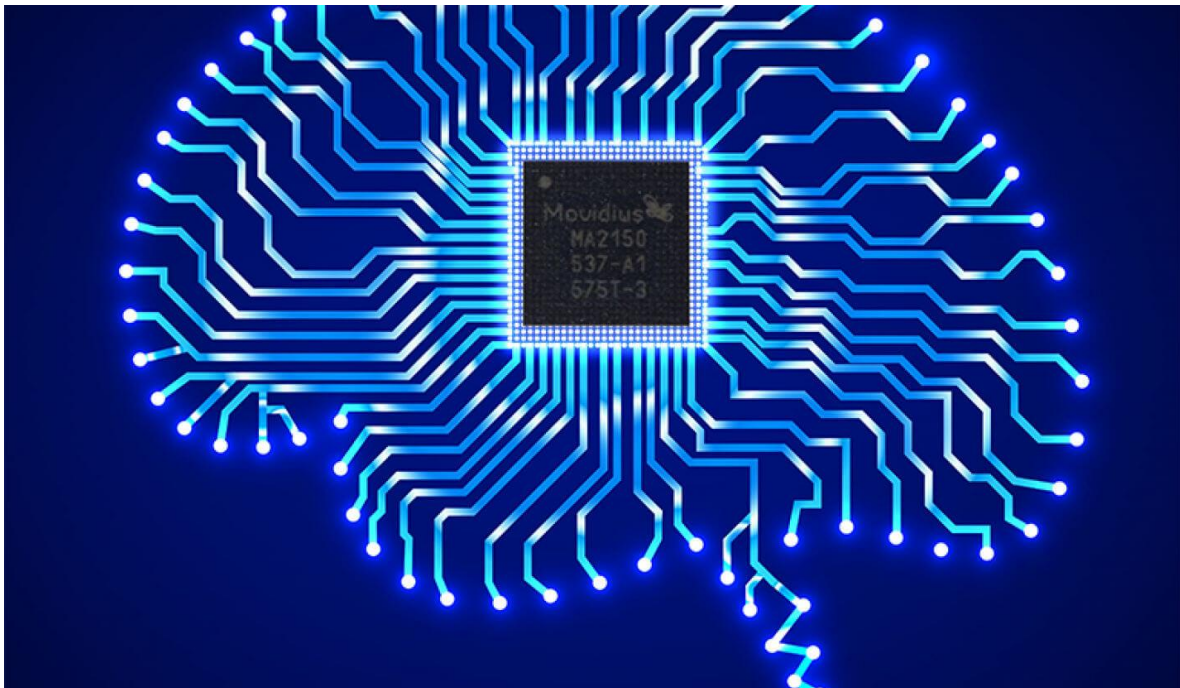


# Integration of Intel Movidius with RobAir

**Baptiste BOLÉAT - Rémy PALOMO**  
*INFO4 - 2019/2020*



## OVERVIEW:

In the context of our fourth year of Computer Science studies at Polytech Grenoble, we chose to work on the incorporation of the Intel Movidius on the RobAir project. Its purpose was to improve our teamwork and our skills in project management.

In this document, you can find a summary of the work we have accomplished since the beginning of the project. It can also be used for people who would want to improve this project.

All the code and the documentation is available on this repository :

[RobAirMovidiusProject](#)

## TABLE OF CONTENT

<b>INTRODUCTION.....</b>	<b>3</b>
Context.....	3
Problem.....	3
Objective.....	3
<b>TECHNOLOGIES USED TO DEVELOP THE PROJECT.....</b>	<b>4</b>
RobAir.....	4
ROS-Kinetic.....	5
Movidius.....	6
<b>PROJECT PROGRESS.....</b>	<b>7</b>
Ros Environment Installation.....	7
Movidius Environment Installation.....	8
Find an object-detection program.....	9
Edit our object-detection program and connect it to the turtlebot.....	10
Modifications on detection program.....	10
How did we connect ros and detection?.....	10
Our set-up.....	13
Results.....	13
<b>CONCLUSION.....</b>	<b>14</b>
Review.....	14
Prospects for the future.....	14

## INTRODUCTION :

### **Context :**

Begun in 2012, at the Fablab laboratory, the RobAir project has been created to provide an open-source and low-cost platform intended to teach ambient intelligence and to enable efficient experimentation. To do this, it has been equipped with multiple sensors as Lidars to detect obstacles or jog wheels to compute the length it has travelled. During the last decade, Artificial Intelligence and especially image detection started to be an essential tool for robots, softwares and tools. That's why important companies like Intel developed the Movidius, a low-cost and energy-saving device which provides enough computing power to use **AI** on small platforms like a Raspberry.

