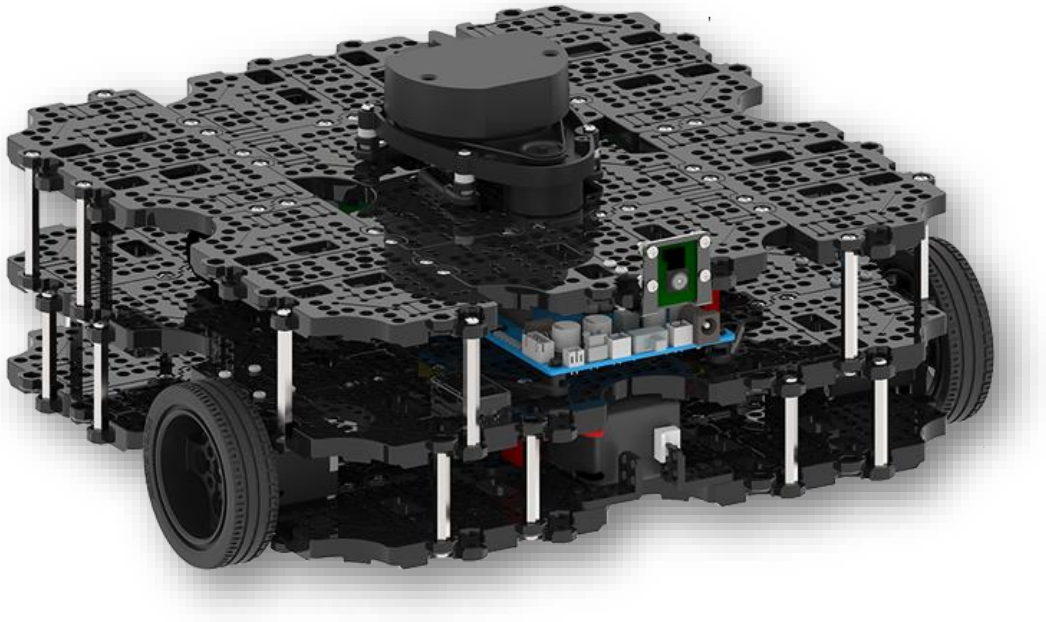
Erwan
DUHAMEL

Jean
SOUDIER

GE5 S-IoT

Speciality project

AI applied to number recognition and robot control



Revision: 11/01/2021

Professors: Sylvain **DURAND**
Renaud **KIEFFER**
Ahmed **SAMET**
François **DE BERTRAND DE BEUVERON**
Floriane **BALLOT-BUOB**

School year 2020/2021

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES STRASBOURG

Erwan
DUHAMEL

Jean
SOUDIER

GE5 S-IoT

Table of contents

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES STRASBOURG

Erwan
DUHAMEL

Jean
SOUDIER

Introduction

Our specialty project consists in detecting numbers through a recognition algorithm before writing them using a robot arm on a distant whiteboard.

The system being quite complex, creating it involves the knowledge from several fields (Big Data / Machine Learning techniques, multitask system synthesis, image processing, mechanical CAD, …) that are luckily taught this year in class.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES STRASBOURG

Specifications

Task: Create a system that recognizes hand-written numbers using an AI and then writes it on a distant whiteboard.

AI design requirements:

- The AI must be able to recognize any number from 0 to 9 (inclusive).
- The AI must be able to recognize commas and several-digit numbers.
- The AI recognition algorithm must be fast- and reliable-enough to be used for general purpose applications.

Robot design requirements:

- The robot must be able to write on a whiteboard the number detected by the AI using an adapted pen.
- The robot must be able to move from its original position to the board on itself, and then move back to it without crashing into walls or people on the way.

Miscellaneous:

- The system must be nice looking and presentable during the open day.
- The system must be able to give a real-time feedback to the user.

Additional objectives:

*These objectives **were planned to be added** if time allowed it.*
They are ordered by order of priority.

- Detection of expression of format *x op y* and parsing.
- Equation detection system, parser and solver.

Functional analysis

Main functions:

- Recognition of hand-written numbers
- Writing of the detected number.
- Real-time or pseudo-real time user feedback.

Additional functions:

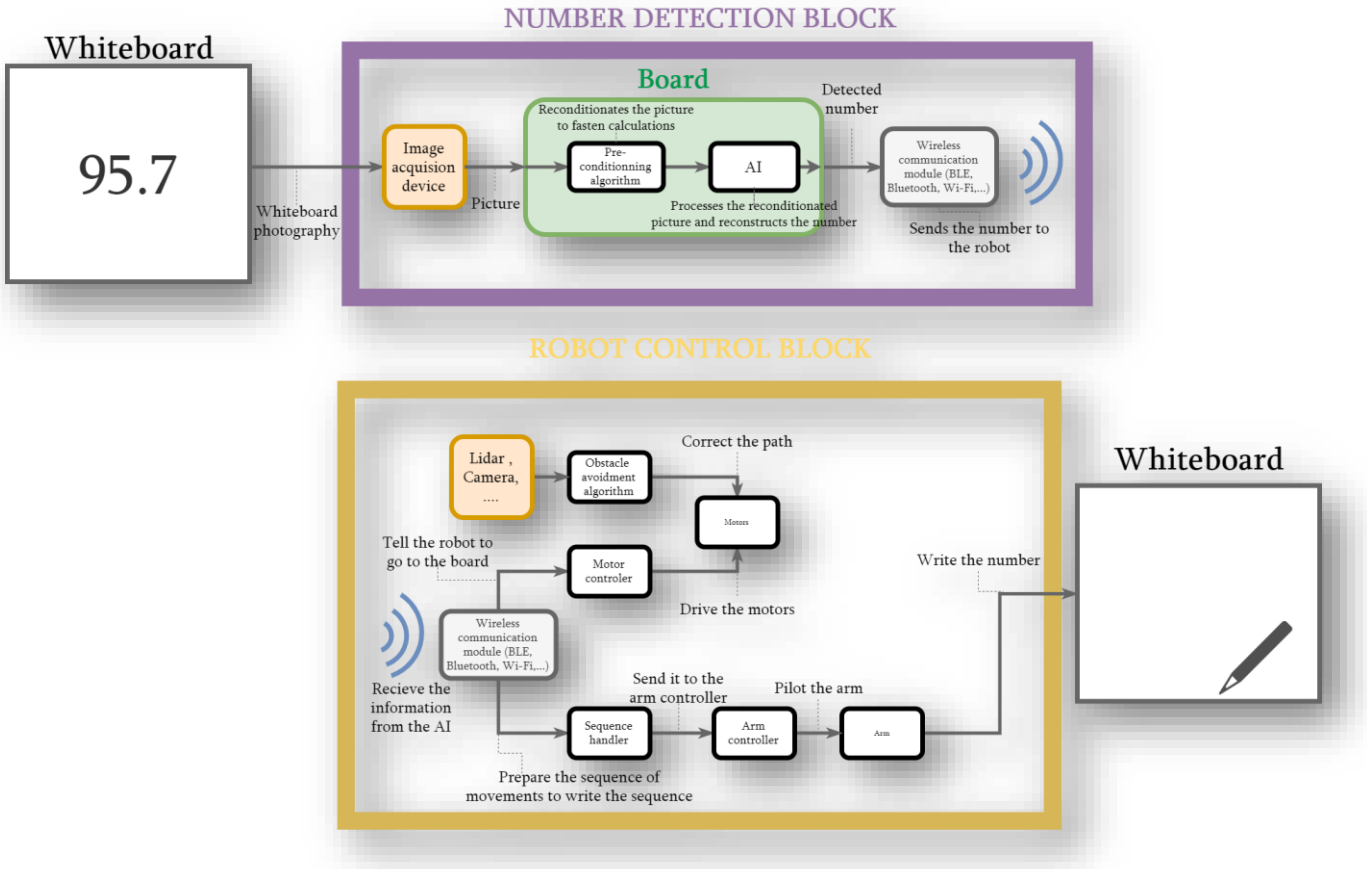
- Must be nice-looking.
- Wireless communication between the two parts of the system.

Constraints:

- Execution time must be limited to its minimum.
- Must be completely safe for the user.
- Must have a user-friendly interface.

- **Functional diagram:**

N.B.: *This diagram is **not** the final version.*



Erwan
DUHAMEL

Jean
SOUDIER

GE5 S-IoT

Equipment

TurtleBot

Introduction:

TurtleBot is a general-purpose robot kit with open-source development software. There are several versions of these (8 including the original discontinued one) that come with different pre-installed equipment.
The model we use (TurtleBot Waffle Pi) comes with a Lidar and a camera in addition to its motors and the boards required to drive them.

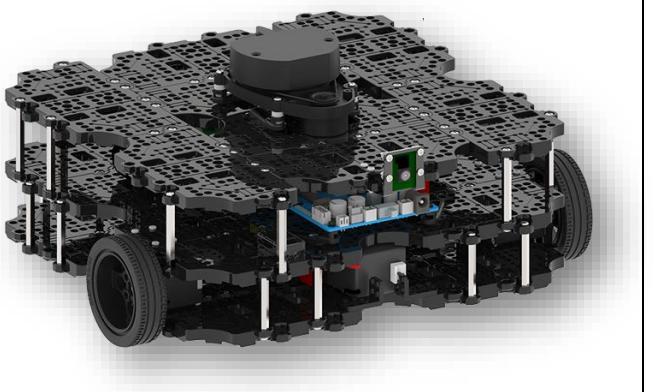


Illustration: *Cyber Robotics Technology Limited*.

