

Final Thesis

Bachelor in Industrial Technologies Engineering

Active vision system for searching, detecting and localizing objects in a real assistive apartment

Spring 2021

Author: Jaume Albardaner i Torras

Director: Sergi Foix Salmerón

Co-Director: Cecilio Angulo Bahón

Call: Spring 2021



Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona



Institut de Robòtica i
Informàtica Industrial



Abstract

Mobility impairment is a disability that affects movement. Said impairment can be caused by many diseases, but in most cases aging is the most significant factor.

People who suffer such disabilities have to limit the actions they can do as the disability increases. In cases where the disability is so great that the disabled person cannot live on its own, an external agent is required. This agent has to be capable of doing anything the disabled person is unable to do, which can be actions as basic and necessary as feeding oneself, or moving around looking for an object.

Usually these people are brought into nursing homes, so there can always be somebody ready to assist the disabled person. In other cases, the assistant is brought into the disabled person's house.

Although it is a common occurrence that disabled people prefer to be looked after by a human, there are situations where the caregiver is unable to assist the disabled person or in cases where his presence represents a threat to the patient's health (e. g. due to disease transmission). In such situations, having a robotic system taking care of the patient helps both parties.

Taking into account that there are countless actions a robotic system can take to assist the disabled person, the European Healthcare Robotics Technologies - Metrified (HEART-MET) competition addresses tasks that may directly impact the user's well-being. Some of the functionalities evaluated in this competition are the ones we will be implementing in this project, so we will take into account their evaluation system when developing it.



Contents

1	Introduction	7
1.1	Objectives	7
1.2	Scope of the project	7
1.3	Environment this project has been developed in	8
1.3.1	Robot Operating System (ROS)	9
1.3.2	HTC Vive	10
1.4	State of the art	11
1.4.1	Object detection	11
1.4.2	Efficient Search	12
2	Object detection	13
2.1	Introduction to object detection	13
2.1.1	How the switch from traditional CV to deep learning happened	13
2.2	Finding what algorithms to use for testing	14
2.2.1	COCO dataset	14
2.2.2	Evaluation criteria for the precision of a model	15
2.2.3	Selecting algorithms to test on	16
2.3	YOLO and EfficientDet: pros and cons	19
2.3.1	You Only Look Once (YOLO)	19
2.3.2	EfficientDet	20
2.4	Datasets used for training	22
2.4.1	YCB Dataset	22
2.4.2	First custom dataset (YCB Pringles extension)	23
2.4.3	Second custom dataset (YCB apple extension)	24
2.5	Training with YOLO	24
2.5.1	Training a single-object (banana) YCB model	24
2.5.2	Results for the single-object (banana) model	29
2.5.3	Training a multi-object model	30
2.5.4	Results for the multi-object model	31
2.5.5	Training and results for the Pringles custom dataset	32
2.5.6	Training and results for the apple custom dataset	33
2.6	Training using EfficientDet	33
2.7	Comparison of results and chosen algorithm	35
3	Efficient Search	36
3.1	Zenithal camera's properties	36
3.1.1	Camera movement	36
3.1.2	Camera Image	39
3.1.3	Camera zoom configuration	40
3.2	Probabilistic map	44
3.2.1	Preparations before the collection of data	45
3.2.2	Clustering algorithm	46
3.2.3	Capturing data using HTC Vive	47
3.2.4	Preparing the camera to look for an object	48
3.2.5	Efficient sorting of the presets	50
4	Integration of our project on the laboratory's system	53

4.1	Node structure of our project	53
4.2	Integration of every ROS node	54
4.2.1	htc_capture:	54
4.2.2	object_database:	55
4.2.3	object_clusters:	56
4.2.4	camera_presets:	57
4.2.5	camera_move:	58
4.2.6	camera_order_presets:	59
4.2.7	search_object_poi:	59
5	Future work	62
6	Project's cost	64
6.1	Personnel	64
6.2	Tools and objects	64
6.3	Electricity	65
6.4	Total cost	66
7	Climate Impact	67
	Conclusions	68
	Acknowledgments	69
	References	70
A	Data obtained during the experiment from Section 3.1.2 to find the most suitable zoom configuration.	72
B	Data obtained from the experiment to calculate the angle/pixel relation.	73
C	Experiment ran to test how fast each method of sorting presets is, starting from a preset.	74
D	Experiment ran to test how fast each method of sorting presets is, starting from a random location.	76
	List of Figures	
1	Flowchart of the actions of our system once fully developed.	7
2	TIAGo mobile manipulator used in our project.	7
3	Assistive apartment in the Perception and Manipulation laboratory, where this project has been developed in.	8
4	Example of an ArUco marker.	9
5	HTC Vive equipment we used to collect simulated data of an object's whereabouts.	10
6	(a) Traditional CV workflow vs. (b) Deep Learning workflow. Figure from [10].	13
7	Example of (a) iconic object images, (b) iconic scene images, and (c) non-iconic images. Figure extracted from [15].	14
8	Equation used to calculate the Intersect over Union (IoU) metric. Figure extracted from www.pyimagesearch.com	15
9	Examples of IoU computations. Figure extracted from www.pyimagesearch.com	16

10	Performance of YOLOv5 on the COCO dataset. Figure extracted from https://github.com/ultralytics/yolov5	18
11	Performance of the top state-of-the-art algorithms on the COCO dataset. Figure extracted from https://github.com/AlexeyAB/darknet	19
12	Process followed by R-CNN (and most two-step algorithms) when performing object detection on an image. Figure extracted from [6].	20
13	Process followed by the YOLO CNN to compute the detections on an image. Figure extracted from [3]	21
14	YOLO's complete detection process.(1) Resizes the image to the CNN dimensions, (2) runs the CNN, (3) thresholds the resulting detections. Figure extracted from [3]	22
15	EfficientDet's architecture. Figure extracted from [8]	22
16	All the objects that compose the YCB Dataset.	23
17	Actions we performed when collecting data for the Pringles dataset to increase accuracy.	24
18	Visual representation of the similarities between the color of our apple and another training prop, while also displaying the apple as a non-geometrical object. .	25
19	Usage of the Yolo-mark program to check labelling and bounding boxes.	25
20	Visual representation of three different bounding box standards. The coordinates are always given according to the reference system seen in (a).	26
21	Type of images we were provided with.	27
22	Depiction of the available material for training.	27
23	Loss graph (with mAP) for a YOLO model across different training iterations, where we can see that in a normal YOLO training process, the loss decreases incredibly fast at the beginning. Its loss decreases less as iterations pass.	29
24	Image of the results obtained by the model trained with only a banana.	30
25	Loss graph of an already trained model, trained for additional classes. In it we can see the loss does not decrease, and that the model does not start training at iteration 0.	31
26	Detection obtained with a multi-object detecting YOLO model.	31
27	Precision obtained on the Pringles can using different configurations for range and camera zoom.	32
28	Precision obtained on the apple using different configurations for range and camera zoom.	33
29	Class loss after 100 iterations on the custom Pringles dataset.	34
30	Overall loss after 100 iterations on the custom Pringles dataset.	34
31	Output obtained from testing our trained EfficientDet model on its training data. .	35
32	Our laboratory's Zenithal camera with its corresponding pitch and yaw axis. . .	36
33	Camera movement around its axis. (<i>Green</i> corresponds to the available positions for the camera. <i>Red</i> marks the unavailable ones).	37
34	Map of the lab generated using all available presets without zooming in.	38
35	Horizontal and vertical distortion due to the camera's lens with no zoom. (<i>Green</i> - distorted, <i>Red</i> - undistorted)	39
36	Correction applied to the images from the camera to remove distortion.	40
37	Locations in which we set the objects to find the best zoom configuration. <i>Green</i> = Close / <i>Blue</i> = Mid / <i>Red</i> = Far	41
38	Average precision obtained for different objects on different zoom configurations. .	42
39	MasterChef props used during the training and testing phase.	43
40	Cropped results obtained in each zoom configuration in a middle-ranged position. .	43



41	Lens distortion with the camera set at 1.5s of zoom.	44
42	Map of the lab generated by using all available presets zooming in 1.5s.	44
43	Change experienced by an occupation map that starts with no data, and finished execution having recorded the movement of a Marker moving in a circular pattern.	45
44	Degrees of freedom the newly implemented Marker has. We also display that we are now able to move the Marker freely to record data in the locations we want.	46
45	Clusters obtained from a Marker moved both manually and in a circular manner.	47
46	Portrayal of the HTC Vive equipment working on the laboratory's ROS environment.	48
47	Clusters obtained from the data recorded using the HTC Vive.	49
48	TIAGo manipulator robot on the POI that corresponds to the sink.	50
49	ROS nodes with the services they use for communication.	53
50	GUI that was implemented to run the htc_capture ROS node.	55
51	GUI used to interact with the search_object_poi node in order to either prepare the system to search for an object or search for one.	60

List of Tables

1	Description of how predictions are classified.	16
2	Experiment ran to test speed of algorithm	52
3	Monetary value given to the time spent by the student and tutor.	64
4	Price for all the objects we used in this project	65
5	Time it has taken to train each YOLO model.	65
6	Table computing the monetary cost	66
7	Total monetary cost of the project	66
8	Experiment ran to test how fast each method works on preset locations.	75
9	Experiment ran to test how fast each method works on random camera positions.	77

1 Introduction

1.1 Objectives

The objective of this thesis has consisted in developing an assistive system that helps in the search, identification and estimation of the position of previously established objects within an apartment. We split this project in two different parts, each covering a functionality:

- Enabling the system to detect and identify objects in images taken from a zenithal camera.
- Minimize the amount of time the system takes to search for an object.

The procedure that the system had to follow is described in Figure 1. The overall process can be simplified into these steps:

1. The name of an object is given to the system.
2. A set of images is taken from a camera located at the zenith of the apartment (the camera can rotate around its pitch and yaw axis in order to point to different locations).
3. Object detection algorithms are applied on the images as they are taken.
4. If the given object is detected, the system has to compute its location so that a TIAGo mobile manipulator (Figure 2) can navigate towards it.

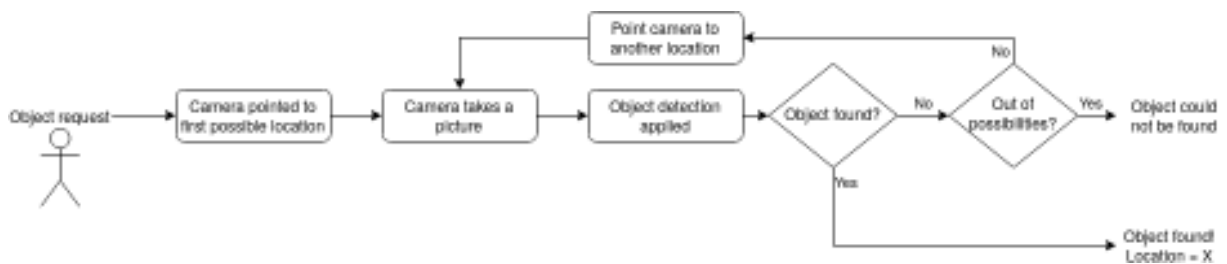


Figure 1: Flowchart of the actions of our system once fully developed.

The part of the project that allows the system to detect and identify objects represents a single iteration of the loop observed in Figure 1. Meanwhile, the second part of the project does not only allow the system to perform the loop by sequentially moving the camera. It also tries to make the whole process as fast as possible by appropriately sorting the order in which locations must be visited.

1.2 Scope of the project

Life expectancy for humans is increasing globally [1], and the conditions impaired people live in has improved thanks to technological advances. Thus, it is becoming more urgent to provide these assistance-requiring collectives with tools that can improve their freedom and quality of life.

Since aging and certain diseases can limit the actions these collectives can perform, a way of improving their quality of life would consist in provid-



Figure 2: TIAGo mobile manipulator used in our project.



ing them with robots that could perform actions that would otherwise be difficult for the person to do.

In order to evaluate how well robots perform different functionalities, the METRICS project¹ was established to organize challenge-led robotics competitions in the topics of Healthcare, Inspection and Maintenance, Agri-Food, and Agile Production. The competition in the topic of Healthcare (Healthcare Robotics Technologies - Metrified, HEART-MET) evaluates different functionalities from the robots. These functionalities are required by the robots to complete tasks that help a patient in a domestic environment. Since the topics we have worked on this project are in the scope of the HEART-MET competition, we have taken into account their evaluation plan when developing our project.

1.3 Environment this project has been developed in

The entirety of this project has been developed in the Perception and Manipulation Laboratory (Figure 3) at the *Institut de Robòtica i Informàtica Industrial (IRI)* research institute. A large part of the laboratory's space is taken by a life-scale mock-up of a small apartment. The apartment hosts two TIAGo mobile manipulator robots that can move freely inside and outside of the apartment.



Figure 3: Assistive apartment in the Perception and Manipulation laboratory, where this project has been developed in.

The apartment provides a variety of tools to work with:

- **Amazon echo dot:** a voice detecting device that may be used to perform a request to the

¹METRICS project website: <https://metricsproject.eu/>

assistive apartment's control system.

- **IoT sensors/actuators:** Internet of Things (IoT) devices that are able to monitor the laboratory's current temperature, control the blinds depending on the UV input detected, informs of what doors are open, and many more things.
- **Amcrest camera:** camera that can rotate horizontally and vertically. It uses the network to communicate, from where it can both receive orders or send feedback.
- **TIAGo mobile manipulator:** robot with wheels that enable it to move around. It also has an arm to which a device can be attached to interact with objects.

For this project we were mainly interested in working with the Amcrest camera and the TIAGo mobile manipulator.

The camera was an interesting tool to choose because it was located at the zenith of the apartment, and it could see it in its entirety. Furthermore, it had been used previously on another project ([2]) to detect ArUco markers (Figure 4), and it performed well.

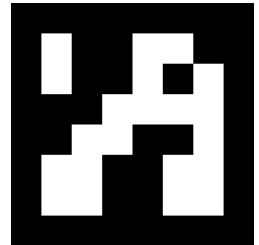


Figure 4: Example of an ArUco marker.

In the making of that project, the camera was linked with the devices we specified previously in this section. They were all connected to a Raspberry that was running an OpenHAB service ². By the time we started making some tests with the camera, we discovered that it was no longer accessible from the OpenHAB platform. In order to communicate with it, we ended up using a Python library and some snippets from the code that was developed in [2].

Secondly, we wanted to use the TIAGo mobile manipulator as a tool to interact with the object once we discovered its location. Unlike with the camera, communication was not achieved with the use of a library. Communication with the TIAGo was achieved using its built-in ROS messages.

1.3.1 Robot Operating System (ROS)

ROS is a flexible framework whose objective is to simplify the writing of robot software. In order to do so, its designers provide its users with a collection of tools, libraries and conventions.

When a robot wants to communicate with another robot, they will both do so using an executable file. The executable that wants to send a message will publish a message onto the local network, and the executable that wants to receive it will have to be constantly checking for new packages in the network. These executables are called "nodes" under the ROS naming system.

Let's say that another node that wants to publish his messages joins the network. Now the node that was initially listening just one node's publications is now listening to two. As a consequence, ROS established a system where a node has to specify the topic at which it wants to publish/receive a message to/from.

²OpenHAB is a home automation open-source platform. It is capable of linking different devices without accounting for the communication protocol followed by each.



Another important property that must be accounted for is that each topic can only accept one type of message. This property was implemented to prevent differences in object types among publishers and subscribers of a same topic. To exemplify, this prevents numbers from being published in a topic where all listeners are expecting words.

Having this communication method as the basis, many more procedures were implemented for communication. Out of all the options these are the ones we have used in this project:

- **Service:** A node is permanently listening to a topic. When a message is published, the node takes its information, performs a computation, and returns its result directly to the node that published the message .
- **Action:** An action is a service that can be cancelled. In cases where the computation done by a service takes too long to complete, it may be more favorable to interrupt it and continue doing something else.

Aside from the equipment the laboratory already had, during this project a device with precise tracking was installed: an HTC Vive. It helped collecting data that portrayed real information of the apartment (sensible coordinates).

1.3.2 HTC Vive

The HTC Vive is a Virtual Reality (VR) ³ kit mostly used for playing immersive video-games, where the player can have a better experience by using this type of hardware.

This hardware was interesting to use because it had great tracking properties, which could lead us to accurate results when creating the probabilistic map. Moreover, since we could communicate with it using ROS, in any time we wanted we were able to check on the locations of each controller and tracker (Figure 5 (a) and (b)). They were detected by the sensors (Figure 5 (c)) and their whereabouts were published in a *tf2_msgs/TFMessage* message.

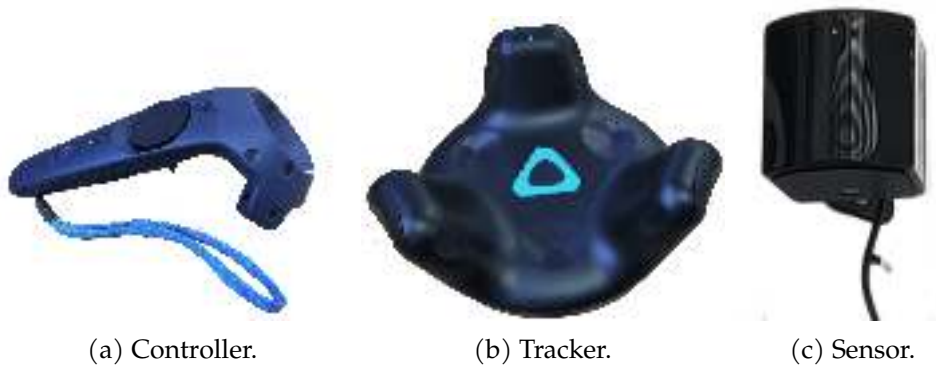


Figure 5: HTC Vive equipment we used to collect simulated data of an object's whereabouts.

³Virtual Reality is an experience where the user immerses itself in a simulation using specialized equipment

1.4 State of the art

As it was introduced in Section 1.1, this project is composed by two well-differentiated parts: object detection and efficient search (by efficient we specifically mean time-efficient).

1.4.1 Object detection

The most accurate and fast object detection algorithms currently used anywhere involve the use of neural networks. These neural networks can usually be described as a "black box" at which one can introduce an input and obtain an output.

There have been lots of algorithms that have worked with different versions of this "black box", and the ones that have become top-performing object detection algorithms are convolutional neural networks (from now on CNNs). These networks are different from normal neural networks because although they are formed by internally connected layers of neurons⁴ too, the connectivity pattern between them resembles the structure real neurons have in the animal visual cortex. Each neuron takes care of part of the introduced image. This change in connectivity entails a lower training period and a higher precision than previously used methods.

Deep learning models can be classified in two groups: *one-stage* and *two-stage* models. Their difference lays on the fact that two-stage models have to run the image twice through the CNN in a sequential manner. One-stage algorithms may run the image more than once through the CNN, but they perform the computations in parallel.

One-stage models were initially designed to minimize execution time, even if this implied being less precise. The most famous (based on both speed and precision) are the YOLO series [3], SSD [4] and RetinaNet [5].

Two-stage algorithms try to maximize precision without accounting for computing time. The most broadly used are the R-CNN series [6].

Until 2017, two-stage algorithms obtained the best results (Faster R-CNN, to be precise). It was with the introduction of RetinaNet [5] that balance shifted towards one-stage. Since then, computer vision (CV) algorithms have been rapidly improving their performance.

As a reference of rapid growth, YOLO has had 5 different versions in the last 6 years. Although the amount of different versions does not necessarily mean a better performance, YOLO currently obtains the best results when working with the most complicated datasets. To be precise, these versions are Scaled-YOLOv4 [7] and YOLOv5.

For this project we have used Scaled-YOLOv4. Despite theoretically performing worse than YOLOv5 (according to YOLOv5's authors), it has been acknowledged by the author of YOLO's original framework as the canonical version. It has also been through rigorous testing.