### **Problem :**

Thanks to all these sensors, RobAir can avoid most of the physical hurdles but some of them remain difficult to detect with only two lidars. For example, if the robot faces a table, the top lidar is too high to detect it and the bottom one is too low. That's why equipping RobAir with a camera seemed to be a feasible option. The Intel Movidius NCS provided us a portable option which we can put on the robot.

**How to make the communication between the camera and the motor functions possible ? How to enable the robot to eye-track something ?**

### **Objective :**

Our purpose is to add a camera to RobAir which will be directly connected to the tablet and linked to the Movidius Neural Compute Stick. Then, the tablet has to compute informations about the objects detected by the Neural Network of the NCS in order to enable a rostopic to control the motor functions of the robot. As the Movidius and ROS are new tools for us, one of the first and the biggest tasks to do was to document ourselves about their installation and their use. Then we had to link these technologies by using the available material on RobAir.

## TECHNOLOGIES USED TO DEVELOP THE PROJECT :

- RobAir :

The RobAir project, developed at FabLab, is actually broken down into several types of behavior, or “**modes**” (**teleoperation**, **autonomous** and **semi-autonomous**) divided into several robots having the same basic skeleton (as opposite) but being equipped of specific tools to their task.

One of the models (visible on the picture) is equipped with a small camera as well as a tablet, making it possible to make online conference for example.

A second one is simply controllable by a remote control, but is also equipped with Lidars sensors allowing for example the robot to detect a wall and warn the pilot or force the stop among others.



After several interviews with the FabLab manager, Mr. Lemasson, we decided that it would be interesting to integrate an artificial intelligence (**AI**) into a semi-autonomous model to detect obstacles (such as chairs or desks) much more effectively than with Lidars and so alert the user in a more relevant way. We could also integrate AI into a autonomous model so that it could look at a persons for example and follow them when they move.

⇒ However, for the rest of this report, you will note that our experiences, our achievements have been tested on a robot simulator (see paragraph about ROS TurtleBot).

This is due to the fact that the health crisis caused by the Covid-19 prevented us from having access to the FabLab and therefore to RobAir when we could start the phase of real tests on it in order to test our programs.

- Ros-Kinetic :

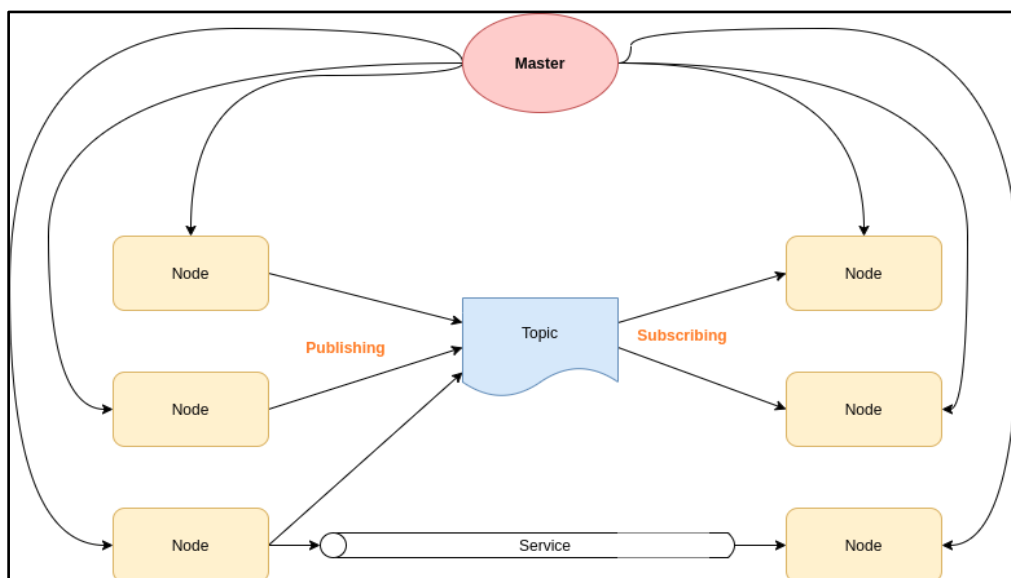
Robot Operating System (ROS) is a set of tools to develop softwares for robotics. It provides an operating system for robots but also high-level features for their development. It had been developed by Willow Garage in 2007 and its initial goal was to manage their flagship product, the **Personal Robot 2**. When it has been generalized, it enabled developers to avoid using a lot of embedded programming while building it.



