 doi:10.20944/preprints202106.0037.v1

Article

Implementation of NAO robot maze navigation based on computer vision and collaborative learning

Daniela Magallán-Ramírez[1,‡], Areli Rodriguez-Tirado[1,‡], Jorge David Martínez-Aguilar[1,‡], Carlos Francisco Moreno-García[2,*], David Balderas[1], Edgar Omar López-Caudana[1]

[1] Tecnologico de Monterrey, School of Engineering and Sciences. Calle Puente 222, Coapa, Arboledas del Sur, Tlalpan, 14380 Ciudad de México, CDMX
[2] Robert Gordon University, School of Computing. Garthdee House, Garthdee Rd, Aberdeen AB10 7GJ, UK
* Corresponding author: c.moreno-garcia@rgu.ac.uk; Tel.: (+44) 1224 262790
‡ These authors contributed equally to this work.

Abstract: Maze navigation using one or more robots has become a recurring challenge in scientific literature and real life practice, with fleets having to find faster and better ways to navigate environments such as a travel hub (e.g. airports) or to evacuate a disaster zone. Many methods have been used to solve this issue, including the implementation of a variety of sensors and other signal receiving systems. Most interestingly, camera-based techniques have increasingly become more popular in this kind of applications, given their robustness and scalability. In this paper, we have implemented an end-to-end strategy to address this scenario, allowing a robot to solve a maze in an autonomous way, by using computer vision and path planning. In addition, this robot shares the generated knowledge to another by means of communication protocols, having to adapt its mechanical characteristics to be able to solve the same challenge. The paper presents experimental validation of the four components of this solution, namely camera calibration, maze mapping, path planning and robot communication. Finally, we present the integration and functionality of these methods applied in a pair of NAO robots.

Keywords: robot navigation, computer vision, camera calibration, mapping, path planning, communication, NAO robot, educational innovation, higher education.

1. Introduction

An autonomous system for maze navigation must be able to take real-time decisions for different and sometimes unexpected problems that may occur. For instance, the robot might be aware of its environment by using different sensors, data processing algorithms or other methods. Recently, one of the most widely used methodologies consists of collecting the information that the robot needs by means of the embedded cameras. This kind of technique have been implemented no just in competitions [1], but also in practical applications, such as airport navigation by a fleet of robots [2]. The problem of using other sensors (e.g. sonar or infrared lights) is that they require conditional programming, which particularizes each solution to a specific type and version of robot. Therefore, academia and industry work hard to find methods that can be applied to different robotic platforms and that would allow real-life applications in areas such as education, industry 4.0, health, etc.

This paper presents the test and comparison of different methods that will conform an autonomous maze solution system for a pair of NAO[1] robots. This will enable them to navigate a maze in an autonomous way by means of four main parts: (1) camera calibration, which allows the acquired images from the first robot to be pre-processed and cleaned prior to their use; (2) mapping, which consist in vision algorithms to analyze

[1] https://www.softbankrobotics.com/emea/en/nao

 doi:10.20944/preprints202106.0037.v1

the images and generate an internal map; (3) path planning, which allows the first robot to make decisions about the navigation to get out in the most optimal way and; (4) communication, which will allow to transmit the generated knowledge to allow a second robot to solve the considering its mechanical characteristics. We have selected this particular robots since they can be programmed using a multi-platform environment called *Choregraphe*, which will allow us to integrate script from languages such as C++, YAML and Python. Moreover, NAO robots possess a variety of sensors which could be integrated to our solution in future work, such as touch sensors, omni-directional microphones and ultrasonic sensors[2]. We will be using the 6^{th} generation of these robots, which are used mainly in education and research.

This is an extension on the paper presented by Rodriguez-Tirado et al. [3], where we introduced an initial proposal which would allow the navigation of robot into a maze by means of the aforementioned pipeline framework. Nonetheless, in this paper we will thoroughly review and validate each step, explaining why certain methods were selected in each step. The modifications implemented in each step have as final objective to fully integrate the system. Finally, this paper presents the migration of this final integration into a real environment.

After the introduction in Section 1, this paper is organized as follows. Section 2 contains a literature review of the existing investigations and applications of similar methodologies in various scenarios. Section 3 describes how to interconnect the four modules proposed to solve robot navigation and introduces the methods used in each module. Section 4 presents bot the results of all tests in each step, and the integration of such modules. Afterwards, Section 5 analyses these results. Finally, Section 6 is reserved for conclusions and future directions.

2. Related Work

2.1. Camera calibration in NAO robots

To improve the response time of a fleet of NAO robots playing football soccer, Kastner et al. [4] implemented a kinematic calibration within the robots. They used a calibration method originally proposed by Zhang et al. [5], which will also be considered in our work. This method is based on a chess pattern, and the calibration result was helpful to compensate the errors of the NAO robots legs, which allowed these robots to be able to better kick the ball. Another objective of such work was to perform a fast calibration before a match, thus resulting on an elapsed time of about 590 seconds. According to that paper, in a ten minute match without the calibration, the robots fell an approximate of 11 times per match, and after the calibration, this was reduced to six times.

NAO robots use camera calibration for other applications, such as portrait sketching as shown by Singh et al. [6]. In this implementation, it was necessary to change the plane from 3D to 2D by transforming the effector position into the NAO, so that the robot, instead of having to compute the coordinates of a tri-dimensional plane, could just perceive the x and y coordinates to sketch a portrait. Authors evaluated three solutions for the 3D-2D transformation, which were: (1) fundamental matrix; (2) 4-point calibration and an; (3) an artificial neural network (ANN) based regression analysis. Comparing the experimental results of the three solutions, the third option was the best since it derived on less square error. Conversely, the method with the lowest computational time was the 4-point calibration.

2.2. Path Planning

One of the most widely used solutions used in this domain the *Tremaux* algorithm, which has been subsequently applied in literature by different authors. For instance, Li et al. [7] implemented this approach with the purpose of having a mobile robot in a virtual environment be able to navigate and find the minimum cost in the shortest time. As a

[2] http://doc.aldebaran.com/2-1/family/index.html

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

result, the robot was able to indicate the shortest path, without actually being able to navigate it or to communicate the solution to a second robot. An important feature of this work is that the robot was given the location of the exit; something that may not necessarily be our case.

Kumar et al. [8] used path planning and other algorithms aimed at avoiding obstacles to solve maze routes, by means of a linear regression method. The robot was trained with 500 different scenarios. In a second part of the work, a second robot with the same algorithm was introduced to the same route. The main purpose was that both robots could solve the puzzle simultaneously without any collisions. The obstacles consisted of cylinders, and the robots had to follow an establish path. Therefore, robots just had to get from one point to another with a given trajectory that would be adjusted depending of physical obstructions. Although this holds similarity to our proposal, authors do not specify whether the communicated knowledge to the second robot considered different mechanical properties of that robot.

Wikström et al. [9] worked on how a robot can process its environment to map and navigate through a maze. Using an infrared light sensor, the robot was able to obtain information from its surroundings. Python 2 was used to program the mapping and path planning modules. In this thesis, the efficiency between the Wall Followers and the *Tremaux* algorithm were compared. It was concluded that the latter was more effective to solve the task at hand. This work has some similarities with our proposal, nonetheless no computer vision algorithms were implemented in that case, which effectively showed a reduction in reliability and accuracy of the obtained solution.

2.3. Communication Protocols

Simoens et al. [10] defined and explained the potential benefits of the Internet of Robotic Things (IoRT). Some of the key benefits of this area of knowledge are perception, motion, manipulation, interaction, adaptability, configuration and decision autonomy. In particular, the latter has the ability of determine the best action for a fleet of robots to undertake. In our work, we propose to connect the NAO robots by means of commercial platforms such as *Ubidots*, in which the data recall, such as the instruction for the robot, can be download and transfer easily. This in effect has some relation to the IoRT concept.

Bechon et al. [11] implemented a swarm code to coordinate the actions of eight NAO robots. As a result of this work, the fleet was able to communicate among them by using a global variable to synchronize the whole NAO robot group in a dance choreography. The robots were synchronized using the Network Time Protocol (NTP), getting all NAO robots synchronized. Distance between the robots was an important factor; while it increases, also the time to detect and correct the next position does. Because of that, it was necessary to implement a mesh network where all robots had access to each other, so that each robot was able to communicate with their closest neighbor. This type of architectures are interesting for a future implementation of this work, provided that we need to synchronize multiple robots simultaneously.

2.4. Other systems

Needless to say, maze navigation can be applied to multiple other robotic standards in the industry. For instance, Gul et al. [12] explained how different navigation techniques can be applied into a mobile robot to follow a dynamic path. The robot had to go from the start to the end by avoiding obstacles between the points and analyzing a 2D plane. Some of the algorithms used were the Dijkstra method (which will be explored in this paper). This was found to be a simple, effective and less time and computational complex method with a very high efficiency for the task.

Delfin et al. [13] aimed at localizing and helping a NAO robot to navigate into an environment by using a visual memory which consisted on a set of key images. First, the NAO got into the environment and tried to solve by mimicking a human. In parallel, images were acquired and used to generate a graph in which a path planning

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

method was applied to help the robot optimize it trajectory in the environment. To our knowledge this is the closest work in literature to our proposal. Nonetheless, we have the added complexity of a second robot which has to solve the same challenge based on a shared knowledge and considering different mechanical characteristics.

3. Methodology

As mentioned in Section 1, we developed this work in four main parts. Each of the steps in the pipeline was developed separately; however, the flow chart in Figure 1 shows the interconnection of these modules. In this section, we describe the technologies used for each stage. Communication sessions are colored in orange, camera calibration in red, mapping are the blue and finally, path planning tasks are in green.

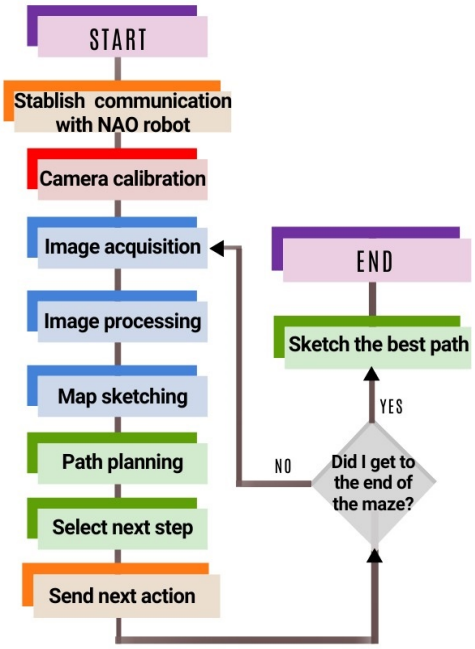


Figure 1. Pipeline of the proposed system.

