UMONS
Université de Mons

Faculté
des Sciences

Connect4NAO

A NAO robot plays Connect 4

Project realised by: Anthony Rouneau
Section: 1st Bloc Master in Computer Sciences
Academic year: 2015-2016

Under the direction of: Tom Mens, Pierre Hauweele
Department: Computer Sciences – Software Engineering

Contents

Part I
Introduction

This project was developed within the context of the first year of a Master's degree in Computer Sciences at the University of Mons (UMONS). The goal is to make NAO, a humanoid robot (introduced in the figure 1), play the Connect 4 game, detailed in the section 2.2, autonomously against a human opponent.

While this objective has already been achieved in a commercial purpose [6], we would like to bring an open-source approach to the subject. Besides, we are using a bigger board than the classical one in order to be more visual, while the commercial version uses the classical board.

1 Goals

With the modern technological growth, we could imagine that, in the future, interacting with a humanoid robot can be an everyday task. Hence, developing or analysing human-machine interfaces is very interesting to see how a robot can or can not interact with a person.

This is why an open-source solution for this project is great. Each part of the project could be useful to someone else that is developing another interface, and beyond that, there is a real demand for games between humans and robots. For example, the ASK NAO program aims to help children with autism by making them play with NAO. ASK NAO is detailed later in this report.

By making NAO play the Connect 4 game with an Open-Source-license, we can provide this solution to anybody interested in human-machine interfaces, it can be used alongside the ASK NAO solutions or maybe someone could just have fun with our interface.

Such projects can also be motivating for future computer sciences students as it is very fun to program a robot.

2 Material Used

To achieve the goals of this project, we had to use specific materials. These are described below.

2.1 Humanoid robot NAO

The NAO robot, engineered by the french company Aldebaran, is a humanoid robot conceived to move, react to and interact with its environment to resemble a human being. The robot can speak, listen and follow sounds, recognize objects, walk, grab things, etc... It uses various sensors to do such things: sonars, bumpers, cameras, microphones and tactile sensors.

The development of NAO has started in 2006, hence, there exists multiple versions of body and head[a]. A NAO robot consisting of a H25 body and a V5 head was used, these are the latest body and head available as this report is written. The H25 body has the particularity of having prehensile hands, which can be used to grab things.

As any humanoid robot, NAO is interesting to researchers in multiple field of study. One can study a human-machine interface, another can investigate into complex movements or even help children with autism to interact with their environment. But beyond that, NAO and its software is designed so that anybody willing to use the robot can control it without a specific knowledge. We will detail some points later in this report.

Figure 1: The NAO Robot morphology is similar to the human morphology.

[a]For more details, see Aldebaran's doc : `http://doc.aldebaran.com/2-1/family/body_type.html`

2.2 Connect 4

The well-known board game, Connect 4, consists of a vertically suspended rectangular grid of circles with six lines and seven columns. A slot is located on the top of each column to receive coloured discs dropped by the players.

The board used during the development of this project is of light wood and it has red discs and green discs. These colors are different enough to be distinguished from one another. The dimensions of the board are bigger than the traditional ones; it is 51.9cm long, 33.7cm high and 6.6cm wide. This big game board was chosen in order to be more visual and to be sure that the robot can interact with the discs. Indeed, if they are too small, the space between NAO's fingers could be too large to hold them.

When the game starts, the players choose a color and then take turns dropping the discs into the board through the slot of his choice. The disk falls and places itself on the lowest position available in the column. The winner is the player that manages to align four pieces of his own colour in a row (horizontally, vertically or diagonally).

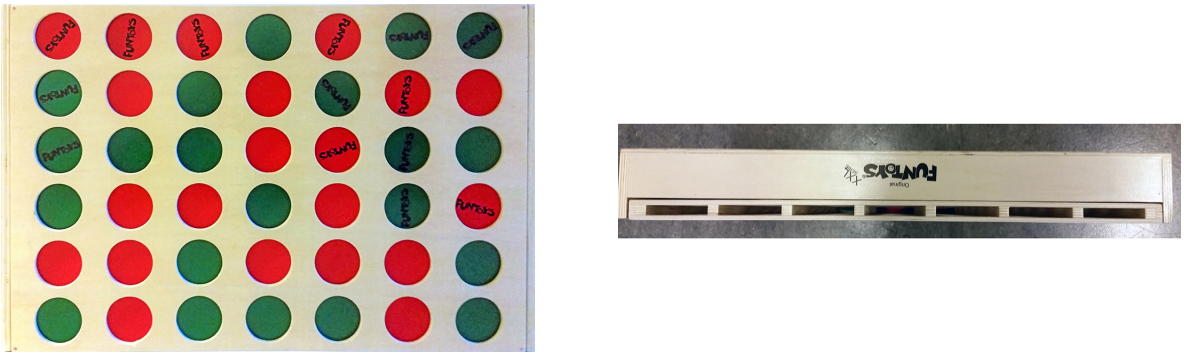


Figure 2: The Connect 4 used for the project consists of a wooden board, red discs and green discs.

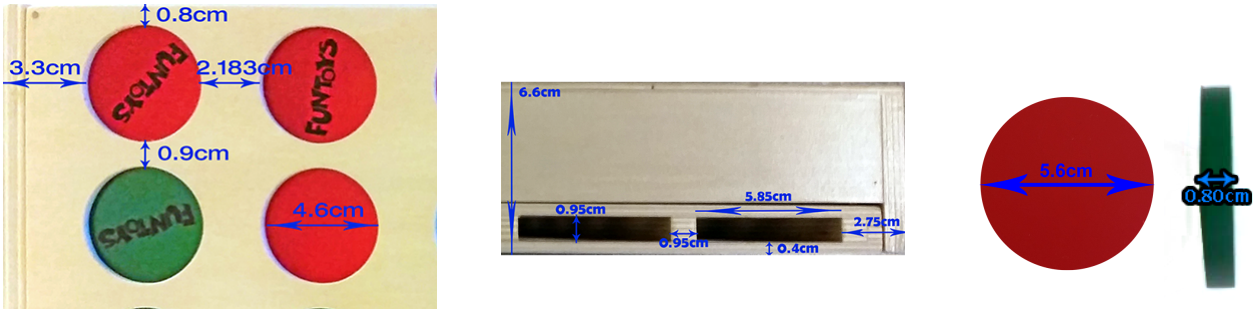


Figure 3: The elements of this Connect 4 are bigger than the usual.

3 Software used

The software used to achieve the goals of our project are described below.

3.1 Controlling the robot

The languages natively supported by NAO are `C++03` and `Python 2.7`. We chose `Python 2.7` because it is very easy to deploy a Python script on the robot, it allows to achieve prototypes very fast and we were more comfortable with the `Python 2.7` syntax rather than with the `C++` syntax. Finally, `Python 3` is not currently supported by NAO.

As well as being easier to deploy, `Python 2.7` will not slow the calculations. Indeed, the main task of NAO for this project is to carry out image processing through OpenCV, and, as mentioned, in the section 3.2 about OpenCV for Python, the calculations are made through compiled C++ and, as it is showed in [21], the speed loss when using `Python 2.7` is negligible.