Ros is based on two languages which are C++ (*roscpp*) and Python (*rospy*). Every node, topic or other component can be written with both of them. Moreover it isn't energy-intensive, that enables users to use this on embedded computers, as is often the case in robotics. For example, a Raspberry Pi can handle ros for sure.

The **Master** is the core process of the **ros computation graph** (set of concepts which are used while ros is running). It mainly contains a name server where **nodes** will declare them. A **node** is an executable which can match with a sensor, a motor or just an algorithm. A **topic** is an asynchronous information bus. When declaring it, you choose a **message type** for every **publisher** and then, every **node** can either **publish** on or **subscribe** to this topic.

In order to run a ros node, you will always have to run the **roscore** command which initialize the **Master** and the **Parameter Server**. The **Parameter Server** is a space that every node share and can add parameters like the speed limit you would like to assign to the robot for example.



**Figure 1 : Architecture of a simple ros computation graph**

One of the main concepts of ROS is that it is in Peer-to-Peer architecture : A Ros Node is linked with different topics which he can either publish messages on or subscribe to. So it mainly uses an asynchronous communication, but it also provides an other kind of synchronous communication thanks to the “Services” that we won’t use in this project.

Eventually, ros-kinetic isn’t the last release of ros but it’s still maintained by Willow Garage. Indeed, they released it in 2016 and will maintain it until 2021. Two new versions have been released since kinetic (Lunar and Melodic) and they are starting to migrate to ros2 which corresponds better with current companies applications. Even with all these new features, keeping maintaining kinetic enables users as us to keep programming in ros on a **16.04** version of Ubuntu.

- Intel Movidius :

The story of the Neural Compute Stick is quite special. Indeed, at the beginning, it was developed by the company named **Movidius** which was specialized in image analysis.

However, when the company was going to sell the NCS, this one was bought by Intel in September 2016, pausing the process.

It’s only 1 year later, in September 2017, that Intel announced the release of its **Movidius Neural Compute Stick**, which will be referred to as **NCS** later in this report.



At first, we imagine that this tool looks like a simple standard USB key, but the physical comparison stops there because the NCS is used for much more complex purposes. Indeed, the stick aims to provide Artificial Intelligence functions to different objects (such as robots with a webcam for example) at lower material and energy cost.

To accomplish this complex task, the NCS has a Myriad 2 VPU which is used by most image analysis devices which therefore require very large computing capacities.

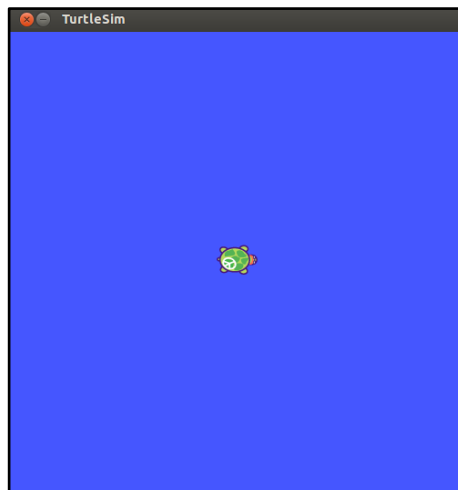
This VPU has the particularity of needing only 1 W to operate, which is very advantageous for the devices which the NCS can be equipped with. Even more if a device already has a limited autonomy.