Characteristics:

Name	Data
Maximum translational velocity	0.26 $\mathbf{m.s^{-1}}$
Maximum rotational velocity	1.82 $\mathbf{rad.s^{-1}}$
Maximum payload	30 **kg**
Size	281 **mm** x 306 **mm** x 141 **mm**
Weight	1.8 **kg**
MCU	32 – **bit** *ARM Cortex® – M7 with FPU* (216 **MHz**, 462 *DMIPS*)
Battery	*LiPO polymer* 11.1 **V** 1800 **mAh** 19.98 **Wh** 5*C*

Source: *Robotis e-Manual*, **Robotis**

Equipment

Widow XL Robot Arm

Introduction:

A Widow XL is a robot arm made out of DYNAMIXEL servo-motors designed to allow very precise movements thanks to their very high resolution.
The device also comes with a sort of tool-kit that allows the user to define his own correctors for the application, making it very flexible.

Illustration: *RobotSavvy*.

Characteristics:

Name	Data
Model	*WidowX*
Weight	1.33 **kg**
Vertical reach	51 **cm**
Horizontal reach	37 **cm**
Strength	30 ***cm*** / 400 ***g*** ; 20 ***cm*** / 600 ***g*** ; 10 ***cm*** / 800 ***g***
Gripper strength	500 **g**
Wrist lift strength	500 **g**

Source: *WidowX*, **ROS components.**

Equipment

Jetson Nano

Introduction:

A Jetson Nano is a development board produced by Nvidia that aims to facilitate the development and deployment of fast, modern and well-trained AIs on embedded environments. It is equipped with modern GPU and CPUs, supports Ubuntu, and may thus be used as a low-cost personal computer.

Illustration: *Nvidia*.

Characteristics:

Name	Data
GPU	$128 - core\ Maxwell$
CPU	$Quad - core\ ARM\ A57$ (1.43 **GHz**)
Memory	4 **GB** $64 - bit\ LLPDR4$ 25.6 **GB**. $\mathbf{s^{-1}}$
Storage	$microSD$
Display	$HDMI\ , DP$
Other ports	$USB, GPIO\ , I2C\ , I2C\ , SPI\ , UART$
Dimensions	100 **mm** x 80 **mm** x 29 **mm**

Source: *Jetson Nano*, **Nvidia.**

Equipment

Raspberry PI

Introduction:

A Raspberry PI is a system very similar to a Jeston Nano (historically much older), but with lower specifications. It also supports Ubuntu, and is also equipped with a CPU. Recent versions with a GPU are also available. Several models exist, with different specifications for different budgets.

Illustration: *Distrelec*.

Characteristics:

For Raspberry 4 boards:

Name	Data
GPU	$VideoCore\ VI\ 3D\ Graphics$
CPU	$Quad-core\ ARM-Cortex\ A72$ (1.5 **GHz**)
Memory	1 *to* 4 **GB** $64-bit\ LLPDR4$ 25.6 $\mathbf{GB.s^{-1}}$
Storage	$microSD$
Display	$HDMI$
Other ports	$USB, GPIO, I2C, I2C, SPI, UART$

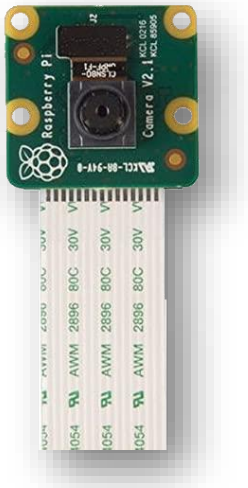
Source: *Raspberry Pi 4 Model B datasheet*, **Raspberry Pi Org.**

Equipment

Raspberry PI Camera module

Introduction:

A simple camera module intended for Raspberry boards, but actually usable on any that supports Ubuntu and with the appropriate connectors. Three models are produced: the V1 one, the V2 one and the HQ one (here ordered by rising image quality). It is designed for general purpose photography and video, and is thus equipped with an infra-red filter that can be removed if needed.
Very light (around 3 **g**), it is well-suited for embedded applications.



For V2 cameras.

Illustration: *Amazon*.

Characteristics:

Name	Data
Size	*Around* 25 x 24 x 9 **mm**
Sensor	*Sony IMX*219
Resolution	3280 x 2464 **px**
Pixel size	1.12 µ**m** *x* 1.12 µ**m**
Horizontal field of view	62.2 °
Vertical field of view	48.8°

Source: *Camera Module*, **Raspberry Pi Org.**

Design solutions

Number detection system

Generalities:

Such a system requires a quite significant computing speed and power, making it impossible or at least very inefficient if not running on an adapted system.
A Raspberry PI is therefore incredibly not adapted for this application, justifying the use of a powerful Jetson Nano.
We moreover need a GPU to process images, as it fastens processing for any image reasonably big enough to compensate for the lag due to having to copy the data onto the GPU (RAM is not shared between CPU and GPU).

Program organization:

To provide a visual feedback, a graphical user interface is be provided. It features the following functions:

- Real-time rendering of camera image.
- Display of the different steps of the number detection algorithm.
- Various information about the project.

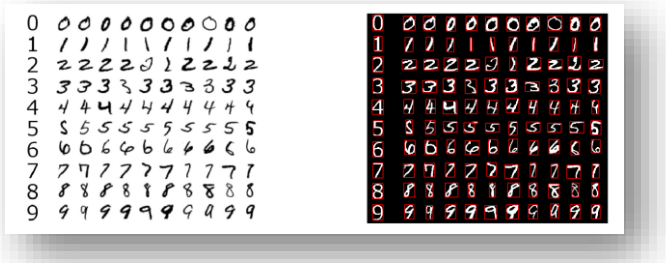
General considerations about data pre-conditioning:

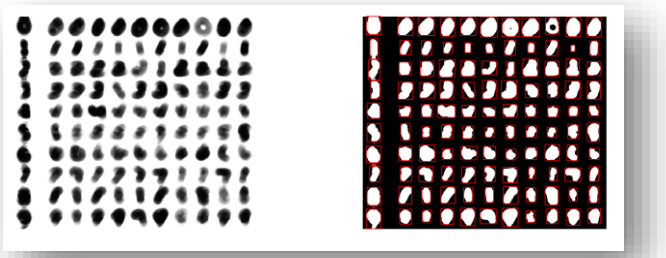
In practice, AI-training sessions will always use patterns that are specifically made to accelerate the learning process. This thus means that real-life usage will require a pre-conditioning algorithm to format the data to a state that is as close to the learning patterns as possible.
Such a program requires either another neural network (which reconstructs an image from the original one) or well-known image processing algorithms that clean the picture, contract it, reduce noise influence and split it into smaller parts to facilitate recognition.
We will obviously choose the second option, as it requires consequently less data.

Data pre-conditioning algorithm:

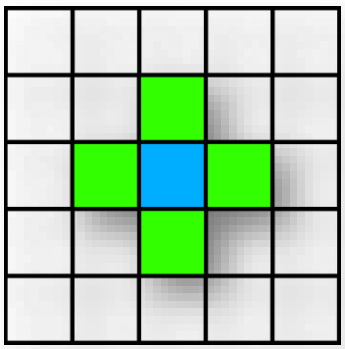
Area splitting:

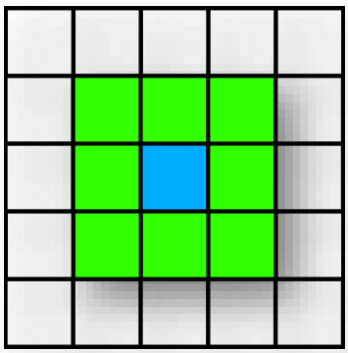
This algorithm assumes an image with low noise, but does not make any assumption on the blur.





This uses the usual 8-connectivity algorithm, that splits an image into several parts connected by eight pixels or more. This method is a bit more permissive than a 4-connectivity algorithm, as it enables the detection of more complex forms, which is what we want.
The pictures below describe the basis of the two versions: the green tiles are the one considered connected to the blue one. On the left, the 4-connectivity, and on the right, the 8-one.





Design solutions

AI

Generalities:

Base concept:

The AI will be designed using machine-learning techniques. This choice is motivated by two main factors: the speed requirement and the reliability one, both imposed by the specifications.
Other design lines such as brutal cross-correlation detection would produce both performance drop and programming overhead, and thus cannot be considered a reasonable solution.

Training:

We originally planned to train the AI using Google TensorFlow (their tool for machine learning). It turns out that it is actually incompatible with our board: indeed, they use an out-of-date version of the GPU Toolkit, which cannot run on a JetsonNano.
We therefore decided to implement it ourselves, from scratch.

Structure:

The most adapted structure is the one of **convolutional neural networks** (also known as **shift invariant neural network** or CNN).
To put it in a nutshell, it is a multi-layer network in which a neuron of one layer is connected to all the ones of the next layer. The output is computed as usual: by weighting each neuron correctly and computing an activation function for every neuron.

- **Network architecture:**

  • An input layer, which contains as many neurons as the input pictures contain pixels (thus $28 * 28 = \mathbf{784}$ cells as MNIST picture are $28 * 28$ pixels).

  • A hidden layer, which contains **80** cells.

  • Another hidden layer, with **60** cells

  • An output layer, which contains **10** cells, as for one per number between **0** and **9**.

- **Neuron architecture:**

  Representation:

  A neuron in our system will be defined as such:

  • A positive non-zero natural number of inputs. These are real numbers (rational on computers) that can hold any value in [0 ; 1].

  • A bias, which is a real number.

  • A vector a real number which each coordinate represents a weight for an input.

  • One output that can be any number in [0 ; 1] computed through a function named "**activation function**" depending on the weights, the inputs and the bias.