Using `Python 2.7`, the NAO robot can be controlled through the `NAOqi for Python 2.7 SDK V2.1`, developed by Aldebaran.

3.2 Computer vision library

To play the Connect 4 game autonomously, the robot must be able to see the game board to know where to put his disc and to analyse the current game state to choose its next action.

The robot can look around by taking pictures, but it can not use what it sees unless an image analysis is performed, using image processing algorithms. This is why we need libraries that provide these algorithms, such as CCV, OpenCV, SimpleCV, VXL, ...

OpenCV is a computer vision library released under a BSD open-source licence. We chose this library because it is integrated in NAO's system, a large support is available online, and because it has an implementation for `Python 2.7`.

Using this library and given an image or a video stream, we can recognize objects, detect circles, isolate a specific colour and much more. The library provides algorithms that aim to get basic, usable information from pictures, but what remains to be done is to use this information to achieve the goals of our project.

We used `OpenCV 3.0.0` for `Python 2.7` which is a binding of the `C++` implementation. We will explain further how we used OpenCV to detect the Connect 4 board and to target the slots in which NAO will drop its discs.

3.3 Dependencies

`Python 2.7` is required to launch the solution. The list below contains all the `Python 2.7` libraries used in this project.

Library	Utility
`OpenCV 3.0`	Provides the computer vision algorithms
`Numpy 1.8.1`	Required by `OpenCV` for the basic data structures
`Scipy 0.16.1`	Used for the K-Dimensional Tree structure used in section 3.3 of the part 3 because it is compatible with `Numpy`'s data structures.
`Hampy 1.4.1`	Used to detect the Hamming markers of section 5.2
`Docopt 0.6.2`	Used to parse the arguments of the command line interface.

The `Hampy` library was chosen because of its popularity for its functionalities on the pypi[1] package repository. The `Docopt` library was preferred to other command line parsers because it allows to create prototypes really fast.

3.4 Source code

The source code of this project is available on GitHub at the following link:

`https://github.com/Angeall/pyConnect4NAO`

To launch the application, please install the requirements listed above. The links to install these requirements are available in the README file and on the GitHub page of the project.

Once the requirements are installed, launch a terminal emulator in the `src` repository of the project. The command line interface allow the user to launch each part of the project, detailed in figure 5, separately, excepted the steps 4(b) and 7, that are only available in the whole prototype of the project that allows NAO to play autonomously.

The following command lists and explains each command available in the software, as illustrated in the figure 4.

```
python connect4nao.py --help
```

```
[redacted]\pyConnect4NAO\src>python connect4nao.py --help
Welcome to Connect4NAO.

Usage:
  connect4nao.py [--help] <command> [-h | --help | <args>...]

Options:
  -h --help         Show this screen.
  --version         Show the version.

Commands:
   game          Launch a virtual Connect 4 game (without the robot)
   board         Tries to detect the game board in front of the robot
   markers       Tries to detect Hamming (7, 4) markers in front of the robot
   state         Analyse the game state of the board in front of NAO
   coordinates   Computes the 3D coordinates of a upper hole
   ik            Make NAO grab and drop a disc in the given hole
   play          (Prototype) play the Connect 4 autonomously
```

Figure 4: The command line interface includes multiple commands

For each command, a specific help section is available. One can display it by entering the following command, replacing `command` by the targeted command.

[1] https://pypi.python.org/pypi/

```
python connect4nao.py <command> --help
```

4 Essential steps

In order to play the Connect 4 game, NAO must follow a sequence of steps. NAO must:

1. **Find the game board** in the room.
2. **Analyse the current game state**.
   (a) Wait until the other player has played.
   (b) Detect if the other player has cheated.
3. **Decide which action is the best**.
4. **Walk towards the board** to be in position to play a disc.
   (a) Detect the 3D coordinates of the board.
   (b) Walk to these coordinates.
5. **Grab a disc**.
6. **Drop the disc** into the hole that was chosen at step 3.
   (a) Detect the 3D coordinates of that hole.
   (b) Place the hand in the good position at these coordinates.
7. **Walk back** far enough to see the entire game board.
8. **Repeat from step 2** until an end is reached.

Figure 5: The essential steps for NAO to play the game

We will see later in this report how each step can be achieved.

Part II
State Of The Art

The state of the art concerning our project can be seen below. Two main subjects can be distinguished: the Connect 4 game and the NAO robot.

1 Connect 4

Some works related to the Connect 4 game are listed below.

1.1 Strategy

In October 1988, Victor Allis introduced a Connect 4 strategy in his master's thesis, VICTOR[1].

There is many strategic moves and rules[2] that can be applied during a Connect 4 game. Victor Allis proved that for each game situation, there is a move that is better than all the others. The purpose of VICTOR is to bind each game situation with its perfect move. Allis also proved that his strategy is unbeatable, and that in the worst case, the game ends in a draw.

Hence we know it is possible to develop an unbeatable artificial intelligence to play Connect 4. If the opponent uses the VICTOR strategy, there is no way to beat him. But if this strategy is used by both players, the game will end up in a draw.

In the context of our project, we can consider some levels of difficulty to play against the robot, the highest level being the VICTOR strategy.

1.2 NAO plays Connect 4

As mentioned in the introduction, the goal of our project has already been achieved. It is the french company *HumaRobotics* that developed *NAO plays Connect 4* [6] to prove their skills in human-robot interfaces. Using it, NAO can play Connect 4 against a human opponent while also simulating human feelings. For example, the robot can pretend to be angry if its opponent cheats or it can pretend to be joyful if it is winning. It is also different from our project by the game board used; they use a classic Connect 4 board whereas we are using a significantly bigger board, see figure 3.

But the code is not Open-Source. This is why we plan to achieve a similar goal by providing an Open-Source solution.

2 NAO and games

NAO has been programmed to play many kinds of games. Here are some examples:

[2] They are detailed in [1]

2.1 NAO plays *Tic-Tac-Toe*

As a proof of concept for a study on inverse kinematics for NAO [19], the researchers Renzo Poddighe and Nico Roos made NAO play *Tic-Tac-Toe*.

Inverse kinematics are kinematic equations used to determine the positions and rotations of joints of a robot/avatar to achieve a certain task.

Let us take NAO as model to explain a little more inverse kinematics. If we know the 3D coordinates of NAO's right hand and the 3D coordinates of the Connect 4 board, inverse kinematics give us a way to determine the new position and rotation of each joint of NAO in order to get his hand over the board.
The use of inverse kinematics will be obviously considered to achieve the goal of our project.

2.2 ASK NAO

The NAO developers made a framework based on the interplay between children with autism and NAO. This framework is named `ASK NAO` [20]: *Autism Solutions for Kids*. `ASK NAO` gathers NAO behaviours which bring children to interact with their environment through the robot's actions.

An example of such a behaviour is a game where the robot asks a child to show a designated animal. The child must pick a drawing between those on the floor in front of him and show it to the robot. NAO congratulates him or encourages him to try again depending on whether the child has picked the right drawing or not.

There are many more behaviours in this framework as dance learning, communication learning, ...[3]

Many studies, including [2], have evaluated this learning method for children with autism, and the results are very encouraging. Children tend to play with each other more spontaneously when the robot is present in their environment. To be part of such solutions can be a motivation to achieve our project.