Meanwhile, YOLOv5 has barely gone through any testing, and the few it has gone through are difficult to reproduce. It is not even a direct improvement upon the original YOLO framework, but rather an improvement of a variant. At last but not least, YOLOv5 does not have a published

⁴Neurons in each layer perform a non-linear sum of their entries (the outputs from the previous layer), apply a weight to the result, and send it to neurons in the next layer



paper even one year after release.

After training our data in the Scaled-YOLOv4 framework, we would be comparing it with another framework that had a similar performance (or even better) in the same datasets. According to Scaled-YOLOv4's paper [7], the best framework to compare it with was EfficientDet [8], which had been the most accurate state of the art object detecting algorithm before the launch of Scaled-YOLOv4.

1.4.2 Efficient Search

Efficient object search algorithms can be classified in two types [9]: those that follow direct methods and those that rely on indirect ones.

- The direct methods create a probabilistic map with the information provided by the environment. Applying a cost function the algorithm then decides what instructions must be followed to detect the object as fast as possible.
- Indirect methods rely on the relations between the object the system is looking for, and those that usually surround it. This way instead of looking directly for the requested object, the system looks for objects related to it. It is when it detects one of these related objects that it knows that it is close to detecting the requested object.

From the two types of algorithms we chose to continue with a direct method. Because creating a probability map with the usual locations of an object was simpler than establishing the relations with all surrounding objects and, as stated by Alejandra C. Hernandez et al. [9], it still provided good results.

After implementing this method we initially wanted to compare it with other popular methods that serve as a baseline:

- **Brute force:** Checking every possible combination in an ordered manner. This can lead to good results if the object to search is located at the first part of the searching sequence. Likewise, it can also lead to bad results if the object is located at the final part of the sequence.
- **Random search:** Checking every possible combination in a random manner. The good part of this option is that it has no preferences when choosing where to look (equiprobability), so in one run we may take a long time to find an object, and in another run we may find it in an instant, despite it being in the same location.

Due to the zoom configuration we ended up having to work with (Sec. 3.1.3), we could not map the entire apartment efficiently. We were only able to point the camera at the likeliest places for the object to be in. Thus, we were unable to conduct the brute force and random search experiments. In their place, we prepared other experiments that we could actually extract conclusions from. They are explained in Section 3.2.5.

2 Object detection

2.1 Introduction to object detection

There has always been a need for robots to understand their surroundings, because it would allow them to adjust their actions depending on what situation they are in. If the robot disposes of a camera, a possible way for them to do so would be by processing wisely the data coming from it.

2.1.1 How the switch from traditional CV to deep learning happened

Different CV algorithms existed for that purpose, which allowed object identification in images (as described in [10]), such as feature descriptors (SIFT, SURF, BRIEF, etc.). Despite working properly, there was one issue with these type of CV algorithms: they required an additional step to be taken (compared to current methods), as seen in Figure 6 (a). The problem with this extra step is that it required the knowledge and time of a CV engineer. His task consisted in selecting what features were important in each image. As the number of classes per image increased, the amount of features to be extracted also increased. This made it difficult to work with large datasets.

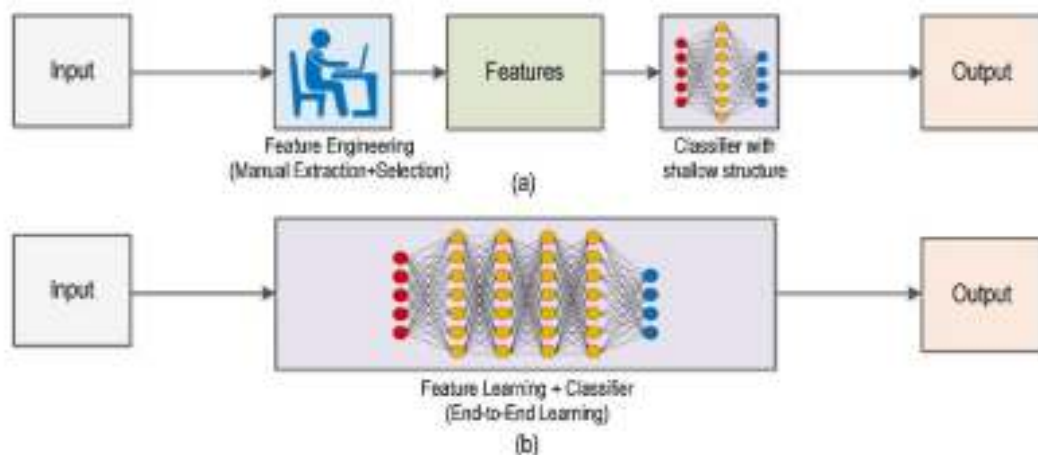


Figure 6: (a) Traditional CV workflow vs. (b) Deep Learning workflow. Figure from [10].

Thanks to hardware advances, training neural networks in GPUs became feasible. The importance of this fact is that the training period was reduced by orders of magnitude: what would have taken weeks to train before, now only took some days. This led to a revolution in Deep Learning⁵. Moreover, the recent introduction of the convolution operation in neural networks (subsequently called CNNs) has pushed the boundaries of what is possible.

Due to this increase in computational power, the role of selecting what features were relevant in each picture was passed onto the Neural Networks. They now took care of the entire process, as portrayed in Figure 6 (b).

⁵Deep Learning is a technique used in Machine Learning to model the neural network after the structure of human brain.



It was now so easy to train a model that its user's only responsibility was to "feed" the neural network with images that contained the class to be trained. After training, when an image is passed through the model, it compares it to the classes it has learned from its training input. As an example, if the model is trained to identify cats, it looks for the idea the method has gained of the concept "cat" (from the training data). With previous methods, it would look for "whiskers", "paws", "tail", etc.. Thus, there is no need for statistical knowledge to train a Neural Network, it is only necessary to know how the Neural Network wants the input to be formatted like.

Now anybody was able to run an object detection algorithm without the need for the statistical knowledge that was once required. In addition to this simplification of the process, if we also take into account the explosive growth autonomous vehicles are having, CV algorithms are seeing an unprecedented progress.

2.2 Finding what algorithms to use for testing

2.2.1 COCO dataset

Since there are many algorithms used for object detection, it is almost mandatory to compare their performance to choose those that perform better. In order to evaluate them it is necessary that they work with the same objects, which have to be objects that the robot might encounter regularly that pose a challenge for the detection algorithms. The CV community made some datasets to test different detection functionalities, like object attributes [11], scene attributes [12], human body parts [13], or indoor information [14].

In 2014 Microsoft released a dataset known as COCO [15] (Common Objects in CONTEXT). This large-scale dataset addresses three main issues found in CV algorithms when detecting objects:



Figure 7: Example of (a) iconic object images, (b) iconic scene images, and (c) non-iconic images. Figure extracted from [15].

- See objects from unusual perspectives (Figure 7): The algorithms explored until that time worked fine when the object to detect was presented to them clearly (iconic images). However, if the object to search is occluded by another object, if it is located at the back of the image or amid clutter, the algorithm has a difficult time finding it. In case it actually detects an object, it has a hard time classifying it.
- Make use of contextual reasoning: The algorithms find it difficult to decide what class a detected object belongs to. In most cases this is due to the fact that the detected object is

small. This problem could be solved by observing the environment in which the detection has been made, and classifying the detection in a class that would be reasonable in said environment.

- **Find the 2D location of objects inside the image:** It is necessary to know how algorithms can compute the bounding box around the objects that compose the image. This would also help the robot get an idea of how objects are distributed around it based on where the objects are located in the image.

In order to obtain good results with all the previously described issues, the COCO dataset provides with 91 different object categories. Out of all of them, 82 are instanced in pictures more than 5,000 times. This dataset is constructed on 2,500,000 labeled instances in 328,000 different images.

Although it has less categories than another famous dataset, such as ImageNet [16], the COCO dataset provides with more instances per category. This can help algorithms locate objects in pictures easily, effectively increasing the precision of the model. This dataset has more instances per image on average than other famous datasets: 7.7 instances per image compared to the 3.0 of the ImageNet dataset or the 2.3 from the PASCAL dataset [17].

It is due to the described properties that COCO is such a renowned dataset when comparing different object detection algorithms.

2.2.2 Evaluation criteria for the precision of a model

There are two functionalities to be evaluated: object classification and 2D location of objects in images. Thus, there are two metrics to be used:

- **IoU (Intersection over Union):** It is a metric used to calculate how similar a calculated bounding box is to the ground truth (hand-labeled) bounding box. The equation used to calculate it is described in Figure 8.


$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$


Figure 8: Equation used to calculate the Intersect over Union (IoU) metric. Figure extracted from www.pyimagesearch.com.

As it can be seen in Figure 9, bounding boxes that overlap the most (like the ones on the



right of the figure) obtain a higher IoU score than any others. Bounding boxes that do not achieve an IoU score greater than a certain threshold are not considered to be correct, and are labeled as *False Positives*.



Figure 9: Examples of IoU computations. Figure extracted from www.pyimagesearch.com

- **mAP (mean Average Precision):** Before trying to understand mAP, we must beforehand know what precision is. Precision consists in measuring the percentage of correct predictions among all predictions made. In Table 1 this corresponds to the sum of the diagonal divided by the sum of all occurrences. In the case of object detection, a certain score cutoff is established: All detections with a score lower than the cutoff are treated as a false positive.

		Actual	
		Positive	Negative
Predicted	Positive	True Positive	False Positive
	Negative	False Negative	True Negative

Table 1: Description of how predictions are classified.

When we run the object detection algorithm in order to detect a certain class, the score cutoff has a direct influence on the obtained precision. This is why the average precision for all values of cutoff is computed. Doing so makes comparison between different models easier, as the average precision computed is non-dependent on the cutoff value.

After calculating the Average Precision of all categories, the mAP (mean Average Precision) can be computed. It is the metric that defines how precise an algorithm can be when detecting any type of object. To calculate it, one just has to compute the mean of the Average Precision of all categories.

2.2.3 Selecting algorithms to test on

When choosing the first object detection algorithm we were going to test on, we took into account the metric we specified in the last part of Sec. 2.2.2: mAP. And taking into account how relevant the COCO dataset was, we would be selecting the algorithm that achieves a higher mAP in this dataset.

There is an issue when selecting an algorithm out of all of them: the rapid evolution of CV algorithms. Although it was a good thing that these algorithms were progressing so rapidly, it

does not help when it comes to choosing one. Mainly because after having made a choice and trained the chosen model, a newer model with a better properties than the one we chose might already have been released.

In order to understand why it was somewhat difficult to choose one we must first take a look at how much algorithms progressed in the latest years.

- **2014:**
 - Microsoft COCO is released, opening a world of challenges for CV algorithms.
- **2015:**
 - Faster R-CNN is released, a two-step algorithm that obtains a mAP of 21.2 ⁶.
 - SSD is released, a one-step algorithm that focuses on inference speed. With the right configuration it can achieve a mAP of 28.8, but Faster R-CNN still outperforms with smaller detections.
 - YOLO is released, an algorithm that is capable of running at 155 FPS by sacrificing a big part of its precision. It is not tested on the COCO dataset on release.
- **2016:**
 - YOLOv2 (also known as YOLO9000) is released. It achieves a mAP of 21.6. Although it is lower than SSD, its inference time is of 25ms, 5 times faster.
- **2017:**
 - RetinaNet makes its appearance, with its mAP of 37.8. Creating a big gap with all the other methods, and setting one-step methods in front of two-step ones.
- **2018:**
 - YOLOv3 is released. This version can perform with a mAP of 28.2 in only 22ms inference time, or it can work with a mAP of 33 at 51ms inference time. This version leaves SSD behind, as it surpasses it in every way. It is also the last version uploaded by one of its original creators ⁷.
- **2019:**
 - EfficientDet is released. It is one of the current best algorithms to use, as it has a mAP of 55.1. The highest one until this point. The only counterpart is that its inference time is really big in order to achieve such precision.
- **2020:**

⁶Unless specified otherwise, all mAP scores are obtained from evaluation with the COCO dataset.

⁷Due to privacy concerns and the project being given military applications, one of the two creators of all YOLO versions, Joseph Redmon, publicly announced he would not involve himself further in this project.



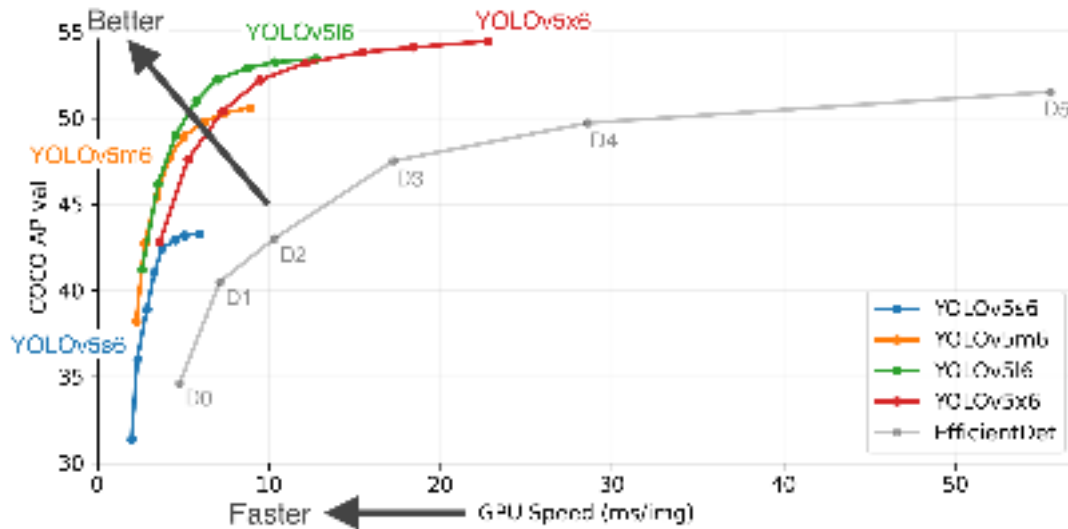


Figure 10: Performance of YOLOv5 on the COCO dataset. Figure extracted from <https://github.com/ultralytics/yolov5>.

- YOLOv4 is released, with a big support from the CV community. Even one of the creators of the YOLO architecture considered this version the canonical version. It achieves a mAP of 43 while running twice as fast as EfficientDet. This speed allows it to run in real-time.
- YOLOv5 is released one month later, without a paper, and without basing itself off of YOLOv4. Moreover, it claims to have the best precision (Even better than EfficientDet) without making any comparisons to YOLOv4, as we can see in Figure 10.
- YOLOv4 gets an update seven months later: Scaled-YOLOv4. With this new update YOLOv4 gets a mAP of 55.5, making it the most precise with the lowest inference time algorithm.

The current state of the top performing algorithms (discarding YOLOv5) on the COCO dataset can be viewed in Figure 11. In this figure we can see in a more visual way what we explained in this section.

Although it should be clear that the algorithm with which we should start is the latest version of YOLOv4, making sure that we were using the right version is another story. When we entered the official GitHub repository of the YOLOv4 algorithm (<https://github.com/AlexeyAB/darknet>), we could see that there was no clear distinction between the different versions. This is why training a model in EfficientDet did not sound like a bad idea at the beginning, as there was only one version no confusion could be made.

Despite the confusion with the different versions, in the end we decided to begin testing with the official YOLO implementation. The reason being that the algorithm was better-documented, allowing its users to not need any previous experience in order to use it.

It would be after gaining experience with YOLO that we would then attempt to train a model

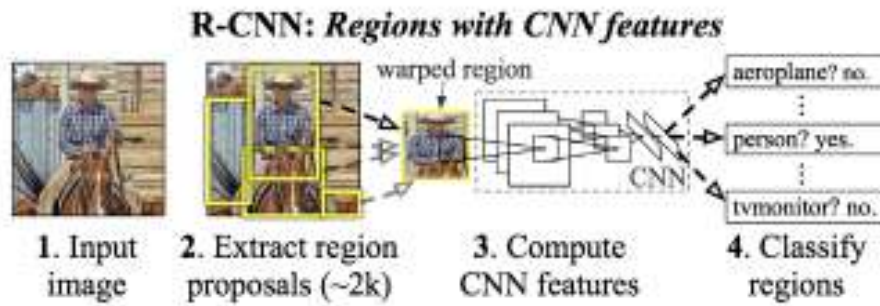


Figure 12: Process followed by R-CNN (and most two-step algorithms) when performing object detection on an image. Figure extracted from [6].

YOLO solved those problems by applying the methodology shown in Figures 13 and 14. By making the algorithm look only once (one-step) at the image, it automatically got rid of all the stated issues, hence its name of You Only Look Once. The process followed by the algorithm is the following:

1. The input image is resized to the dimension the Neural network has been trained in.
2. The image is sectioned in a grid, and in each area of the grid the two following operations are applied:
 - Bounding boxes for possible detections are computed.
 - The odds at finding a certain class in each section is computed.
3. After the computation, the results are brought together to select the boxes located in sections of the image where a class has been detected.

Although this algorithm solved the main issues presented in two-step algorithms, its precision was not initially as good. Despite this, it did a good job at running the Convolutional Neural Network the fastest at that time.

One year after the release of the first version, YOLOv2 was released. It allowed the model to be trained on a dataset with 9000 classes. With this it became the first model to be able to predict this amount of classes in real-time.

Further changes were done on the following versions (YOLOv3, YOLOv4, Scaled-YOLOv4, YOLOv5). These changes were mostly focused on improving performance. Although in itself it may not seem like something worth mentioning, these changes led YOLO to currently become the most accurate and fastest object detection method to be ran on the COCO dataset.

2.3.2 EfficientDet

In Section 2.2.3 we explained how in 2017 the introduction of RetinaNet [5] represented an important advance towards more accurate object detection. Although in this case it was a lightweight model, that was not the case for all high-precision models. With time, models that

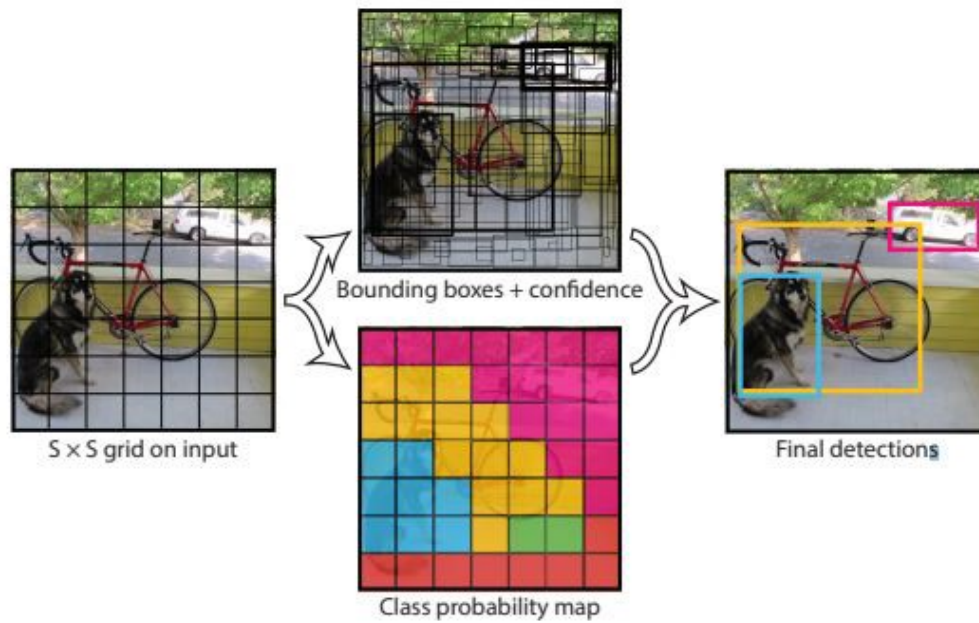


Figure 13: Process followed by the YOLO CNN to compute the detections on an image. Figure extracted from [3]

aimed for the highest accuracy had been getting increasingly heavier. Even the slightest increase in precision could result in a heavier demand of resources to run the model (even higher for training).

As a consequence, trying to make a model as efficient as possible was becoming a trend. By the time EfficientDet came out, alternate models that focused on efficiency had already been released, like YOLO (v3 at the time), SSD, or RetinaNet. Efficient models usually ended up sacrificing accuracy in order to use less resources.