To put it simply, the Movidius has a very large computation and image analysis capacity that it can put to the service of an embedded program on a device, and all this at a very low energy cost. Therefore, the Movidius is the perfect tool to use an artificial intelligence program on any machine !

## PROJECT'S PROGRESS :

### ❖ ROS Environnement's installation

As you will see lower, one of the requirements of our project was he had to be effective on **Ubuntu 16.04** because of our choice concerning the neural network we put in the Neural Compute Stick. That's why we chose to work with **kinetic** in terms of ros distribution. The fact is that, in the way we use it, there are relatively few differences between kinetic and a later version of ros. As we hadn't access to RobAir so often and it required to come to the Fablab, ros also provided us a simulator named **Turtlebot** which enabled us to test our code when we wanted. As we said into the RobAir's part, it turned out it has been essential since we couldn't have access anymore to the Fablab because of Covid-19.



*Figure 2 : The TurtleBot's window when we run it*

In addition to this simulator, a message type, **/turtle1/cmd\_vel**, was in the default packages installed with ros. So we didn't had to create new one since **/turtle1/cmd\_vel** contained all the informations we could have to give to the robot. Indeed, this message type is made of a linear velocity and an angular velocity, both comprised of 3 coordinates (x, y and z).

To help us to understand how ros works, we created a *TalkerListener* package which sets up an exchange of informations between two nodes via a topic. We commented it on our own word in

[https://gricad-gitlab.univ-grenoble-alpes.fr/Projets-INFO4/19-20/7/docs/-/tree/master/catkin\\_ws%2Fsrc%2Fbeginner\\_tutorials%2Fsrc](https://gricad-gitlab.univ-grenoble-alpes.fr/Projets-INFO4/19-20/7/docs/-/tree/master/catkin_ws%2Fsrc%2Fbeginner_tutorials%2Fsrc)

## ❖ Movidius Environment's installation

As part of our project, we first had to install the necessary environment in order to be able to run programs using Movidius. This environment was only available on **Ubuntu 16.04**, so we had to install a virtual machine running this operating system. Once the VM was properly initialized (installation of git, configuration of the USB 3.0 ports in particular), we were able to install the package called **ncsdk 1.0** (version 2.0 was available but only compatible with Movidius NCS 2 !).

The main features of this package, according to its official documentation, are :

- Profiling, tuning, and compiling a **DNN model** on a development computer (host system) with the tools provided in the NCSDK.
- Prototyping a user application on a development computer (host system), which accesses the neural compute device hardware to accelerate DNN inferences.

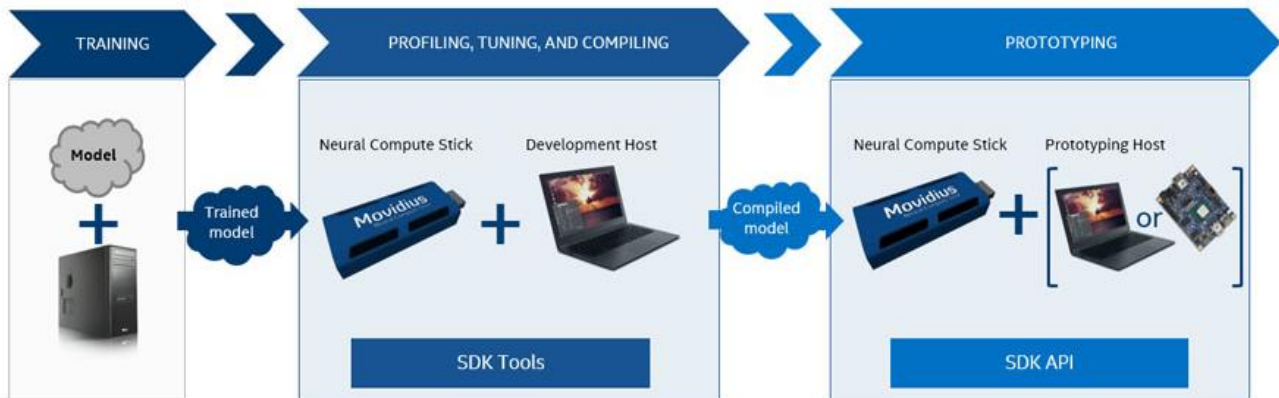


Figure 3 : How we use the Movidius (<https://movidius.github.io/ncsdk/>)



## ❖ Find an Object Detection program

Then, we looked on **Github** if a possible object detection model, using the Movidius, existed in order to be able to use it directly (and avoid us of having to configure such a model ourselves). We finally found a repository called **yoloNCS**, realized by Denis Gudovskiy (Available at the following address : <https://github.com/gudovskiy/yoloNCS>).