The system starts by establishing the communication between the robot and the computer to control the robot. Then, camera calibration is applied before the robot gets into the maze, as the robot must use the cameras to solve the maze. Afterwards, the robot is ready to start the maze. Thus, the NAO will take and process an image to begin generating a section of the maze map. That section of the map will then be used in path planning related tasks to tell the robot what step to take in order to exit the maze. When the robot finishes its movement, it will continue acquiring these images to create the internal map and advance according to the process that was mentioned before. This process will repeat until the image that the robot detected depicts the exit. When the robot exits, it will then know a route to exit the maze, and will also have acquired an internal map of the part of the maze where it passed. This information will be used to select the best path according to the robot's experience, and transmit this to the second one. Due to the nature of this process, we will present the four main parts in order of usage. In the following subsections, we will describe the integration process of these four modules.

3.1. Camera calibration

Camera calibration is a process that helps minimize image imperfections caused by of intrinsic and extrinsic camera parameters. This pre-processing step is used to accurately establish a relationship between 3D point (in the real world) and its projection in a 2D plane (image pixel). Moreover, the camera must be calibrated so that it captures images

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

that are not distorted, and that the results obtained from the computer vision algorithms are less prone to errors. Furthermore, the robot will collect information such as the distances between the images, and thus the importance of this stage.

To achieve this, we implemented the method presented by Zhang et al. [5] which generates a calibration matrix to correct these distortions by taking multiple images of a well-defined object, in this case, a checkerboard. Some parameters such as brightness, saturation, and focal length must be manually adjusted to get the best possible image. Then, we generate the matrix by taking the corner patterns of the checkerboard which has dimensions of 6×5. Afterwards, we detect the corner and get the camera matrix and its distortion coefficients using modules contained in the OpenCV framework[3]. Finally, with the obtained information we generate a new matrix that will be used to un-distort the images.

3.2. Mapping

Robotic mapping refers to the ability of an autonomous robot to construct a map while being able to position itself within it. Navigation and mapping are closely related, and these concepts rely on different algorithms. However, while investigating the existing approaches, we found that most of them use different sensors (e.g. infrared sensors, odometers, GPS, etc.) which contrasts from our proposed scenario and therefore, adapting existing solutions is not an affordable task. As a result, we will be presenting a proprietary solution in this paper based on computer vision as the input.

Moreover, due to the COVID-19 worldwide pandemic, access to the NAO robots and the material for building up a test environment was restricted at the beginning of this project. From that perspective, the best solution was to take advantage of existing software tools to start the mapping module development. To this end, we used Coppelia Sim software[4], which is a 3D model of a maze which was deployed virtually to record a video simulating the first robot traversing the maze. Then, that video was used to try out the map reconstruction with the proprietary code. To simulate the navigation a robot model that comes by default in Coppelia (called Ackerman), steering car having ease of control with the keyboard arrows was used. Additionally, a camera was attached to an Ackerman robot, and using OBS Studio video editing software, the video was recorded. Figure 2 shows the simulation environment including the Ackerman model and the resulting view from the camera.

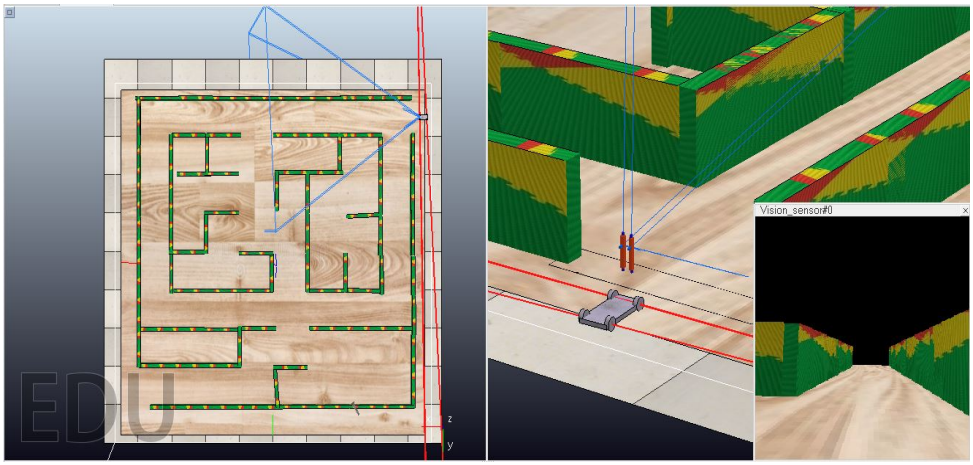


Figure 2. Simulation environment.

To achieve the mapping, the first step consists on applying the Canny algorithm [14] to divide the image into different areas which allow to better control what the robot

[3] https://opencv.org/
[4] https://www.coppeliarobotics.com/helpFiles/index.html

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

detects and maps, as shown in Figure 3. Three zones are obtained: (1) the top zone (i.e. the farthest); (2) the middle zone and; (3) the bottom zone (i.e. the closest). The robot uses just the middle zone to sketch the maze, and anything above or beyond zones 1 and 3 is discarded. To make the distinction between zones, the Canny edges of each section are stored in different arrays. This way, we can manage them individually as appropriate. Afterwards, the probabilistic Hough transform [15], [16], followed by a merge algorithm [5], is applied to each zone to find the lines which define the maze. This merging algorithm was considered since it makes the resulting array of probabilistic Hough lines smaller, thus making then easier to manipulate it frame by frame.

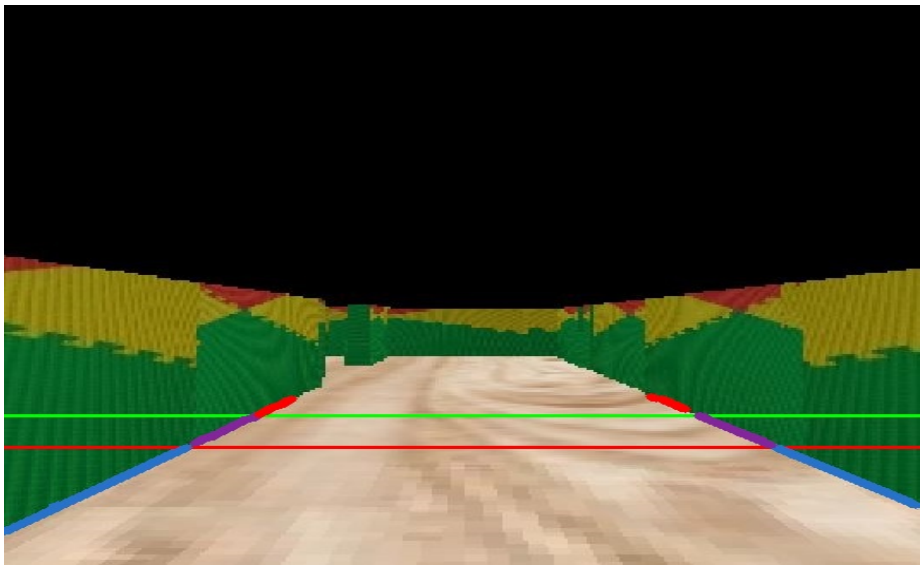


Figure 3. The three areas obtained by the robot, divided by green and red lines.

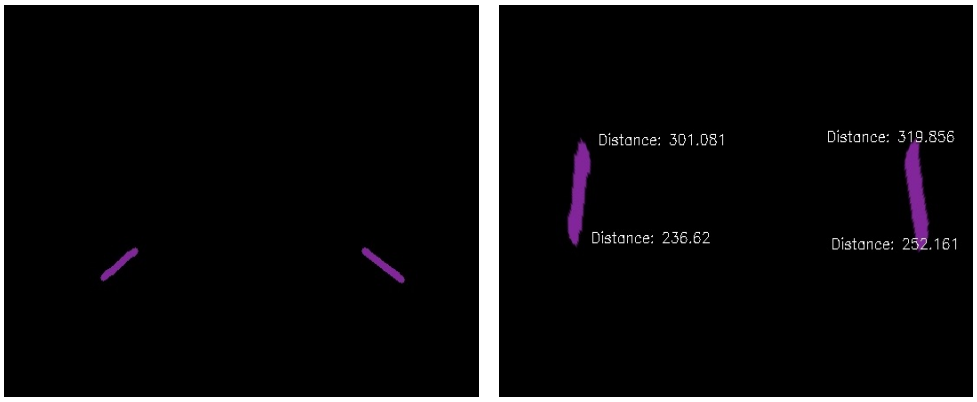
At this point, wall detection is carried out by changing the state of the two flags within the code, depending on whether the left or right wall is detected. If there is no wall, the Hough algorithm returns an empty array for the analyzed region, causing the flag to be deactivated within the code. Otherwise, if the array has elements, the flag is activated (provided that there is a wall). Then, the obtained middle lines (if any) are superimposed over a black image using a transform algorithm [6] which allows a better perspective of the space in front the robot, allowing it to determine approximate distances from the surrounding environment. For the estimation of distances, we used the Euclidean distances based on the pixels.

Subsequently, in a new black image, like Figure 4 , the map sketching takes place by using a previously defined starting point, and then taking the distances to a 10 : 1 ratio, so that the entire map fits into a single image. The lateral lines that the robot visualizes in each frame are drawn from the starting point and when the sketching for this frame ends, the starting point is updated, taking as the new starting point the last coordinate of the drawn lines. In this way, the next segment of the map will be sketched just after what has already been plotted.

As the tests were conducted, it was decided that during the orientation changes, the mapping would stop drawing the maximum possible number of useless lines as possible; the rationale behind this will be explained in more detail in Section 4.2. Moreover, as the tests were made by means of a video, and there was no way of knowing when the robot was positioned, we had to define in advance when to stop mapping. This sketching process resumes once the presence of both walls is detected, as this indicates that it is rejoined into a corridor.

[5] https://stackoverflow.com/questions/45531074/how-to-merge-lines-after-houghlinesp
[6] https://github.com/ndrplz/self-driving-car

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1



(a) Middle zone superimposed. **(b)** Middle zone transformed.