On the 28th of May from 2019, EfficientNet [18] was released. It was not an algorithm for object detection (hence why we did not introduce it in Section 2.2.3), but rather for image classification. In any case, this algorithm was developed in a way that precision adapted to the device's resources: if the device had more resources, precision would be higher. Although this is how current models work, they used to be developed at a fixed resource budget.

The same developers decided they would use EfficientNet as a backbone for an object detecting model, which would preserve the scalability while increasing accuracy (by implementing a special layer in the CNN). They chose to name it EfficientDet (Det for Detection), and its architecture can be seen in Figure 15.

Although its architecture now looks similar to YOLO's, when it first came out it was very innovative. That was what made EfficientDet be the best object detecting method, at least until the arrival of Scaled-YOLOv4.



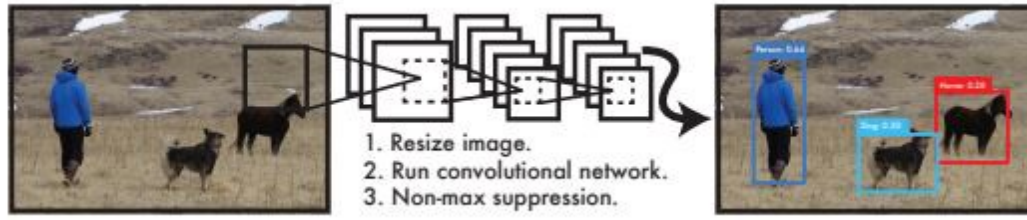


Figure 14: YOLO's complete detection process.(1) Resizes the image to the CNN dimensions, (2) runs the CNN, (3) thresholds the resulting detections. Figure extracted from [3]

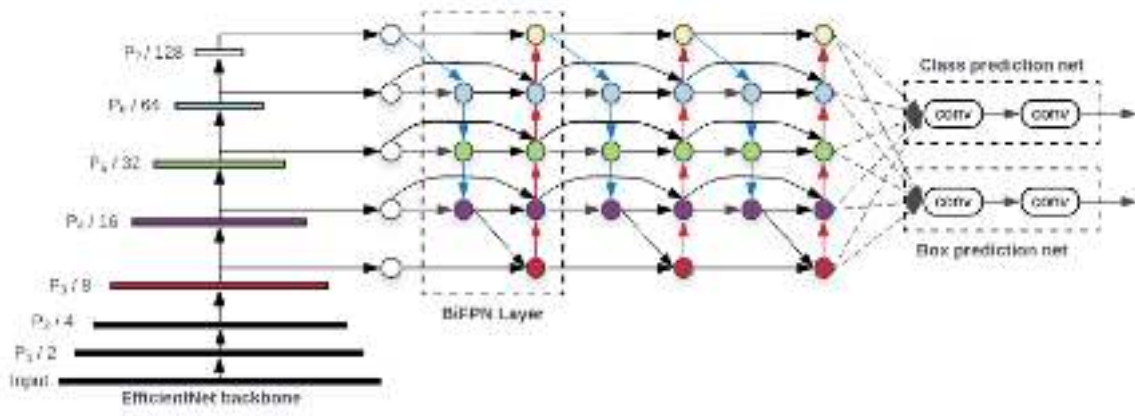


Figure 15: EfficientDet's architecture. Figure extracted from [8]

2.4 Datasets used for training

In this section we have grouped all the datasets we have used (and created) to train both YOLO (in Section 2.5) and EfficientDet (in Section 2.6)

2.4.1 YCB Dataset

Although a lot of attention has been brought onto the COCO dataset, some of its classes do not appear in the environment this project will be implemented in. As an example, it is highly unlikely that an elephant will be found inside an assistive apartment. And it is much less likely that the system will be asked to interact with it. Thus there is no need to detect such classes.

Although having more classes in itself is not a bad thing, the main issue why we have not used this dataset (COCO) is because it does not have the classes that are present in our apartment. The dataset that has classes that overlap the most with what we have present in the apartment is the YCB dataset [19] (as it was bought a few years ago). An additional reason to choose this dataset was that it has been used in the past in the HEART-MET competition.

The classes we find in this dataset are separated into five categories: Food items, Kitchen items, tool items, Shape items and Task items. The entirety of the dataset can be seen in Figure 16.

In order to train our model we required images where different classes appeared with their



Figure 16: All the objects that compose the YCB Dataset.

respective bounding boxes. The entire work would have had to be done by us had we not found a source with a subset of the dataset. In a website ⁸ we were provided with 133827 frames in which a total of 21 objects appeared. Despite 21 object not coming close to the 77 that compose the dataset, it served as a good estimation if in the future we wanted to cover the entire dataset.

With this dataset, we trained a pair of models (Sec. 2.5.1 and 2.5.3), which did not perform as good as we expected. Furthermore, we also wanted to know what process was needed to train an object of our choice. Thus we decided to train a custom object.

2.4.2 First custom dataset (YCB Pringles extension)

The first thing that is needed when creating an image dataset of an object, is choosing what object to train. The properties of each object must be accounted for before making a choice.

Out of the different props we had selected as candidates to create a custom dataset with, we chose a Pringles can. The reason being that it should be an easy to find object, due to its physical properties:

1. **Easy geometry:** A cylinder is an easy to identify object. Its surfaces are well defined.
2. **Easy colour:** Pringles cans are famous for having a vivid red, which is not common to find at the lab's apartment.
3. **It has text:** Even if the model may not be able to learn the past two properties, it will most likely be able to read "Pringles" in an image, as long as sufficient training data has been provided.

After choosing an object, obtaining the images that would compose the dataset was rather simple. Just like the dataset's author had done, we recorded a series of videos out of which we extracted the frames.

When filming the videos, some rules were to be followed if we wanted our model to be as accurate as possible while still remaining mergeable with the downloaded dataset (just in case we ever want to train a model that uses both the custom and downloaded dataset).

⁸Website's URL: <https://kyouma9s.com/ycb-video-dataset-download-mirror/>



1. **Resolution:** The images had to have a 640x480 resolution. This way the custom dataset could be combined with the downloaded one.
2. **Illumination:** The images must be taken in different lighting conditions (Figure 17 (b)).
3. **Object pose:** The object must be recorded in different configurations, be it a simple change of location or a rotation (Figure 17 (a)).
4. **Surrounding objects:** There has to be at least an object that looks similar to the target object in color or/and shape. It is also highly recommended for surrounding objects to occlude the target object somehow in some pictures (Figure 17 (c)).

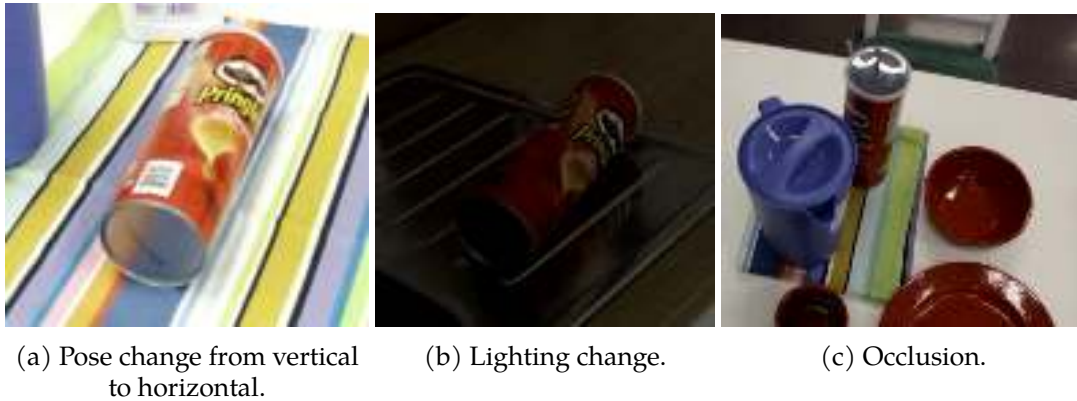


Figure 17: Actions we performed when collecting data for the Pringles dataset to increase accuracy.

When we had the videos, we extracted and labeled a total of 2969 frames from them using Yolo-mark. This program was introduced when we trained for the first time, in Section 2.5.1.

After obtaining great results with the first custom dataset trained in YOLO (Sec. 2.5.5), we wanted to create a more difficult dataset for a model to work with. This way we could test the limits of the object detection algorithm we were working with.

2.4.3 Second custom dataset (YCB apple extension)

The object we chose for the second custom dataset was a plastic apple. It lacks all the aspects the Pringles can had that made it an easy object to find: it is a non-geometrical monochrome object that shares the same red tone with other objects from the apartment. These facts can be seen in Figure 18.

For this model we made sure to add more pictures, as it would not be beneficial to create a harder dataset without allowing our model to achieve the same results than with the first custom dataset: a total of 2814 images were used for training and 704 for validation.

2.5 Training with YOLO

2.5.1 Training a single-object (banana) YCB model

We trained the first YOLO model with the dataset we described in Section 2.4.1.



Figure 18: Visual representation of the similarities between the color of our apple and another training prop, while also displaying the apple as a non-geometrical object.

After obtaining the images and the files with the bounding boxes of the classes that appeared in each image, it was time to prepare the data to train it under YOLO.

According to the indications given on the YOLO repository, before continuing any further we had to check the correctness of the annotations in each image. For that purpose we used a simple program developed by the author of the YOLO repository. As no bounding boxes were displayed in the program (Figure 19 (a)) we could conclude that the algorithm used to save those bounding boxes used a different standard than the one YOLO required.

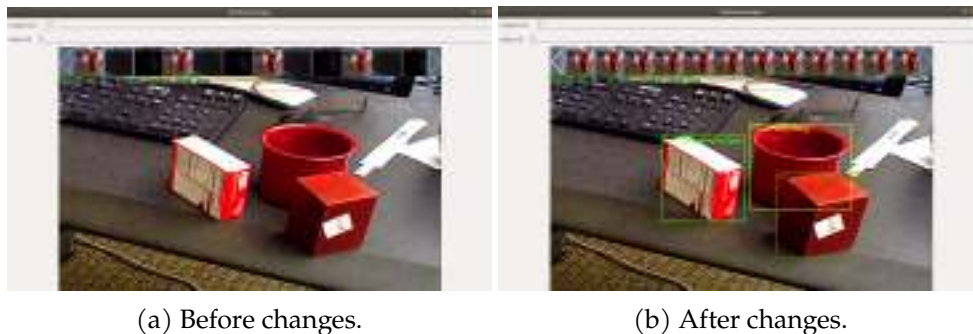


Figure 19: Usage of the Yolo-mark program to check labelling and bounding boxes.

When describing bounding boxes in image processing there are many different ways to proceed. As an example here are some that we have come across in this project (visually described in Figure 20):

- Writing clock-wise: consists on writing the x and y coordinates of the bounding boxes' vertex clock-wise (Figure 20 (b)). The format is: $x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4$.
- Describing from the center: consists on writing the coordinates of the center of the bound-



ing box with its width and height (Figure 20 (c)). The format is: x, y, w, h .

- Describe its diagonal: consists on saving the top-left and bottom-right coordinates (Figure 20 (d)). The format is: $x1,y1,x2,y2$.

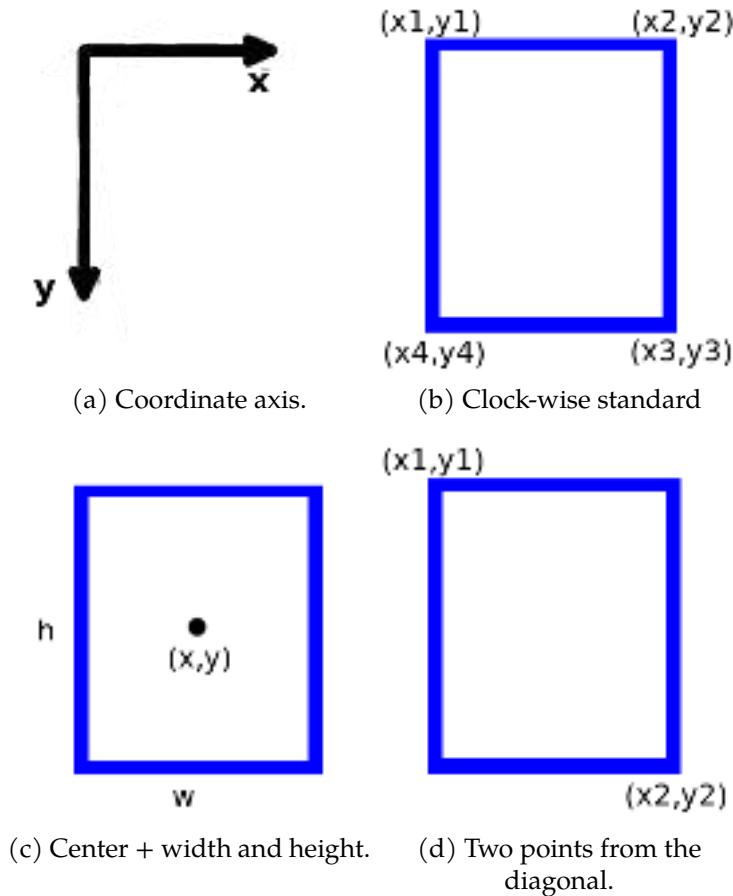


Figure 20: Visual representation of three different bounding box standards. The coordinates are always given according to the reference system seen in (a).

As it turned out, the way the bounding boxes were initially given to us was in the third way we described (Figure 20 (d)), by giving us the coordinates of the pixels of the top-left and bottom-right points of the bounding box. YOLO does not take the data in this format, it uses a variation of the second method (Figure 20 (c)): it requires the coordinates of the center of the bounding box and its width and height. The variation is that everything must be relative to the width and the height of the image (instead of using pixel values), thus the values introduced must range from 0 (not included) to 1.

Another change that had to be made was that classes had to be enumerated in each *.txt* file, where the first number represented the line of another file where the classes were listed (In our case there was only one object, so that number had to be 0). This is how the described file changed:

011_banana 147.74 192.05 321.79 314.73

↓
0 0.36682 0.52790 0.27195 0.25558

We made a script to apply these changes to all the *.txt* files, and started training afterwards. However, the training algorithm warned us that there were incorrectly formatted coordinates. We found out that the algorithm that was used to create the bounding boxes took into account the relative position between the objects. So when an object went out of frame, the coordinates still appeared in the *.txt* file, but with values greater than the width or height, or even negative. After the change all images displayed properly the bounding boxes (as seen in 20 (b)).

For training we also had to discard some images that were provided in the dataset. The discarded images were not the usual RGB images displaying what the camera saw (Figure 21 (a)). Those described object borders on the image (Figure 21 (b)) and segmented the image (Figure 21 (c)). Leaving these images out was not a necessity but a commodity, as having two duplicates of a picture that did not bring new information occupied much space, even more so if we have 133827 pictures.

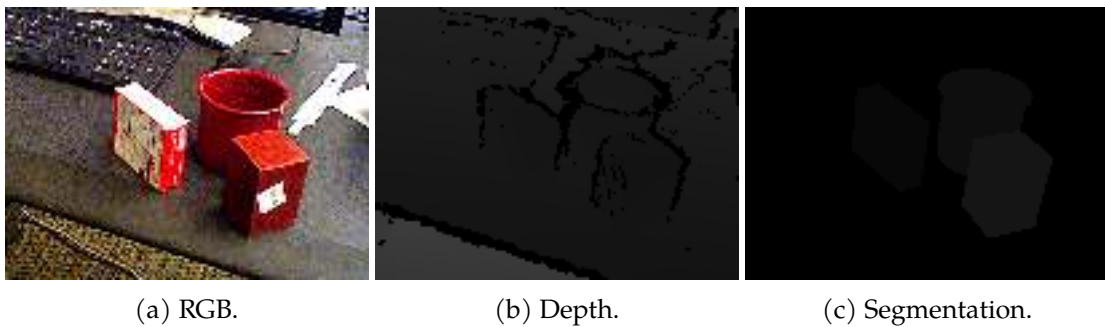


Figure 21: Type of images we were provided with.

Lastly, since we have prepared the entire dataset for training, it would be a good idea to make an initial test with a subset of elements. These elements have to be present in the laboratory's prop inventory. Because, as we explained in Sec. 2.4.1, despite some of the props belonging to the YCB set, the lab does not dispose of all of them. If we also add the fact that the downloaded dataset only has some of the elements, the objects that we can train and test are a handful. The depiction of this is seen in Figure 22.

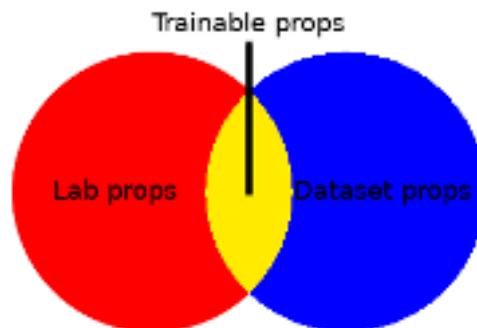


Figure 22: Depiction of the available material for training.



As a first element we chose a banana. Given its color and shape, it should not be a difficult element to find in the lab. Mainly due to the lab not having many yellow objects that can trick the model into thinking that there is a banana in places where there is none.

After selecting those images that contained a banana, a total of 28793 images were left for training. Although more images could be added to possibly improve precision, where the banana would not be present, the massive amount of images we already had made us discard this option.

As a reminder, in the YOLO's GitHub repository, a minimum of 2000 images per category were recommended. Furthermore, in the COCO dataset not all of the classes made it to 5000 instances. On the one hand, this means that having 28000+ instances of the 'banana' class will probably yield to good precision. On the other hand, those instances have been extracted from several videos, so a good part of these instances look like other instances. This means that although we may have lots of images, if they are not different enough between them, we will not achieve good results. Additionally, even if they are different between them, the conditions these pictures were taken in must be similar to the testing conditions in the lab for the model to detect the banana properly.

Before training, we had to make sure we included all the required files used by YOLO's training script:

1. **object.data:** File with the following information:
 - (a) Number of classes
 - (b) Absolute path to the .train file
 - (c) Absolute path to the .val file
 - (d) Absolute path to the .names file
 - (e) Absolute path to the weight backup folder.
2. **object.cfg:** File where the hyperparameters are set, like the input image's resolution, the number of iterations to perform, or number of classes.
3. **object.names:** File containing the names of the classes to be found in the images.
4. **object.train:** File containing the absolute address of all .png (or.jpg) files to use during training.
5. **object.val:** File containing the absolute address of all .png (or.jpg) files to use during validation.
6. **yolo.conv.137:** Weights that have been initialized to speed up convergence.