2.3 NAO plays Poker

The previously mentioned enterprise, HumaRobotics, has developed a NAO behaviour [7] that allows it to play Poker. Unfortunately, it is once again proprietary-licensed.

[3] A video illustrating some behaviours included in `ASK NAO` is available here:
`https://www.youtube.com/watch?t=236&v=lm3vE7YFsGM`.

Part III
Development

In this section, the techniques developed to achieve the goals of our project will be introduced. The development includes the steps, listed in figure 5, that led to NAO playing the Connect 4 game.

1 Robot vision and cameras

The steps 1, 2, 4(a) and 6(a) of the figure 5 are entirely based on the robot vision. While a human player can easily do these things, for a robot, his vision consists only of camera pictures, which are matrices of pixels that must be processed so that they can be made understandable. Hence we will need to use `OpenCV`, the computer vision library previously chosen, to handle what it sees and to make it play.

1.1 Devices

NAO has two cameras, shown on the figure 6. The robot can be connected by an Ethernet cable or by a `WiFi g` connection. The `WiFi g` is slower than a Gigabit Ethernet or even a 100Mbps Ethernet. As a result, the number of frames per second that the robot can send to the computer will depend on the chosen medium. Below are the frame rates of NAO's cameras as a function of the resolution of the picture and the medium.

Resolution/Medium	Gigabit Ethernet	100Mbps Ethernet	WiFi g
40x30	30fps	30fps	30fps
80x60	30fps	30fps	30fps
160x120	30fps	30fps	30fps
320x240	30fps	30fps	11fps
640x480	30fps	12fps	2.5fps
1280x960	10fps	3fps	0.5fps

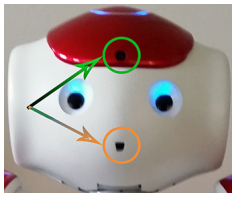


Figure 6: Cameras of the NAO robot

During the development, we chose the 320x240 resolution, which is the higher resolution that allows a reasonable frame rate of 11fps when using the `WiFi g`. Hence, our solution is working for resolutions $\geq$ 320x240, but if the frame rate is too slow, the robot can be very slow to play. So, in order to use a big resolution, one should consider to connect the robot with an

Ethernet cable.

Another technique would be to override NAO's default camera methods to compress the pictures before sending them which could improve the frame rate. As the resolution and frame rate mentioned above were high enough for the image processing of this project, we did not develop this technique.

1.2 Camera calibration

To estimate the 3D coordinates of the game board and its holes from a 2D image, we need the camera's intrinsic parameters. They will be used to transform the pixels into coordinates of known objects, modelled using their measures. The parameters needed are the following:

$$The\ camera\ matrix\ :\ \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

$$The\ camera\ distortion\ coefficients\ :\ [k_1, k_2, p_1, p_2, k_3]$$

Where :
f_x, f_y are focal lengths in pixel units[4].

(c_x, c_y) is the principal point in pixel units[4],
which would ideally be at the centre of the image.

k_1, k_2 and k_3 are radial distortion coefficients.

p_1, p_2 are tangential distortion coefficients.

[4]It means that these values have to be scaled accordingly to the resolution used.

1.2.1 Calibration using a 2D object

The objective of such a calibration is to be able to map 2D coordinates into 3D coordinates. To estimate the needed parameters, we will map pixels with 3D coordinates of a theoretical model of our object. Hence, we need to detect the modelled object in the image.

`OpenCV` provides methods to easily detect a chessboard pattern in an image. Hence, will use one to calibrate the camera.

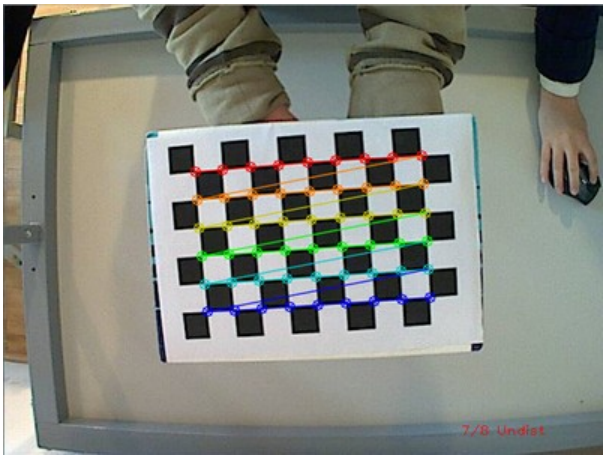


Figure 7: The chessboard can be easily detected by `OpenCV`.
©OpenCV docs

This method is quite simple as the chessboard can be detected in any orientation. Indeed, the algorithm can detect a rectangular chessboard from the top to the bottom, from left to right, or from the bottom to the top, from left to right. In both case, it will not affect the mapping because it is used to detect the perspective and the distances in pixels between the squares of the chessboard, and these do not change with the orientation of the chessboard.

To perform a good calibration, the chessboard must be rotated and moved in front of the camera and multiple mappings must be retained. They will be all used to approximate the intrinsic parameters. These movements are used to estimate the different parameters for multiple cases (sloped to the right, close to the camera, far from the camera, ...).

The results for this calibration were not accurate enough to play a disc into the game board. We performed multiple tests where we tried to put NAO's hand on top of a hole, detecting its 3D coordinates using the technique explained in the section 5.2. The results of the tests were represented on the figure 8, where the radius of the circle is the mean of the errors between the expected position (the white circle) and the actual position of the hand. The mean error was 1.43cm.

Figure 8: The mean error of the calibration using a 2D object.

1.2.2 Calibration using a 3D object

As the results obtained with the 2D calibration were not accurate enough, we tried another technique introduced in [24], that uses a 3D object to calibrate the camera.

We needed a 3D object, easy to detect and not too hard to model to perform the calibration. The same way that we did for the previous calibration, we will use a chessboard as the object to detect. But this time, we will use three perpendicular chessboards of different colours to build a 3D object, which is shown on figure 9.

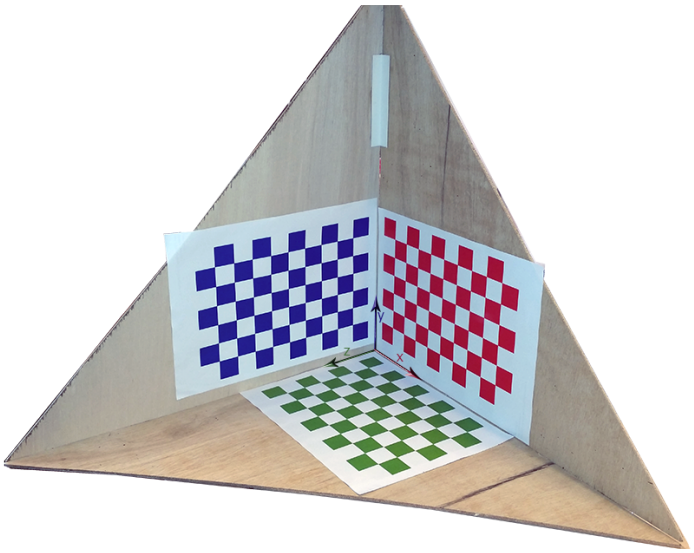


Figure 9: The calibration using 3D objects gives better results
than the calibration using 2D objects.