Figure 4. Perspective Transform.

3.3. Path planning

For this stage, we require a method which allows the robot to navigate the maze without having a notion of where the exit is, while simultaneously intending to calculate the best possible route. To this end, the first option we contemplated was the Dijkstra Shortest Path algorithm presented by Ably et al. [17]. This method generates a tree of shortest paths from the starting point to all other points in the plane. Each possible step is a node that consists of two values, the distance to the start point and the last neighbor to get to that point with the least cost. There is a queue while it is not empty, as there are unknown possible nodes to visit. Also, there is a list that contains all the visited nodes with its characteristics. This method is based on discovering the new areas by extending a circular trajectory, and it is necessary to go back to the beginning to calculate distances. In this way, it ensures that the entity traversing the route will discover each possible step in all directions until the exit is found.

The second option we considered was the Trémaux algorithm, which is widely known as an efficient method to get out of a maze [18]. It mark all the steps and the direction were the robot comes from. In this method, the navigator avoids traversing the same path twice by means of two lists. The first list contains all visited nodes in the order that they were found. The second list stores the path towards the exit. In principle, this second list will be a copy of the first one, but each time that the robot comes back to a node, all preceding nodes will be erased. For the actual implementation in the robot, we set priorities to four possible moves in the following order: (1) front step; (2) left step; (3) right step and; (4) back step. Figure 5 shows a schematic depicting these four options.

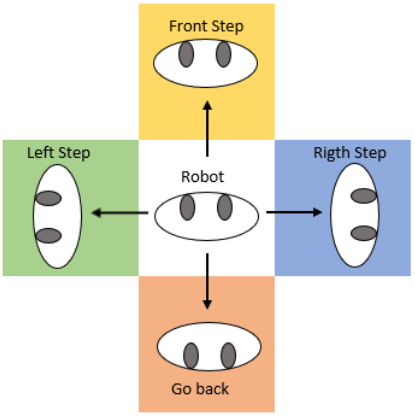


Figure 5. Priority of the steps that a robot can take when traversing the maze.

There are some reasons behind this order. Firstly, taking a step back is something that usually has to be avoided, because it means that the navigator will go back to a node

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

that it has already visited. With that logic in mind, the back step has the last priority. For the other steps (i.e. front, left and right) this rule is not applicable, and it does not really make any different whether one has a highest or lowest priority between them. Figure 6 shows values for each node representing the number of times that the robot has passed. The black parts represent the walls. The robot is in the green node, so it can go through the four possible steps. According to our approach, the robot should avoid the back step because it has already been there. Still, the robot does not have any clue about where is the exit, thus the other steps have a 33.33% of leading to the correct path.

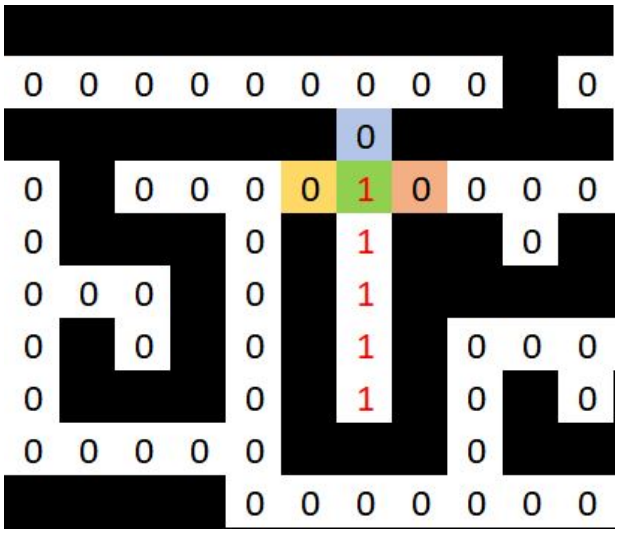


Figure 6. Priority of steps within a maze.

We made an exception to the Tremaux´s algorithm rule of never having to go back to a node twice because this could get the robot caught in some situations like the one shown in Figure 7. In that case, the node marked in red has been crossed more than once, however we allow the robot to cross again to find the exit, even when there is a priority to go to the nodes with less visits.

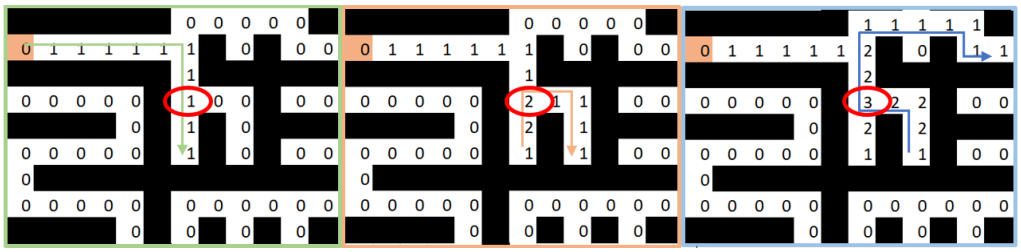


Figure 7. An image representing why a robot could pass 3 times in the same node while looking for an exit.

3.4. Communication

The NAO robot can communicate with a computer via Ethernet and Wi-Fi to be programmed. In our case, Wi-Fi was selected because the NAO robot will be in constant motion as it traverses the maze. Evidently, it would not be comfortable walking behind the NAO robot with the Ethernet cable connected to a computer. The connection via Wi-Fi has the advantage of connecting with other web services, and any data can be transmitted and received directly between the computer and the NAO robot, such as temperature measure of some motor. This communication architecture is similar to Ad-hoc, but in this case, there is an access point that works as an intermediary. As mentioned before, Choregraphe is the official software developed by Aldebaran Robotics for the programming of the NAO robot, which uses the NAOqi framework so that it

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

can be programmed in different programming languages, such as C ++, Python, Java, Matlab, Urbi and .Net, amongst others. To establish the wireless communication, the robot uses the registered port 9559. Note that registered ports are only used by some specific application or service[7].

4. Test Bench

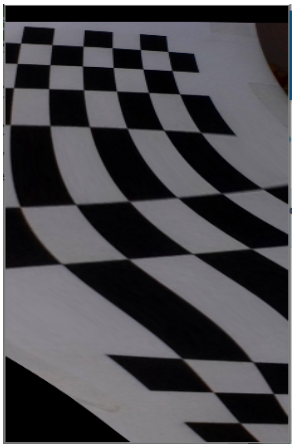
4.1. Camera calibration

By taking several pictures of a checkerboard in different angles and under different lighting conditions, we generate an image database for the calibration. This calibration focuses on a ROI, which is the checkerboard itself. Evidently, the more images that are obtained, the better results that can be achieved. Nonetheless, we decided to stop at 51 images, since the matrix calibration parameter adjusted well to the image ones, as shown in Table 1. The dimension of the images are 680×420 and the calibration matrix use the center of the image, i.e. cx and cy, so base on those we calculate the error (%cx and %cy) between the real centers and the ones of the matrix. We can see that the error of both centers at 51 images is of less than 0.05.

Images	cx	cy	%cx	%cy
10	170.003	42.4261	0.469	0.823
20	159.479	28.9455	0.502	0.879
30	248.704	222.34	0.223	0.074
35	349.239	222.55	0.091	0.073
40	377.333	205.08	0.179	0.146
45	331	233	0.033	0.028
47	394	130	0.232	0.459
49	357	198	0.117	0.177
51	320.229	236.477	0.001	0.015

Table 1: Comparative table between the centers of the real image and those of the matrix (the best results achieved are highlighted in red).

Moreover, Figures 8a and 8b show how the calibration works with different number of samples.



(a) Undistort image with 10 samples.

(b) Undistort image with 20 samples.

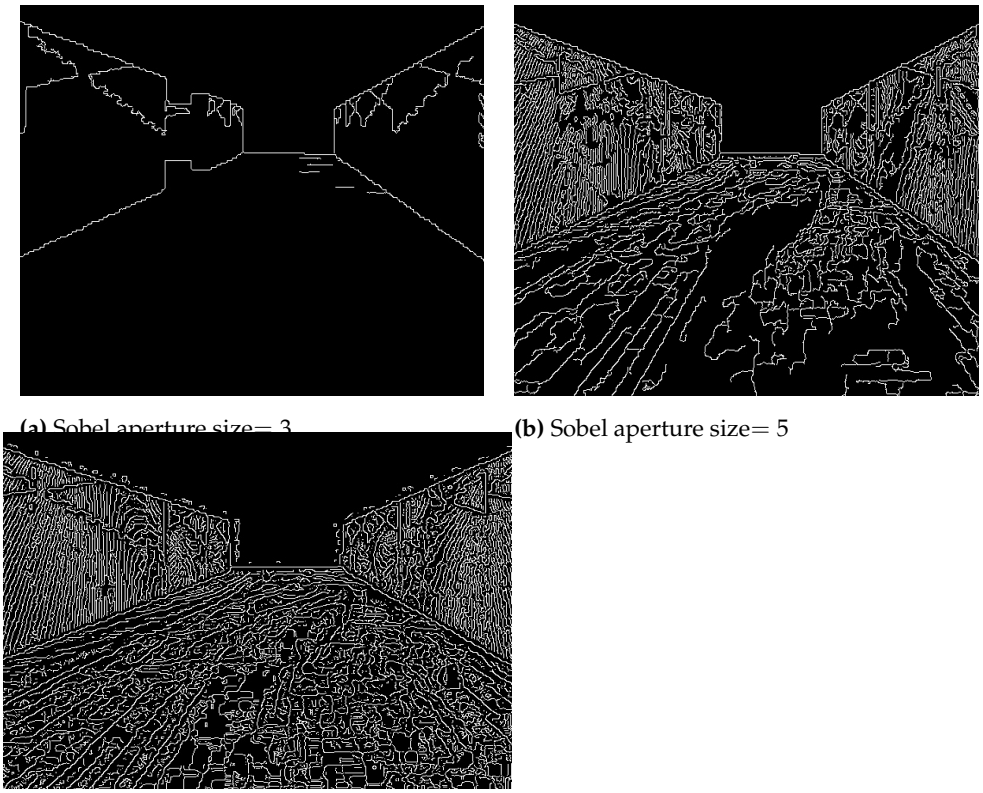
Figure 8. Calibration examples.