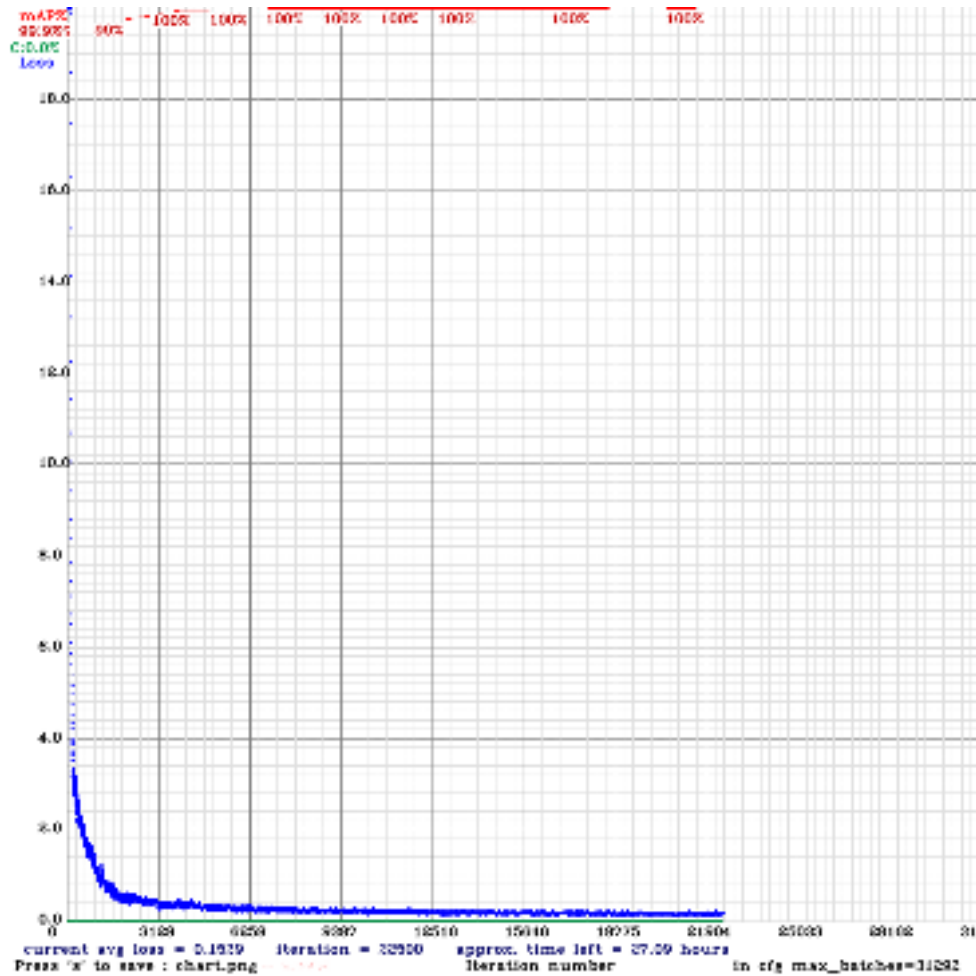


Figure 23: Loss graph (with mAP) for a YOLO model across different training iterations, where we can see that in a normal YOLO training process, the loss decreases incredibly fast at the beginning. Its loss decreases less as iterations pass.

2.5.2 Results for the single-object (banana) model

After letting the model train during approximately 80 hours, we obtained a graphic similar to the one shown in Figure 23⁹. In this graph we can observe that the loss function for the model decreases, meaning that we are achieving a bigger Intersect over Union (IoU) (explained in section 2.2.2), a greater precision, or both.

Since in our case precision is more important than IoU, by enabling the "-map" option during training we now have a red line in the graph that describes how the mAP progresses as we keep training. As it can be seen, the line reaches a mAP value of 100% really quickly. This means that at least with the validation data the model works perfectly.

We then did some testing around the lab to check that the banana could be found accurately,

⁹It was not exactly this one because due to power cuts, training was halted. When restarting the training from the last point, the system does not update the old graph, but rather creates a new one.



and obtained the detection seen in Figure 24. In it we can see that the banana can be found with a very high accuracy.



Figure 24: Image of the results obtained by the model trained with only a banana.

2.5.3 Training a multi-object model

An apartment usually holds more than one object in it, and it is our case too. So instead of having to select a different model (one trained with each object) each time we want to look for an object, it would be great to train a single model that could look for various objects simultaneously. It so happens that YOLO has been able to detect large quantities of objects ever since its second version (as we explained in Section 2.2.3).

The dataset had already been prepared for the first model we trained: the banana-detecting model. Thus, training more objects only required selecting more images, and making sure each object was correctly labelled. Training with more objects in itself did not bring anything to the project that had not been introduced before. So the reason why we still pursued this objective was because we wanted to test whether it was possible to reuse an already trained model, by adding the new objects to it.

After selecting all the images that contained the objects we wanted to look for, we started training the banana-detecting model with the new images. We obtained an output we did not expect: The loss graph did not converge.

In the first place, in Figure 25 we can see that the training process does not start at iteration 0, but rather continues from the iteration the banana-detecting model stopped training at. This means that at least the training starts where it should start. Secondly, it can easily be observed that loss does not seem to converge. The most likely reason behind this is due to the model being already prepared for one of the classes. In every iteration, the training algorithm modifies the weights of the neural network, in an attempt to minimize loss. But when a class has an average loss close to 0, any change in the neural network will cause the loss to increase.

Having one class' loss increase almost for sure, whilst not knowing if the other classes' loss will decrease or increase, leads to the results we see in Figure 25. We can conclude that in this case

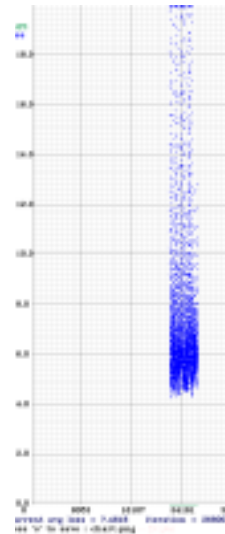


Figure 25: Loss graph of an already trained model, trained for additional classes. In it we can see the loss does not decrease, and that the model does not start training at iteration 0.

we have reached a local optimum.

A way to exit the local optimum in search for a global one (or at least a better local optimum) would be to tune the hyperparameters that define the progress the training algorithm has. Another way (the one we have taken) is to start training from scratch using the weights file that we introduced previously in Section 2.5.1 and that is provided in YOLO's repository: [yolo.conv.137](https://github.com/yololab/yolo.conv.137).

2.5.4 Results for the multi-object model

Training the multiple-object model from scratch led to the same results the banana-detecting object provided. A loss graph that looked like Figure 23, and a high accuracy for more than one object (Figure 26):



Figure 26: Detection obtained with a multi-object detecting YOLO model.

This model was trained with three different models. Additionally, unlike in the single-object



2.5.2 testing, this model was tested thoroughly, with the results of the test being stored in Appendix A. While doing those experiments we discarded the third object, as our prop was different than the training prop (check Section 3.1.3 for more information on this issue).

Thanks to thoroughly testing this model we noticed that the models we had trained until this point were only performing well under iconic conditions (as seen in Figure 7). When the objects were placed in weird positions, or even just by changes in the lighting of the room, precision dropped.

At this point we proceeded by extending the YCB dataset. Since the downloaded images did not contain all the objects the dataset has, we chose some trainable props (Figure 22) and created our custom dataset (Section 2.4.2).

2.5.5 Training and results for the Pringles custom dataset

For training, we made an 85-15 train-validation split. This means that we would be training the dataset with 85% of its images. Although in the original dataset an 80-20 split was made, for this custom dataset we wanted to preserve more images for training, as its size was smaller.

After training, we performed the same test we had ran on the last model (Figure 27). Initially it did not come as a surprise that the custom model performed better, as it was trained under our apartment's environment. What was surprising was that it outperformed the previous model in every configuration.

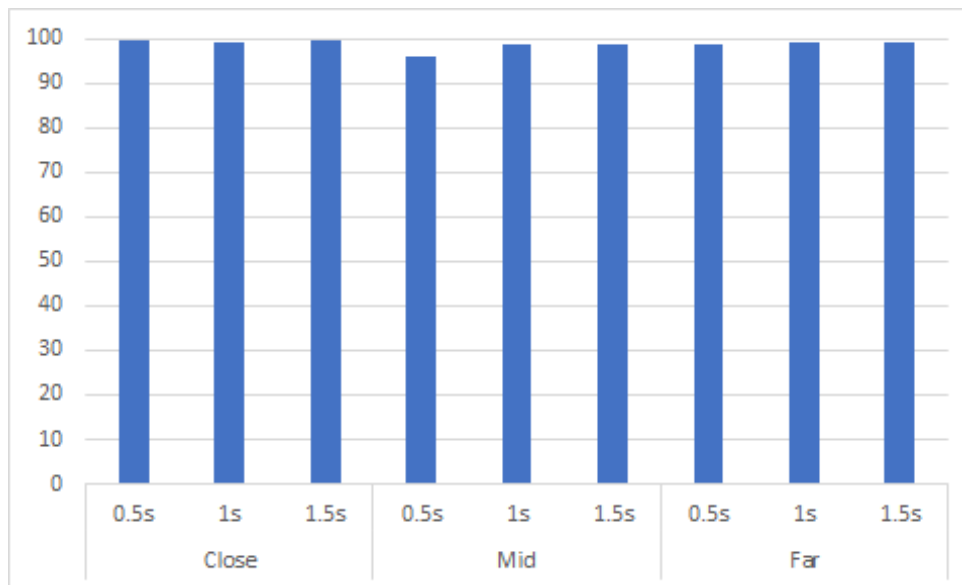


Figure 27: Precision obtained on the Pringles can using different configurations for range and camera zoom.

Like we stated previously in this section, the Pringles can should be an object easy to find. Seeing these results we decided we had to test the system again with a harder to find object 2.4.3.

2.5.6 Training and results for the apple custom dataset

Since we had already trained three models up to this point, there was nothing new to explain when it comes to testing.

It was quite surprising that an object far smaller than the Pringles can could still be almost perfectly detected in every configuration (Figure 28), just barely dropping a little accuracy.

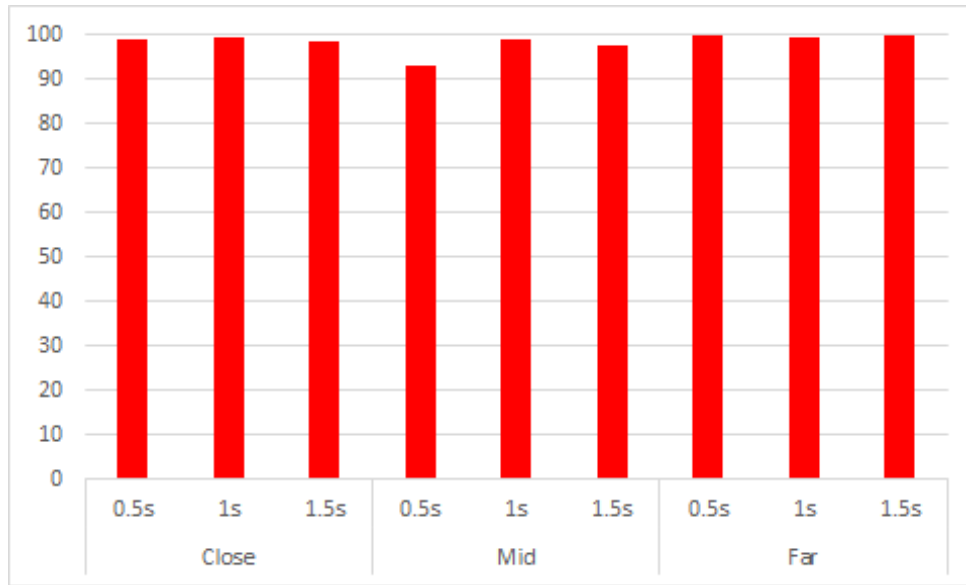


Figure 28: Precision obtained on the apple using different configurations for range and camera zoom.

Now that we had obtained a really good base with YOLO (both knowledge-wise and model-wise), we proceeded by testing if EfficientDet could perform as good as YOLO did with the first custom dataset.

2.6 Training using EfficientDet

Wanting to check if YOLO was the best candidate for an object detecting model, we had to compare it against another state-of-the-art model. In Section 2.2.3 we introduced the other candidate, which was the method with the highest mAP on the COCO dataset prior to the newest YOLO version release: EfficientDet.

Like we did with YOLO's training process (Section 2.5.1), before training we have to adapt the input to what the model can read. In this case the bounding boxes were formatted differently. As explained previously in section 2.5.1, there are different ways to work with bounding boxes. In this case it occurs that YOLO and EfficientDet do not use the same standard. To be precise, they do not even have the same way of storing the information at all.

Unlike YOLO, EfficientDet does not store the information for the bounding boxes in separate files (one for each picture). It stores all the information for the model inside a single .json file



(even the class names). We found a separate GitHub that made this conversion: ¹⁰.

After converting the dataset to something that could be processed, we started by making a test run with a lightweight model. We started with EfficientDet's D2 model. This model theoretically only occupies 1321 MB in the graphics card, but according to the repository's author, during training it takes 3x its size (issue # 673).

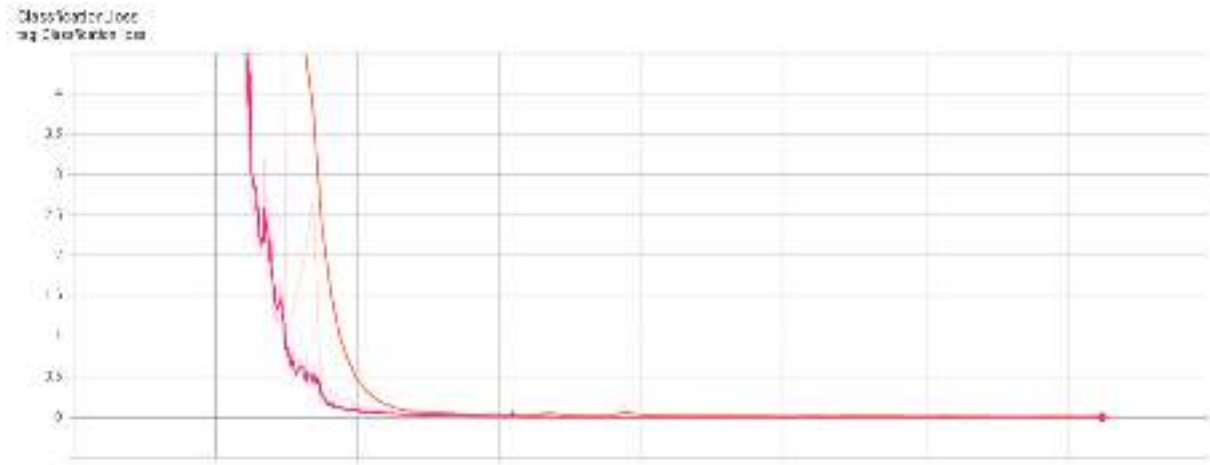


Figure 29: Class loss after 100 iterations on the custom Pringles dataset.

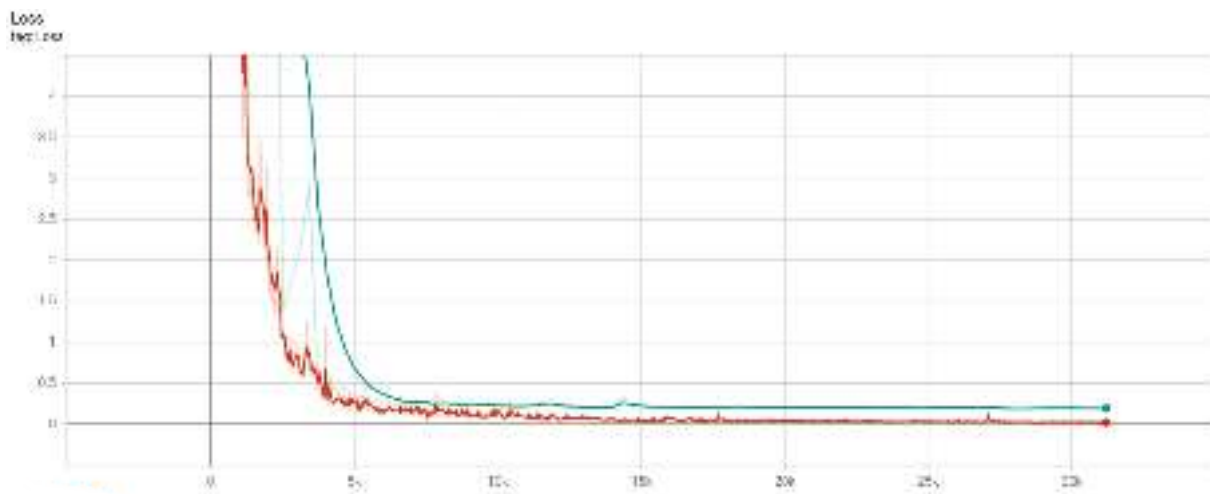


Figure 30: Overall loss after 100 iterations on the custom Pringles dataset.

We managed to train our model during 100 epochs in this dataset. Despite obtaining a decreasing loss that converged on 0.X (Figures 29 and 30), the model did not perform properly. Not even with the training images. In some images there were so many false positives that no image can be seen, only bounding boxes. On other images there are barely any bounding boxes, but they do not contain any objects. These two problems can be seen in Figure 31.

¹⁰<https://github.com/Taeyoung96/Yolo-to-COCO-format-converter>



Figure 31: Output obtained from testing our trained EfficientDet model on its training data.

After obtaining such a bad result, we discovered that despite having performed the conversion from YOLO to EfficientDet properly, we did not take into account that most CNN re-scale the input into a square. In Figure 31 a gray frame can be seen below the picture. The fact that the input was a 640x480 dataset deformed all images, and since nothing was done to correct the bounding boxes they now pointed at the wrong locations, hence why the bad result.

These corrections on the bounding boxes are called "anchors", and amidst the issues from the repository, there were users that had successfully used an alternate repository¹¹ to generate some for their datasets.

Despite almost finding a solution, the program was unable to find adequate anchors for our dataset. Although this meant that we could not longer train our dataset properly on EfficientDet, it had not been the only cause. During the search for a solution to the anchors problem, an attempt to train with a better EfficientDet model was made (to try to cover for the bad results obtained with the D2 model). Unluckily, all models threw an "Out of memory" exception, meaning that the graphics card did not have enough memory to handle the model.

Navigating through the "issues" section of the GitHub repository we found that many more people were having the same issue, with even better graphics cards than us.

2.7 Comparison of results and chosen algorithm

After the issues we had with EfficientDet we barely managed to train a D2 model, and it did not even provide good results due to the issue setting the anchors. In terms of precision this model is equivalent to the lowest YOLOv4 implementations, as seen in Figure 11. Thus, we decided that trying to tune the hyperparameters until we achieved a normal result was not an option. Hence the reason why we have chosen YOLO as our final model.

¹¹Repository to calculate the anchors of a dataset: <https://github.com/mnslarcher/kmeans-anchors-ratios>



3 Efficient Search

When an object is being looked for by someone, the places that are searched first are usually those that are the closest to the person, and they are places where the object is usually seen at. Afterwards, if the object is yet to be found, the person starts looking in places located further away. If the object cannot be found after all, it is common to ask others where they have seen the object recently, and check those places. Lastly, if none of the previous options have brought a successful outcome, the person usually proceeds to look everywhere.

This is the approach we have followed in this project, in which we focused on the three topics that make this method work efficiently:

- Time-based mapping: The location of an object depends on the time of the day it is. If a subject usually uses a mug during breakfast and asks the system to find it in early morning, the system will take this into account and will look for it in the dining room.
- Location-based mapping: The location of an object also depends on where it has been in historically. Applying this principle we will not have the system prioritize looking for an object in places where it usually is not (i. e. it is uncommon to look for a bike in a kitchen, the garage should be checked instead).
- Position-based priority: The order in which different locations are checked depends on the initial position of the searcher.

The process that follows the described searching pattern requires the use of a probabilistic map, as explained in [9]. Although the cited paper only focuses on location-based mapping, by adding the current timestamp to each newly recorded location we covered the time dependency. Furthermore, we were able to position the searcher (the zenithal camera) within the map, making it easier to find what spots are closer and thus faster to reach.

In this project the searcher is a zenithal camera, the one we introduced in 1.3. We could have chosen the mobile manipulator, but it is faster to have the camera check every position rather than having the mobile manipulator do the same job. Even if after using the camera we must still move the robot in order to interact with the object.

3.1 Zenithal camera's properties

The zenithal camera used in this project is an Amcrest IP3M-941W-UK (Fig. 32). It was introduced in the lab during a study made by a researcher (Technical Report: [2]), whose work includes a program that controls the movement of the camera. We communicate with this camera via wifi to exchange information and send commands, as it is connected to the apartment's local network.

3.1.1 Camera movement

The user can interact with the zenithal camera by connecting any device to the same network the camera is connected to (or gaining access to it via VPN).



Figure 32: Our laboratory's Zenithal camera with its corresponding pitch and yaw axis.