As the `OpenCV` method can detect a single chessboard per image, we will apply color filters on the image to detect each chessboard, one after another. But this time, the orientation of the detected chessboards is important, as the model depends on the three chessboards and relies on their orientation. Indeed, if one chessboard is detected upside down and the two others are detected correctly, we cannot match the model as it is, because the origin of the model is the common corner of the three chessboards. The chessboard detected upside down would have its common corner at the opposite of the common corner of the real one. Hence, if one chessboard

is detected upside down, we must rectify it.

We can detect when a chessboard must be rectified by comparing the distances between the chessboards. Indeed, a square from the bottom line of the blue chessboard will always be closer to any square of the green chessboard than a blue square from the upper line. We can apply this principle with any pair of chessboards, even if the second chessboard is upside down itself.

The mean error was measured the same way than for the previous calibration, and the results were better. We have now a mean error of 0.51cm. We will see in the section 6 how we can tolerate this error.

Figure 10: The mean error of the calibration using a 3D object.

1.3 Head movement

If the game board is not in NAO's field of vision, it will not be able to detect the board. Hence, we must discuss on the possible techniques to make NAO look at the game board.

As the distance between the board and NAO is unknown, multiple distances must be considered. We will see in the section 3 that it is not necessary to know exactly that distance, and that an approximation of this distance can be used. In practice, we will try to detect the game board, using the technique in the section 3 at multiple distances until the board is detected.

Once the distance is known, or approximated, we must make NAO look to the correct position. In fact, the board is a little lower than the robot, hence the head of NAO must be inclined. The value of this inclination can be determined as a function of the distance.

First, we will introduce the Al-Kashi theorem, also called the law of cosines, to use it later. The Al-Kashi theorem sets a relation between the sides of a triangle and its angles. For the triangle given in the figure 11, these relations are:

$$
\begin{aligned}
a^2 &= b^2 + c^2 - 2 \cdot b \cdot c \cdot \cos\alpha \\
b^2 &= a^2 + c^2 - 2 \cdot a \cdot c \cdot \cos\beta \\
c^2 &= a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos\gamma
\end{aligned}
$$

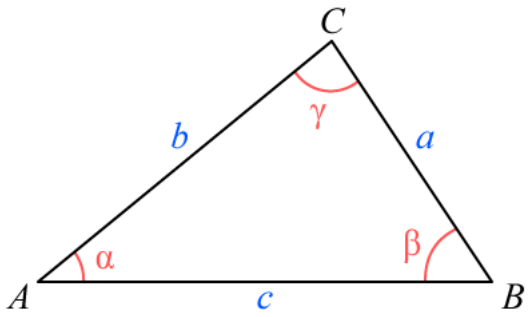


Figure 11: The Al-Kashi theorem defines relations between the sides of a triangle and its angles

Using this theorem, we can find the inclination of the head to look at the assumed position of a game board located at a distance $dist$. We will take the figure 12 to explain the computation. The side a is the distance ($dist$) between the robot and the game board, and b is the height of the robot compared to the center of the board ($16.5cm$). c is the hypotenuse of the triangle, which has a length of $c = \sqrt{a^2 + b^2}$. It remains to apply the Al-Kashi theorem to obtain the angle β.

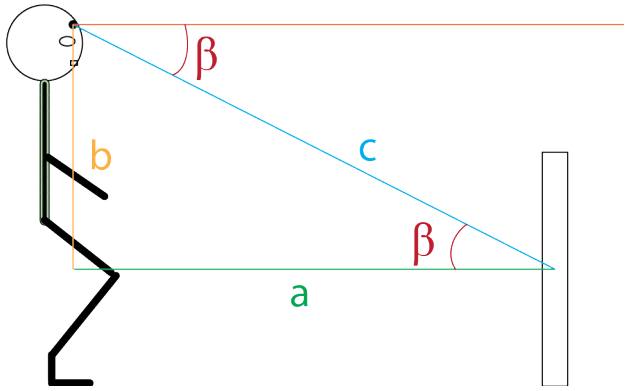


Figure 12: The head angle, β, can be obtained using the Al-Kashi theorem

2 Circles detection

To play correctly, the robot must "be aware" of the game board position. In fact, the robot is not necessarily in front of the board when the game starts, hence it must be able to detect the board in the room.

Let us first consider that the board is facing the robot and is visible in its field of vision. It needs to discern a Connect 4 board in the picture retrieved by its camera. The main characteristic of a Connect 4 board is its rectangular grid of circular shapes. Hence, the circles detection is the first step of the game board recognition.

OpenCV provides what is needed to detect circles in a picture. The algorithm used is called the Hough Transform for circles, which is detailed in [3]. The method in `OpenCV` that implements this algorithm is called `houghCircles`[5].

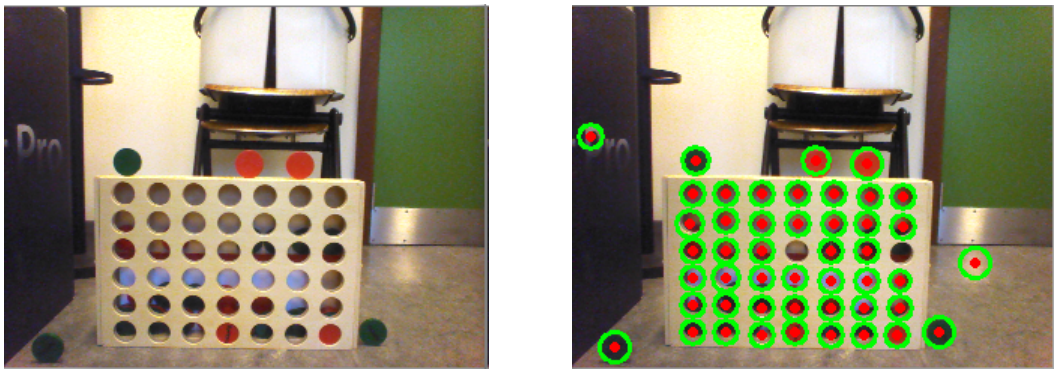


Figure 13: Circles detection with `houghCircles`

2.1 Parameters tuning

The figure 13 shows that noise is sometimes detected outside the game board (false positives) and that some circles may be missed during detection inside the grid (false negatives). The method can be fine tuned using 5 main parameters: `minDist`, `param1`, `param2`, `minRadius` and `maxRadius`

Three test batteries will be performed: the first will be performed to adjust `minRadius`, `maxRadius` and `minDist`, the second will help to adjust `param1` and the third will help to adjust `param2`. each test can be performed separately because the parameters of the different tests are not dependant on one another. For those tests we will use 320x240 pictures as explained in the section 1.

But to perform such tests, we must evaluate the results of each test. Hence a score will be given to each test, this score will tell if the value chosen in the test was good or not by comparing it with the score of the other tests.

Hence we must define a way to determine the score of a test. As we want to detect circles in a picture, we can imagine that the score will count the number of circles detected in the picture

[5](for more details, see [15])

and compare it with the expected number of circles. The closer these two numbers are, the better the score.