Construction and description:

Let $\boldsymbol{n} \in \mathbb{N}^*$ be the number of inputs in a neuron.
Let $\boldsymbol{w} \in \mathbb{R}^{\boldsymbol{n}}$ be the vector containing the weights for each input of a neuron.
Let $\boldsymbol{a} \in [\boldsymbol{0}\,;\boldsymbol{1}]^{\boldsymbol{n}}$ be the vector containing the inputs value of a neuron.
Let $\boldsymbol{b} \in \mathbb{R}$ be the bias of a neuron.

Let $\boldsymbol{z}$ be a real number so that:

$$z = < w|a > + b$$

Where $<\bullet \mid \bullet>$ is the usual dot product of two vectors that defines the norm $||\bullet||_2$.

· **Activation function**: *Exponential Linear Unit (ELU)*

Let $\boldsymbol{z}$ be the real number defined as above.
Let $\boldsymbol{\alpha}$ be a positive real number.

$$\forall z \in \mathbb{R}, f(z) = \begin{cases} \alpha\,(\exp(x) - 1) & if\ x \leq 0 \\ x & otherwise \end{cases}$$

Note that f has the following properties:

$$f\ is\ C^0\ on\ its\ definition\ interval, or\ C^1\ if\ \alpha = 1$$
$$f\ is\ piecewise\ C^\infty$$
$$\forall x \in \mathbb{R} \cup \{\pm\infty\}, f(x) \in [-\alpha\,; +\infty]$$
$$\lim_{x\to\infty} f(x) = +\infty\ ;\ \lim_{x\to-\infty} f(x) = -\alpha$$
$$\forall x \in \mathbb{R}, \frac{df}{dx}(x) = \begin{cases} \alpha \exp(x)\ if\ x \leq 0 \\ 1\ if\ x > 0 \\ 1\ if\ x = 0\ and\ \alpha = 1 \end{cases} ;\ \forall x \in \mathbb{R}, \frac{df}{dx}(x) \in [\alpha\,; 1]\ or\ [1\,; \alpha]$$
$$f\ is\ monotonic\ if, and\ only\ if\ \alpha \geq 0$$
$$\frac{df}{dx}\ is\ monotonic\ if, and\ only\ if\ 0 \leq \alpha \leq 1$$

In our case, $\boldsymbol{\alpha} = \frac{\boldsymbol{1}}{\boldsymbol{5}}$, and thus both f and $\frac{df}{fx}$ are monotonic.

· **Number of inputs**: same as the number of cells of the previous layer.

Note that other activation functions also exist. Among them, the famous sigmoid function, the hyperbolic tangent, etc.

Network initialization:

When a neural network is created, it is necessary to initialize each weight for each neuron. This, however, cannot be and should never be made using poor random, as it will have disastrous consequences on the training performances, especially for the ones relaying on back-propagation using stochastic gradient descent or anything similar (which is the case for almost all neural nets).

It is indeed possible to show that any activation function regular enough fed poorly initialized weights will either lead to an exploding- or to a vanishing-gradient problem which are plagues for every neural net.

The appropriate conditions for optimal back-propagation in feed-forward neural nets have been prooved by **Xavier Glorot** and **Yoshua Bengio**, in their paper *Understanding the difficulty of training deep feedforward neural networks*[1]. They can be simplified into these two guidelines:

- The expected value (in the probabilistic meaning) of each activation must be **0**.
- The variance of the activations must **remain identical across** layer.

Using equations:

Let $L \in \mathbb{N}^*$ the total number of layers.
Let $a^{[l]}$ be the activations of a layer l.
Let $n^{[l]}$ be the number of neurons in layer l.

$$\forall l \in [|0\,;L|], E\left(a^{[l]}\right) = 0$$
$$\forall l \in [|1\,;L|], Var\left(a^{[l]}\right) = Var\left(a^{[l-1]}\right)$$

This leads to initialize the weights $W^{[l]}$ and bias $b^{[l]}$ using the following guidelines:

$$W^{[l]} \sim \aleph\left(0\,, \frac{1}{n^{[l]}}\right)$$
$$or\ W^{[l]} \sim \aleph\left(0\,, \frac{2}{n^{[l-1]} + n^{[l]}}\right)$$
$$or\ W^{[l]} \sim \boldsymbol{U}\left(\frac{-\sqrt{6}}{\sqrt{n^{[l-1]} + n^{[l]}}}\,, \frac{\sqrt{6}}{\sqrt{n^{[l-1]} + n^{[l]}}}\right)$$
$$\forall l \in [|1\,;L|], b^{[l]} = 0$$

Where:

$\aleph(\mu\,,\ \sigma^2)$ refers to the normal distribution of expected value μ and variance σ^2.
$\boldsymbol{U}(a\,, b)$ refers to the uniform distribution in $[a\ ; b]$.

Network training:

Training a neural network means tuning the neuron parameters (weights and bias) in order to minimize the cost function, which indicates how precise the predictions are.

There are numerous existing methods to train a neural network, but the main 4 are based on the same principle, which is error backpropagation. The following paragraphs describe in a few words how they work.

Notations:

- J : cost
- W : weights
- α : learning rate
- n : integer (epoch number)
- $dW = \frac{\partial J}{\partial W}$: variation of the cost relative to the parameter variations
- β, β_1, β_2 are real numbers.
- $\epsilon \in \mathbb{R}_+^*$ so that $\epsilon \ll 1$.

Stochastic gradient descent (SGD):

Number of parameters: 1
This method is the simplest one: it simply consists in approximating the derivative of the cost function, and "slide down" until a minimum is reached.

It originates from a specific part of calculus, *differential calculus*, in which the properties of multi-variable functions are studied.
In our case, doing it with calculus and not numerically would expand computation time by a few million years, which is very likely to not be appreciated.

SGD with momentum:

Number of parameters: 2
Stochastic gradient descent with momentum is an extension of the previous method. It incorporates an intermediate term in the equations, that is meant to simulate a momentum of a ball sliding down a slope (keeping the same example as before).

It keeps in memory the previous variation, and computes the next one as a linear combination of the gradient and this previous update
SGD with momentum provides faster convergence than simple SGD, but much less stable, and thus extremely sensible to the initial state and its parameters.

RMSProp:

Number of parameters: 3
RMSProp is the abbreviation of **R**oot **M**ean **S**quare **P**ropagation (method).
It is meant to adapt learning rate at each iteration in order to find the global minimum of the cost function much more precisely than the previous ones.

Adam [2]:

Number of parameters: 4
Adam is again an abbreviation, for **Ada**ptative **M**oment Estimation Method.

It combines the advantages of all the previous methods, and is thus very frequently the optimizer chosen for global purpose or specific purpose neural nets.
Quite recent (2015), it rapidly gained popularity as its implementation does not differ much from the RMSProp one.

Equations:

SGD	SGD with momentum
$W_{n+1} = W_n - \alpha * dW$	$V_{dW_{n+1}} = \beta V_{dW_n} + (1-\beta)dW$ $W_{n+1} = W_n - \alpha * V_{dW_{n+1}}$
RMSProp	**Adam**
$S_{dW_{n+1}} = \beta S_{dW_n} + (1-\beta)dW^2$ $W_{n+1} = W_n - \alpha * \frac{dW}{\sqrt{S_{dW_{n+1}}} + \epsilon}$	$V_{dW_{n+1}} = \beta_1 V_{dW_n} + (1-\beta_1)dW$ $S_{dW_{n+1}} = \beta_2 S_{dW_n} + (1-\beta_2)dW^2$ $W_{n+1} = W_n - \alpha * \frac{V_{dW_{n+1}}\sqrt{(1-\beta_2^{n+1})}}{(1-\beta_1^{n+1})\sqrt{S_{dW_{n+1}}} + \epsilon}$

N.B.: the notations dW and W may seem ambiguous as it may be considered to only refer to the weights and not the bias.
It is not the case. A clearer but heavier notation would look like this:

$$W_n = (w_n ; b_n)$$

Where w_n reflects the weights evolution and b_n the bias one.

Network representation and implementation [3]

General considerations:

Cleverly representing a neural network is not actually as trivial as it may look like, especially if performance is critical in the application.
The usual, intuitive way beginners create a neural network is either by using pre-made libraries (which is perfectly fine if comprehension is not required), or layer by layer and neuron by neuron.

This is unfortunately incredibly inefficient, and this is why the following representation is used.

Model:

Definitions and notations:

Let N be the number of layers of the neural network (the input layer is not taken in count).

Let $(C_n)_{n\leq N}$ be the sequence so that each term represents the number of neurons in the layer n. C_0 is the size of the input.

Let $\{W_n\}_{n\leq N}$ and $\{B_n\}_{n\leq N}$ be the two sets of matrices constructed so that:

• Each W_n represents the weights of the neurons in layer n, so that each term $W_{i,j}^{(n)}$represents the weights of the j^{th} input of the i^{th} neuron of the n^{th} layer.
• Each $B_{i,1}^{(n)}$ represents the bias of the i^{th} neuron of the n^{th} layer.

$$\forall n \in [|1\ ;\ N|],\ W_n \in \mathcal{M}_{C_n \times C_{n-1}}(\mathbb{R})$$
$$\forall n \in [|1\ ;\ N|],\ B_n \in \mathcal{M}_{C_n \times 1}(\mathbb{R})$$

Let $(f_n)_n$ be the application sequence in $\left(\mathcal{M}(\mathbb{R}) \to \mathcal{M}(\mathbb{R})\right)^n$, the sequence of the **activation functions of the network's layers.**

Let I be the input of the neural network.

Theorem:

A convolutional neural network is integrally described by two sets of matrices of size (number of layers) and set of applications in $\mathcal{M}(\mathbb{R}) \to \mathcal{M}(\mathbb{R})$.