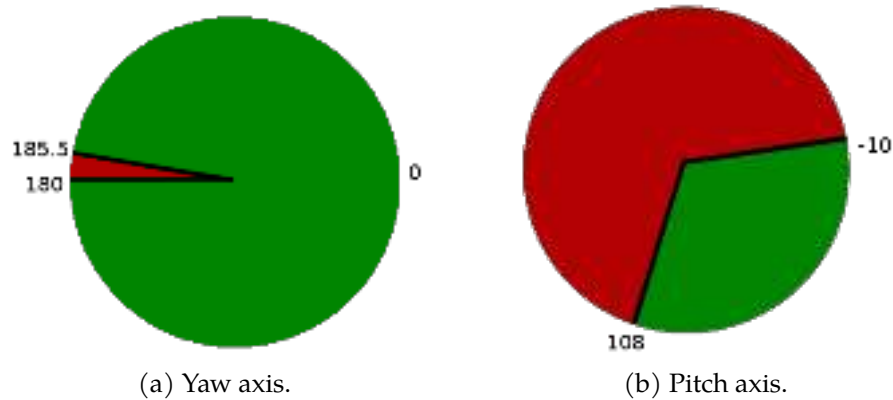


Figure 33: Camera movement around its axis.

(Green corresponds to the available positions for the camera. Red marks the unavailable ones).

The first thing that needs to be noted from this camera is that it cannot spin around its yaw or pitch axis 360° . As portrayed in Figure 33 (a), there is an inaccessible zone between 180° and $185,5^\circ$ around the yaw axis. Although the code that controls the movement of the camera (written by [2]) already has a solution for this, we have had to account for it when applying the position-based priority.

When it comes to the movement around the pitch axis seen in Figure 33 (b), we can see that it cannot rotate 180° . This has not been a problem as we have been able to make a combined pitch-yaw movement, enabling the camera to view every part of the apartment.

Amcrest provides an HTTP API ¹² that allows the users to use all available functionalities in the camera. Despite this, the software used by the camera is not Open-Source ¹³. Thus, the HTTP API was implemented into a Python library, to allow easier access to the camera's functionalities.

There was one important functionality which could not be used (neither using the HTTP API or the Python library), which was the direct movement of the camera from its current configuration to a given one. The only allowed movements on the camera were time-based movement around the yaw and pitch axis. Although initially we thought it was due to a firmware update not being applied, it turned out that the camera itself lacked that functionality.

It was with such limited actions that [2] made a manual positioning algorithm based on error minimization, starting with the movement around the yaw axis and following up with the pitch axis. The algorithm followed this procedure:

1. Rotate around the desired axis during a calculated time at a calculated speed.
2. Check the current position and compare it to the objective.

¹²An API is a set of instructions one can follow to interact with an object. In this case the interaction is provided via HTTP protocol.

¹³Open Source software allows total control by the users, which can access the code. Freely being able to use and modify it.



3. If the difference is lower than a previously-defined threshold, stop. Otherwise go back to step one.

There are two main reasons as to why this process was far from ideal:

- **Network overload:** Since the algorithm needs to send a "start" and "stop" command, when there is a lot of traffic on the laboratory's network, the camera never reaches its destination.
- **Too slow:** Even in times where the algorithm works, it is by far slower than asking the camera to move to a preset location (2x-3x slower).

The camera has an internal storage where it can store up to 25 presets. Since using these is a much faster option than waiting for the movement process established by [2], we were going to investigate whether it was feasible to use them.

The first thing we needed to do if we wanted to use those presets to its maximum potential, consisted in checking whether we could describe the entire laboratory with them.



Figure 34: Map of the lab generated using all available presets without zooming in.

As we can see in Figure 34, we were able to describe the entire apartment with 25 presets, and we still had some presets to spare. There is a catch to this, though: the availability of presets was directly related to how much we zoomed the camera in/out. The higher the augmentations, a lower amount of presets would be available.

To sum up, we had to find out the best zoom configuration: the one that maximized accuracy. It would only be then that we would know which of the two following approaches we could take:

- **Everything in a preset:** This is the ideal case, where we have enough presets for the entire lab and the object to search. We can search for the object in the lab if it is not found in its usual places.
- **Only the object in a preset:** In this case we will only be able to look for the object in its usual places. We cannot look for the object in the rest of the lab afterwards.

Before finding the best zoom configuration, we had to check how the images we obtained from

the camera looked. This way we would be able to correct any issues before running the CNN on them, maximizing accuracy.

3.1.2 Camera Image

When working with a camera, it is common for it to have some factory quirks. As an example, in order to achieve 90° coverage around the yaw axis, our camera was manufactured with a convex crystal lens. The shape of the lens distorts the image the camera perceives.

This distortion cannot only be found horizontally, but also vertically. In Figure 35 we could see that the lines representing how horizontal and vertical lines should look (red lines) were not even close to how these lines really looked in the image (green lines).

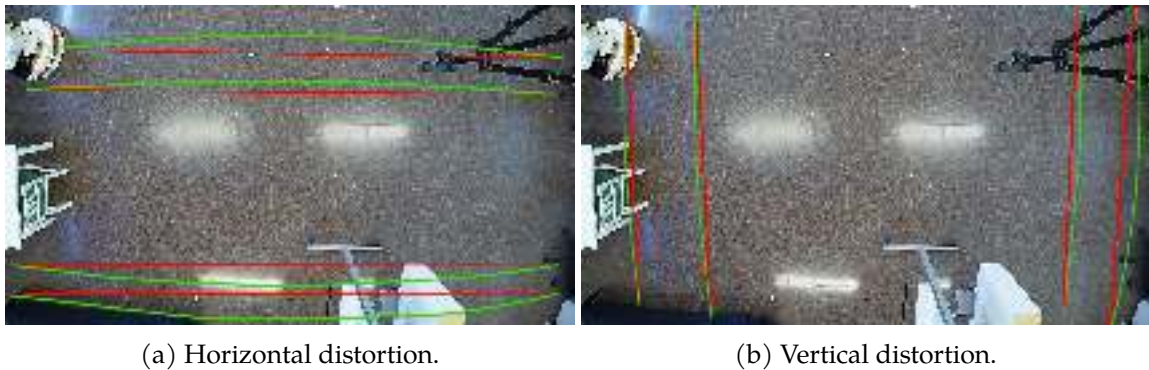


Figure 35: Horizontal and vertical distortion due to the camera's lens with no zoom.
(Green - distorted, Red - undistorted)

Since the images we used to train the model were not distorted, we required that the image taken by the camera was also not distorted. Only then would we achieve a higher accuracy.

Thanks to OpenCV, the process of getting rid of the distortion was really easy. There is a pre-built method that uses chessboards to undistort the image.

This method goes as follows:

1. The program is fed lots of pictures containing a chessboard, and its dimensions (the chessboard's) are given.
2. The program finds the chessboard in the image, and recognizes if all cells are even or if they are distorted in any way.
3. The program applies a displacement in the pixels of the image in order to obtain a non-deformed chessboard, and saves the applied displacements in a matrix.
4. After running this process in each image, the program now has the complete displacement matrix, that describes how each pixel must be moved to obtain its correct representation.

After following the procedure, the matrix we obtained allowed us to undistort any image taken with the camera. In Figure 36 we can see how in the left we have the original image obtained



from the camera, and after running it through the displacement matrix, we obtain the image in the right. This image does not have any distortion at all, it only has black areas where pixels have been moved from. If needed, the image can be cropped to remove the black areas. We have chosen not to crop it as other parts of the picture would also be removed, which leads to a loss of information.

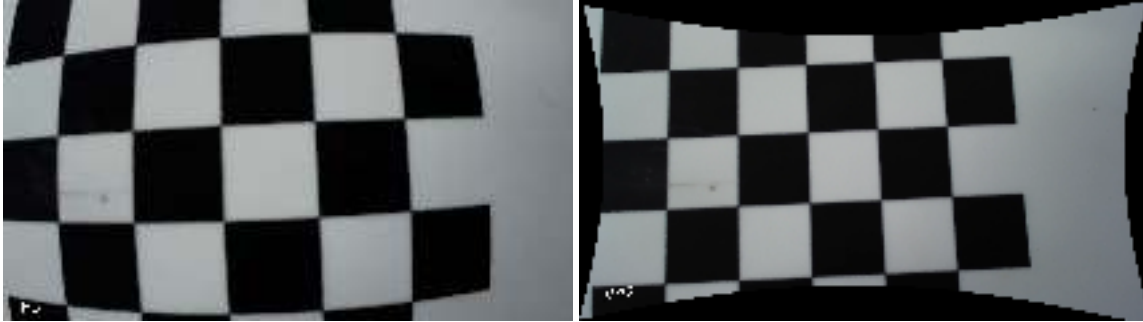


Figure 36: Correction applied to the images from the camera to remove distortion.

Now that we had found a way to ignore the distortion applied by the lens, we had to solve the issue presented in section 3.1.1. We had to find the best zoom configuration for the camera.

3.1.3 Camera zoom configuration

The first issue we came across after removing the distortion was that the camera still had a hard time finding the trained objects in the images, as they appeared too small. We then decided we needed to tinker around with another parameter: the zoom.

Although this camera's zoom is digital, the resolution of the images that the camera takes is always the same. As a consequence, the object detection algorithm's performance may differ under different zoom configurations. Thus, in this section we will be focusing on choosing the zoom configuration that works better with our camera.

Just like what happened with the rotation around the pitch and yaw axis, this camera cannot be set to a specific zoom via code. It can only be asked to start/stop zooming in/out. To solve this we have discretized the range of motion into three different configurations, with an interval of 0.5s between each. With this we have the zoom configurations of 0.5, 1 and 1.5 seconds. The reason why the 0 second (full zoom-out) configuration was discarded was because it could almost never detect objects properly, and in those cases where it could, it did so with a low precision.

The way the best configuration was chosen was by setting different objects in different places (Figure 37) and writing the precision the algorithm had at finding them for each zoom configuration. The average precision that was obtained from the experiments (recorded in Appendix A) are shown in Figure 38. It has to be noted that since the camera updates its footage continuously, every time we asked the camera to send us a picture of what it saw it sent us a different image. This is why despite even not moving the camera we have obtained different results in each run.

Although we have obtained relatively good results for the banana and the tomato soup can, the precision with which the neural network detects the *MasterChef* can is far too low to be



Figure 37: Locations in which we set the objects to find the best zoom configuration.
Green = Close / Blue = Mid / Red = Far

acceptable.

After the initial shock of the system not detecting the *MasterChef* can despite having trained for it, we found a somewhat reasonable explanation for this fact. We made the mistake of selecting a prop to train that corresponded to a newer version of the one provided on the dataset we used for training.

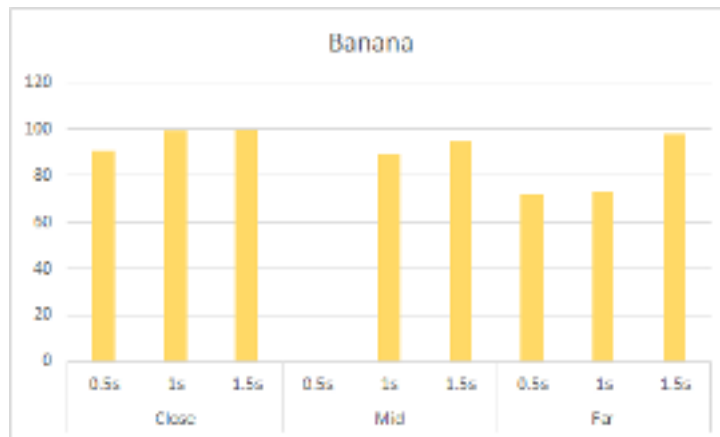
In Figure 39 we observe that despite both of them having a cylindrical shape and being blue, they have some differences. To begin with, their caps are not the same colour. Secondly, their shapes are not exactly similar, as the one the model was trained with has some slight curves whereas the test one is plain. Lastly, the text written in each can is entirely different, and they cover different portions of the label.

In order to test this hypothesis, we made another test with the three objects, where we made the camera perceive our test *MasterChef* can as similarly as the training one.

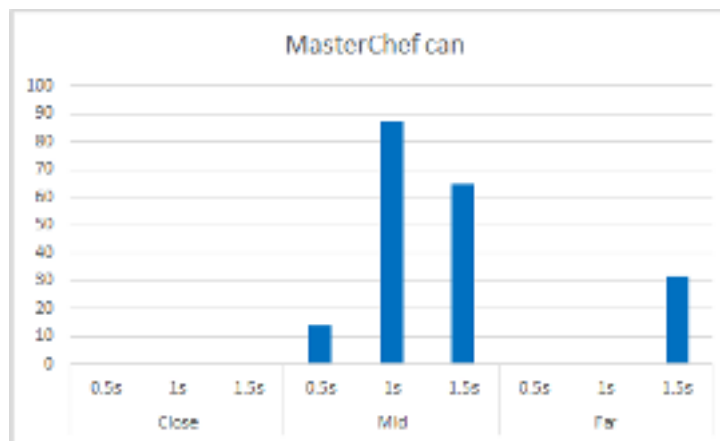
Firstly, we took care of the difference in the cap by putting the prop upside down. Making it reflect the light from the ceiling, acquiring a clearer top. Secondly, we took care of the text in the label by rotating the can, making the text look the other way. Lastly, since we could not do anything about the difference in blue tone or shape of the can, we added a variable to the test: distance.

The results of this tests can be seen in Figure 40. We can see that the precision with which the

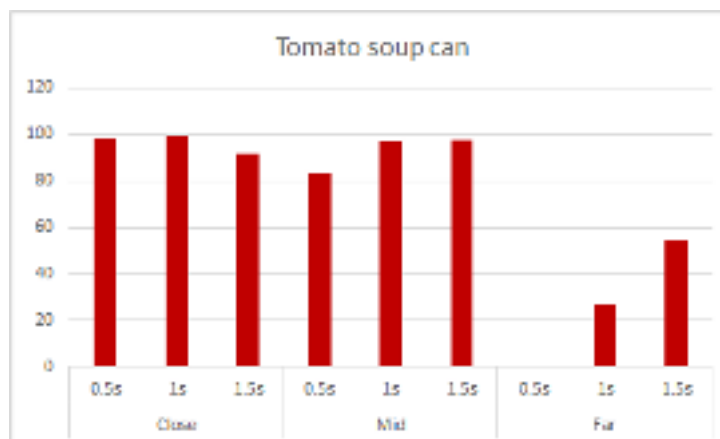




(a) Banana.



(b) MasterChef can.



(c) Tomato soup can.

Figure 38: Average precision obtained for different objects on different zoom configurations.



Figure 39: MasterChef props used during the training and testing phase.

model perceives the *MasterChef* can drops as we increase the zoom. This would be due to the fact that the camera cannot see the difference in shape until it has zoomed in enough. With all images we can see that the precision goes from 82.96, to 46.61, to 0 as we zoom in.

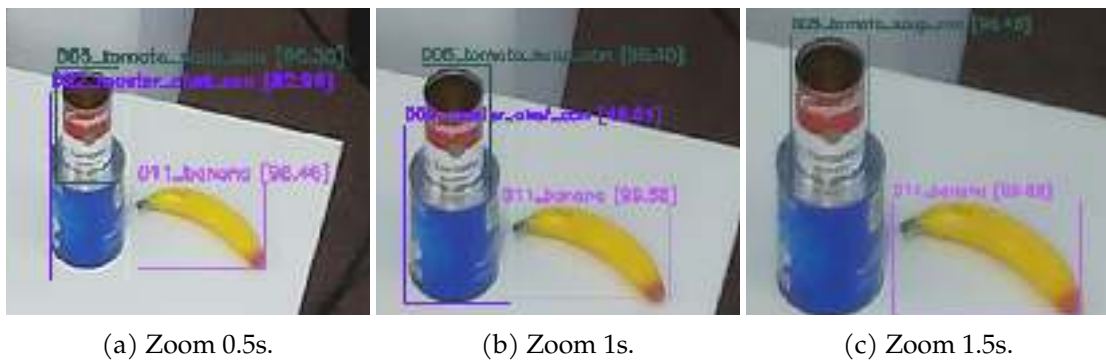


Figure 40: Cropped results obtained in each zoom configuration in a middle-ranged position.

After obtaining this result, we could have probably repeated the tests performed in Appendix A. Instead, we chose to discard the prop, as any small change in the environment around the prop (change of position or lighting), may have unexpected results on the output of the neural network. Thus we chose to stick to the *banana* and the *Tomato* can as our reference models.

From the results in Figure 39 we observe that setting the zoom at 1.5s maximizes precision in any location. It also helps detect objects located in the most remote positions from the zenithal camera.

After finding out that the best option turned out to be 1.5s, we must check if a correction is still needed due to the lens distortion. After checking the images perceived from the camera seen in Figure 41, we can conclude that there is no need to apply a calibration. In said Figure red lines (true straight lines) barely differ from green lines (distorted lines due to the camera).

After finding out the best zoom configuration for our camera, it was time to check how many



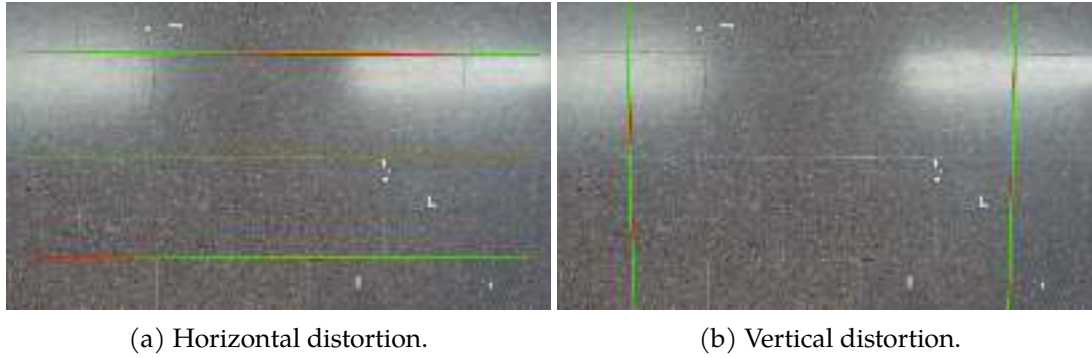


Figure 41: Lens distortion with the camera set at 1.5s of zoom.

presets we could describe the apartment in.



Figure 42: Map of the lab generated by using all available presets zooming in 1.5s.

After taking many pictures around the laboratory, we used up all of our presets just to be able to look at the whole image of it.

Since the camera did not have enough presets for both an object and the lab, we decided we would only be setting up the object in the presets, like we stated in Section 3.1.1.

Although now we will not be able to perform the brute-force and random search we talked about in Section 3, it now makes more sense to build a probabilistic map. If we had elected a zoom configuration where we saw most of the apartment, finding the best order in which to visit the different places would be meaningless, as just a picture or two would suffice.

3.2 Probabilistic map

In order to build a probabilistic map that portrays previous locations where something (objects in our case) has been seen, we first need the occurrences. Collecting those occurrences can be done in two ways: realistically or via accelerated simulation.

- **Realistic collection:** In this case the collection of data is performed by saving the location of the real desired objects periodically under a real time-stamp basis. Therefore, the process can take, days to weeks.... as long as required.
- **Accelerated simulation:** As the name indicates, in this case the collection of data is accelerated by generating lots of occurrences in a short amount of time. Afterwards, data are

counted as occurrences obtained in a long period of time.

In our setup, the realistic collection of data is unfeasible. The laboratory simulates a real apartment, but we do not have 24 hours users that make use of it. Besides that, it would take too much time. Therefore, we opted for acquiring the data using accelerated simulation. Although it did not provide real results, it helped to check that the the system works properly. Once the results under the simulation are as expected, the realistic collection of data can be applied. At this point, if any new issue arises, we would be much more confident that the problem would be related with the uncertainties of the acquired real data.

Before starting the simulation (hence the collection of the data) we needed to make sure we had everything prepared.

3.2.1 Preparations before the collection of data

In order to check that we are gathering the data correctly, it needs to be plotted in a map. Thus, the first step we did consisted in selecting a ROS message that could relay the following information in a map: the position of each recorded occurrence, and the amount of occurrences that have been recorded in each position.