The object detection program, named **object\_detection\_app.py**, makes it possible to identify several types of entities such as people, a car, a computer, a bottle, etc. from an input image.

However, it is also possible to recover several images per second thanks to a webcam, and send these images to the program, in order to have a real time detection, which perfectly corresponds to the behavior that we wanted to add to RobAir.

It should be noted that the detection capacities are determined from a “**caffemodel**” type file which, as its name suggests, contains a pre-trained **Caffe** model (see below) for the detection of objects on a picture.

Indeed, it's necessary to remember that before being able to use an analysis program based on an artificial intelligence, it's first necessary to create and configure this artificial intelligence !

To do this, we must create a model which we will assign the desired capacities (detecting various types of objects for example). For the case of our program, we use a **Caffe model** which is a deep learning framework (more informations about Caffe here : <https://caffe.berkeleyvision.org/>).

This model will consist of layers of neurons, each neuron containing parameters which will be adjusted through a training phase. It is therefore necessary to give a large amount of input images so that the model can progress and thus improve the accuracy of its judgment on a given image that it has potentially never seen before.

This training phase is costly in time, memory and energy for a normal machine because it requires very large computing power.

Finally, when we consider that our model is fairly trained, it is possible to export it as a **.caffemodel** file.

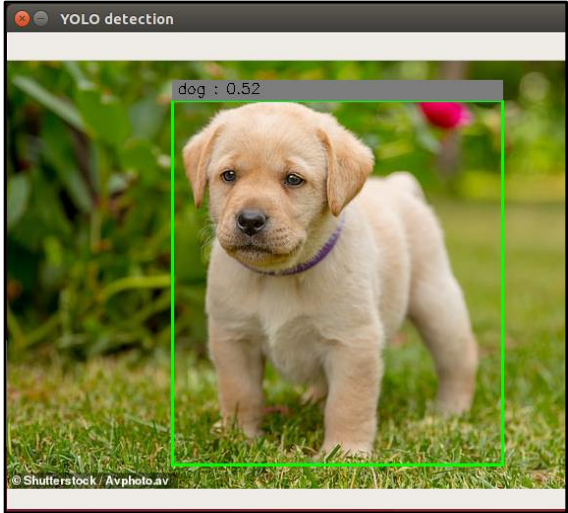

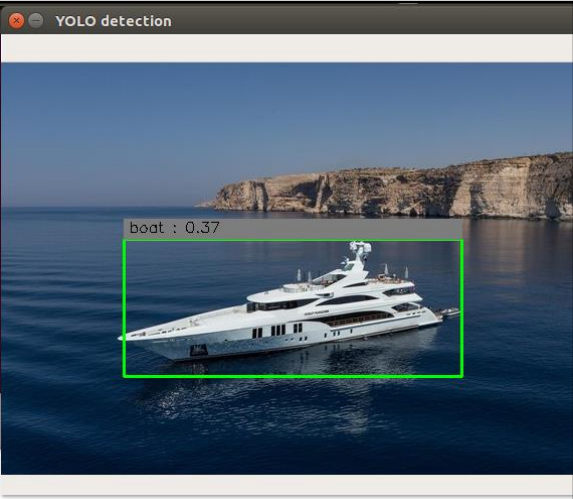
This is the file that we got from the Github repository, allowing us to greatly facilitate the Artificial Intelligence part of our project.

It turns out that the **ncsdk** environment offers the possibility of compiling a model in a “**graph file**” from a pre-trained model file, all this thanks to the command

**mvNCCompile**. It's this graph file which will then be used by the movidius to apply the various calculations relating to image analysis.

