor angles, using *CUnit* the framework [4]. The code successfully passed all the tests.

6 Issues

Here is a brief list of the issues encountered during the physical implementation of the project. Most of these can be simply avoided by using better hardware.

- Some tiny regions of the touchscreen are insensitive and consequently return wrong position values, as if the ball was not present. A low-pass filter on position error variations is able to mitigate problems arising from this defect most of times.
- The field mask is very rough and poorly cut, also its positioning is a bit sloppy. Thus, the physical prototype, despite corresponding well enough to the model, is not 100% exact. Using better materials and cutting tools can definitely help to obtain a better system.
- Wires are plugged onto GPIO pins loosely, because of their low-quality manufacture. Soldering them would completely solve this connection instability.

References

[1] Evidence. *ErikaOS*. http://erika.tuxfamily.org/drupal/.

[2] ST Microelecronics. *STM32F4 Discovery board*. https://www.st.com/en/evaluation-tools/stm32f4discovery.html.

[3] ServoDatabase. *ACE RC 1903 Specs*. https://servodatabase.com/servo/ace-rc/s1903.

[4] Sourceforge. *CUnit*. http://cunit.sourceforge.net/.

[5] Sparkfun. *4-pin touchscreen interface*. https://www.sparkfun.com/datasheets/LCD/HOWDOESITWORK.pdf.

[6] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[7] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. PhD thesis. King’s College, Cambridge, 1989.