Proof:

Trivial. Only relies on:

- The representation of each neuron explained before
- The link between matrix product and dot product.

Property:

The output of a neural network represented as such is obtained by computing the sequence $(R_n)_{n<N}$ defined by:

$$R_0 = I$$
$$\forall n \in \mathbb{N}, n < N\ , R_n = f_n\big((W_n * R_{n-1}) + B_n\big)$$

The output is then simply the last term of the sequence.

Proof:

Trivial. Strictly analogue to the theorem proof.

Note that these definitions, this theorem and this property are only applicable with the previous definition and construction of a single neuron, and are merely a way to represent the network using the most appropriate way.

Regarding the benefits of the matrix implementation

- The "naïve" implementation is incredibly slow, as it requires to loop through all layers AND through all the neurons.
  It also is impossible to gain computation time using a GPU. Indeed, data redundancy is close to zero, making it impossible to compensate for the time lost while copying data to the GPU RAM.

- On the other hand, the matrix implementation allows:
  → Simple code vectorization
  → Optimal GPU usage
  → More flexible structure
  → Simpler implementation

Design solutions

Arm control - Generalities

Control protocol:

The servo-motors are linked through a chained one-wire UART bus (which is not that common). The arm itself is designed to be controllable using the ROS protocol, which is designed to control several devices independently from a single central master device.

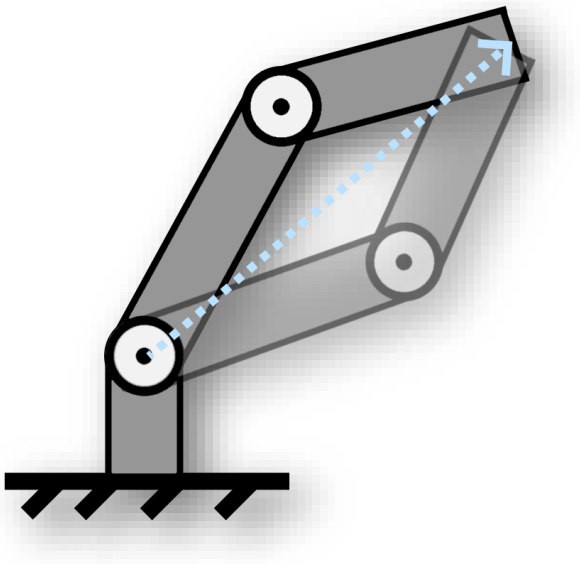
Implementation guidelines

To allow a more flexible and generic control, we decided to reimplement the protocol ourselves. It now uses parallel computing to reduce the time needed before one movement is made.

The implementation also should feature several safety measures that limit its positions to extrema computed at compile time.

Control logic:

Several algorithms exist to control an arm. Among them, one of the most efficient is the control through inverse kinematics which we decided to implement.
Such algorithms consist in determining the movement of each motor needed to reach a given point in space. Through basic maths and geometry in $\mathbb{R}^4$, it is possible to compute very easily (and more importantly, very quickly) the required command signals to send to the robot arm.



It is however not as simple as it looks like, as it is necessary to find the sequence of movements that minimize the energy used to reach a position through an infinity of possibilities.

Design solutions

Arm control - Drawing

Sequence model:

Computing positions in space at runtime is a very long and costly process for the processor or the GPU to which the task is given. It is also absurd to do so when the moves follow known patterns that only vary from a translation (same for humans: writing "4 "on a paper sheet and writing it on another one next to it *should* not require to think that much).

We thus decided to make another program (that is not supposed to run on the system but on a computer) which generates both XML and CSV files describing a sequence of points. In practice, its main goal is to provide a 3D preview for a visual confirmation.

Implementation:

Due to compatibility problems with our Raspberry board, only the CSV files are used. Parsing is made using plain C++, but provides the same performances as the Qt-C++ version.

File generation:

Two possibilities to generate the files containing the sequences:

- Use some software to place the points, and tune them in real life.
- Use a controller to move the arm, save the points, and tune them in some software.

The second one, much safer, is the one preferred for our application.

Constraints due to the usage:

Because the pen held by the arm must stay horizontal and perpendicular to the board, additional constraints must be taken in count.
Let $\{\theta_n \mid n \in [|1\,,6|]\}$ be the set of the angles of the n^{th} articulation (starting from the bottom).
The constraints can be expressed by:

$$\theta_1 = \theta_5$$
$$\theta_2 + \theta_3 + \theta_4 = \pi$$

Design solutions

Robot control

Generalities:

The robot itself is split into two parts: the motor-driving one and the arm-control one, the latter being described above.
The former is handled by the same multitask program as the arm, thus reducing the need of cross-board communication to its minimum. Both are however (almost) completely independent.

Control method:

The robot can be driven both in speed and in position. It detects the environment using a Lidar, that is also controlled with the same program (once again to reduce inter-board communication).
The Lidar data points are passed to an ICP algorithm, that determines the current position of the robot depending on its previous one.

Finally, a state-feedback control is applied, controlling the position of the system.

Program organization:

The program is organized into several threads (POSIX threads to be precise, which provide a very convenient platform-independent execution model) in order to provide full parallel execution. Such an organization is the only viable on a system that is required to act fast and to perform multiple tasks synchronously.

Several threads are used to control the robot:

- One is used for the Bluetooth communication handler
- A second is used to poll the Lidar at regular intervals
- A third one is used to compute an ICP algorithm on the Lidar points, and to control the motors so that the robot position is as intended
- A fourth and final one is consumed in the main program and moves the arm.

Positioning algorithm:

As the robot moves, it is necessary to determine its position in order to provide feedback for the control algorithm.

This is a complex problem, and the dedicated sensors are not as simple as speed ones, as they always require data processing. The most used are:

- Absolute coders
- Cameras
- Radar-like devices (a Lidar is one of them).

Our robot already being equipped by the last two types of sensors, adding another one would not make much sense, and explaining why camera-based positioning algorithm require a quite consequent work should not be necessary.

Among the possibilities using a Lidar, it was crucial to choose a fast algorithm that does not require much computing speed. Only a few of them fit these requirements, and the simplest one is named ICP, abbreviation of **I**terative **C**losest **P**oint.

Iterative Closest Point:

Note that we will restrain the explanations to 2-dimensional cloud of points for simplicity reasons.

The principle is very simple: between two pictures, find two matrices R and T as defined below, so that the distances between each point are minimized.

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \qquad T = \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

Where θ is the rotation angle between the two pictures, and T_{axis} the translation on the axis in index.

To do this, we proceed using brute force as below:

- For each point in the second cloud, find the closest one in the original one.
- Estimate $(\theta\ ; T_x\ ; T_y)$ using any adapted technique.
- Transform the source points with the obtained transformation.
- If the error is higher than a certain threshold, reiterate.

Note that these are simplified explanations.

Design solutions

Mechanical design - Supports

Generalities:

To partially replace the old mechanical structure made from scratch parts, a support for the camera and for the Jetson Nano used for the number detection is planned.

Design guidelines:

Pen:

The pen support must:

- Be able to hold a pen correctly (i.e., negligeable position fluctuations).
- Be light enough to not modify the arm mechanical response.
- Compensate for the effort applied on the board.

Camera support:

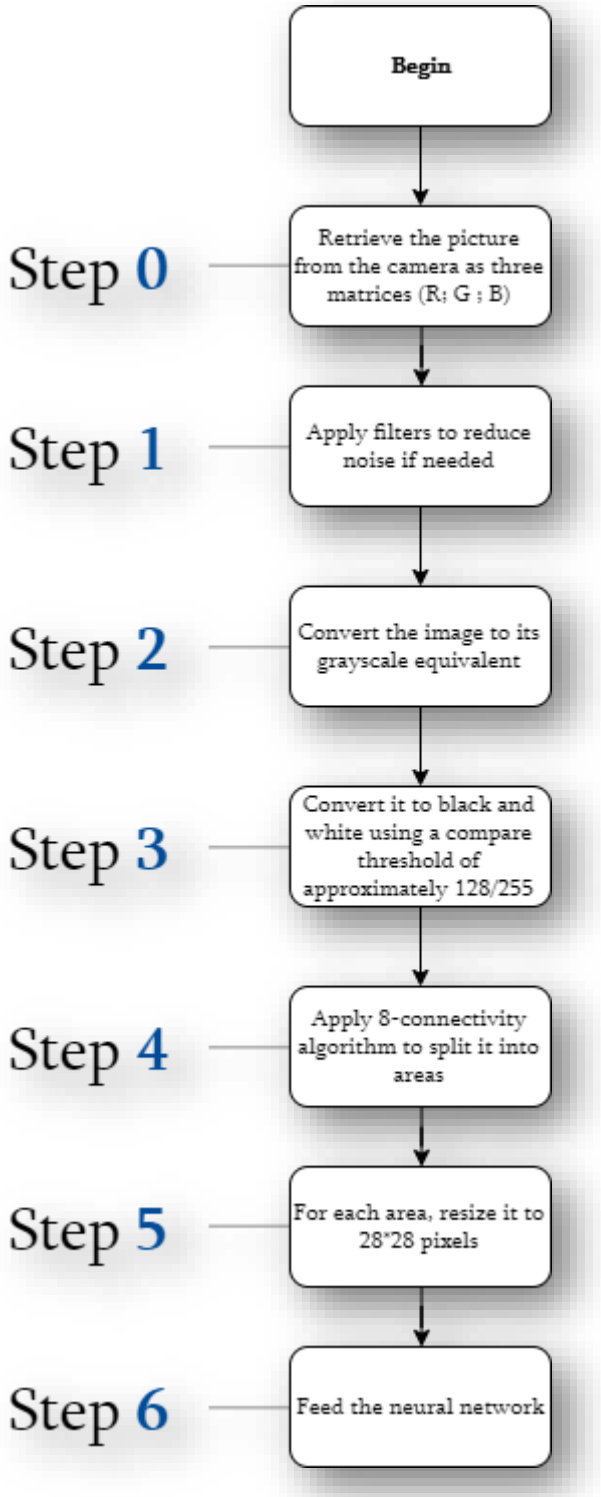
The camera support must:

- Fit the camera
- Have adaptable height and orientation.
- Be light enough to hold with a few screws.