[7] https://www.iana.org/form/ports-services

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

4.2. Mapping

The Canny edge detection function contained in Python's *skimage* package[8] requires the configuration of three main parameter: two thresholds (one upper and one lower) and the aperture size for the Sobel operator. Therefore, the performance of this function was tested with different values to find the ideal configuration for this application. The first parameter that was experimented was the size of the Sobel operator, since this is used on the second step of the Canny algorithm after smoothing the image and so, it is the most significant parameter. This operator allows to acquire the orientation that each pixel is pointing at, by returning the value of the first derivative for the horizontal and vertical direction. We initialized the parameter with a value of 3, given that during our literature research we found that this is the standard size used; however, tests were run with aperture sizes of 5 and 7 as well. Ultimately, we confirmed that an aperture of 3 would yield the best performance. The results obtained are shown in Figure 9.



(a) Sobel aperture size= 3

(b) Sobel aperture size= 5

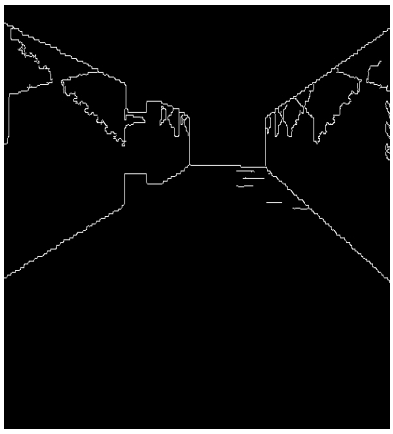
(c) Sobel aperture size= 7

Figure 9. Tests for the Sobel operator aperture size.

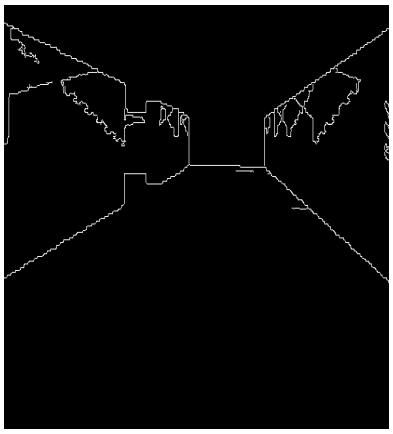
To determine the upper threshold, we decided to start the tests with a threshold equal to 200, which was arbitrarily assigned. We noticed that its performance was not poor; however, the image returned showed many lines on both the walls and the floor. The threshold value was increased in steps of 10 until the optimal value was achieved. As seen in Figure 10, at a value of 250 the number of lines detected diminished. Furthermore, when reaching a threshold of 300 we observed that the images of the walls were significantly cleaner, and only one line was detected on the floor. From that threshold onward, the changes began to be less significant; when the threshold reached 400 we observed that a few lines were cleaned on the walls, but there was no significant changes. Finally, at a threshold of 500 no improvement was noticed, and thus this value (and any further) were discarded.

[8] https://scikit-image.org/docs/dev/auto_examples/edges/plot_canny.html

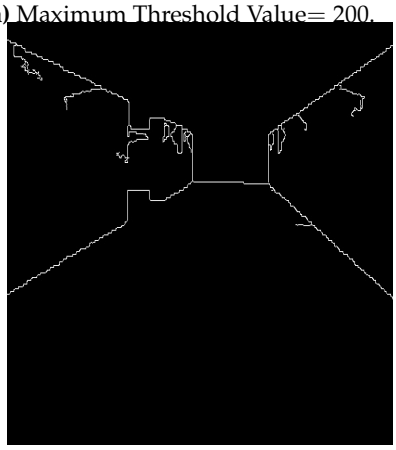
Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1



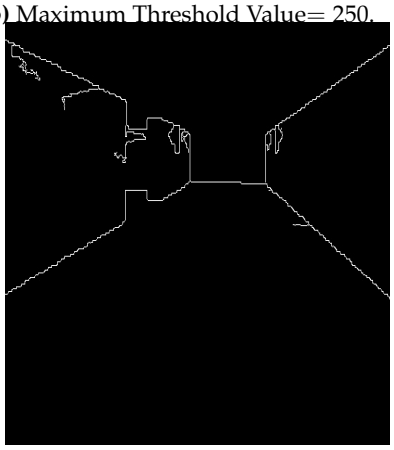
(a) Maximum Threshold Value= 200.



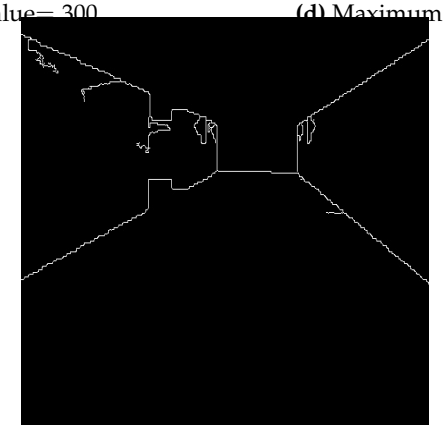
(b) Maximum Threshold Value= 250.



(c) Maximum Threshold Value= 300.



(d) Maximum Threshold Value= 400.



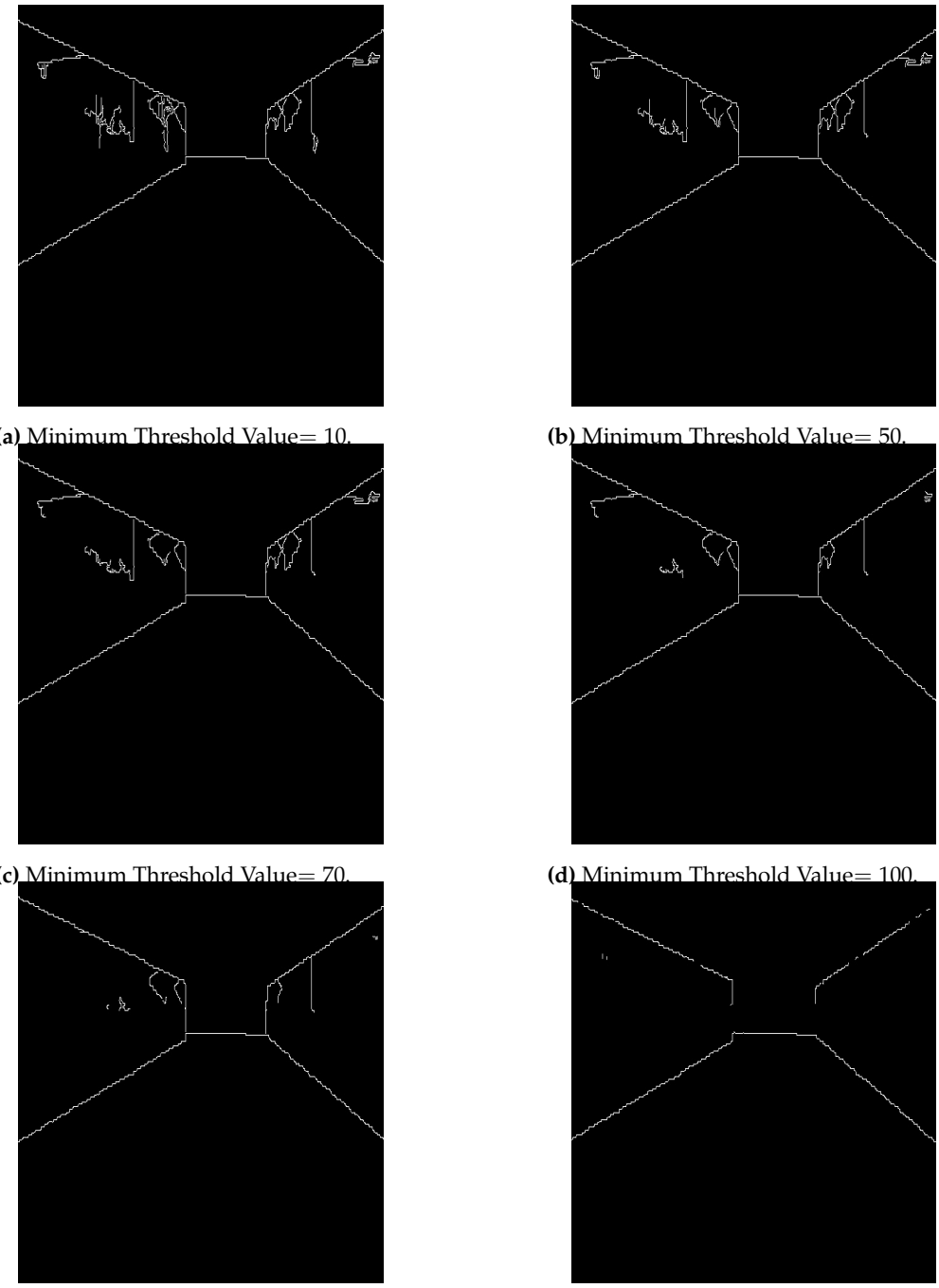
(e) Maximum Threshold Value= 500.

Figure 10. Testing of different upper threshold values.

Therefore, it can be concluded that an acceptable value for the upper threshold could be found at the range of 250 to 400. It was then decided to use a value of 300 for this threshold, leaving it at an arbitrary but at the same time optimal value. On the other hand, in order to clean the image more and achieve better results, the lower threshold was adjusted as well. Figure 11 shows that the best image is obtained with a value of 150 since, if we using a filter of 200, certain empty spaces in the image begin to appear as full. This can cause problems in the lines detection, hence it was decided that it was better to leave the threshold at a 150; however, it is observed that the image is not completely cleaned.

As mentioned before, with thresholds 150 and 300 the walls were not completely clean; this raised concerns regarding whether to use an additional Gaussian filter to

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1



(a) Minimum Threshold Value= 10. **(b)** Minimum Threshold Value= 50.

(c) Minimum Threshold Value= 70. **(d)** Minimum Threshold Value= 100.

(e) Minimum Threshold Value= 150. **(f)** Minimum Threshold Value= 200.

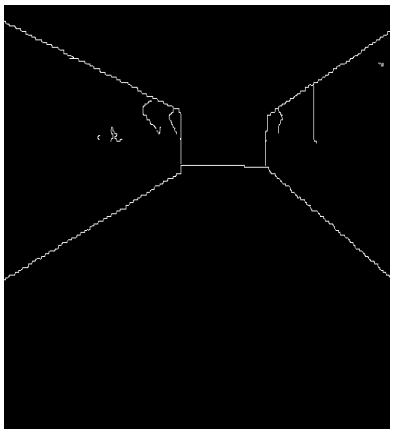
Figure 11. Testing of different lower threshold values.

remove these imperfections to prevent that they would not affect the line detection. Figure 12 shows a comparison of performance with a second filter and without it, some improvement can be seen. However, recall from Section 3.2 that, in order to limit of the robot's vision to make processing easier, only a part of the image is being taken, which can be seen in Figures 12d and 12c. Therefore, the second filter has no effect on the final image, and thus er decided not to include it for the simulation.