The message we selected for this purpose was *nav_msgs/OccupancyGrid*. One of its components is *int8[] data*, which is an array of integers in the range $[0,100]$. In our case, we will use each integer as the probability for an object to be exactly in the coordinates the integer corresponds to.

We started by creating a simple script, where we had a cube (described using the *visualization_msgs/Marker* message) spinning around in circles. The terrain under it, which used the message *nav_msgs/OccupancyGrid*, stored the value of the coordinates of the cube periodically, increasing the integer corresponding to that position. This made those points become darker with time. We can see the transformation in Figure 43.

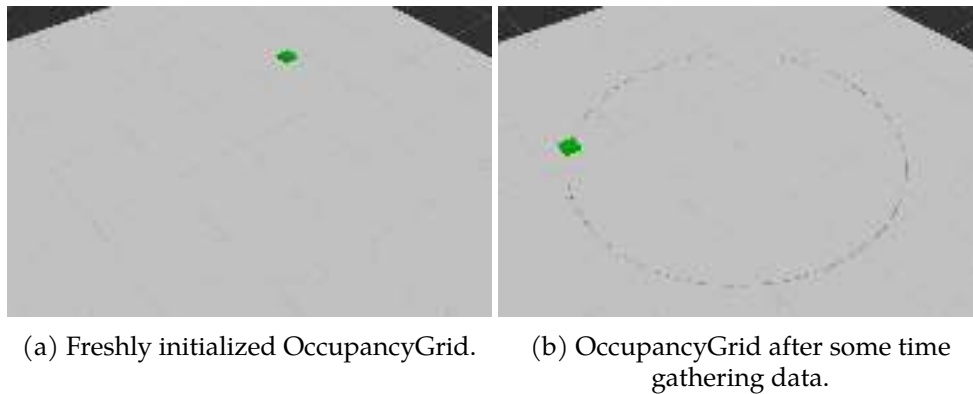


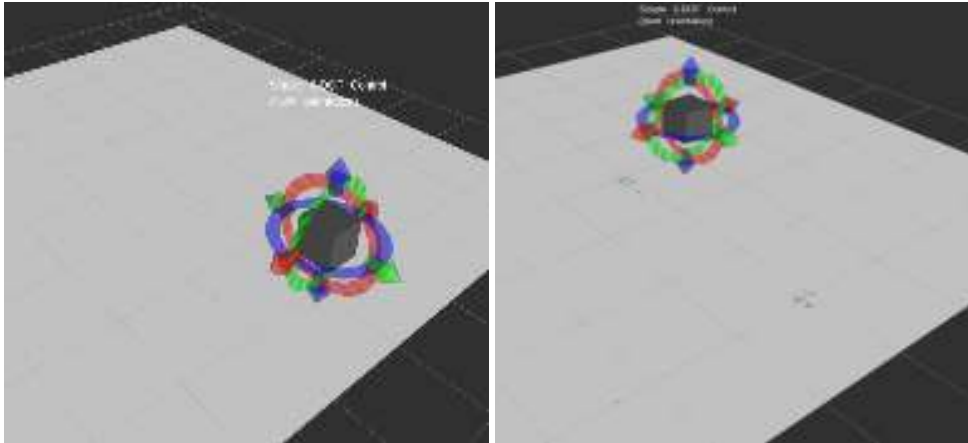
Figure 43: Change experienced by an occupation map that starts with no data, and finished execution having recorded the movement of a Marker moving in a circular pattern.

After proving that we are able to capture the occupancy of an object making continuous movements, it was necessary to prove that the system held strong with erratic movements too. Mainly because erratic movements are closer to real movements than continuous ones.



The clusters formed from this data will also be more realistic, and will display how a real object may occupy the apartment in a realistic manner.

Thanks to the examples provided in the rviz tutorial webpage (wiki.ros.org/rviz/Tutorials), we were able to make a cube with such properties easily. In Figure 44 we can see how we can now create realistic occupation maps, which we can now run through a clustering algorithm.



(a) Freshly initialized OccupancyGrid.

(b) OccupancyGrid after some time gathering data.

Figure 44: Degrees of freedom the newly implemented Marker has. We also display that we are now able to move the Marker freely to record data in the locations we want.

3.2.2 Clustering algorithm

The clustering algorithm we selected was the OPTICS method from the Python's scikit-learn library, because it works well with uneven cluster sizes and densities. There is only one parameter required to run this method: the minimum cluster membership. This parameter can be used to generate the clusters that are formed by a percentage of occurrences.

The first time we ran some data through the clustering algorithm, we also checked the *save* and *load* functionalities of the map. Since the real map data will have to be updated every time a new occurrence is detected, we have tested this by saving the data from the circular movement, and loading it before applying manual movement. This has led to what we see in Figure 45, where the gray dots represent those that do not belong to any cluster, and the coloured ones form three different clusters:

- *Blue*: [0.9081762 -1.781803]
- *Red*: [0.05315908 3.04101533]
- *Green*: [0.0776345 -1.60381879]

The only reason why a third cluster is created is due to the fact that the circular movement of the square was stopped before the recording of its position was halted. Despite this, it still takes the last position, because the algorithm sorts the clusters by the number of occurrences.

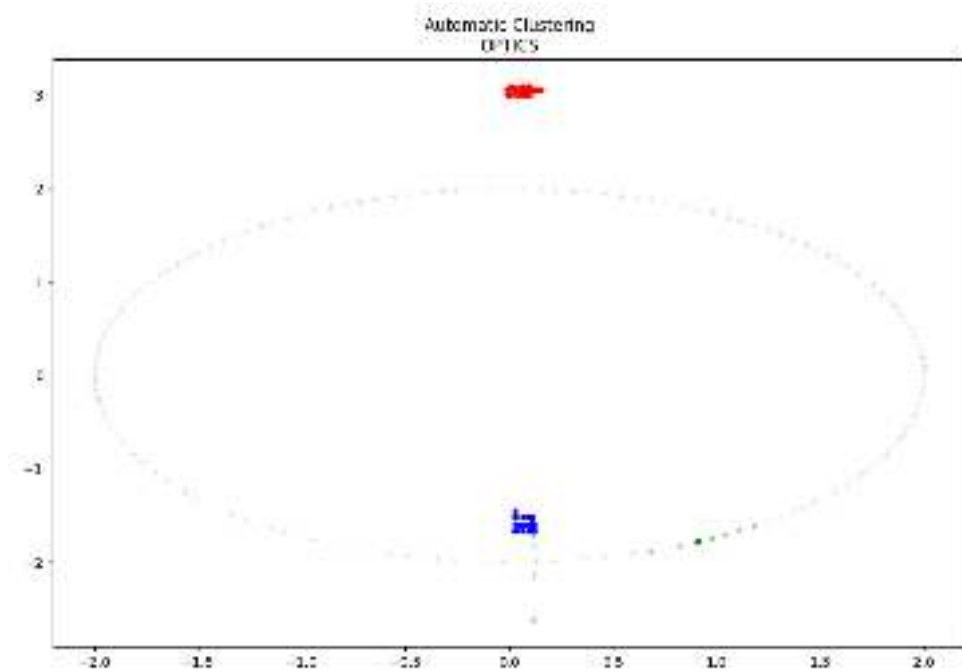


Figure 45: Clusters obtained from a Marker moved both manually and in a circular manner.

Having confirmed that the clustering algorithm worked correctly, we proceeded with the following step: the capture of the object's coordinates using the HTC Vive.

3.2.3 Capturing data using HTC Vive

In Section 1.3.2 we explained that we were able to obtain information from the HTC tracker thanks to a *tf2_msgs/TFMessage*. This message consists of an array of *geometry_msgs/TransformStamped* messages. This message has an attribute called *child_frame_id*. This attribute allows us to search for the object we are tracking. As an example, in Figure 46 we could choose to track "controller_2" and/or "tracker_1", and extract their coordinates from the published message.

After setting up the system in order to capture "tracker_1", we ran an accelerated simulation, capturing the tracker's position once every 0.1s. This would be the equivalent to having the system storing the object's position once per second during 10 days.

With the simulation we stored the results seen in Figure 47. In order to obtain three clusters we had to manually change the minimum cluster membership, because most of the recorded locations (>90%) are in the same location. This fact initially led us to having just one cluster. For this Figure we have set the parameter so that it creates a cluster if 5% of the points are close in the Euclidean space.

The clusters in this case each corresponds to the following coordinates:

- Blue: [-3.78361695 -2.80463991]
- Red: [-3.99296416 1.14484564]



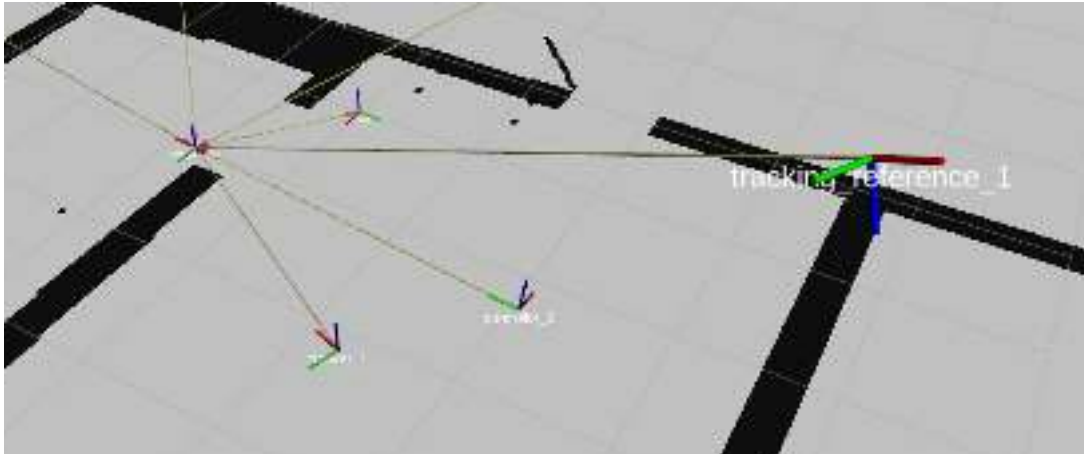


Figure 46: Portrayal of the HTC Vive equipment working on the laboratory's ROS environment.

- *Green:* $[-1.25167972 -2.76846008]$

Now that we finally had the possible locations for the object to be in, the only thing left to do was to move the camera so that it pointed towards each cluster.

3.2.4 Preparing the camera to look for an object

In order to calculate the Euler angles the camera should have when pointing at each cluster, we started by setting the camera's location at $[-1,1,4]$ (in the lab's reference). The only issue we encountered when doing the calculations was that the probabilistic map was to be set at some height. Otherwise calculations could not be performed. We started by setting the height of the probabilistic map at 1m, which was inside the actual range of values for the height we detected when we used the HTC: $[0.8-1.4]$.

Although initially we were invested in calculating the x, y and z coordinates of the object, after looking through TIAGo's documentation we discarded the idea. Moving the robot is not as simple as: sending the coordinates makes the robot get as close as possible to that location and stops while looking towards it. The way Pal Robotics (TIAGo's manufacturer) implemented this method is completely different: sending the coordinates makes the robot get as close as possible to that location, but once an obstacle is encountered, the robot starts spinning looking for alternate routes. When it finally decides to stop, it turns to face along the x -axis direction.

The new method we implemented consisted on segmenting the laboratory, giving the tag "sink" to the area surrounding the sink, "bookshelf" to the area around the bookshelf, etc. Afterwards, we established a location and orientation (also known as a Point of Interest, POI for short) that allowed the TIAGo to see the whole area corresponding to a tag. As an example, in Figure 48 we can see the TIAGo on the "sink"'s POI, standing in a place and facing towards a direction where it can see the sink.

As a consequence, now we only had to find the correct Euler angles to set them as a preset within the camera. Achieving this only required a single step:

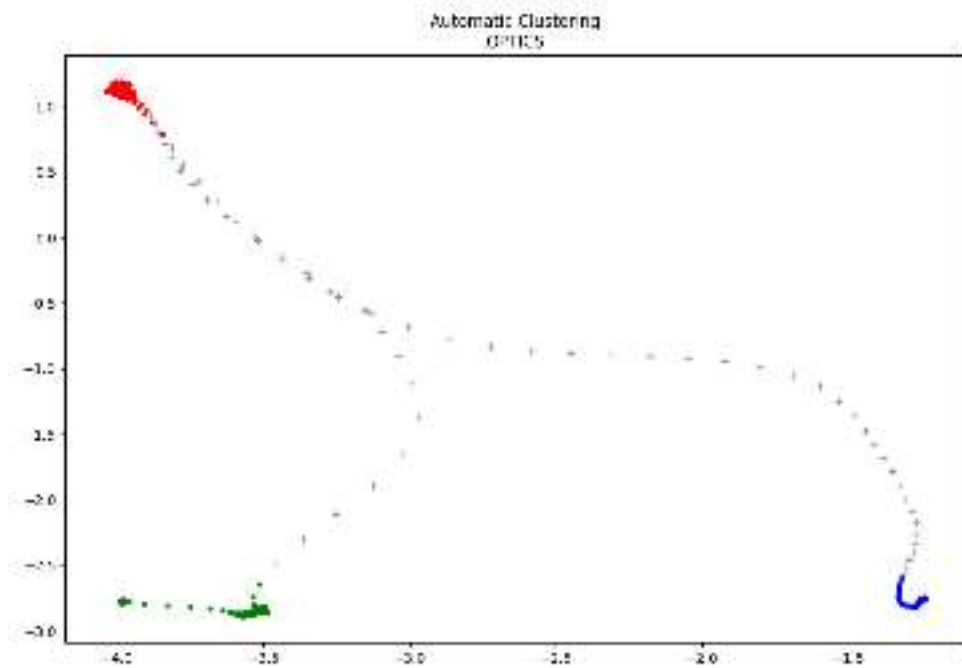


Figure 47: Clusters obtained from the data recorded using the HTC Vive.

Calculating the object's possible location:

This first step consisted on getting the camera to look at a cluster. The Pitch (horizontal angle) and Yaw (vertical angle) axis are calculated as follows:

$$\begin{aligned} X_c &= \text{x component of the camera;} & Y_c &= \text{y component of the camera} \\ X_p &= \text{x component of the cluster;} & Y_p &= \text{y component of the cluster} \end{aligned}$$

$$dX = X_c - X_p \quad dY = Y_c - Y_p$$

$$Z_c = \text{z component of the camera;} \quad Z_p = \text{z component of the cluster}$$

Horizontal angle:

```

if dX>0 and dY>0:
    phi=atan(dx/dy)
elif dx>0 and dy<0:
    phi=180-atan(-dx/dy)
elif dx<0 and dy>0:
    phi=360-atan(-dx/dy)
else:
    phi=180+atan(dx/dy)

```

Vertical angle:

```

d=sqrt((Xc-Xp)**2+(Yc-Yp)**2+(Zc-Zp)**2)
theta=asin((Zc-Zp)/d)

```



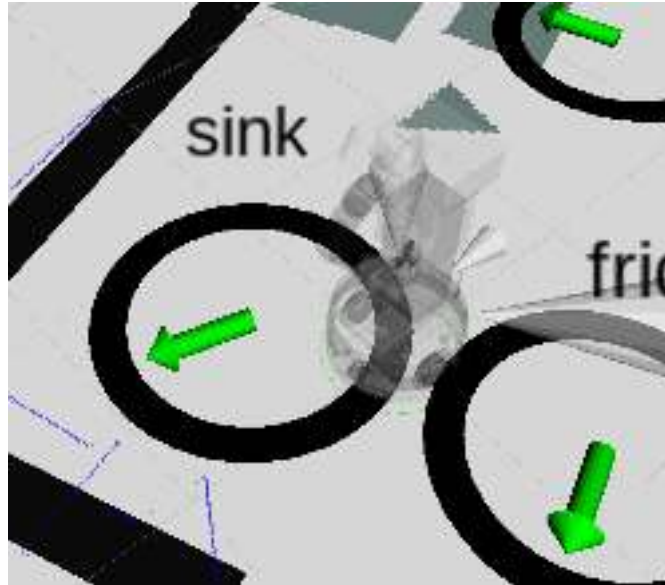


Figure 48: TIAGo manipulator robot on the POI that corresponds to the sink.

After calculating the Euler angles that corresponded to the configurations of the camera that pointed towards each cluster, we zoom in during 1.5s, as we calculated in 3.1.3. Then, using the algorithm developed in [2], we move the camera to every computed configuration prompting the camera to save the current configuration as a preset.

Every time we store the configuration as a preset we also store in a .txt file the preset number we just set with the angles it represents and the tag of the region the camera is pointing towards.

3.2.5 Efficient sorting of the presets

In the last section we set the camera's presets so that each of them pointed at a location where we may find the object.

Those locations do not share the probability we have at finding the object in them. In some locations we will have a higher amount of data recordings than in others. Those with a higher amount of data recordings will yield a higher probability of housing the object.

As obvious as it may sound, this makes visiting locations with higher probability of housing the object before others an appealing idea. Nevertheless, if the camera is initially pointing towards a location where there are many low probability clusters, it may also be a good idea to visit the low-probability ones first, minimizing unnecessary camera movements.

After coming up with these two variables to minimize, we had to check which one was better than the other. For this purpose, we created a C++ executable that evaluated the probabilities each location had, and took into account the distance that separated them.

For this executable we established three possible models to evaluate:

- **Sorting by %:** This model checks each preset in a decreasing probability fashion.

- **Current configuration's best:** This model takes into account the current position of the camera and moves towards the one that presents the highest probability \times proximity value.
- **Brute force:** This model takes into account the current position of the camera and recursively checks every possible path the camera can take. It then selects the one that presents the highest score. The value each path has consists of the sum of scores from each time the camera must move to a new configuration.

Four tests were prepared for the evaluation of those models:

- **T1: Last set preset:** We tested this option because in the way we have implemented the configuration of the camera's presets, the camera remains in the last position set. It also turns out to be the one with the least probability of housing the object.
- **T2: Highest probability preset:** After testing the first option, we wanted to check if moving the camera to the preset with the highest probability made a difference.
- **T3: Random location:** This test was made in order to test if implementing an algorithm¹⁴ that left the camera looking in a random position affected performance.
- **T4: Random location with delay:** Every time we move the camera from one location to another, there is a delay between the moment when the camera stops moving and when the image stops moving. We implemented a wait every time the camera stopped moving, and checked if there were noticeable different results. This test highly penalizes methods that stop on the most places without finding the object.

We set the object in each preset and ran 10 iterations per model per test. We then calculated the weighted average time it took each model to find the object:

$$model_i = \sum P_j * T_j$$

Where P_j is the probability a preset j has at holding the object, and T_j is the average time of the 10 runs it did in that preset.

In the tests we conducted, the probability the object had at being in every preset were the following:

1	2	3
0.57028489	0.23620627	0.19350884

After running the tests, which can be found in Appendixes D and C, we obtained these results:

From the Table 2 we can observe that there has been no improvement after the implementation of any of the complex models that we programmed. Even though we see a very slight improvement

¹⁴the algorithm could be something from human-tracking to a scan around the apartment to record the location of every detectable object.



Weighted average time				
Method	T1 (s)	T2(s)	T3(s)	T4(s)
0	7.1232	7.4752	6.7325	8.3401
1	7.0686	7.6516	6.9656	8.5542
2	7.4111	7.4446	7.5666	9.3366

Table 2: Experiment ran to test speed of algorithm

in Test 2 with method 2 over 0, it only represents a speedup of 1.004. The same happens in Test 1, where the speedup gained by using method 1 over 0 is 1.007.

4 Integration of our project on the laboratory's system

Up until this point we had managed to create a system that efficiently looked for an object within the laboratory's apartment. The next step in our project consisted on sending a TIAGo mobile manipulator to the location where the object was found at. It has to be noted that sending a robot to interact with an object (like bringing medicine to an ill person) are functionalities evaluated by the HEART-MET competition, so it is a good idea to set a basis from which develop this functionality further. However, we could not do that without connecting our system to the laboratory's system.