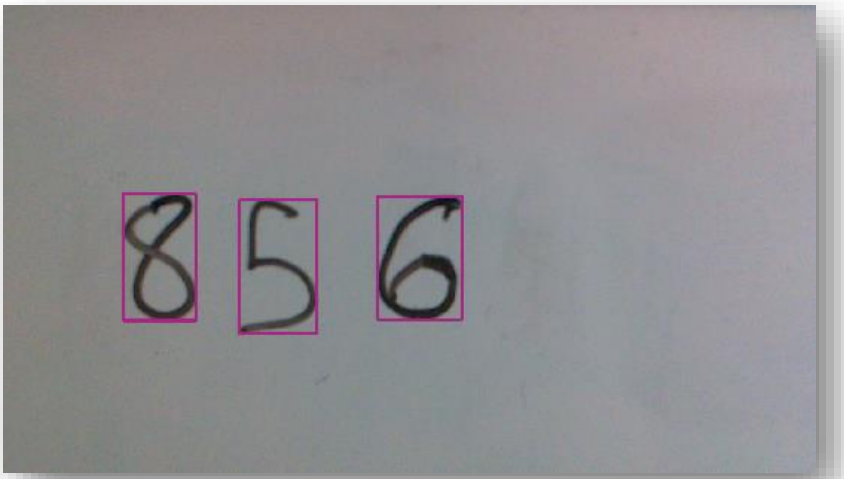
Technical achievements

Number detection system

- **Algorithm:**



- **Results:**



- **Implementation notes:**

• Steps 2 to 5 do not even require to use GPU as they are both fast and do not need much computing power. It is of course faster using one, but not crucial.

• All other steps are however greatly sped up when executed by a GPU (up to hundreds of times).

- **Notes:**

• In practice, step 1 is not implemented, as the system is not supposed to be used in highly electromagnetically polluted environment.

• **It is not possible to re-orientate the pictures before sending them to the network**. Being able to perform such an operation would mean knowing what number is represented on each part before even starting to compute the result of the neural net, which is absurd.

• It is also necessary to take in count the areas containing others, and remove the contained ones, otherwise it will detect the area inside 0,6,8 and 9.

Technical achievements

AI

Implementation notes:

• Uses a global purpose library of our creation, intended for flexible, ready-to-use, easily and highly customizable neural networks.

• Uses C++ and CUDA, but can also run on CPUs simply by using the macro *#define USE_GPU* before including any file related to this library.

• Possibility to train any neural network using SGD, SGD with momentum, RMSProp or Adam by calling one function.

Notes:

• MNIST original data format is incredibly unpractical, and has therefore been exported to CSV.

Data conditioning:

In image-feature recognition, the following parameters always have to be taken in count in order to pre-process an image, as they influence in a non-necessarily measurable way the performances of the system:

→ **Lightning**
→ **The size of the item in the image**
→ **The colour of the item in the image (if any)**
→ **Noise**
→ **Blur**

In our case, only the first two have a noticeable influence, so much the neural network predictions can reach 0% to 100% depending on these factors.

Of course, a design with more complex pattern recognition, or even a second neural network dedicated to fixing these issues could be implemented, but having near-100% accuracy is not the purpose of the application.

Performance:

To evaluate the library performances, nothing faster than some benchmarks. Below, a comparison of the time needed to train the neural network with the library and with Matlab.

Epochs	Batch size	Architecture	Layers	CPP (s)	Matlab (s)
30	100	784 - 80 - 60 - 10	4	3	1800
		784 - 80 - 60 - 40 - 10	5	7	2000
		784 - 80 - 60 - 40 - 20 - 10	6	17	2400
		784 - 200 - 80 - 60 - 40 - 20 - 10	7	24	2700
		784 - 400 - 200 - 80 - 60 - 40 - 20 - 10	8	30	3300
		784 - 400 - 200 - 100 - 80 - 60 - 40 - 20 - 10	9	47	

These benchmarks **are biased in MATLAB's favour**: the time has been **rounded down** to the nearest hundred, the Matlab implementation makes uses of the Matlab optimizer, which among others, unrolls loops.

Nevertheless, the C++ implementation is more than one hundred times faster, for the same results.

Conditions:

- CPU ONLY.
- Intel® i5-8300H @2.30 **GHz** (overlock at 4.00 **GHz**)
- 16 **GB** of RAM (thus meaning that Matlab does not have to spread its operations)

Limits:

- As the neural network is only trained to recognize numbers on a picture with a single digit, it is very sensitive to the size of the items in the pictures.

- As the pictures are in black and white, it is necessary to have a lighting as pure as possible, and that does not reflect on the whiteboard.

- The design being very simple, the bottleneck effect due to the cost of having to copy data on the GPU RAM is not compensated by the gain of treatment speed.

Technical achievements

Arm control - Generalities

Global considerations:

The arm is controlled using a sequencer which also contains a list of sequences read from XML files. The sequencer itself runs in parallel on a thread that also moves the robot to a point from which writing on an empty space on the board is possible.

Communication protocol:

ROS base UART communication protocol requires to send several messages (one per position), which is extremely slow. In this application, having to wait for each line of a character would not only be frustrating, but also quite strange as the arm specifications would not have been used to their maximum potential.

The new home-made protocol is thus as follow:

Byte	**Byte 0**	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**
Meaning	**Start**	**Arbotix Id**	**Message length** **N.B:** *size* is in BYTES	**Instruction**	**Time to execute the movements**	
Value	255	253	$size + 2$	12	Time LSB	Time MSB

Byte	**Byte 6**	**Byte 7**	$[\boldsymbol{Size - dependant\ values}]$			
Meaning	**Successive positions coordinates**					
Value	$Pos(size - 1)$ LSB	$Pos(size - 1)$ MSB	…	…	$Pos(0)$ LSB	$Pos(0)$ MSB

Byte	$[\boldsymbol{Size - dependant\ value}]$
Meaning	Checksum
Value	$255 - ((sum\%256) + 1)$

Inverse kinematics:

For various safety reasons, it proceeds in several steps:

- It first determines if the position is in the arm workspace.
- It secondly determines if the position is not in the robot itself (as it would break it)
- It thirdly determines if the position is not under the arm (on the TurtleBot)
- Finally, if all tests are passed, the arm is moved to the position specified.

Implementation notes:

- Uses only plain C++

- Parses CSV files to avoid having to make heavy computations at runtime.

- Runs in a thread to avoid having to cut the Bluetooth communication that links the AI and the robot while writing numbers.

- It is also possible to use either a **keyboard or a gamepad** to place the arm.

- USB ports had to be renamed in order to assign a fixed device name to the USB port.

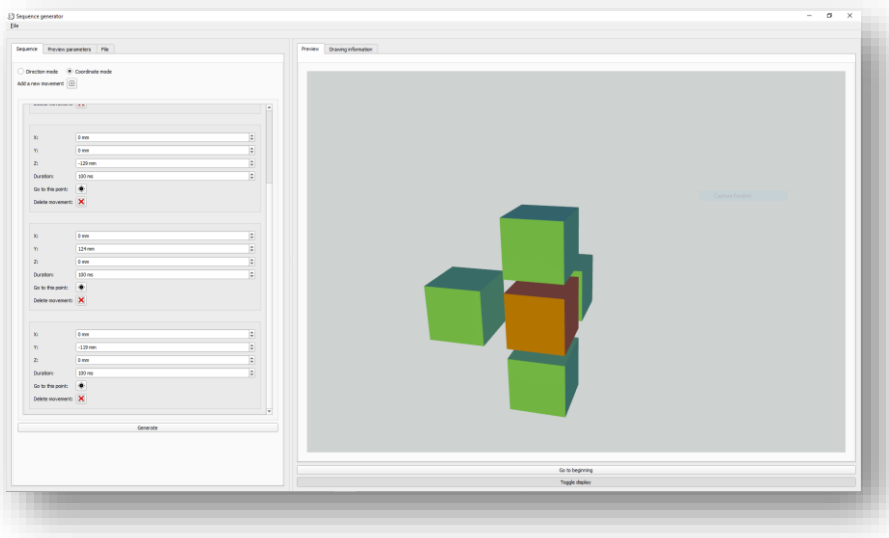
- Arduino-compatible, but requires the IDE version to be **1**. **0**. **0**.