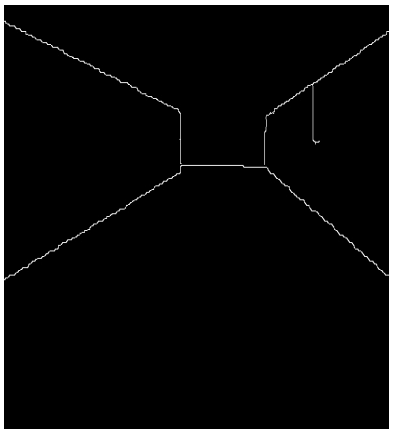
In spite of the simulation results, the second filter was not discarded as an option for real time tests since, as can be seen on Figure 13, on a real environment the floor actually presents many lines that could affect the performance of the algorithm.

On the testing phase, the possibility of using a corner detection algorithm instead of a line detection algorithm was considered; however, corner detection had a reduced

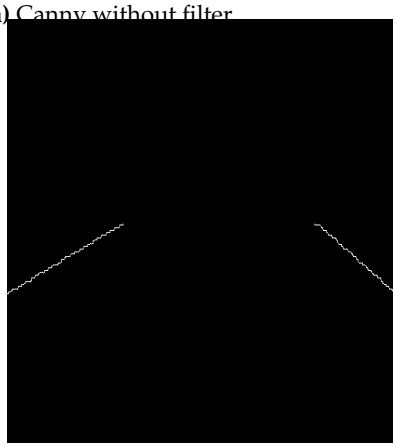
Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1



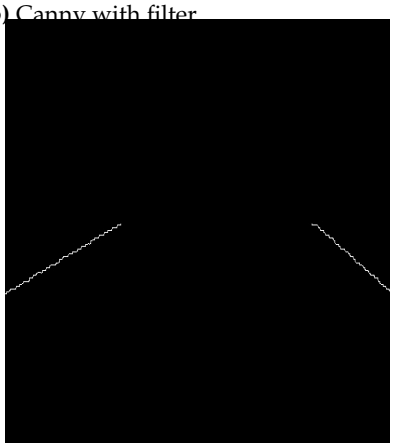
(a) Canny without filter.



(b) Canny with filter.



(c) Cropped image without filter.



(d) Cropped image with filter.

Figure 12. Testing of different Gaussian filter values.

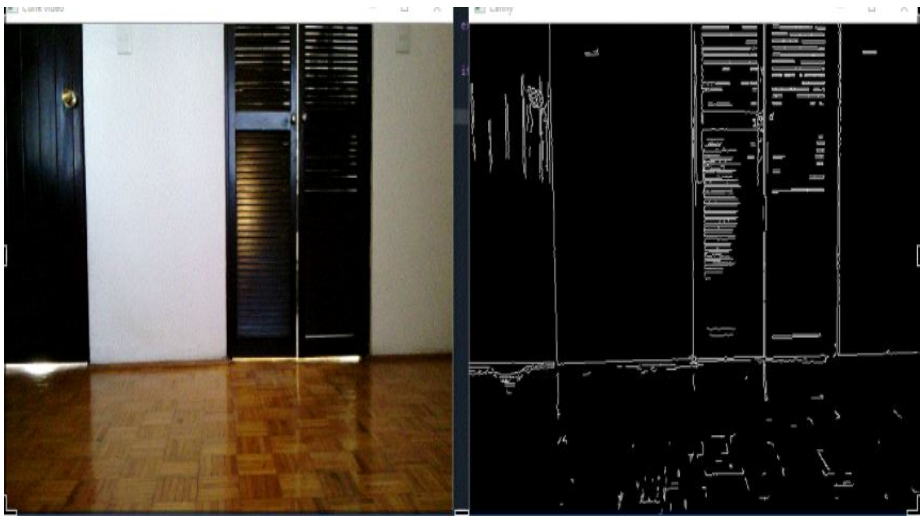


Figure 13. The Canny algorithm deployed on a real-world environment without Gaussian filtering.

performance on detecting corners of the rendered image, and thus, in a real situation with smoother images, those characteristics would not be present. A comparative of both algorithms implemented in the simulation environment is shown in Figure 14 and Figure 15, we can see line detection seems to be more precise. Also, we did not want to limit the functionality of the system to a maze with corners, as we expect that this project can be implemented in different tasks and scenarios. For those reasons, this option was

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

discarded and wee worked with lines detection using probabilistic Hough transform, as mentioned in Section 3.2.

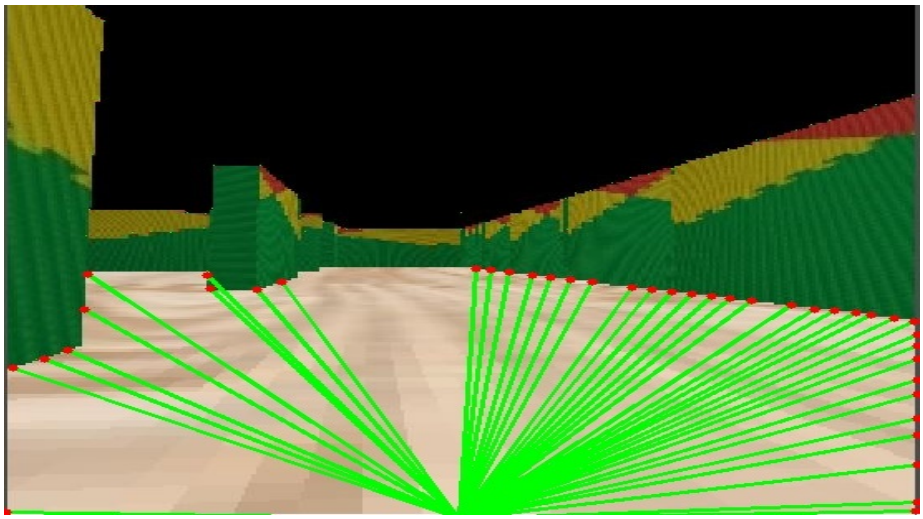


Figure 14. Corner detection algorithm performance.

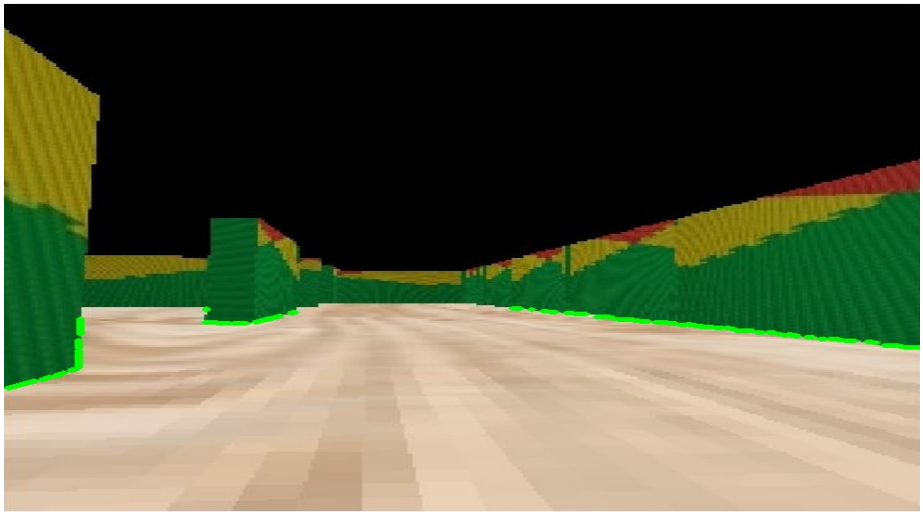


Figure 15. Line detection algorithm performance.

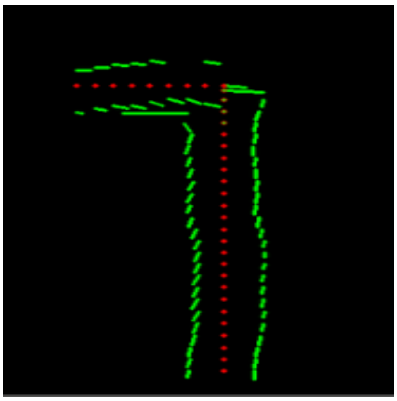
For the sketching part, many tests were carried out and therefore, as considered this to be one of the most challenging stages. Firstly, as mentioned in Section 3.2 we had to stop mapping during turns because the sequence of the hallways was lost at these points. Figure 16a shows the precision of the sketching before the decision was taken; whereas Figure 16b shows the improvement achieved by changing the condition of the adjustment for the new center, and stopped mapping during the laps.

Finally, it was found that, when a wall was not being detected, the distance was computed from the center to the first pixel of the image, which caused again useless line traces. This gave us yet another reason to keep using the flags which indicates the presence or absence of walls. Figure 17 shows some further visual errors found.

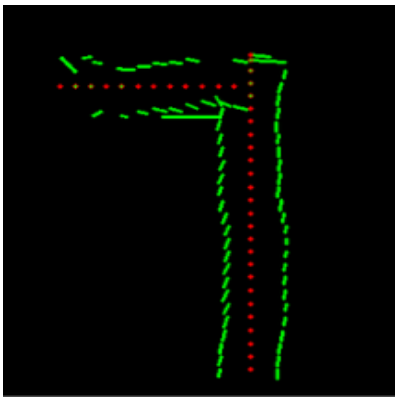
4.3. Path planning

When running the Dijkstra algorithm, a circle search is initiated as shown in Figure 18, It can be seen that the maze navigator covers all possible areas of the maze. The color intensity of the each pixel creates circles, depending on the cost of that point to the starting point. The colors of the circles repeat given the limitations of the available colors; therefore, this does not mean that such pixels have the same cost.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

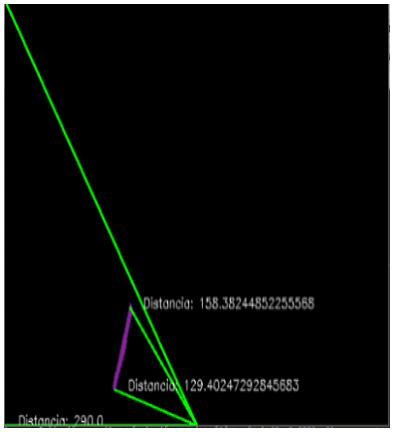


(a) First experiment.