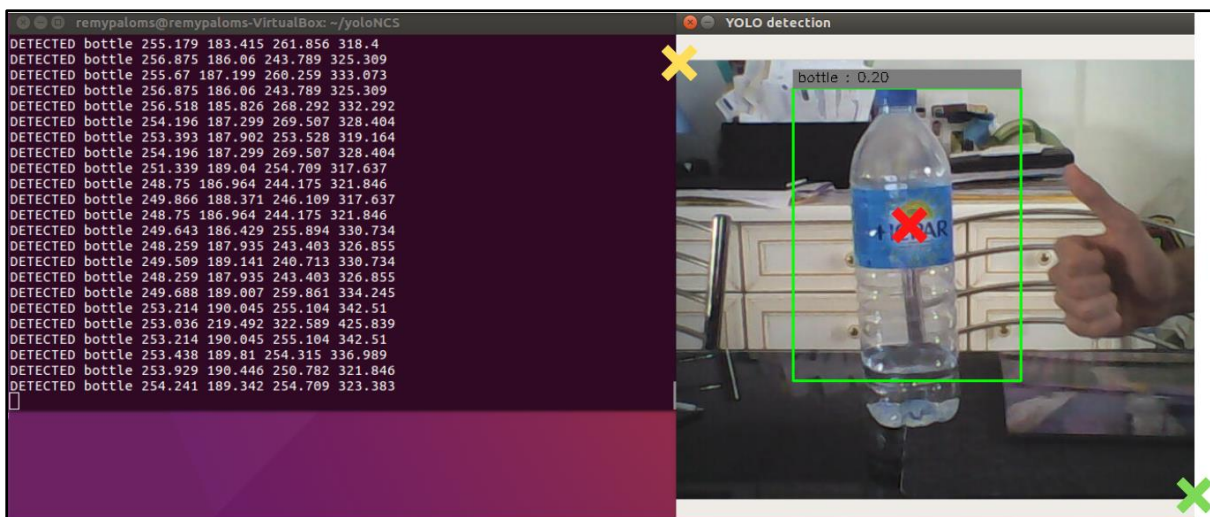
Once the compilation was completed, we were able to test the program. Its speed, its precision and the result is rather very satisfying !

→ Below are examples of using the program with a simple **input image** :

<i>Input Image</i>	<i>Detection program's Output</i>
	
	

We can observe that the program is able to frame a detected object, to indicate its type and to give a “confidence index” between 0 and 1.

→ An example with video input from a webcam :



## ❖ Edit our Object Detection program and connect it to the TurtleBot

⇒ *Modifications made on detection program :*

First, we had to analyse the output of the program and also find useful data for us like the object's type, its position on x-axis and on the y-axis.

As you can see on the above example, the *yellow cross* corresponds to the (0,0) coordinates and the *green cross* to the (448,448). An entity is localized by its center, for example, on this picture, the bottle's coordinates are the same as the *red cross* (254.241, 189.342).

All this data is contained in an array called **results**. Then, we simply returned the desired information in the form :

**"DETECTED    Object's type    Position X    Position Y    Width  
Height"**

The modified code can be seen in the file here : [https://gricad-gitlab.univ-grenoble-alpes.fr/Projets-INFO4/19-20/7/docs/-/blob/master/code\\_for\\_Movidius/yoloNCS\\_modified/py\\_examples/object\\_detection\\_a\\_pp.py](https://gricad-gitlab.univ-grenoble-alpes.fr/Projets-INFO4/19-20/7/docs/-/blob/master/code_for_Movidius/yoloNCS_modified/py_examples/object_detection_a_pp.py) (from line 177).

⇒ *How did we connect ros and recognition program :*

Once our object-detection program was able to compute every type recognized and his location, a major part of our work was to be able to collect these informations, analyzing them and finally sending instructions to the robot depending on them. The

goal we fixed ourselves was that RobAir (or rather our simulated robot) was capable of **eye-track** an entity identified by the object-detection program. With our simulation code, we assume that the camera would be on the robot's body and the simulated turtle would represent its head. So if an entity appears in front of the webcam and moves to the left of it, the turtle should rotate on her left and if the entity slightly moves to the right in the robot's point of view, the robot should slightly rotate on his right side.