Technical achievements

Arm control - Drawing

Sequence generation software:

The sequence generator software is fully functional and generates XML files and renders such as those below. The CSV exportation format is very similar but less readable.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<SEQUENCE>
    <FILE_FORMAT>1</FILE_FORMAT>
    <MOVEMENTS>
        <MOVEMENT>
            <X>-141</X>
            <Y>0</Y>
            <Z>0</Z>
            <DURATION>100</DURATION>
        </MOVEMENT>
        <MOVEMENT>
            <X>0</X>
            <Y>0</Y>
            <Z>-129</Z>
            <DURATION>100</DURATION>
        </MOVEMENT>
        <MOVEMENT>
            <X>0</X>
            <Y>124</Y>
            <Z>0</Z>
            <DURATION>100</DURATION>
        </MOVEMENT>
        <MOVEMENT>
            <X>0</X>
            <Y>0</Y>
            <Z>0</Z>
            <DURATION>100</DURATION>
        </MOVEMENT>
    </MOVEMENTS>
</SEQUENCE>
```

Results:

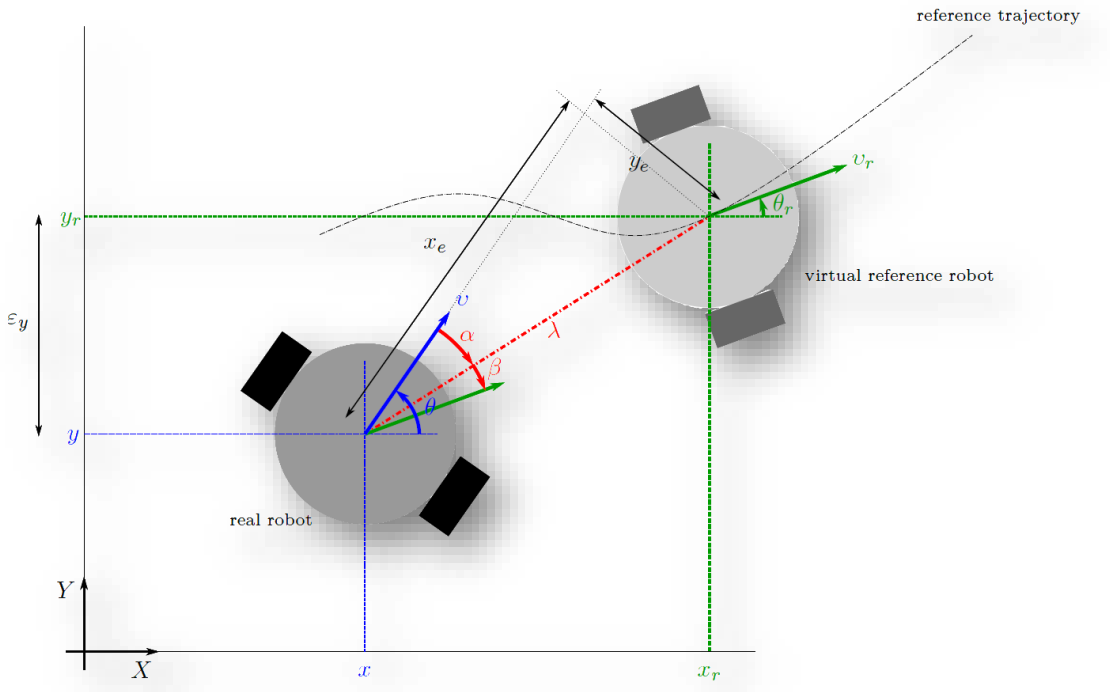
On the left, the shapes of numbers between 0 and 9 (inclusive).

Technical achievements

Robot control

Position tracking - Principle:

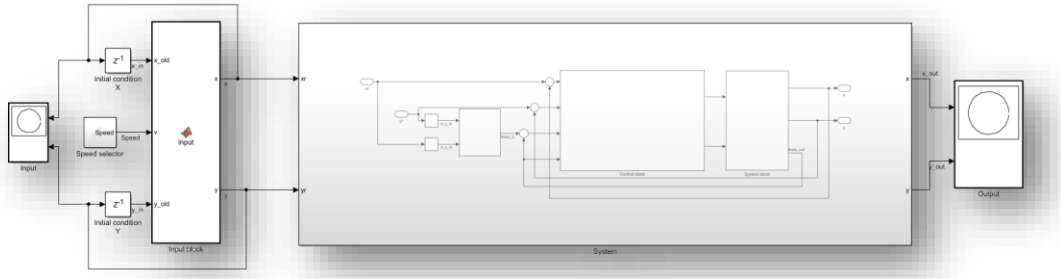
The current robot position and orientation are compared to a virtual reference, that moves as intended. Three parameters, λ, α, β are chosen as system state variables. They are defined as below:



Source: *Introduction to Mobile Robotics*, Sylvain Durand [4]

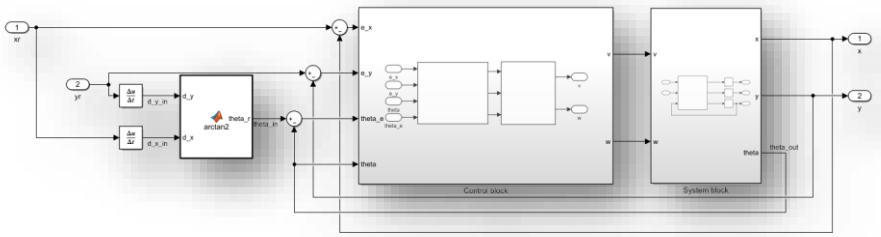
Position tracking - Algorithm:

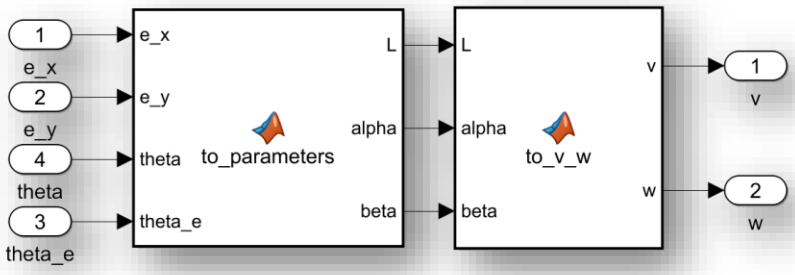
The algorithm is easily representable using a Simulink schematic:

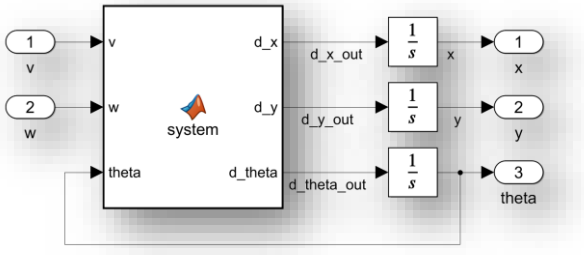


Or in a less but still quite obfuscated form:

Where ***xr*** and ***yr*** represent the reference trajectory equation.

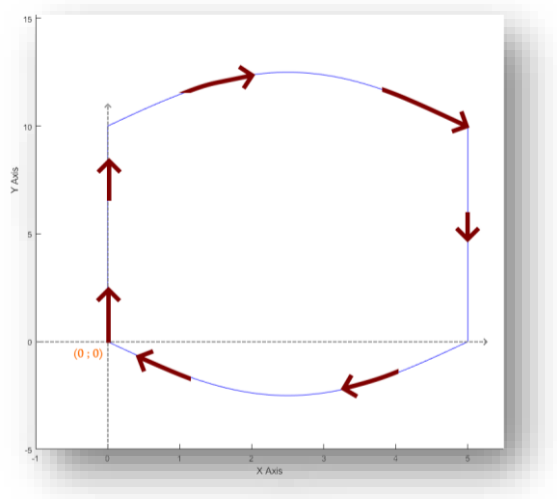




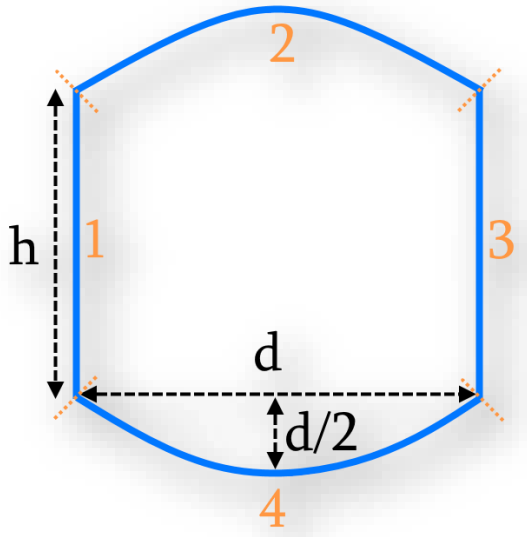


Position tracking - Generating the equations of the trajectory:

The robot must follow a trajectory of this form:



Which is of course described by not really nice-looking equations.
Let $(h\,;d) \in {\mathbb{R}_+^*}^2$ so that:



The equations are then:

$$x(t) = \begin{cases} \mathbf{0}\ if\ y(t) < h \\ \boldsymbol{t}\ if\ x(t) < d\ \wedge y(t) \geq h \\ \boldsymbol{d}\ if\ x(t) \geq d\ \wedge y(t) \geq 0 \\ \boldsymbol{d} - \boldsymbol{t}\ otherwise^* \end{cases}$$

$$y(t) = \begin{cases} \boldsymbol{t}\ if\ y(t) < h \\ \boldsymbol{h} + \frac{\boldsymbol{d}}{\boldsymbol{2}}\boldsymbol{\sin}\left(\frac{\boldsymbol{\pi t}}{\boldsymbol{d}}\right)\ if\ x(t) < d\ \wedge y(t) \geq h \\ \boldsymbol{h} - \boldsymbol{t}\ if\ x(t) \geq d\ \wedge y(t) \geq 0\ ^* \\ -\frac{\boldsymbol{d}}{\boldsymbol{2}}\boldsymbol{sin}\left(\frac{\boldsymbol{\pi t}}{\boldsymbol{d}}\right)\ otherwise^* \end{cases}$$

*: The equations are oriented, as shown above.

Note that the top part is actually a sine, not a half circle. This is due to floating point error, because of which the equation of such a form becomes undefined at some points.

These equations are in their continuous form, which are not easily programmable on a computer. Let us rewrite them as numerical sequences x_n and y_n. Furthermore, let us change the variable t to $T_s * v$ where T_s is the sampling time of the system in **s** and $v \in \mathbb{R}_+$ the Euclidian norm of the speed of the robot in $\mathbf{m.s^{-1}}$.

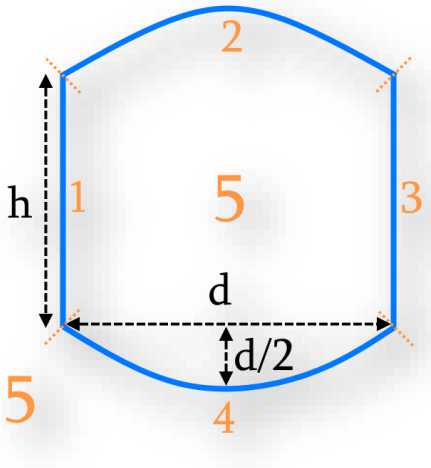
$$x_0 = 0 \ \ and \ \forall n \in \mathbb{N}, \qquad x_{n+1} = \begin{cases} \mathbf{0} \ if \ x_n = 0 \wedge y_n < h \\ \boldsymbol{x_n + T_s v} \ if \ x_n < d \wedge y_n \geq h \\ \boldsymbol{d} \ if \ x_n \geq d \wedge y_n \geq 0 \\ \boldsymbol{x_n - T_s v} \ if \ none \ of \ above \ and \ x_n - T_s v > 0 \ * \\ \mathbf{0} \ otherwise^* \end{cases}$$

$$y_0 = 0 \ \ and \ \forall n \in \mathbb{N}, \qquad y_{n+1} = \begin{cases} \boldsymbol{y_n + T_s v} \ if \ x_n = 0 \wedge y_n < h \\ \boldsymbol{h + \frac{d}{2} \sin\left(\frac{\pi x_{n+1}}{d}\right)} \ if \ x_n < d \wedge y_n \geq h \\ \boldsymbol{x_n - T_s v} f \ x_n \geq d \wedge y_n \geq 0 \\ \boldsymbol{-\frac{d}{2} \sin\left(\frac{\pi x_{n+1}}{d}\right)} \ otherwise \end{cases}$$

*: Due to sampling time, it is possible that $x_n - T_s v$ become negative, which is not possible for the continuous versions of the equations.

This is however not enough for a correct implementation. It is indeed necessary to compensate for the non-perfection of the system, otherwise large divergence will occur.

In this application, a simple additional condition corresponding to the area the robot is not supposed to be is enough to keep the system stable.



Position tracking - Corrector determination:

Let $(k_\lambda\,;k_\alpha\,;\,k_\beta)$ be the three parameters of the corrector, respectively pondering the state variable in index.

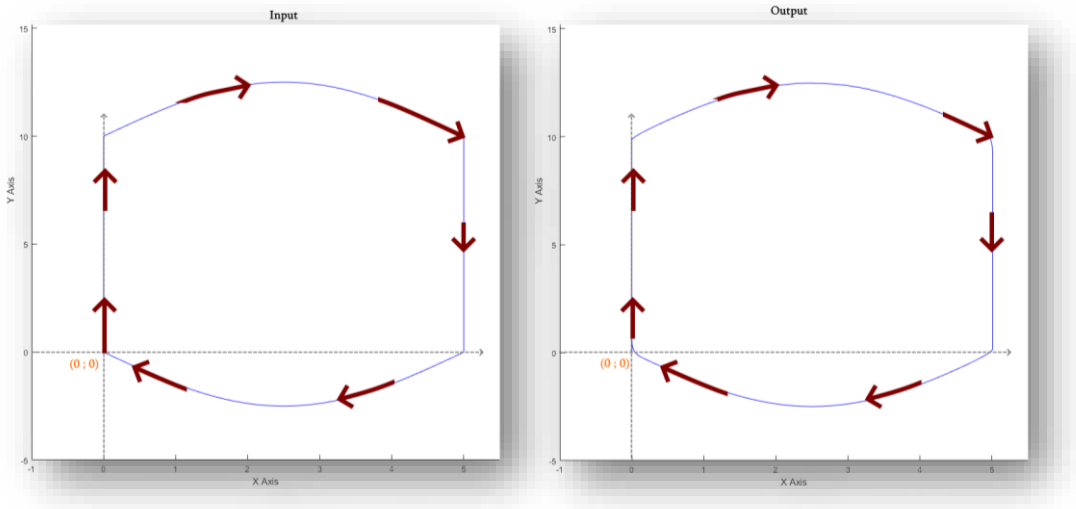
• Corrector coefficients are determined empirically, as they vary non-linearly when $(T_s\,;h\,;d)$ changes.

• For various reasons, we pick $(T_s\,;h\,;d) = (20\text{e}-3\,;10\,;5)$. T_s is in $\boldsymbol{s}$, h and d are in $\boldsymbol{dm}$. The units of the last two do not matter much, as they can be interpreted before or after the control unit of the system.

• At the particular conditions above, $(k_\lambda\,;k_\alpha\,;\,k_\beta) = (-2.1\,;\,-50\,;1)$. This assures stability $\forall(h\,;d)$ so that $h \leq 10$ and $d \leq 5$, but not that the performances will be as expected when the values become too small.
No guarantee is given for stability when $h > 10$ and $d > 5$.

Position tracking - Simulation results:

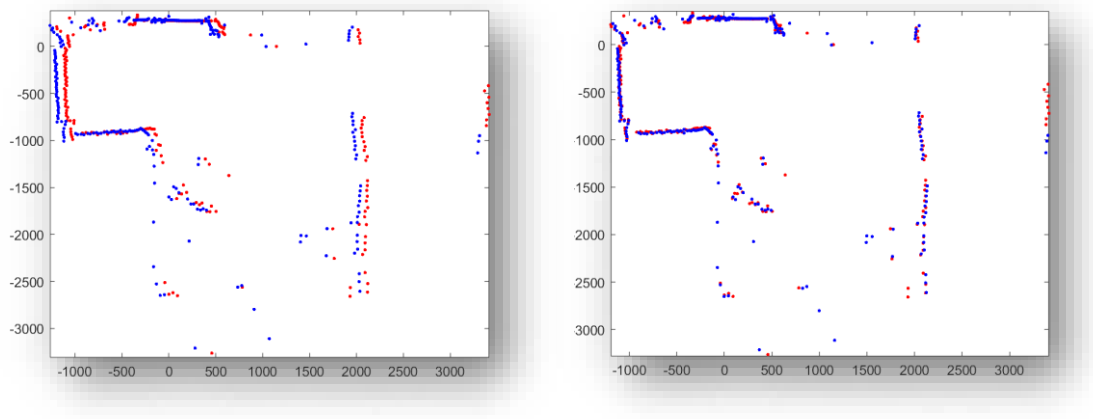
As shown above, the output follows the path as expected.