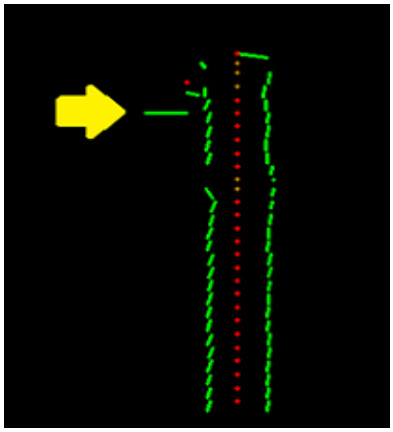


(b) Experiments after mapping correction.

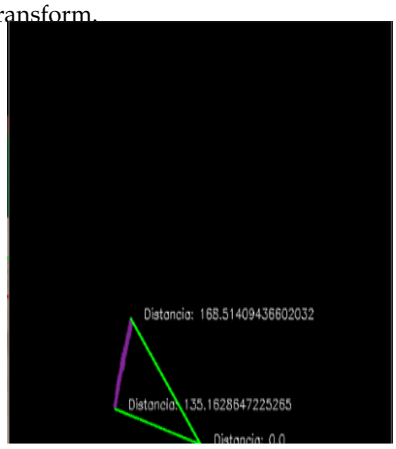
Figure 16. Corners mapping.



(a) Before wall detection flags. Perspective Transform.



(b) Before wall detection flags. Maze map.



(c) After wall detection flags. Perspective Transform.

(d) After wall detection flags. Maze map.

Figure 17. Wall detection correction.

Conversely, when we applied the Trémaux algorithm on the original maze, some loops are obtained in the trajectories, as shown in Figure 19.

Thus, we adjusted the map that means we left each row and wall with the size of one pixel, which resulted in a bigger difference as shown in Figure 20. Notice how the route of this maze seems more natural than the previous one. Most importantly, the robot does not have to walk everywhere in the maze, because the walls and paths are better defined. The intensity of each pixel represents how many times the robot has visited that node of the maze.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

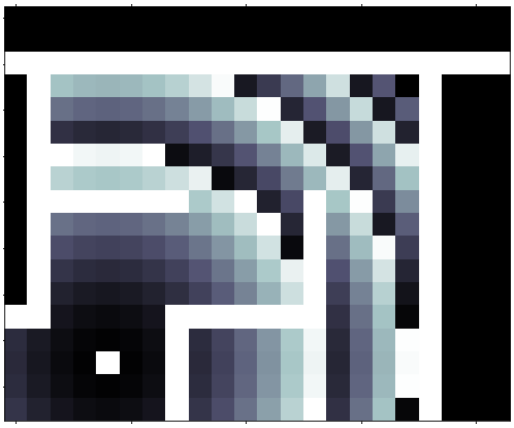


Figure 18. Circular pattern of the Dijkstra algorithm.

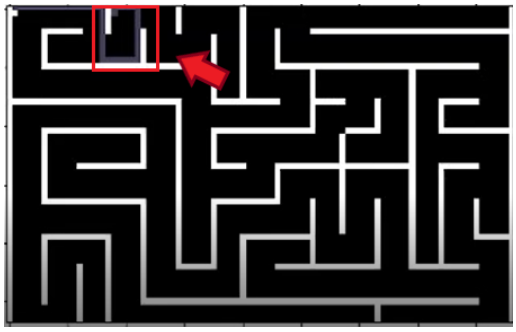


Figure 19. The Trémaux algorithm running in the original maze.

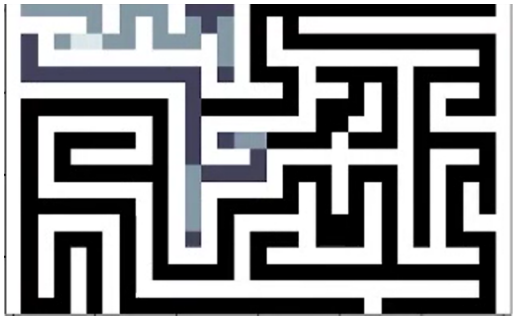


Figure 20. The Trémaux algorithm running in the adjustment maze.

4.4. Communication

Libraries that only work with Python 3.6 or higher were used in the camera calibration, mapping, and path planning modules, so the programming codes could not be executed locally, as the NAO is only programmable with some sub-version of Python 2.7. This would cause a delay in the development, because modules would have to be reprogrammed with some sub-version of Python 2.7, and even some libraries would not work due to incompatibility, for example, OpenCV. The solution to this paradigm was to run two Python scripts, one compiling with Python 2.7.17 and the other with Python 3.7.7. We opted to work using a master-slave architecture, where the computer is the master and the NAO robot (called Curie) is the slave, as shown in Figure 21.

Curie receives a script in Python 2.7.17 attending two tasks: (1) acquisition and storage of an image from the robot's upper camera, and (2) its control i.e. moving or lifting the arms. In parallel, a Python 3.7.7 script was executed in another terminal on the same computer, which retrieved the stored images, processing them with one of the computational vision algorithms i.e. Canny. This approach solved the Python library compatibility issue.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

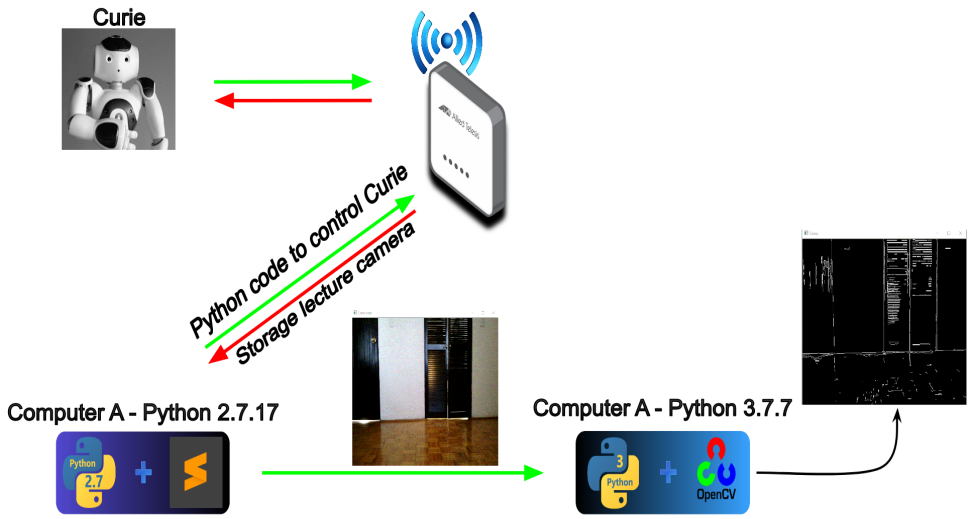
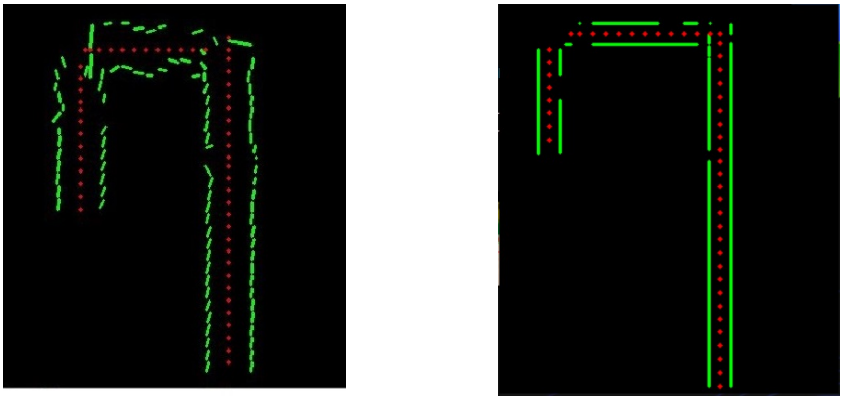


Figure 21. Communication module diagram.

4.5. Integration

Once all previous algorithms were selected and optimized, the integration of the four modules into a single system took place. Prior to this, it was decided to merge the mapping and path planning modules and validate them in the simulation environment. To do so, we had to consider as priority to draw the map lines completely straight (as in Figure 22b, in contrast to Figure 22a); always respecting the calculated distances discussed above. It was found that, by implementing this small change, the resulting algorithm was able to make more accurate decisions.



(a) Map before adjustment. **(b)** Adjusted map.

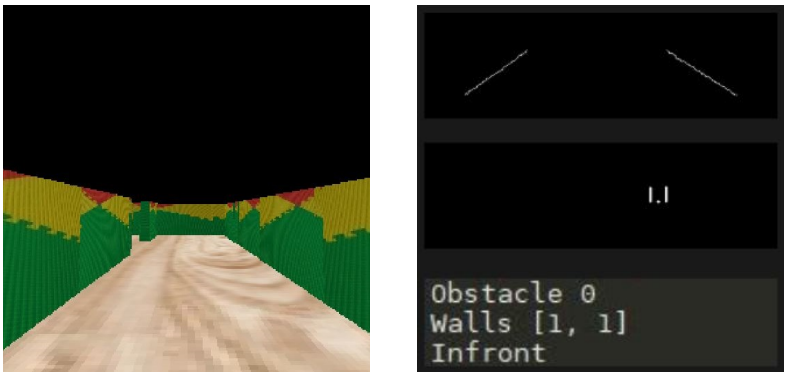
Figure 22. Sketch adjustment.

Figure **??** shows an example of these modules working together. It can be observed that, when using an image of a clear hallway as the one depicted in Figure **??**, the robot detects no obstacle and moves forward. Figure **??** illustrates the sketch of the stretch seen by the robot and the output obtained on the console, confirming that the system is working correctly.

Figure 24 presents another scenario where it can be seen that the robot is recognizing a corner as an obstacle, while at same time identifying that there is no wall to the left. As a result, the robot decides that the next action should be to turn left.