To do that we first had to create two bash scripts (*run\_roscore.sh* and *run\_turtlebot.sh*) which would setup the ros environment by sourcing a *setup.bash* file in ros-kinetic, running roscore and starting the simulator. Then, we implemented two other scripts, the first one *run\_movidius.sh* setting-up the Movidius environment and then, launching the object-detection program (we can interrupt it with the "q" key). And the second one, *run\_process\_output\_movidius.sh* collecting what is written on the standard output and calling an awk file on each line of this output.

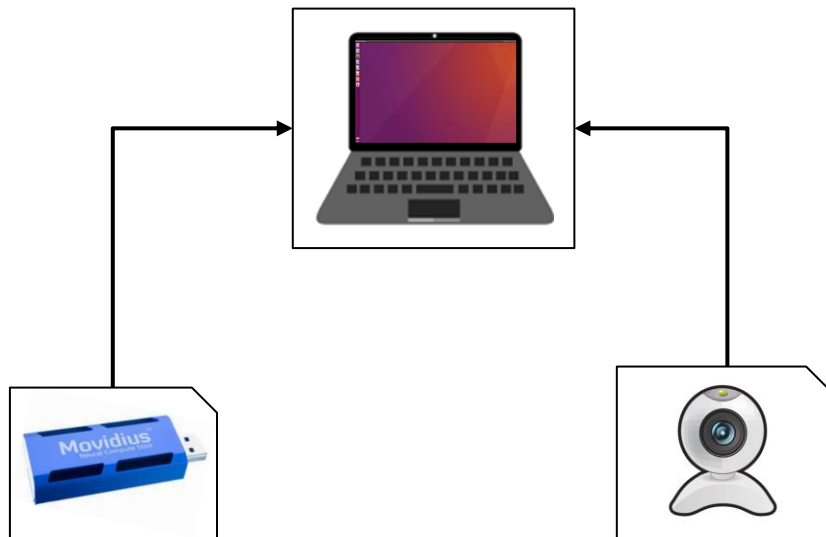
This awk file named **Motion.awk** is taking the first word of each line and if it is "**DETECTED**", it corresponds to the output of the object-detection program so it just have to analyse the third word which is the **x-motion** i-e the horizontal movement since the last call 1 second before. Depending on the value of this **motion value**, we wanted the robot to more or less turn right or left in order to always watch the entity as long as it is in his field of view. So this awk runs a command named **rosparam set angular x** with x being this motion value.

An additional script is calling these two previous scripts by piping the first with the second, which enables us to run the two parts of the process in concurrence.

*Angular* is the name of the the parameter we set in the **parameter server** of the **roscore**. So each second, it's updated by our script. Then, another script *setnode.sh* runs a node we created which is called **testnode.py**. This node firstly start a "Listener" Thread which is designed to get every second the new value of *angular* on the parameter server. Eventually every second, the main process of this file gets this angular variable and sends a *velocity\_message* to the simulator with *angular.z* which equals to its value.

Finally a last script named **yoloMain.py** run all these scripts with a "&" at the end in order to run them each in a subshell console in a concurrent way.