There exists a well known technique to determine the score of correctly retrieved data inside a database (we can consider the picture on which the test is performed as a database). This technique is called the balanced *F-Score*.

To define the balanced F-score, we must define the `Precision` and the `Recall` of a data cluster when trying to detect a specific type of data.

$$Recall = \frac{Amount\ of\ data\ correctly\ detected}{Amount\ of\ data\ expected}$$

$$Precision = \frac{Amount\ of\ data\ correctly\ detected}{Total\ amount\ of\ data\ detected}$$

$$Balanced\ F-Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

In our case, we want to detect circles on the game board. Hence, the data correctly detected is a detected circle that belongs to the game board, the amount of data expected is the number of circles on the game board (which is 42) and the total amount of data detected is the total number of detected circles. The balanced *F-Score* was chosen because we assign the same importance to the *Precision* and the *Recall*. Indeed, the *Precision* penalises the false positives and the *Recall* penalises the false negatives.

For each test, multiple pictures will be taken with the robot from a certain distance of the game board. For each distance, pictures will be taken during 15 seconds. For each picture, the parameter(s) to adjust will vary and we will retain the parameter(s) that led to the best score for the picture. But between two pictures, the best parameter(s) can differ. Hence, we will take the parameter that occurs the most. The different distances are between 0.4m and 3m, 0.4m being the minimum distance from which the robot can see the entire grid and 3m being the length of the test room.

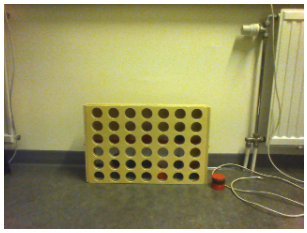


Figure 14: The test pictures were taken with the robot at different distances from the board.

2.2 Circle radius estimation

• What size could be the circles we are looking for ?

When the game begins, NAO does not know at all where the board is. Hence, it is also unable to know the distance between two grid circles or the radius to detect. As a result, we need to adjust the radius of the circles to detect in the image.

We saw on figure 3 that the minimum distance between two circles in the grid is the vertical distance between two neighbours in the grid (0.9cm) + a circle diameter (4.6cm) (because we search for the distance between two centres) = 5.5cm. Knowing that and the real radius of a grid circle (2.3cm), we can adjust the minimum distance: $\frac{5.5}{2.3} = 2.391$. Hence we know that we must keep a 2.391 ratio between the circle radius and the minimum distance between two circle centres. In addition, a small error in the detected radius detected can be admitted because of the camera projection (all circles of the detected grid will not have the same size, see figure 15).

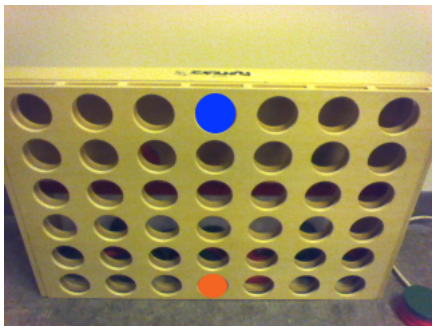


Figure 15: A picture taken by the robot when it was standing 0.4m away from the board. We can see the size difference between the blue circle and the orange circle, due to the image perspective on the rectangular grid.

But the radius to detect is in fact a value between `maxRadius` and `minRadius`.

- `minRadius`: The minimum radius to look for in the image.
- `maxRadius`: The maximum radius to look for in the image.
- `minDist`: The minimum distance between two detected circle centres.

• **Can we predict the radius of a circle ?**

We made multiple tests with different couples (`minRadius`, `maxRadius`). The minimum distance (`minDist`) will be `minRadius`$\cdot 2.391$.
The results will tell if we can find a connection between the optimal parameters and the distance.

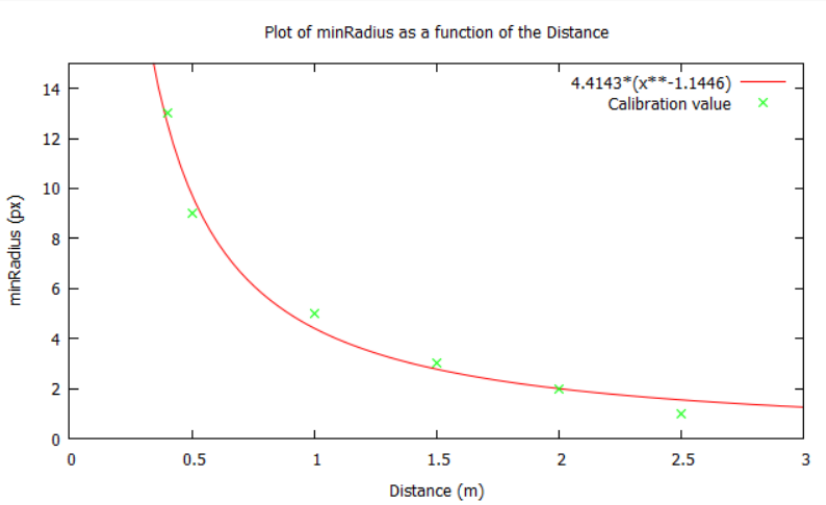
Here are the results of the tests:

Distance	Optimal (`minRadius`, `maxRadius`)
0.4m	(13, 17) px
0.5m	(9, 13) px
1m	(5, 9) px
1.5m	(3, 7) px
2m	(2, 5) px
2.5m	(1, 4) px
3m	Grid never detected

Three meters away is apparently too far away from the board to detect it with this resolution. We could try a higher resolution, but then the `WiFi g` transmission time of the picture between NAO and the computer would be longer as we explained in the section 1.

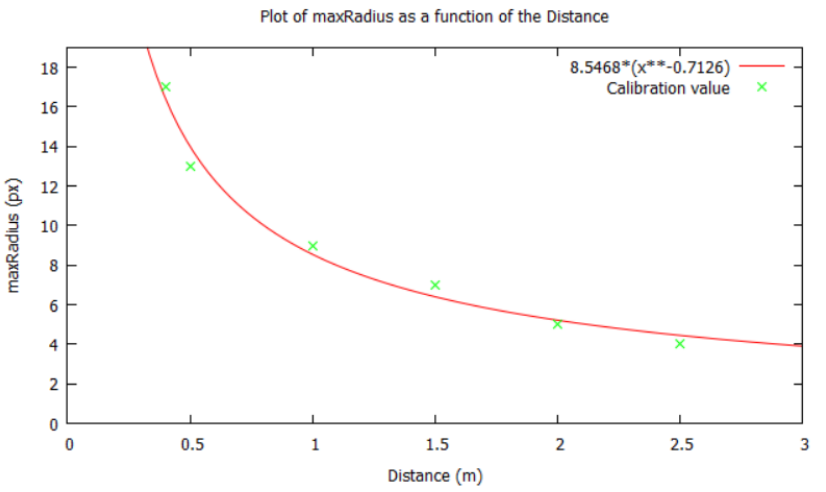
We can see on figure 16 and figure 17 that the `minRadius` and the `maxRadius` both fit an exponential curve as a function of the distance. We know that they fit an exponential curve because it is trivial that the values will not suddenly rise again (as in a quadratic polynomial curve). Indeed, the size of a circle to detect will not increase as we get farther from the board. Hence we can estimate the value of `minRadius` or `maxRadius` for a given distance by an exponential function.