Please notice that it does not return exactly at $(0\,;0)$ for mechanical reasons.



Position tracking - Positioning using an ICP algorithm:

Experimental results:

Rather than a long text explaining the implementation, the two pictures below should be enough to prove the reliability of the algorithm:



Where the points in red and the ones in blue represent two successive acquisitions of Lidar points.

Implementation notes:

• Plain C++.
• Off-distance points are deleted while keeping the sizes of the two cloud of points equal.
• Limited to two-dimensions in order to limit the computation time (and also because the robot is not supposed to suddenly lift off).
• Very precise to determine the rotation angle (error is less than 0.1%), but less to determine the translation (approximately 8% to 9%).
• Criterion: Cauchy criterion function (i.e., usual sequence convergence criterion).

Limits

• This algorithm is REALLY noise-sensitive.

• In this application, the environment changes each iteration, and reconstructing it is a heavy process not suited for such a system. This added to noise from the Lidar-obtained cloud of points greatly decrease performances, **making it impossible to use** for feedback positioning is this application.

Alternatives:

• Calculating a transformation matrix (impossible due to environment changes each time)
• Using Fourier Transform to determine the spectre shift (possible but heavy), and noise-sensitive.

Position tracking - Workaround:

Under the following hypothesises, it is possible to use the input of the system as the feedback itself:

- The robot is slow.
- The rising- and falling-time of the motors speed are neglectable as the speed variations are slow.
- The transfer functions of the motors are close to 1 due to the built-in speed- and current-loops.
- The whiteboard does not move between two turns.
- The terrain does not change between iterations (no obstacles are added while the robot is running).
- The terrain is flat.
- The only flat surface on the left of the robot is the whiteboard or a wall with the same angle.

These hypothesises of course induce a divergence in the trajectory at each iteration.
To compensate for this, the Lidar is used to correct the angle by detecting the nearest flat surface on the left, and aligning the robot with it.

Position tracking - Implementation:

- Plain C++

- Doing the maths to determine the equations could have been quite long, especially considering the input.
It was therefore decided to create a Simulink-like library, allowing the declaration of a system through blocks, linked together as the user specifies.

- Sampling period is fixed to 20 **ms** for stability reasons. It also ensures that the computation will be faster than the next call to the control loop functions.

Position tracking - Performances:

- Approx. 1 **cm** repeatability pre-Lidar correction.
- Approx. 0.5 **cm** repeatability post-Lidar correction.

System interface:

Due to Raspberry Pi running a quite special OS, C++ $std :: thread$ may not produce the same results as on other, more classic systems.
As a consequence, a portable execution model had to be chosen to have a fully functional system.
Among the numerous available, the implemented solution makes use of POSIX (***P****ortable* ***O****perating* ***S****ystem* ***I****nterface*).

Global system organization:

The system must be able to perform several operations at the same time, being:

- Control and track its position in *real time*.
- Read the information from the AI at any time without stopping the current operation.
- Control the arm.
- Poll the Lidar points without missing data from the Bluetooth communication.

System performance drastically drop when the number of currently active threads rises above the maximum number of threads per core of the CPU, thus leading to a 4-threaded design.
Note that the OS threads are of course not taken in count, as they are most of the time in sleep mode.

Real time position tracking:

This part of the code is executed in the *main* function. It avoids having to allocate a thread to it while doing nothing in the rest of the program.

The sampling period is maintained as constant as possible using hardware timers, but without interruptions.
Every time the timer runs out, the control loop is updated, fixing the robot position.

This approach reduces the system load to its minimum, **but is not adapted to lower specifications hardware**, as the time to compute the output of the control loop may take more time than the sampling period especially if the control logic is consequently big. This is fortunately not the case in this application.

Communication with the AI:

Being able to read the data received by Bluetooth during execution, without missing any byte is one of the core features of this project. This therefore requires the system to perform this task in parallel to the others.

CPUs are, unlike FPGAs, limited in parallelism, which means that system load induced is quite consequent. This is however indispensable, as missing even a single byte could lead to do severe glitches, especially if the missing ones are the beginning- or stop-bytes.

This single functionality also explains why the thread sequencer load must be reduced to its minimum. Indeed, having it switches between a large number of threads would mean taking the risk of it becoming too slow compared to the Bluetooth communication, and thus missing a message or worse, a message part.

The message itself is in the following format:

Byte	**Byte 0**	**Byte 1**	**Byte 2..2 + size**	**Byte 3 + size**
Meaning	**Start**	**String length (Bytes)**	**Message**	**Checksum**