The laboratory's system was a network of devices. In this network when one device wished to communicate with another, it published a message onto the laboratory's network. In our case, such system has been implemented using the Robot Operating System (ROS), introduced in Section 1.3.1.

4.1 Node structure of our project

Although up to this point we have been setting a clear line between object detection and efficient search, the design is not modular enough. If per chance a change has to be made to a code pertaining to the object detection module, the developer has to make sure that the change does not affect the functioning of all the other code contained in the object detection module.

Overall, modular projects are easier to implement, as each module implements a single functionality. This also makes them easier to understand and work on. Thus, we analyzed our project thoroughly, and separated it into the nodes we see in Figure 49.

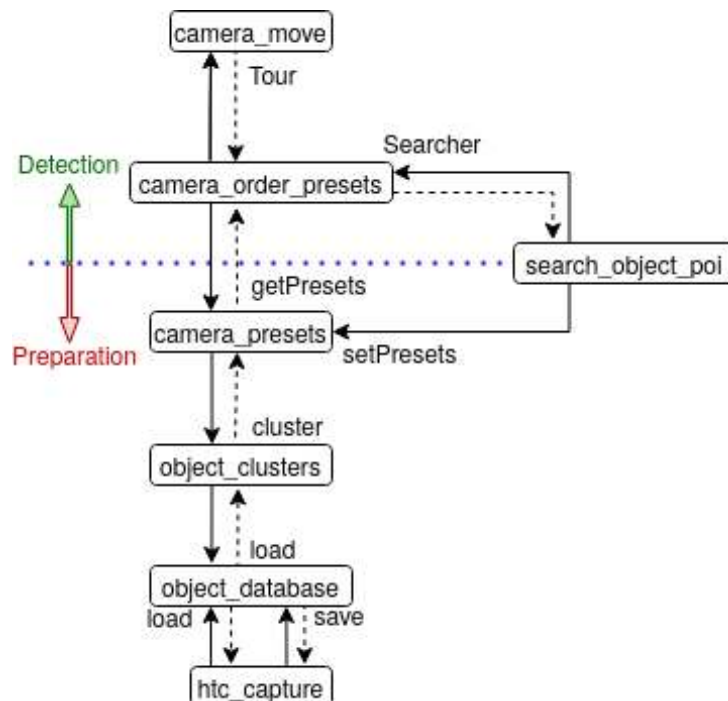


Figure 49: ROS nodes with the services they use for communication.

In said Figure, it can be observed that the project has been split. We did this because of the



conclusions we reached in Section 3.1.1. This has classified the nodes into those that prepare the camera to look for an object, and those that look for the object once the camera has been prepared.

The functionality each of those nodes represents in the system is the following:

- **htc_capture:** Records the position of an HTC tracker or controller.
- **object_database:** Serves as a storage for all previously saved detections.
- **object_clusters:** Generates clusters from the data stored in the database, which represent where the object usually is.
- **camera_presets:** Prepares the camera to look at the usual locations of a given object.
- **search_object_poi:** Asks the system to prepare an object for searching, or searches it. It can also send a TIAGo manipulator robot to a location.
- **camera_order_presets:** Given the camera's configuration, it computes the order in which locations have to be visited.
- **camera_move:** Moves the camera to a desired location and runs the object detecting algorithm to check for the presence of a desired object.

Following the Perception and Manipulation lab's directives, all of the code for the nodes stated above has been uploaded onto the laboratory's GitLab environment:

https://gitlab.iri.upc.edu/perception/lab/iot_apartment/alexa_openhab

4.2 Integration of every ROS node

4.2.1 htc_capture:

Although the inner workings of this node have already been explained in section 3.2.3, back then there were several parameters that had to be changed in the code for the system to work as expected. Lucky for us, the ROS framework has a package that exactly does that: `rqt's dynamic_reconfigure`.

Rqt is a development tool for ROS that allows its users to perform many different tasks using its numerous plugins. In our case the plugin we required was called `dynamic_reconfigure`. As the name implies, this plugin allows its users to reconfigure the functioning of a node in runtime by changing the value of certain specified variables.

Using this tool, we saved the user from having to look through the code in order to change a certain parameter. Furthermore, selecting the variables that the user can modify and isolating them makes the program safer to run. Simply because the only font of error can come from the modification of those variables.

The variables the user can change can be seen in the GUI we programmed (in Figure 50). It can be seen there that some additional functionalities (apart from "load" or "save") were added to allow the user an even better experience. Here is a description on what the purpose behind each parameter is:



Figure 50: GUI that was implemented to run the htc_capture ROS node.

- **recording:** boolean that can easily toggle recording on/off.
- **dataFile:** name of the object we want to operate with.
- **dataPeriod:** establishes the frequency with which we want to save the data. Represents the amount of data recordings per second we want.
- **tracking:** here we must introduce the name of the tracker we want to store the data from.
- **Load:** makes a request to the database for the data corresponding to the object with the name specified in the "dataFile" variable.
- **Save:** sends the information recorded locally to the database, and requests that it be saved.

4.2.2 object_database:

As we just saw in the htc_capture node, whenever we want to load or save data, we interact with a database. This is a functionality we implemented to reduce the responsibility the htc_capture node had. Moreover, it is not conceptually correct to ask for an object's data to a program whose function is not explicitly to store it.

The functioning of the node is standard to say the least, we have two functions: loading and saving data. We have implemented these two services using ROS' service methodology:

- **Loading data:** Requested input parameters from the service:
 - *string* stating the name of the object.

Output obtained from the service:

- *string* where all the data has been concatenated into.



- **Saving data:** Requested input parameters from the service:
 - *string* stating the name of the object from which we are saving the data.
 - *string* where all the data has been concatenated into.

Output obtained from the service:

- *string* message that displays if the operation has been completed successfully.

Although as of now the string returned from the "saving" service does not have an application, this node has been implemented like this in case in the future the database is stored online. Because in that case for every request there will be a message stating whether the operation has been successful.

4.2.3 object_clusters:

Now that it was possible to consult an object's data, we were able to implement a ROS service that could return its clusters.

The clustering algorithm we used has already been described in Section 3.2.2. Despite this, when implementing we made some tweaks to it. In order to explain the additional procedures we implemented, we must explain the input required from this service:

- **Cluster request:** Requested input parameters from the service:
 - *string* stating the name of the object we want the clusters from.
 - *int64* stating the time in which we center the interval search for clusters.
 - *int64* states how wide the range we want to look at is.
 - *int64* (positive) marks how many seconds pass between each log of data we take into account.

Output obtained from the service:

- *string* where a list with all the object's clusters have been concatenated into.

The additions we made to the clustering algorithm consisted on adding a time variant to it. Like we stated in Section 3, it is common to check where an object is depending on the time of the day. So the clustering algorithm firstly makes the clusters for all data, and then selects the data within a certain margin around the time of the request and reruns the clustering algorithm with that data only.

Now we had two lists of clusters, the one that is not time-dependent (as clusters are obtained from data of the entire day), and the one that is. It has to be noted that the OPTICS algorithm returns them in descendant order of importance: clusters with more instances of data come first. Consequentially, we have two lists of locations sorted by their likelihood of having the object placed in them. These two lists were fused into one in the following procedure:

```
timdep_list=list of time-dependent clusters
```

```

ntimdep_list = list of non time-dependent clusters
sel_clusters= empty list

For each i in timdep_list:
    For each j in ntimdep_list:
        if i is within a certain range to j:
            cluster i is fused onto j

    if i has not been fused:
        insert i in sel_clusters

final_list = join(sel_clusters, ntimdep_list)

```

When we fuse a cluster onto another we keep the location of the non time-dependent cluster and add the number of instances they both had. After obtaining the list of all clusters with their number of instances, we compute the probability the object has at being at each location:

$$Probability_i = \frac{\#instances_i}{\sum \#instances_i}$$

4.2.4 camera_presets:

This node implements two different services: one for setting up the camera's presets, and another to check what object the current presets are set for. These are the expected input/outputs from each of these two services:

- **Camera setup:** Requested input parameters from the service:
 - *string* stating the name of the object we want the clusters from.
 - *int64* stating the exact time of the request.
 - *int64* stating the time in which we center the interval search for clusters.
 - *int64* states how wide the range we want to look at is.

This service does not return anything. If there are errors they are displayed on the console.

- **Get camera's current setup:**

Requested input parameters from the service:

- *string* stating the name of the object we want to check if the camera is ready for.

Output obtained from the service:

- *boolean* displaying whether the camera is prepared to look for the given object.
- *string* where a list of probability-preset-location objects have been appended into.
- *float64* displaying the current yaw angle from the camera.



- *float64* displaying the current pitch angle from the camera.

As setting up the camera's presets requires using the cluster service explained in the last section, additional parameters are required when calling this service.

After the list of clusters is obtained, each cluster is run through the operations we described in Section 3.2.4. Moreover, the cluster is also run through a discretization process, where the x-y coordinates are substituted by the tag of the place they are contained in.

Afterwards, using the manual positioning of the camera (from [2]) and zoom (using Amcrest's Python library), the presets are set.

Additionally, the configuration information is stored in a .txt file inside the node, so that they can be consulted when the camera's configuration is requested.

The configuration information consists of two things: the first one being the name of the object that has been prepared for searching, and the second one a list of probability-preset-location objects. A sample of the content of such file can be observed below:

```
c_Pringles
57.0284890011 3.8200322089 27.9005755286 1 bookshelf
23.206274793 92.8151063913 33.7107427056 2 sofa
19.3508835197 33.2080387945 23.491598644 3 sink
```

When a service request is performed to ask for the camera's current configuration, the first thing that is checked is whether the name of the object stored in the .txt configuration files correspond to the one provided by the service. If this yields true, the node makes a query to the camera to know its current pitch and yaw angles. It then returns all the information we have specified to the node that requested the service.

4.2.5 camera_move:

This node iterates through the presets set by the camera_presets node, and sorted by the camera_order_presets node. When the camera is located in each preset, it takes a picture and runs an object detecting algorithm. If the object is detected in that location, the algorithm stops and returns the name of the location the object has been seen in.

This service has the following properties:

- **Cluster request:** Requested input parameters from the service:
 - *string* stating the name of the object we want to look for in all images.
 - *string* string-formatted list of the presets the camera must visit and the location they represent.

Output obtained from the service:

- *string* displaying the location where the object has been found. Empty string otherwise.

This service mostly follows the object detection methodology we used when performing tests in Sections 27 and 2.5.6. However, in that section we did not take into account the noise that appeared around the image, we only accounted for the precision on the bounding box that surrounded our object.

Now that we needed the system to identify whether the requested object was within all the detections, we had to set a threshold. We chose the number basing ourselves off of the experiments performed with both the downloaded YCB dataset and the custom ones, and established it at 80% precision.

4.2.6 camera_order_presets:

This node implements a service that checks if the camera has been prepared to look for the specified object. In case it has, it computes the order in which locations must be visited, and prompts another service to perform the search.

- **Search request:** Requested input parameters from the service:
 - *string* stating the name of the object we want to look for.
 - *int8* representing what method wants to be followed when computing the order in which presets must be visited..

Output obtained from the service:

- *string* displaying the location where the object has been found. "-1" if the object has not been set. Empty string otherwise.

Although in Section 3.2.5 we developed a C++ recursive algorithm that found the order in which we should look at the presets, we have been developing the entire project using Python. Even though ROS does not have an issue communicating scripts written in C++ and Python, this node uses the "getPresets" service from the camera_presets node. In this service we are returned a list codified in a string using a Python built-in function.

If we used C++ we would have had to create a function to decode the string and transform it into a list before operating with it. So we decided to create a Python layer that would pre-process the information before running the C++ algorithm with it.

4.2.7 search_object_poi:

This node has been left in charge of interacting with both the part that prepares the system to search for an object, and the part that searches for it.

Since no other node has to interact with this one, like we saw in Figure 1.3.1, this node does not have its own custom services.

Despite that, since this node links the two sections of our ROS implementation, it needs to be input with all the parameters that are required in the other nodes. We can see the requested inputs in Figure 51.

As we can see in the Figure, most of the variables that can be modified by the user have been



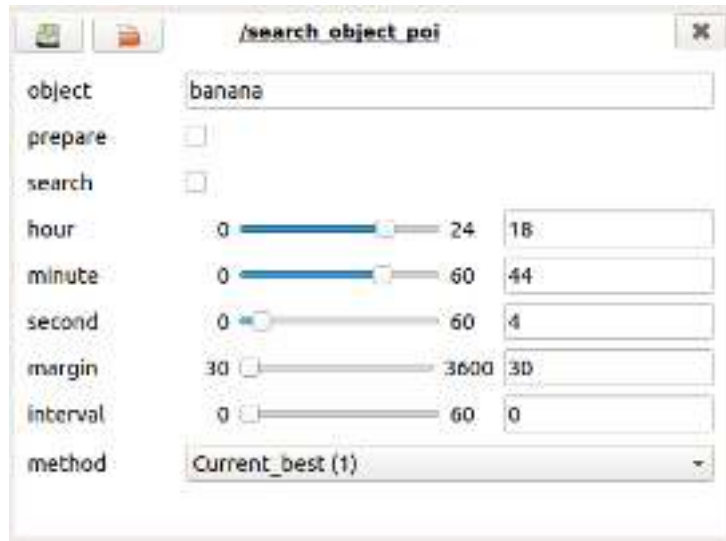


Figure 51: GUI used to interact with the search_object_poi node in order to either prepare the system to search for an object or search for one.

mentioned in the last sections (when talking about the services that interconnect nodes). The complete list of variables the user can interact with is as follows:

- **object:** name of the object we want to search or prepare the system to search.
- **prepare:** boolean that acts as a button. When pressed it will only stop being pressed after the attempt at preparing the system for the object has been completed.
- **search:** boolean that acts as a button. When pressed it will only stop being pressed after the attempt at finding the object has been completed.
- **hour:** positive integer that represents the hour in which we want to center the search for the object.
- **minute:** positive integer that represents the minute in which we want to center the search for the object.
- **second:** positive integer that represents the second in which we want to center the search for the object.
- **margin:** positive integer that helps establish the amount of data we want to process for the time-dependant clustering algorithm. We take all the data that is within "Center \pm margin" seconds.
- **interval:** positive integer that marks how many seconds pass between each log of data we take into account when calling the clustering algorithm.
- **method:** integer with three different values (0-2):
 - 0: Iterates only accounting for the probability of having the object each location has.

- **1:** Iterates accounting for both the probability and the angles the camera has to rotate to move to the location.
- **2:** Uses a brute-force algorithm to find the best configuration taking into account both the angles the camera must rotate and the probability for the object to be in each location.

The only fact that differentiates this node from all the rest is its dependency to the PAL Robotics action: `/poi_navigation_server/go_to_poi`. Without this dependency this node would not be able to sent the TIAGo to a POI, we would only be able to find the POI in which the object is found.



5 Future work

While developing this project we have come across different ideas that could improve the project in the future. These improvements can both be changes in the current project, or different projects altogether that could complement this project.

- As we have displayed in the latest section 4.2, in this project we have developed a Graphical User Interface capable of receiving the requests the users may bring forth. An interesting functionality that could be developed would consist in developing a voice recognition model that could recognize commands.

The laboratory already disposes of an Amazon Echo device, so if a model like this can be implemented it would not only be compatible with our project, but with many more aspects of the lab.

- A possible change in this project could consist on developing an algorithm to compute the goal position and orientation for the TIAGo. This algorithm, given the coordinates of the object that has to be interacted with, has to compute the nearest position the TIAGo can stand on. Moreover it also has to calculate the right orientation, so that when it reaches the computed coordinates, it must look in the direction of the object.

This change would eliminate the need to discretize the laboratory, allowing us not to depend on Points of Interest (as seen in Section 3.2.4).

- Another improvement to this project could consist on creating a method for the TIAGo to calculate its approximate position in the lab by itself.

Every time in this project that we have used the TIAGo, before we could send it to a Point of Interest (POI), we had to either move it around a bit or set its position in the lab manually. Not until we had done that were we able to do anything. Understandably enough, having somebody calibrate the TIAGo every time it does not know its whereabouts is not ideal in an assistive environment.

- Now that the TIAGo goes to a location where it can see the object, a project that could complement the one we just finished would consist on applying another CV algorithm that verified the presence of the object. Additionally, if the object is truly in front of the TIAGo, a second step could consist on grasping the object and taking it to wherever it is ordered to.
- A well needed improvement to the laboratory consists on changing the camera to one that can either store more presets or move freely to some given angles. It would also be a good idea to purchase a camera with a lower delay when sending the video feedback. As of right now one has to wait for a few seconds everytime the camera is moved.

This way the system would be able to search quicker for the object even if it is not in one of its usual places.

- Another possible improvement is to implement a human tracking algorithm, be it with the zenithal camera or, if not possible, with a set of cameras distributed around the household. This could help users keep track of their activities during the day, and then notify a ther-

apist or a relative if certain restrictions are not being upheld.

- One last improvement for this project has already been presented, in Section 3.2.5. In said section we introduced the idea of monitoring the different objects. This improvement would not be difficult to make and it would allow the system to generate a real dataset, which adapts to whatever changes there may be in the room along the years. The ability robots have at gathering data is one of the HEART-MET evaluated functionalities. Thus, if our laboratory is ever planning on assisting the competition, it may be interesting to apply this improvement.



6 Project's cost

In this section we calculate how many resources have been used to bring this project to life. We have divided them in three categories: time spent by the team, machines that have been used, and resources required to make the machines work.

6.1 Personnel

According to the Polytechnic University of Catalonia (UPC), a final thesis has a workload that corresponds to 12 ECTS credits. Since each ECTS credit corresponds to 30 hours of work, a normal thesis should correspond to 360 hours of work.

Despite that, this thesis has not been developed in the 4 months that are accounted for in those 360 hours. An extension was requested for this project, making it last two more months. Thus, those 360h have received an extension as well.

Since those 360 hours are initially distributed among 4 months (mid-February to mid-June), that leaves 90 hours per month. Hence why we will be considering that our project took $6 \times 90 = 540$ hours of work to be finished.

During those months, the student received weekly advice from his tutor, averaging at 1 hour per meeting. Although that would account for 27 hours, the tutor has also put extra hours. Some of them preparing material for the student (like setting up the HTC Vive in the lab), and other hours were spent in counseling and reviewing his work. Although those extra hours cannot be precisely accounted for, an approximate of at least 10 extra hours were spent by the conjoined effort of the tutor and co-director (who can also be considered a tutor).

In order to put an economical value to those hours, an official state document (in our case *Boletín Oficial del Estado*) was consulted [20]. According to that document, a recently graduated engineer should be paid a minimum of 11.40 €/hour. This value was obtained by dividing the annual income of a graduate student (20,424.25 €) by the maximum working hours of a year (1,792).

On the other hand, the tutors, who have a higher status, would be paid 26,323.57 € annually. This represents 14.69€/hour.

Personnel	hours spent	€/hour	Price (€)
Student	540	11.40	6,156.00
Tutors	37	14.69	543.53
Personnel cost:			6,699.53

Table 3: Monetary value given to the time spent by the student and tutor.

6.2 Tools and objects

The tools and objects we have taken into account in this section are all of those present in the lab that have been required for the functioning or development of the project. All the purchase prices have been included in Table 4

- Reflex camera: Used for the creation of the custom datasets.

- Amcrest camera: Used for detecting objects.
- YCB props: Used for the creation of new datasets and testing.
- TIAGo: Mobile manipulator we move to the object's location.
- Computer setup: Environment we have used to train our models and develop the project.
- Router: Device used to communicate with the elements in the laboratory's network.

Object	Price (€)
Reflex camera	479.99
Amcrest camera	63.42
YCB props	1,185.94
TIAGo	60,500
Computer setup	2,151
Router	125.25
Tools and objects cost:	63519.66

Table 4: Price for all the objects we used in this project

6.3 Electricity

The exact amount of Euros spent on electricity on this project cannot be calculated, as we have used many different electronic devices in a non-continued manner. The only electrical component that has been monitored has been the computer's graphics card, thus it will be the sole accountant for the electrical cost of the project.