The best fitting exponential function is $y = ax^b$. We used a curve fitting algorithm [10] to find the a and b parameters for the two curves. These parameters are detailed in the figure 16 and in the figure 17. The resulting curves fit very well. The curve approximation for `minRadius` has a R^2 ratio of 0.9867 which is very close to 1 and it is the same for the curve approximation of `maxRadius`, which has a R^2 ratio of 0.9827.



$$y = ax^b$$
$$a = 4.4143$$
$$b = -1.1446$$

Figure 16: The `minRadius` estimation curve



$$y = ax^b$$
$$a = 8.5468$$
$$b = -0.7126$$

Figure 17: The `maxRadius` estimation curve

• **What if the board is sloped ?**

If we consider that the board is somewhere in the room, and not necessarily in front of the robot, we must take a larger range for the radius of the circles (a detection that uses the following technique will be called a detection with the `sloped` flag set to true).

The value of the inclination can be determined using the Al-Kashi theorem introduced in the section 1.3.To detect the board, a maximum angle between the robot and the game board will be set. We chose 45 degrees, because an angle . Doing so, if the game board is sloped more than 45 degrees from the farthest point the robot can see on the board, the robot will not be able to detect the board from its current position and it should move to see it more clearly.

This maximum angle is illustrated in the figure 18; the angle $A\hat{B}C$ is 45 degrees.

We will take the figure 18 to explain how to obtain the radius in that case.
Using the law of cosines, we can find the ratio of α and β of the figure 18. This ratio is the relation between the `maxRadius` of the farthest point and the $\texttt{maxRadius}_2$ of the closest point.

In figure 18, β is the visual angle of the farthest circle on the game board and α is the visual angle of the closest circle on the game board. The `maxRadius` and `minRadius` of the farthest circle can be computed with the approximation curves in figures 16 and 17 for a given distance x. In this case, x is the distance between A and B. By their definition, α will be bigger than β. Hence, we can find the $\texttt{maxRadius}_2$ of the closest circle: $\texttt{maxRadius}_2 = \frac{\alpha}{\beta} \cdot \texttt{maxRadius}$

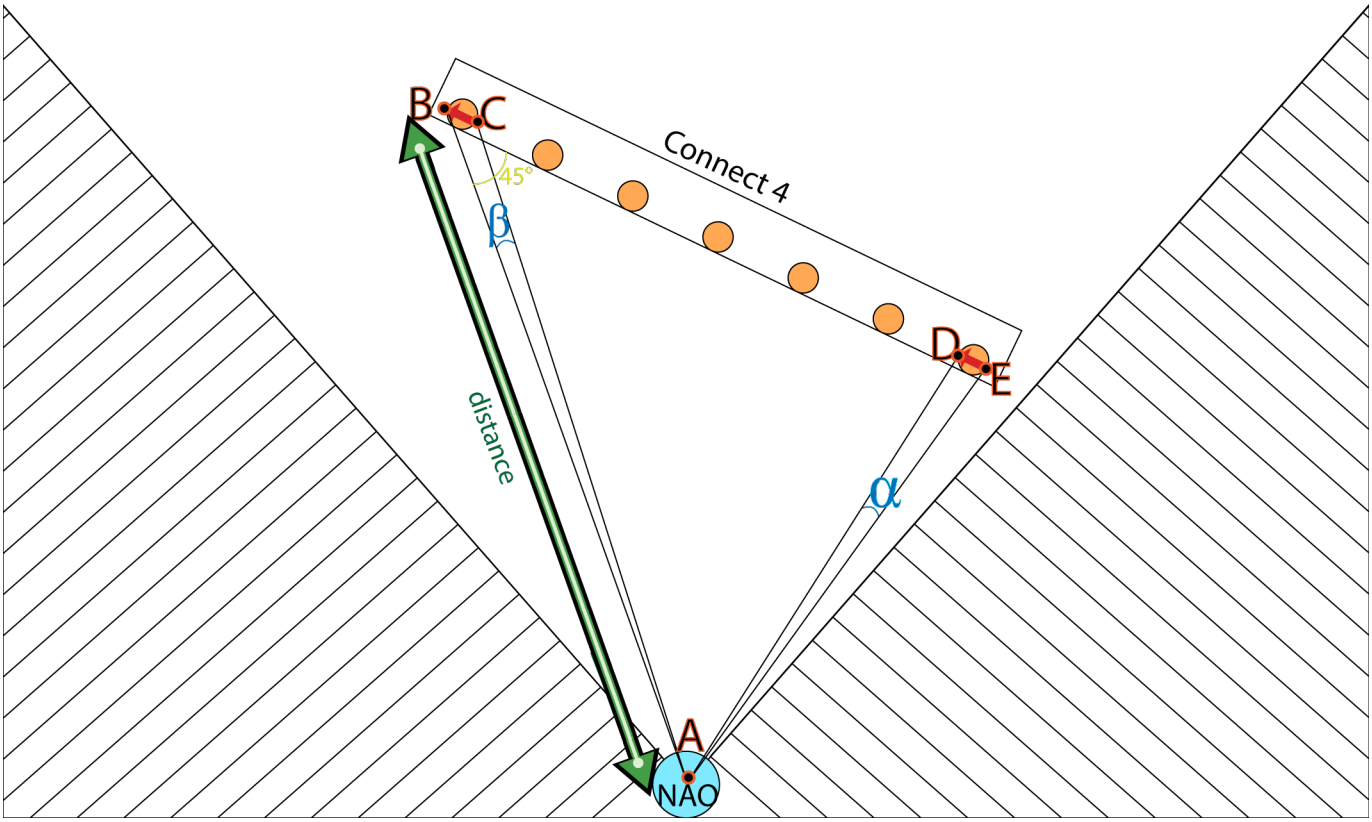


Figure 18: The game board must not be inclined by more than 45 degrees from the robot so the robot can detect it.

For the triangles ABC, ABD, ABE and ADE of the figure 18, we have the following relations (with the measurements of the figure 3).

$$|BC| = 4.6cm; \quad |DE| = 4.6cm; \quad |BD| = 51.9 - 4.6 = 47.3cm$$
$$|AB| = distance$$

$$|AC|^2 = |AB|^2 + |BC|^2 - 2 \cdot |AB| \cdot |BC| \cdot \cos 45°$$
$$|AD|^2 = |AB|^2 + |BD|^2 - 2 \cdot |AB| \cdot |BD| \cdot \cos 45°$$
$$|AE|^2 = |AB|^2 + |BE|^2 - 2 \cdot |AB| \cdot |BE| \cdot \cos 45°$$
$$\beta = \arccos \frac{|AB|^2 - |BC|^2 + |AC|^2}{2 \cdot |AB| \cdot |AC|}$$
$$\alpha = \arccos \frac{|AD|^2 - |DE|^2 + |AE|^2}{2 \cdot |AD| \cdot |AE|}$$

With these relations, we developed an algorithm that takes the distance between the robot and the farthest circle in the grid and that returns the ratio $\frac{\alpha}{\beta}$. With this ratio, we saw above that we can have the new $\texttt{maxRadius}_2 = \frac{\alpha}{\beta} \cdot \texttt{maxRadius}$. `minRadius` does not change because a little circle that is far from the robot will be detected with a smaller radius than a little circle that is close to the robot.