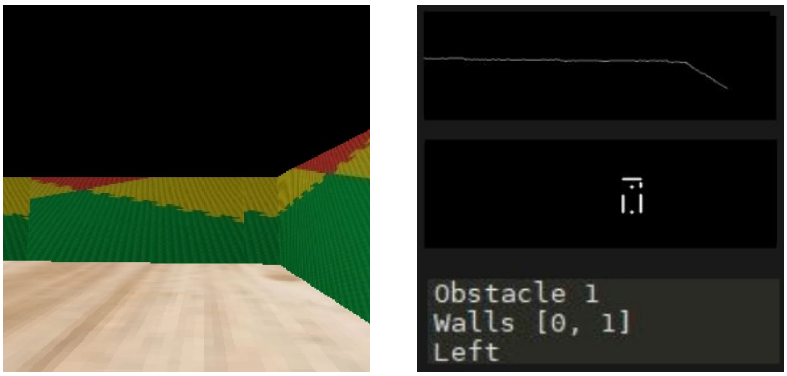
Once the integration of the path planning and mapping modules was achieved, the camera calibration was embedded into the NAO robot. Figure 25 shows samples of the camera calibration, which was evaluated with 24 photographs captured by Curie, where the center of the image is: $(160px, 120px)$.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1



(a) What the robot sees. **(b)** Algorithms output

Figure 23. Mapping and path planning integration. First demo.



(a) What the robot sees. **(b)** Algorithms output

Figure 24. Mapping and path planning integration. Second demo.

Figure 26a and Figure 26b show a photograph captured by Curie and the calibrated camera respectively. In general, it can be observed that the photograph has deformations, both on the ROI (i.e. the chessboard) and the corners. The effects of these image deformations in the performance of the system will be detailed in the Section 5.

A third integration took place during this phase, again with the path planning and mapping modules. This time, an adjustment had to be made to these modules to change the path from where it was retrieving the images. This means that images from the simulation were no longer in use, in favor of those taken by Curie. As all the modules were already integrated, a small labyrinth hall was built with foamy blocks and a yoga mat was placed as a floor, as shown in Figure 27, so that there was a greater contrast between the walls and the floor.

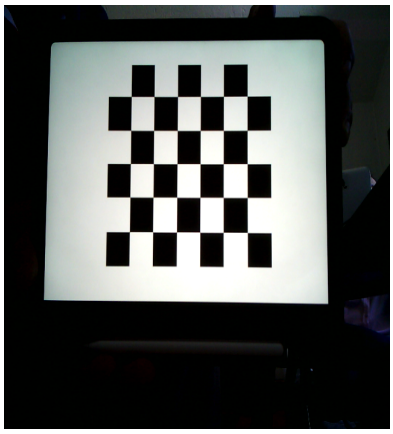
Figure 28a shows a photograph captured by Curie with the path planning, mapping and communication modules and on the right side the image captured by adding the calibration module. In contrast, Figure 28b shows how the image looked after the integration of all modules. Notice the distortion on the bottom of the image; this issue will be discussed in the next section.

5. Analysis of results and discussion

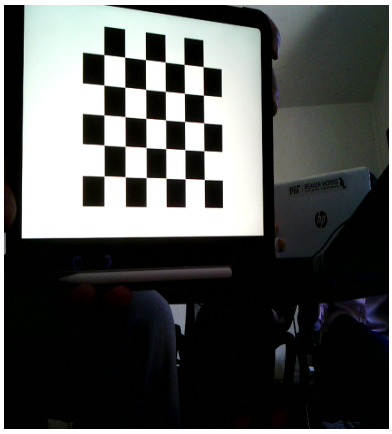
5.1. Camera calibration

As mentioned before, the calibration module was evaluated with 24 photographs captured by Curie, where the center of the image is: $(160px, 120px)$ and the results obtained are shown in Table 2. It was observed that with 21 photographs, the lowest percentage of error was obtained when the center of the image was located with errors of 3.67% for the x axis and 5.6% for the y axis. As the number of photographs increased, so did the error, as evidenced by the results of using 24 photographs, where the errors obtained were 12.38% and 22.91% for the x and y coordinates respectively. Notice that the pho-

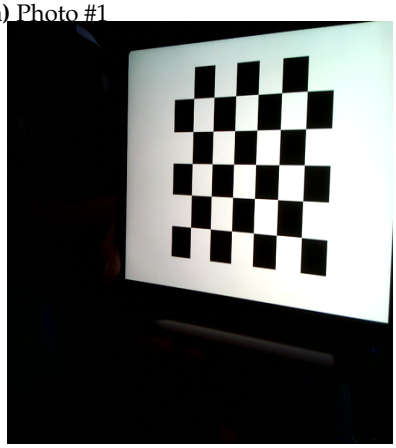
Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1



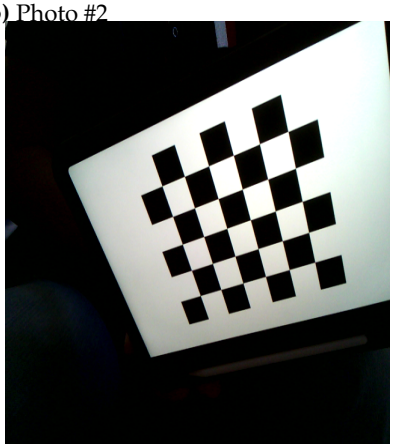
(a) Photo #1



(b) Photo #2

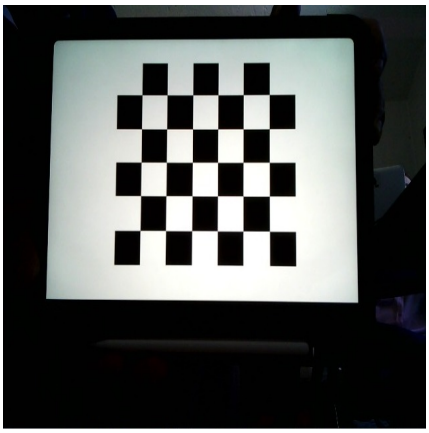


(c) Photo #13

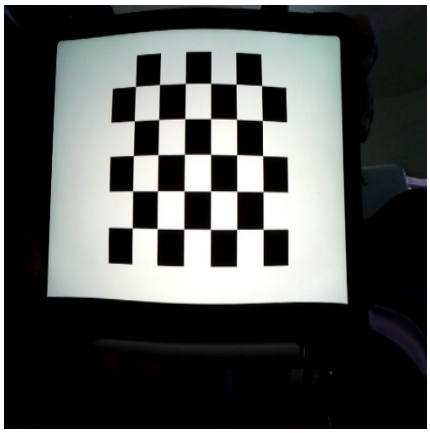


(d) Photo #18

Figure 25. Photos used for the calibration module.



(a) Photo original captured by Curie



(b) Photo after applying Calibration Module

Figure 26. Photos used for Calibration Module.

tographs presented considerable distortions as shown in Figure 28, which affected the performance of the mapping and path planning modules considerably, given that the edges of the hall could not be correctly detected. This led to a higher contrast with the wall, even when using the yoga mat as the floor surface. We experimented removing the calibration module so that the mapping and path planning modules would directly process the photograph captured by Curie. Interestingly, this benefited the performance of the entire system, obtaining more acceptable performance. We suspect that this is

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

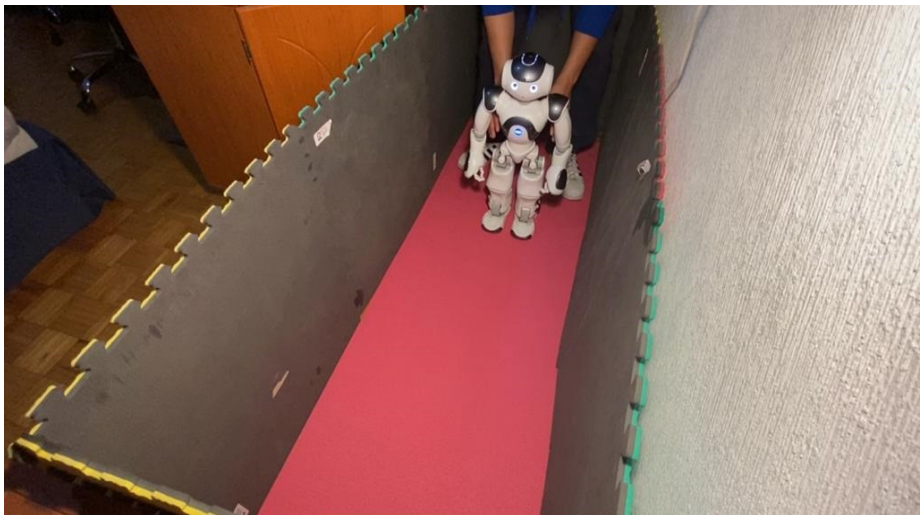


Figure 27. Maze hall with Curie

(a) Photo original captured by Curie in the maze.

(b) Photo captured by Curie after the integration.

Figure 28. Photos used for calibration module.

due to the fact that the NAO robot already contains a calibration module embedded, and thus, this step is not required in this architecture. Nonetheless, it is important to realize that in other applications such as autonomous vehicles [19], this module must be considered.

Images	X	% Error X	Y	% Error Y
3	75.725	0.52671875	5.43	0.95475
6	92.115	0.42428125	14.55	0.87875
9	102.7	0.358125	24.59	0.795083333
12	117.375	0.26640625	39.815	0.668208333
15	128.21	0.1986875	62.8	0.476666667
18	148.915	0.06928125	82.9	0.309166667
21	154.125	0.03671875	113.18	0.056833333
24	179.81	0.1238125	147.5	0.229166667

Table 2: Comparative table between the centers of the real image retrieved from Curie and those of the matrix (best values highlighted in green).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

5.2. Mapping

As mentioned in previous sections, three fundamental solutions had to be implemented in order to improve mapping. First, we had to stop mapping while orientation changes occurred to prevent the disruption of the sketch. Second, wall detection flags were required with two main purposes: (1) to avoid the computing of distances when a wall was not being detected, and; (2) to re enable the sketching after a turn took place. Finally, also in orientation changes, the center had to be adjusted to avoid overlapping the corner walls. The final result can be observed in Figure 29.

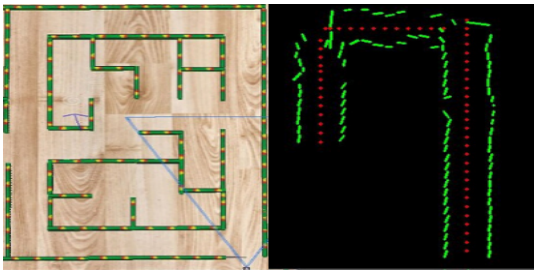


Figure 29. Sketched maze.