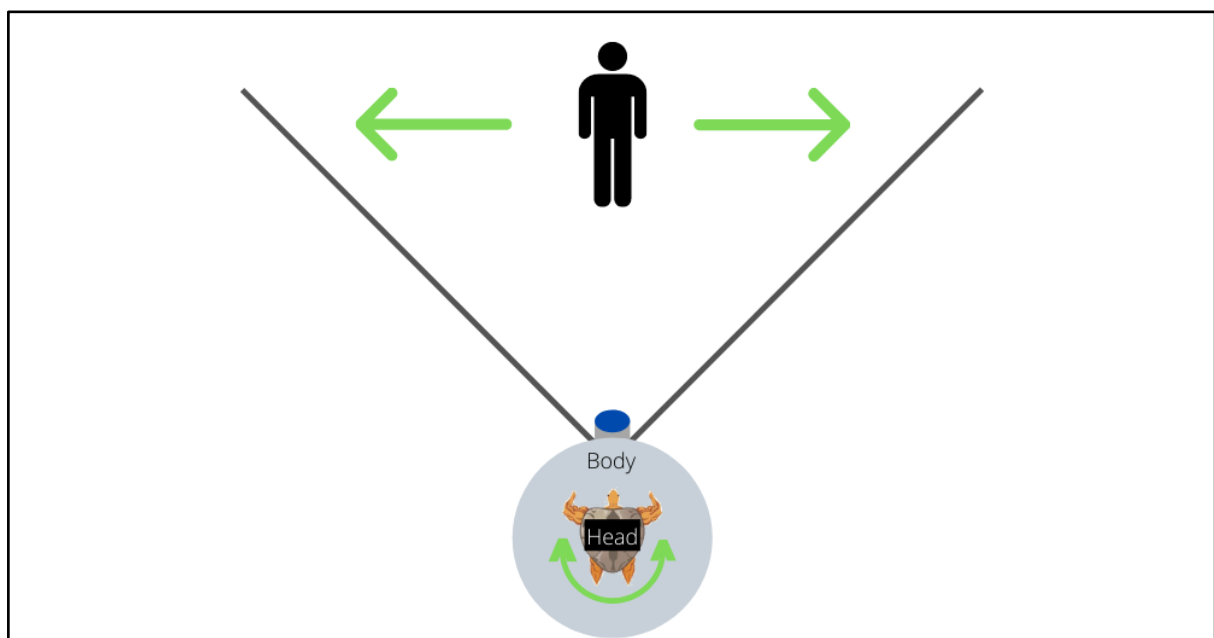
⇒ *Our set-up :*



## ❖ Results

As a result, our program is able to detect a person which is in the webcam's field of view, compute its location, analyzing it and send instructions to the robot in order to make him rotate. With our simulation code, we assume that the camera would be on the robot's body and the simulated turtle would represent its head. Si if an entity appears in front of the webcam and moves to the left of it, the turtle should rotate on her left.

We chose this way of simulation because, as we are confined, we couldn't simulate well the fact that the webcam's view is changing when the turtle is rotating.



So we have succeeded to make the turtle **eye-track** us. Our problem was *how to make the communication between the camera and the motor functions possible ?*

We made the communication possible by including ROS instructions into the object-detection program in terms of its outputs. Then, if we go on the left, our location on x-axis change with a certain way, the program detect it and sends an instruction to the TurtleBot the make it turn on the left like us.

A demo is available here : [https://gricad-gitlab.univ-grenoble-alpes.fr/Projets-INFO4/19-20/7/docs/-/blob/master/Eye\\_Tracking\\_Demo/eye\\_track\\_demo.mp4](https://gricad-gitlab.univ-grenoble-alpes.fr/Projets-INFO4/19-20/7/docs/-/blob/master/Eye_Tracking_Demo/eye_track_demo.mp4)  
It shows how we run the program and how the Turtle follows movements.

## CONCLUSION :

### Review :

As students, we chose this project because the subject differed a lot comparing to what we study. Indeed, we didn't have the opportunity to study robotics programming and it could be rewarding to discover it. During the first part of the semester, we struggled to progress in our project because of the environment that Movidius and Ros requires and the fact there are relatively few sites which demystify these technologies except their creator site. That's why we wanted to write documentation in our own words so as to enable future students working on these tools to have further informations and a documentation.

Once these two environments were successfully installed, the second part of the project was more interesting when we had to find a way to communicate the object detected and to make the robot move. However, because of the confinement, Rémy kept the Neural Compute Stick so he was the only one able to run the Movidius part of the program. So in order to test each part of the code we wrote, Baptiste always had to send it to him, that's why Rémy did the biggest part of the last commits on the repository.

### Prospects for the future :

As the second part of the project was during the confinement, we couldn't try to deploy our code and test it on RobAir. So as a prospect, we could adapt our code in order to make it work on RobAir by adapting some files.

For now, it can only eye-track somebody, another feature to implement would be to make RobAir be able to circumvent the people, or to stop before colliding. Finally, currently, if the person goes out of the camera field, the robot stops. Another feature would be to make this robot rotate until it finds back the person.