2.3 Canny threshold tuning

`param1` is the parameter used as second threshold in the Canny edge detection [12] used in the algorithm. (The first Canny threshold is half of this one).
The Canny edge detection takes an image in input and transforms it into a black and white image where the white pixels are the gradient peaks (where the gradient difference with the pixel neighbours exceeded the Canny threshold). The higher its value, the less circles will be detected, but if its value is too small, more noise will be detected.

Here are the results of the tests:

Distance	**Optimal `param1`**
0.4	98
0.5	100
1	88
1.5	102
2	98
2.5	96

We can not choose an optimal value with the previous results. Hence we will look at the mean score for every value. Below are the best mean scores with their variance.

`param1`	**Mean**	**Variance**
50	0.76	0.06
53	0.76	0.06
55	0.76	0.07
57	0.76	0.07
59	0.76	0.06
60	0.76	0.06
65	0.76	0.07
75	0.76	0.06
76	0.76	0.06
77	0.77	0.07
78	0.76	0.07
79	0.76	0.07
81	0.76	0.07
82	0.76	0.07
85	0.76	0.07
86	0.76	0.07
87	0.76	0.07
88	0.76	0.07
193	0.76	0.08
194	0.76	0.08
195	0.76	0.08
196	0.76	0.08
198	0.76	0.08
199	0.76	0.08

We will choose 77 for every distances (as `param1` is not distance-dependant) as it is the score with the best mean value and its variance is not too big.

2.4 Accumulator threshold adjustment

This last parameter, `param2`, is an accumulator threshold for the Hough Transform. The smaller it is, the more false or noisy circles may be detected. But the bigger it is, the more legit circles may be missed.

Here are the results of the tests:

Distance	Optimal `param2`
0.4	10.25
0.5	10.75
1	11.25
1.5	11.75
2	9.75
2.5	9

The results show that the short and the long distance need to be more tolerant to noise. This is explainable by the fact that when we are very close to the board, the circles of the bottom of the grid seems more like ovals than circles, like in the figure 15. And when the robot is far from the board, the circles to detect are smaller and therefore less easy to detect.

If we take a look to the mean score of each value, shown in figure 19, we see that the best mean value is obtained when `param2` is 9 or 9.25 and that the results for those values have a very small variance. So, we will take 9.25 for every distance. In fact, it is not a problem that some noisy circles are detected. We will se in the next section how such noise can be tolerated.

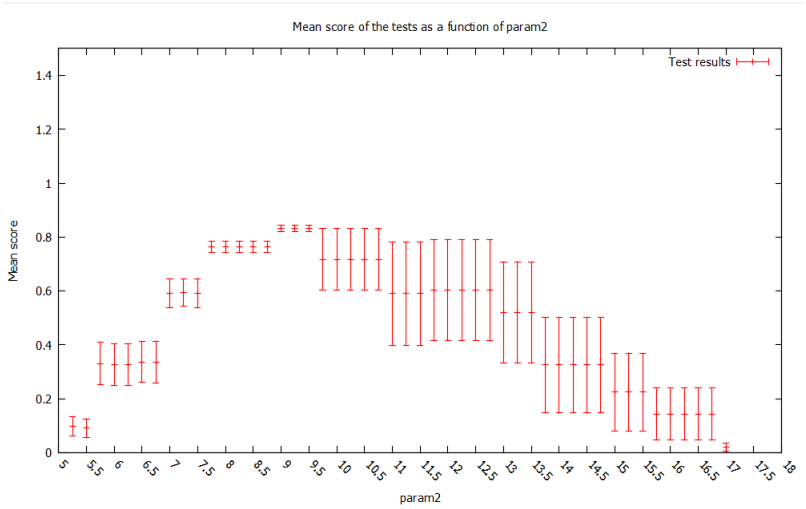


Figure 19: Mean score as a function of `param2`

3 Game board recognition

Now that we know how to detect circles in a picture, by section 2, we will use some tricks based on the implementation of OpenCV's `findCirclesGrid`[13] in order to detect a circle grid pattern.

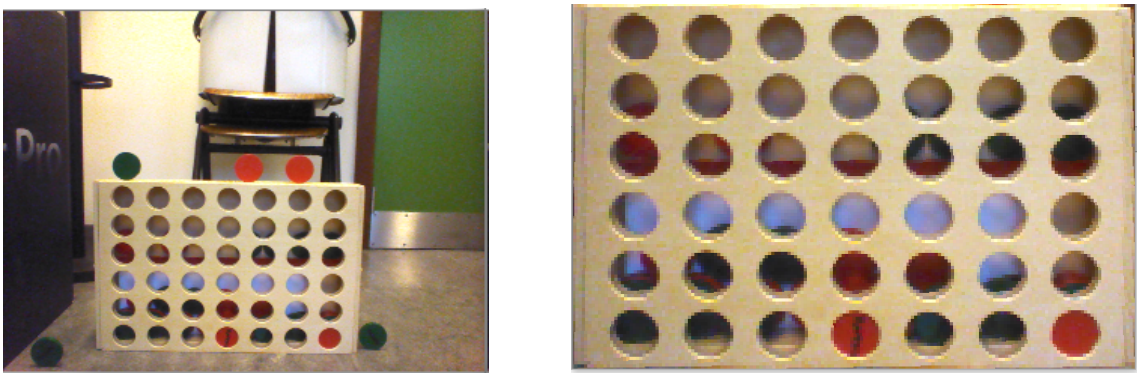


Figure 20: Original picture followed by the result of the grid detection

Using the methods introduced in the previous section, a circle detection is launched. If at least `min_circles` circles are detected in the grid, we continue to the next step. `min_circles` is a parameter $\in [12, 42]$. 12 is the minimum number of detected circles necessary to detect a game board using the technique introduced in the following sections. The minimum set of detected circles is shown further in this report in figure 37. 42 is the number of circles in the circle grid of the game board, hence it is not necessary to set the minimum number of detected circles to a number that is bigger than 42. If the number of detected circles is smaller than `min_circles`, we consider that we are sure we are not facing the board and that it is pointless to continue the analysis. In practice, 28 is the value that gave the best results in the detection. When there is no board, there are less than 28 circles detected, and when there is a board, there is more than 28 circles detected.

Now that the circles have been detected in the picture, we want to make sure that those circles belong to the game board. Indeed, we see on figure 21 that some detected circles were outside the game grid.

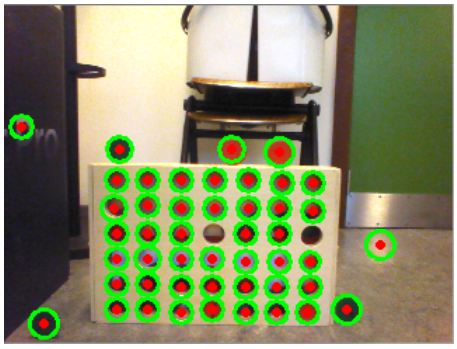


Figure 21: Result of step 1

3.1 Time complexity

To evaluate the quality of the developed techniques, we will use the *Big-O* notation, which defines the time complexity of an algorithm. This notation has the advantage of abstracting totally the hardware and the software characteristics.