Value	255	$size$	$string\ (ASCII)$	$sum(message + size)\%256$

Controlling the arm:

When reaching the whiteboard, the robot must move its arm to write the numbers. This is once again a supplementary task that has to be done simultaneously, to avoid the aforementioned problems.

Computing the arm movements is already a heavy process in itself, and has thus been lighted by the use of pre-computed positions stored in files (please refer to the dedicated part for more details), allowing fast execution to fit the execution time constraints.

Controlling the motors:

Once the output of the control loop has been updated, it is obviously necessary to update actual position too.
Both of the motors are controlled in speed, and through an UART communication in the following format:

Byte	**Byte 0**	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**
Meaning	**Start**	**Speed (right wheel)**		**Speed (left wheel)**		
Value	255	$RSpeed$ LSB	$RSpeed$ MSB	$LSpeed$ LSB	$LSpeed$ MSB	$sum(speed)\%256$

Polling the Lidar:

At the end of each turn, the Lidar is polled to determine the position error.
This thread was originally designed to be executed at small intervals in order to generate the position feedback, which implies that it had to be synchronized with both the Lidar transmission signal and the control loop.
As a consequence of the ICP performance, the second synchronization became less important (but is still involved when adjusting the position at the end of each turn and when the system boots).

The messages received follow the format below:

Byte	**Byte 0**	**Byte 1**	**Byte 2**	**Byte 3**
Meaning	**Start the Lidar**	**Number of frames (Bytes)**	**Motor speed**	
Value	0xFA	nb + 0xA0	$Speed$ LSB	$Speed$ MSB

Byte	**Byte 4**	**Byte 5**	…	$[Size - dependant\ values]$	
Meaning	**Intensity**				
Value	$Intensity[nb-1]$ LSB	$Intensity[nb-1]$ MSB	…	$Intensity[0]$ LSB	$Intensity[0]$ MSB

Byte	[***Size dependant values***]	…	$[Size - dependant\ values]$	
Meaning	**Range**			
Value		…		

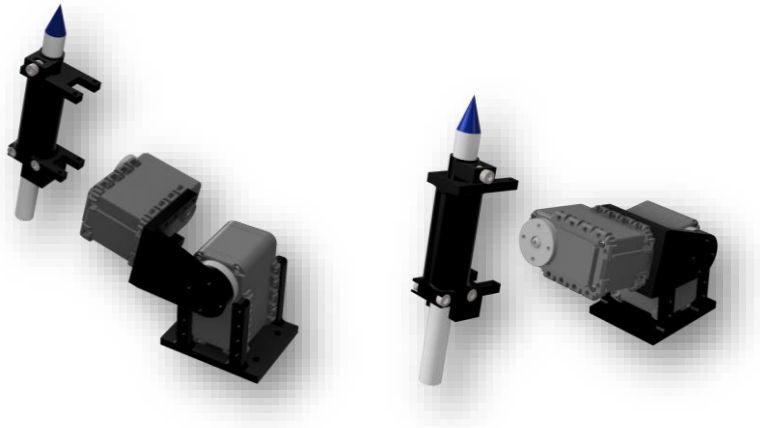
Starting the communication: $'\mathbf{a}' == 0x61$.
Ending a communication: $'\boldsymbol{e}' == 0x65$

Technical achievements

Mechanical parts

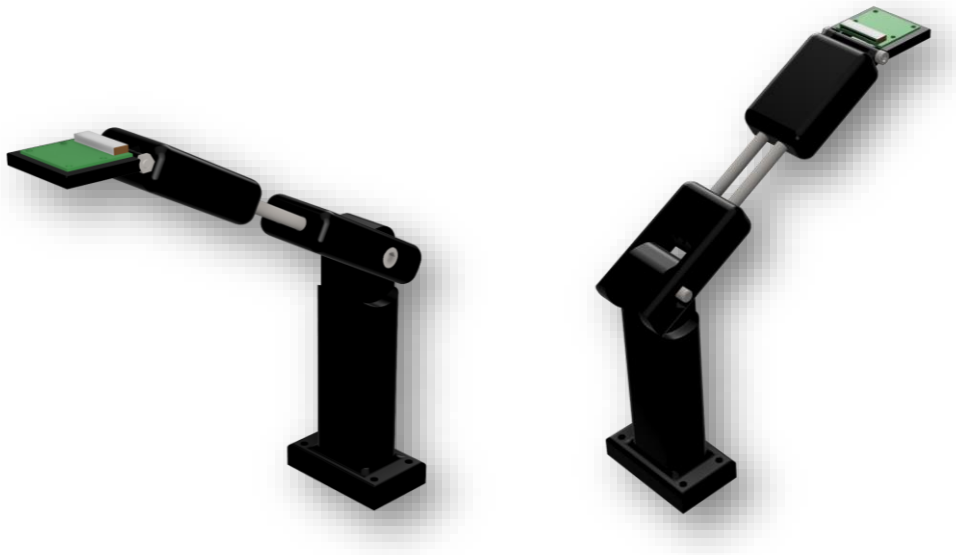
Pen support:

To compensate for the support (whiteboard) mechanical reaction, it is necessary to either design a fairly complex algorithm that corrects it using an equivalent mechanical model or a very simple support with a spring inside it. Needless to say, the second option was chosen.



Camera support:

The camera support is designed as follows. Please note that the grey part length is adjustable.



Conclusion

Number detection system:

By using simple image processing, it was proven possible to detect numbers on a whiteboard with satisfying accuracy.
It is however very sensitive to lighting, and may therefore be improved using for example another neural network, dedicated to image feature recognition.

Neural network

Fast, simple to use and to train, this neural network implementation performs really well on perfectly conditioned images. Its performances could be improved by changing the architecture for a more complex one (several additional layers, with intermediate thresholding using ReLU function for example), but as reaching 100% is not the core of this project, these improvements will certainly not be implemented.

Its implementation is capable of beating Matlab in term of speed, and even TensorFlow under certain conditions, making it competitive for general use.

Robot- and arm-control:

The arm performances (≈ 1 **mm** repeatability) are really satisfying as it is the standard precision in the industry.

The mobile robot control is on the other hand not quite as great as we could expect. Diminishing the error would require other sensors, such as encoders on the wheels, another wheel only used for position monitoring, or indoor GPS.
The cheaper solutions are of course the first two ones, but their reliability should be proven first (especially when the encoders move by a few millimetres, or even break). The indoor-GPS-solution is however much more reliable, but restrain the use of the system in a defined area.

Mechanical structure:

Plastic is quite fragile, and is susceptible to break if not enough attention is paid (and it already happened).

Components and equipment list

- **NVIDIA:**

  - Jetson Nano.
  - CUDA development Kit

- **Raspberry Pi Foundation:**

  - Raspberry PI
  - Raspberry PI camera module

- **OpenCR board**

- **ROS components:**

  - TurtleBot 3 Waffle Pi
  - WidowXL robot arm

- **Other:**

  - HC-05 Bluetooth module
  - 360° LiDAR for SLAM and navigation

Sources

[1]: X. **Glorot** and Y.**Bengio**, *Understanding the difficulty of training deep feedforward neural networks*, DIRO, Université de Montréal, Montréal, Québec, Canada, 2010

[2]: D. P. **Kingma** and J. **Lei Ba**, *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*, University of Amsterdam, OpenAI , University of Toronto, 2015

[3]: M. **Nielsen**, *Neural Networks and Deep Learning*, 2019.

[4]: S. **Durand**, *Introduction to Mobile Robotics*, INSA Strasbourg, 2020

[5]: **NVIDIA**, CUDA Toolkits

[6]: **NVIDIA**, Jetson Nano datasheet

[7]: **The Qt Company**, Qt documentation

[8]: **ROS**, ROS standard communication and command protocols

[9]: G. **Henderson**, WiringPi C library, 2013-2019

[10]: Y. **LeCun**, C. **Cortes**, C. J.C. **Burges**, THE MNIST DATABASE, Courant Institute (NYU), Google Labs (New York), Microsoft Research (Redmond), 1998.

[11]: Domoticz, Assign fixed device name to USB port, 12 May 2018.