5.3. Path planning

We discarded the Dijkstra algorithm because even if it is able to find the best path, there is still a need to traverse the whole maze, which is not a scalable solution, despite being an algorithm that does not need to know the endpoint and always finds an option for each position. Furthermore, It would be counterproductive to implement this solution in a robot, because it would force it to return to the beginning every time that the optimal distance has to be calculated. Therefore, the Trémaux algorithm, which demonstrated to be the fastest and most efficient method, was embedded into the model. Figure 30 shows that, even when the robot get into a dead end (as some situations depicted with gray pixels), it is also able to keep looking for the optimal path to the exit (white line).

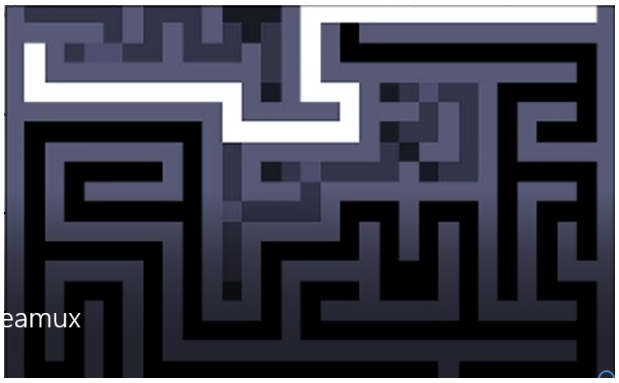


Figure 30. Trémaux algorithm: Path to the endpoint.

5.4. Communication

In this module, we encountered no problems regarding loss of connection between Curie and the computer, or high latency in transmitting the flag generated by the mapping and path planning modules to Curie. Table 3 shows the possible values of the flag based on the analysis carried out by the path planning and mapping modules, and Table shows the latency in transmitting this value to Curie, being on average 333.12ms. The path planning and mapping modules take approximately 10 seconds to perform all the processing of the photograph captured by Curie to throw the previously mentioned flag. Efforts were put to optimize the programming codes for both modules, unfortunately the time could not reduced.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

Movement to execute Curie	Flag value
Walk	1
Turn left	2
Turn right	3
Stop	4

Table 3: Equivalence of flag values and movement to be executed (Curie).

	Latency to send flag
	0.35641644
	0.31901984
	0.30859224
	0.39965194
	0.33136234
	0.37186354
	0.33310064
	0.33050654
	0.24758604
Average (seconds)	0.33312217

Table 4: Latency (in seconds) to send the movement flag from the computer to Curie.

5.5. Modules Integration

We ran the three following tests for our integrated system: (1) with a straight path as shown in Figure 31; (2) with a turn to the left as in Figure 32 and; (3) with a turn to the right as in Figure 33. All of these images show the robot traversing the path and how the maze is being sketch. In all three cases, the robot detected the walls and corners of the maze, which allowed it to follow the correct instruction. Also with our experimental results, we calculated that the time response between each robot action was about 10 seconds.

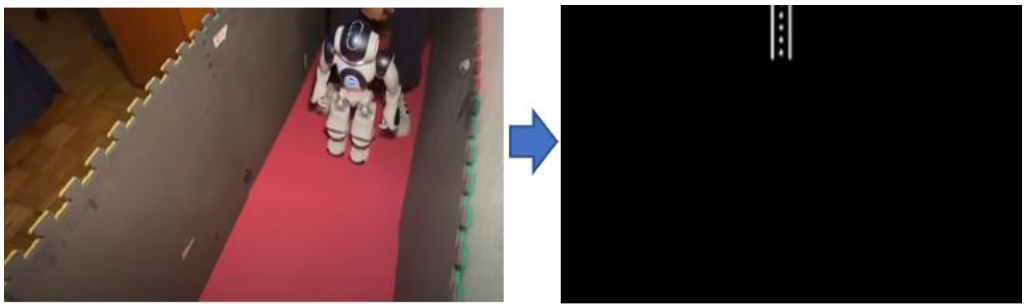


Figure 31. Test: straight path.

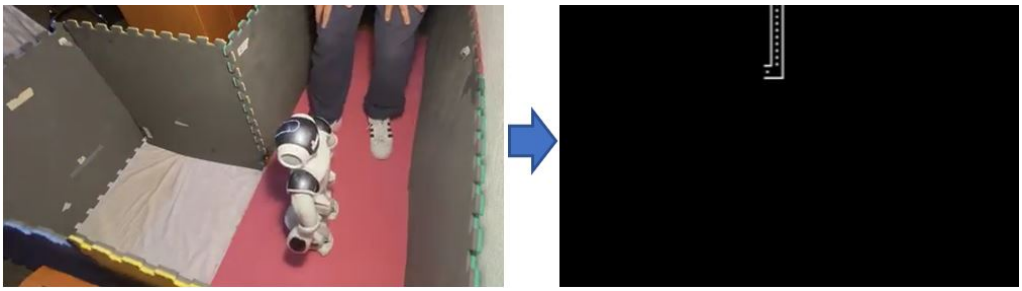


Figure 32. Test: left turn.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 June 2021 doi:10.20944/preprints202106.0037.v1

Figure 33. Test: right turn.

6. Conclusion

In this paper, we presented our latest findings towards implementing a NAO robot maze navigation solution based on different algorithms, which range from computer vision, camera calibration, path planning, mapping, and communication protocols. With this, we aim at deploying a robot that is not only capable of solving a maze, but to share that knowledge with a second robot which will be also able to solve the maze considering its own mechanical characteristics. Several algorithms were tested and validated for each stage, showing the viability of this project.

The calibration module needs a second check, because at the moment, it is actually affecting the performance of the path planning and mapping instead of improve it. Nevertheless, base on the results of the test, we can infer that the NAO cameras are already sufficiently calibrated for the purpose of this task. Conversely, path planning, mapping and communication work really well together, but could be enhance by improving the response time of the robot to make it more continuous. Finally, we realized that uploading and retrieving information from Ubidots was what was causing the biggest delay on the performance. Once fixed, the NAO robot was capable of follow instructions based on its environment.

In future work, other path planning algorithms will be tested to further improve the time to solve the maze. We also need to do more tests with different maze structures and surfaces to see how the performance can be enhanced. Finally, at the communication stage, we will implement the aspect of sharing the generated knowledge with another robot, to take advantage of first robot's experience.

7. Acknowledgments

The authors would like to acknowledge the financial and technical support of Writing Lab, TecLabs, Tecnologico de Monterrey, Mexico, for the support of this work.

8. Bibliography

1. Horn, B.K.P. *Robot vision*; MIT electrical engineering and computer science series, MIT Press, 1986.
2. Cortés, X.; Serratosa, F.; Moreno-García, C.F. Semi-automatic pose estimation of a fleet of robots with embedded stereoscopic cameras. *Emerging Technologies and Factory Automation (ETFA)* **2016**.
3. Rodriguez-Tirado, A.; Magallán-Ramírez, D.; Martínez-Aguilar, J.D.; Moreno-García, C.F.; Balderas, D.; López-Caudana, E.O. A pipeline framework for robot maze navigation using computer vision, path planning and communication protocols. *13th International Conference on Developments in eSystems Engineering (DeSE)* **2020**.
4. Kastne, T.; Rofer, T.; Laue, T. Automatic robot calibration for the NAO. *RoboCup* **2014**, *LNAI 8892*, 233–244.
5. Zhang, Z. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **2000**, *22*, 1330–1334.
6. Singh, A.K.; Nandi, G.C. NAO humanoid robot: Analysis of calibration techniques for robotsketch drawing. *Robotics and Autonomous Systems* **2016**, *79*, 108–121.

7. Li, L.K.; Annaz, F. Implementation of the Trémaux maze solving algorithm to an omnidirectional mobile robot. *13th International Conference on Electronics, Information, and Communication (ICEIC)* **2014**.
8. Kumar, A.; Kumar, P.B.; Parhi, D.R. Intelligent navigation of humanoids in cluttered environments using regression analysis and genetic algorithm. *Arabian Journal for Science and Engineering* **2018**, *43*, 7655–7678.
9. Wikström, R.; Sjögren, M. Amazeobot: The construction of a maze mapping robot. *Inom Examensarbete Teknik, Grundniva (Bachelor Thesis)* **2016**.
10. Simoens, P.; Dragone, M.; Saffiotti, A. The internet of rbotic things: A review of concept, added value and applications. *International Journal of Advanced Robotic Systems* **2018**, *15*, 172988141875942.
11. Bechon, P.; Slotine, J.J. Synchronization and quorum sensing in a swarm of humanoid robots. *arXiv* **2013**.
12. Gul, F.; Rahiman, W.; Alhady, S.S.N. A comprehensive study for robot navigation techniques. *Cogent Engineering* **2019**, *6*.
13. Delfin, J.; Becerra, H.M.; Arechavaleta, G. Humanoid localization and navigation using a visual memory. 2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids), 2016, pp. 725–731.
14. Canny, J. A computational approach to edge detection. *Readings in Computer Vision* **1986**, p. 679–698.
15. Hough, P.V. Method and means for recognizing complex patterns.
16. Matas, J.; Galambos, C.; Kittler, J. Progressive probabilistic Hough transform. 1998.
17. Ably, T.; Pang, H.; Williams, C.; Klim, J.; Ross, E. Dijkstra's shortest path algorithm. *Brilliant.org* **2016**, *Retrieved from:*.
18. Wilkins, J.S.; Nelson, G. Tremaux on species: A theory of alopatric speciation (and punctuated equilibrium) before Wagner. *History Philosophy of the Life Sciences* **2008**, *30*, 179–205.
19. Cortés Gallardo Medina, E.; Velazquez Espitia, V.; Chípuli Silva, D.; Fernández Ruiz de las Cuevas, S.; Palacios Hirata, M.; Zhu Chen, A.; González González, J.; Bustamante-Bello, R.; Moreno-García, C.F. Object Detection, Distributed Cloud Computing and Parallelization Techniques for Autonomous Driving Systems. *Applied Sciences* **2021**, *11*, 2925.

 doi:10.20944/preprints202106.0037.v1

 doi:10.20944/preprints202106.0037.v1