We have trained a total of 4 models in this project, and each has taken different times to complete. The training hours per model have been described in Table 5.

Model	Time to train (h)
Banana	80
Banana+ Tomato can + MasterChef can	130
Pringles	4
Apple	5

Table 5: Time it has taken to train each YOLO model.

Knowing that the graphics card is at 100% power usage during training, and that it consumes 180W, we can know the kWh we have used for each model.

Since we trained the models in different dates, we have consulted the average price per kWh for each of those periods. Taking into account that the price kept increasing, we cannot average them out. Even more so when the periods that took the longest to train were trained in the cheapest episode. We have calculated the price for each model in Table 6.



Model	Total used energy(kWh)	kWh Price(€/kWh)	Resulting price(€)
Banana	14.4	0.12847	1.849968
Multi-object	23.4	0.13120	3.070080
Pringles	0.72	0.18081	0.1301802
Apple	0.9	0.18978	0.17080
Training cost:			5.22

Table 6: Table computing the monetary cost

6.4 Total cost

Even though all costs pale in comparison to the one due to physical objects, they must be taken into account. In Table 7 we display the total cost of our project.

Subject	Cost (€)
Personnel	6,699.53
Tools and objects	91,575.9
Electricity	5.22
Total cost:	98,280.65

Table 7: Total monetary cost of the project

7 Climate Impact

Calculating the carbon footprint each and every component of our project has is not an attainable objective. Mainly because we do not dispose of such precise information about every tool or device we have used in this project.

Setting the TIAGo as an example, the carbon footprint it has created upon manufacturing is not published in the manufacturer's website. It then becomes impossible to know the process each component has been through before arriving into the TIAGo.

Consequentially, we have only taken into account the carbon footprint created by the generation of the electricity that powered our graphics card during training. Although it may not be representative of the impact the whole project has had, it establishes a minimum.

From the data we recorded in Section 6.3, we know that we have used a total of 39.42 kWh. We can then proceed with the conversion into CO_2 kg:

$$39.42 \text{ kWh} \cdot 7.09 \times 10^{-1} \frac{\text{kg of } CO_2}{\text{kWh}} = 27.95 \text{ kg of } CO_2$$

The conversion we have used to calculate kg of CO_2 from kWh has been provided by the United States Environmental Protection Agency.

With this calculation we can assert that our project has emitted at the very least 27.95 kg of CO_2 .



Conclusions

We can proudly announce that in this project we have achieved the initial objectives that we set. We have been able to create a system capable of identifying household objects using state of the art Deep Learning algorithms. Moreover, we have enabled our system to look for those objects taking into account their past locations, making the process faster.

We have given the users of our system the freedom to add more objects of their choice in the database, so that they may expand the system's ability to interact with the apartment's residents.

Even though we had to extend the project's duration, it was thanks to that extension that we ended up implementing parts of the system that were initially considered inside the *Future work* section.

This project we have created and implemented onto the laboratory will now also be usable by many other types of projects. As an example, projects that require obtaining data from the HTC Vive equipment set on the laboratory may use the `htc_capture` node we have implemented in this project to do so. Or other projects that want to cluster any type of data with a time dependency.

With the whole system functional, we have made a first step towards helping people search and interact with an object of their choice in our assistive apartment. Despite currently not being able to interact with the object yet, the system can already be used to find it. This can be useful for people that have memory issues (dementia, first phases of Alzheimer's disease, etc.) and have a hard time remembering where they last left a certain object.

Acknowledgments

Before anybody else, I would like to express my most sincere gratitude to Sergi Foix. He has shown a great deal of patience when working with me and has helped me in any way I needed. Without his help this project could not have been a success.

I would also like to give my thanks to the entire personnel at the laboratory. They have helped me with all problems I have come across during my stay there, even if they were as simple as some firmware issues with the camera. Furthermore, they set up many resources for me to use, like the HTC Vive we used to collect data.

Also many thanks to Cecilio Angulo for accepting us in this project without doubting that we would be able to make it.

Lastly, I would like to thank my family, who have been supporting me in any way possible from the beginning to the end.

Thanks for everything.



References

- [1] World Health Organization. "Life expectancy and healthy life expectancy". In: *Global Health Estimates Technical Paper WHO/DDI/DNA/GHE/2020.1* (2020).
- [2] Kévin Bedin, Sergi Foix, and Guillem Alenyà. "Robots and IoT devices for assistive automation". In: *IRI Technical Report* (Aug. 2019).
- [3] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV].
- [4] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *Lecture Notes in Computer Science* (2016), pp. 21–37. ISSN: 1611-3349. DOI: 10.1007/978-3-319-46448-0_2. URL: http://dx.doi.org/10.1007/978-3-319-46448-0_2.
- [5] Tsung-Yi Lin et al. *Focal Loss for Dense Object Detection*. 2018. arXiv: 1708.02002 [cs.CV].
- [6] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2014. arXiv: 1311.2524 [cs.CV].
- [7] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. *Scaled-YOLOv4: Scaling Cross Stage Partial Network*. 2021. arXiv: 2011.08036 [cs.CV].
- [8] Mingxing Tan, Ruoming Pang, and Quoc V. Le. *EfficientDet: Scalable and Efficient Object Detection*. 2020. arXiv: 1911.09070 [cs.CV].
- [9] Alejandra C. Hernandez et al. "Efficient Object Search Through Probability-Based Viewpoint Selection". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 6172–6179. DOI: 10.1109/IR0S45743.2020.9340989.
- [10] "Advances in Computer Vision". In: *Advances in Intelligent Systems and Computing* (2020). ISSN: 2194-5365. DOI: 10.1007/978-3-030-17795-9. URL: <http://dx.doi.org/10.1007/978-3-030-17795-9>.
- [11] Ali Farhadi et al. "Describing objects by their attributes". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 1778–1785. DOI: 10.1109/CVPR.2009.5206772.
- [12] Genevieve Patterson and James Hays. "SUN attribute database: Discovering, annotating, and recognizing scene attributes". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2012, pp. 2751–2758. DOI: 10.1109/CVPR.2012.6247998.
- [13] Lubomir Bourdev and Jitendra Malik. "Poselets: Body part detectors trained using 3D human pose annotations". In: *2009 IEEE 12th International Conference on Computer Vision*. 2009, pp. 1365–1372. DOI: 10.1109/ICCV.2009.5459303.
- [14] Nathan Silberman et al. "Indoor Segmentation and Support Inference from RGBD Images". In: *Computer Vision – ECCV 2012*. Ed. by Andrew Fitzgibbon et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 746–760. ISBN: 978-3-642-33715-4.
- [15] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV].
- [16] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.
- [17] M. Everingham et al. "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88 (2009), pp. 303–338.
- [18] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv: 1905.11946 [cs.LG].
- [19] Berk Calli et al. "The YCB object and Model set: Towards common benchmarks for manipulation research". In: *2015 International Conference on Advanced Robotics (ICAR)*. 2015, pp. 510–517. DOI: 10.1109/ICAR.2015.7251504.

- [20] Migraciones y Seguridad Social Ministerio de Trabajo. *Resolución de 7 de octubre de 2019, de la Dirección General de Trabajo, por la que se registra y publica el XIX Convenio colectivo del sector de empresas de ingeniería y oficinas de estudios técnicos*. 2019.



A Data obtained during the experiment from Section 3.1.2 to find the most suitable zoom configuration.

Object	Location	Zoom	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10
Banana	Close	0.5s	90,28	91,43	92,8	91,36	90,25	92,33	88,59	91,65	92,48	89,04
		1s	99,56	99,56	99,59	99,54	99,59	99,57	99,58	99,45	99,57	99,67
		1.5s	99,87	99,83	99,92	99,9	99,92	99,82	99,89	99,9	99,88	99,88
	Mid	0.5s	0	0	0	0	0	0	0	0	0	0
		1s	82,65	85,54	87,07	90,97	91,12	88,73	94,21	90,21	93,52	93,14
		1.5s	95,57	94,59	92,07	95,43	94,42	96,19	96,46	94,65	95,69	93,21
	Far	0.5s	77,68	51,11	75,84	71,81	78,54	78,08	76,41	75,5	68,09	64,93
		1s	73,05	79,63	76,01	74,28	77,06	71,12	76,66	74,28	64,78	69,27
		1.5s	98,54	96,51	98,08	98,01	98,42	98,46	96,52	97,67	98,54	97,25
MasterChef	Close	0.5s	0	0	0	0	0	0	0	0	0	0
		1s	0	0	0	0	0	0	0	0	0	0
		1.5s	0	0	0	0	0	0	0	0	0	0
	Mid	0.5s	34,5	0	0	0	0	0	0	54,81	26,32	23,67
		1s	94,74	89,9	86,51	90,34	88,14	75,08	84,25	76,25	94,81	91,02
		1.5s	73,62	74,67	82,96	64,37	48,42	74,49	56,17	69,24	53,86	47,39
	Far	0.5s	0	0	0	0	0	0	0	0	0	0
		1s	0	0	0	0	0	0	0	0	0	0
		1.5s	0	95,41	0	0	52,06	50,18	85,05	34,6	0	0
Tomato soup	Close	0.5s	97,6	98,31	98,52	98,19	98,19	98,46	98,53	97,8	98,58	98,19
		1s	99,14	99,3	99	99,53	99,38	99,33	99,34	99,09	99,4	98,79
		1.5s	88,9	89,68	76,63	99	89,12	97,51	96,66	98,71	85	96,33
	Mid	0.5s	84,69	72,4	89,93	76,16	88,3	74,92	87,45	88,78	84,33	84,14
		1s	97,52	97,86	97,29	97,24	97,2	96,75	95,12	97,67	97,03	97,83
		1.5s	98,7	98,14	98,77	98,23	95,41	98,28	98,33	97,09	97,8	97,05
	Far	0.5s	0	0	0	0	0	0	0	0	0	0
		1s	42,93	58,79	27,67	65,07	0	35,03	0	37,85	0	0
		1.5s	39,9	90,12	0	56,87	82,12	0	51,75	95,26	25,53	97,59

B Data obtained from the experiment to calculate the angle/pixel relation.

Deg($^{\circ}$)	Pixel(px)
51.9	1680.18139648438
55.1	1614.35302734375
58	1545.68176269531
60.7	1481.87841796875
63.4	1453.27722167969
66.4	1394.87963867188
69.6	1324.93627929688
72.3	1295.57800292969
75.2	1239.87036132813
78.4	1177.19213867188
81.7	1131.521484375
84.4	1076.82666015625
87.3	1001.3798828125
90.3	950.402160644531
93.2	909.123413085938
96.4	874.994873046875
99.1	817.626831054688
101.8	777.937072753906
104.8	717.7470703125
107.7	658.462463378906
110.7	617.664123535156
113.4	557.690185546875
116.1	501.145202636719
119.3	435.194793701172
122.2	376.131103515625

Deg($^{\circ}$)	Pixel(px)
55.5	301.023834228516
52.7	330.185180664062
49.5	422.391723632813
46.4	471.05419921875
43.3	565.895568847656
40.5	619.708251953125
37.3	686.763427734375
34.5	784.604370117188
31.4	840.0126953125
28.6	905.890380859375
25.5	968.500854492187
22.3	996.925598144531
19.5	1047.67785644531
16.4	1065.91906738281

Left: Calibration for the Phi angle. Right: Calibration for the Theta angle.



C Experiment ran to test how fast each method of sorting presets is, starting from a preset.

Test	ObjectPreset (1-3)	Method (0-2)	T1 (s)	T2 (s)
1	1	0	5.440745	4.108289
2	1	0	4.931125	4.905578
3	1	0	4.862637	4.720624
4	1	0	4.742256	4.805722
5	1	0	4.705323	3.860643
6	1	0	5.352498	3.821826
7	1	0	4.952634	4.769585
8	1	0	5.60212	4.805721
9	1	0	4.802607	4.878949
10	1	0	4.764583	4.815277
1	1	1	5.012756	4.774413
2	1	1	4.490681	4.944368
3	1	1	4.956216	4.860143
4	1	1	4.864151	4.755765
5	1	1	4.915378	5.140073
6	1	1	5.392666	4.531414
7	1	1	5.35137	4.839542
8	1	1	4.820868	4.822848
9	1	1	4.353036	4.982794
10	1	1	4.853458	4.811543
1	1	2	7.699577	4.727278
2	1	2	6.502057	3.924356
3	1	2	8.367234	4.845201
4	1	2	7.137763	4.810908
5	1	2	7.870453	4.907037
6	1	2	7.763679	4.707196
7	1	2	8.261545	4.885301
8	1	2	8.488456	4.921242
9	1	2	7.794417	4.806834
10	1	2	7.810523	4.984634
1	2	0	7.997948	10.173258
2	2	0	8.40131	8.803617
3	2	0	8.550154	9.051141
4	2	0	8.39631	9.308785
5	2	0	8.570898	9.108183
6	2	0	7.883772	10.144775
7	2	0	8.296108	9.232687
8	2	0	7.496016	9.10504
9	2	0	8.391494	9.215083
10	2	0	7.457804	9.116818

Test	ObjectPreset (1-3)	Method (0-2)	T1 (s)	T2 (s)
1	2	1	8.526379	9.230194
2	2	1	8.420078	9.195289
3	2	1	8.465677	9.022929
4	2	1	8.439693	9.143587
5	2	1	8.446753	10.179373
6	2	1	8.551007	9.182487
7	2	1	8.510614	9.1384
8	2	1	8.500418	9.118257
9	2	1	8.47856	9.138288
10	2	1	8.417107	9.182004
1	2	2	10.628758	13.154621
2	2	2	10.292102	12.063919
3	2	2	10.544353	14.197284
4	2	2	10.480666	13.181307
5	2	2	9.428504	13.104914
6	2	2	10.612905	13.147137
7	2	2	10.412317	13.117277
8	2	2	9.511157	13.330504
9	2	2	9.48687	12.03952
10	2	2	10.457186	13.014444
1	3	0	12.472714	14.047441
2	3	0	11.682189	12.763835
3	3	0	11.930436	14.760452
4	3	0	11.930604	12.93618
5	3	0	12.541095	14.551868
6	3	0	13.054847	12.917886
7	3	0	11.848854	14.807478
8	3	0	11.833205	12.877573
9	3	0	11.911335	14.783823
10	3	0	11.678804	13.947188
1	3	1	11.843059	13.814468
2	3	1	11.651696	13.824818
3	3	1	11.878094	13.846315
4	3	1	11.908419	14.945102
5	3	1	12.037466	13.81289
6	3	1	11.45235	13.890492
7	3	1	11.082211	13.846155
8	3	1	11.825238	13.90164
9	3	1	11.800295	13.777056
10	3	1	11.912134	13.985395
1	3	2	2.750406	8.039391
2	3	2	2.829938	7.845414
3	3	2	2.850347	7.81546
4	3	2	2.946135	8.814209
5	3	2	2.827387	8.675027
6	3	2	2.895413	8.953923
7	3	2	2.819215	8.777783
8	3	2	3.903325	8.89656
9	3	2	2.87333	8.881379
10	3	2	2.986938	8.863754



Table 8: Experiment ran to test how fast each method works on preset locations.

D Experiment ran to test how fast each method of sorting presets is, starting from a random location.

Test	ObjectPreset (1-3)	Method (0-2)	Initial Yaw	Initial Pitch	T3 (s)	T4 (s)
1	1	0	296	85	5.355835	6.198715
2	1	0	161	15	4.292737	4.6041
3	1	0	147	57	5.111652	5.675354
4	1	0	341	47	5.661327	6.12894
5	1	0	280	12	3.753126	4.285337
6	1	0	346	25	4.62042	4.925435
7	1	0	230	51	3.894468	5.379351
8	1	0	114	23	4.542976	4.599553
9	1	0	214	50	3.43888	4.419661
10	1	0	214	39	5.14193	5.096834
1	1	1	115	56	5.19944	6.095889
2	1	1	336	48	4.210424	5.011021
3	1	1	135	60	5.708997	6.742437
4	1	1	163	69	6.499028	6.416729
5	1	1	341	26	3.448264	4.814528
6	1	1	54	21	4.36307	5.561234
7	1	1	54	81	4.452569	5.326326
8	1	1	330	41	4.323724	4.885993
9	1	1	24	31	3.600964	4.841056
10	1	1	276	77	4.290774	6.051575
1	1	2	115	56	10.590639	13.45973
2	1	2	336	48	4.183825	5.145088
3	1	2	135	60	11.932914	12.70448
4	1	2	163	69	10.276366	12.947578
5	1	2	341	26	3.569508	4.555827
6	1	2	54	21	11.582718	13.195234
7	1	2	54	81	10.19756	12.801815
8	1	2	330	41	4.173595	5.164854
9	1	2	24	31	3.536538	4.641839
10	1	2	276	77	8.129184	9.872927
1	2	0	115	56	8.966817	10.52799
2	2	0	336	48	7.636756	9.772862
3	2	0	135	60	8.840411	11.665037
4	2	0	163	69	9.360278	11.74866
5	2	0	341	26	8.063144	10.027717
6	2	0	54	21	7.865696	9.984597
7	2	0	54	81	8.326564	10.21037
8	2	0	330	41	8.239071	10.267279
9	2	0	24	31	7.855822	10.396547
10	2	0	276	77	8.825725	10.663517

Test	ObjectPreset (1-3)	Method (0-2)	Initial Yaw	Initial Pitch	T3 (s)	T4 (s)
1	2	1	115	56	9.862791	11.301649
2	2	1	336	48	8.404968	10.341044
3	2	1	135	60	9.176358	11.462538
4	2	1	163	69	10.02533	11.648452
5	2	1	341	26	8.800029	9.891036
6	2	1	54	21	8.324117	10.360915
7	2	1	54	81	8.807452	10.324859
8	2	1	330	41	7.779609	9.635132
9	2	1	24	31	7.386253	9.496755
10	2	1	276	77	8.459748	10.806296
1	2	2	115	56	3.526864	4.739981
2	2	2	336	48	11.345025	13.319632
3	2	2	135	60	3.947531	5.655431
4	2	2	163	69	4.704318	6.207249
5	2	2	341	26	10.015776	13.834064
6	2	2	54	21	4.122974	5.09466
7	2	2	54	81	3.61455	4.664015
8	2	2	330	41	11.162609	13.219937
9	2	2	24	31	10.528772	14.119736
10	2	2	276	77	12.413286	14.510691
1	3	0	115	56	11.606246	15.26129
2	3	0	336	48	10.636896	14.484901
3	3	0	135	60	11.795585	16.161145
4	3	0	163	69	12.34224	16.548732
5	3	0	341	26	11.371668	14.887249
6	3	0	54	21	11.357694	14.640394
7	3	0	54	81	11.158462	15.768765
8	3	0	330	41	12.750702	13.868296
9	3	0	24	31	4.201119	13.451427
10	3	0	276	77	13.174252	16.206728
1	3	1	115	56	13.172516	16.179411
2	3	1	336	48	10.942178	14.559342
3	3	1	135	60	11.947202	15.88583
4	3	1	163	69	12.827482	15.20901
5	3	1	341	26	11.766131	14.610677
6	3	1	54	21	10.922475	14.351548
7	3	1	54	81	11.640892	15.407188
8	3	1	330	41	10.641409	14.695605
9	3	1	24	31	11.88302	14.185465
10	3	1	276	77	12.138822	14.189849
1	3	2	115	56	6.31213	8.621667
2	3	2	336	48	7.521075	8.608123
3	3	2	135	60	7.982072	9.721359
4	3	2	163	69	8.276324	10.912534
5	3	2	341	26	6.748288	8.67344
6	3	2	54	21	7.296669	8.671228
7	3	2	54	81	6.583487	9.059422
8	3	2	330	41	6.657609	9.130657
9	3	2	24	31	6.535875	8.712575
10	3	2	276	77	4.712756	5.508226



Table 9: Experiment ran to test how fast each method works on random camera positions.