Let $f(n)$ and $g(n)$ be two complexity functions ($\mathbb{N} \to \mathbb{R}$),

$$f(n) \in O(g(n)) \iff \exists c > 0, \exists n_0 \in \mathbb{N} \ \ such\ that \ \ \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

This can be applied to algorithm by defining that a basic operation can be declared as a complexity function $\in O(1)$. A basic operation is an operation which takes a constant time to be performed, i.e. that is not dependent on any variable or sequence size. A counter example is an algorithm that takes a list as input, and the bigger the list, the longer the execution lasts.

In the same vein, an algorithm a that executes n basic operations, for a variable n, implies that $a \in O(n)$.

In practice, a competitive algorithm has a time complexity under $O(n^2)$.

3.2 Graph connection

Once we have detected enough circles in our image, we have to make sure that we have the board in the image and we want to delete the possible noise in the detection (detected circles that are not part of the game board). To do that, we will start by connecting the circles in a directed graph. The nodes of the graph will be detected circles. We will build the arcs as following:

Let $\mathbf{X}$ be the set containing the nodes of the graph (the detected circles)
and $\mathbf{U} \subseteq \mathbf{X} \times \mathbf{X}$ be the set containing the directed arcs of the graph.
Let i, $j \in \mathbf{X}$, and $dist(x, y) \in \mathbb{R}^+$ (x, $y \in \mathbf{X}$) be the distance between x and y.

$$(i,\ j) \in \mathbf{U} \iff \nexists\ k \in \mathbf{X}\ such\ that\ (dist(k, j) < dist(i, j)\ \ \textbf{and}\ \ dist(i, k) < dist(i, j)) \qquad (1)$$

We developed a naive algorithm to connect the detected circles but it had a complexity $\in$ $O(\mathrm{n}^3)$ and was therefore really slow. Nevertheless, it appears that there exists an algorithm in $O(\mathrm{n} \cdot \log \mathrm{n})$ that solves the *Delaunay triangulation*, and we will see that the *Delaunay triangulation* nearly respects (1) and that we can use that algorithm to connect the circle centres.

We found an implementation of the *Voronoï/Delaunay* problem in OpenCV, `Subdiv2D` [17], which, given points, constructs the Voronoï diagram and the Delaunay triangulation. All we have to do is use the results to obtain the needed relation (this is what the algorithm in the figure 24 does).

The *Voronoï diagram* is defined as follows:
Let $\mathbf{S}$ be a set of 2D points. The Voronoï diagram has a plane partition for each point $\in \mathbf{S}$.
Let $\mathbf{V}(\mathbf{p})$, $p \in \mathbf{S}$ be the partition created from the point p.
Those partitions are such that if we take a 2D point $q \notin S$ such that $q \in \mathbf{V}(\mathbf{p})$, and $p \in \mathbf{S}$, then, the nearest point from q in $\mathbf{S}$ is p. An example of a Voronoï diagram is shown on the figure 22.

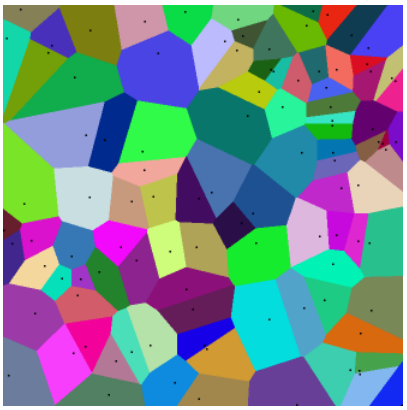


Figure 22: A Voronoï diagram. The points $\in \mathbf{S}$ are in black.
The partitions $\mathbf{V(p)}$ are coloured around p.
©Maksim, Wikimedia.

Now we will take a look to the *Delaunay triangulation*, which can be found using a Voronoï diagram. In fact, this triangulation is found by drawing the dual graph of the Voronoï diagram. In other words, the Delaunay triangulation is a graph built in such a way that $\forall(u,\ v) \in \mathbf{S^2}$,

$$(u,\ v) \text{ is an arc of the graph} \Leftrightarrow V(u) \text{ and } V(v) \text{ are neighbouring areas.}$$

Doing that, a point is connected to its nearest neighbours.
An example is shown on the figure 23.

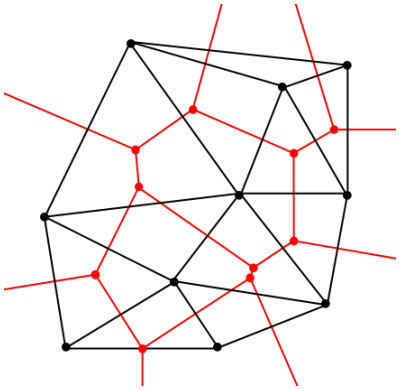


Figure 23: The Voronoï diagram in red.
The Delaunay triangulation and the points $\in \mathbf{S}$ in black.
©Nü es, Wikimedia.

But we do not have what we are looking for yet: our graph must respect the relation (1). But if we take out the biggest border of each triangle, by definition of the Voronoï diagram and the Delaunay triangulation, our graph respects (1), an example is shown on figure 25. This modification is represented by the algorithm below.

Algorithm *A1*:

```
For each triangle (A, B, C) in the Delaunay triangulation graph:
    Remove max(AB, BC, CA) from the edges of the triangulation.
Return the resulting graph
```

Figure 24: The algorithm *A1* takes out the biggest edge of each triangle of the Delaunay triangulation

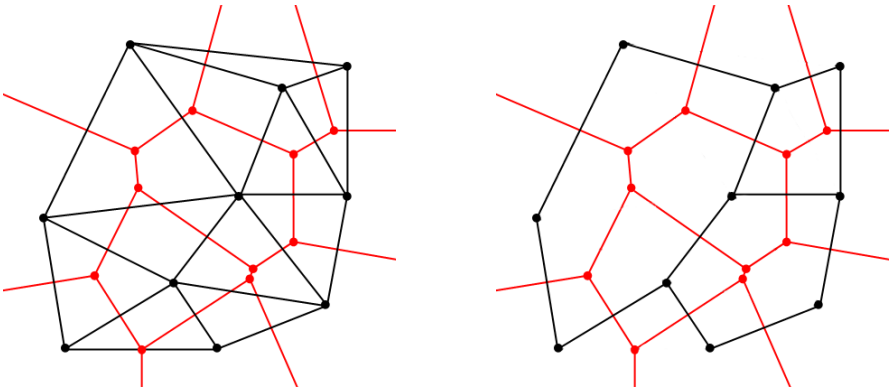


Figure 25: On the left, the Delaunay triangulation in black
On the right in black: a graph that respects the relation (1)
©Nü es, Wikimedia.

We can proof that this modification of the Delaunay triangulation respects the relation (1)

1) **There is no more triangles** after the algorithm *A1* is performed. We can prove it with a reductio ad absurdum.
Assuming that there is still a triangle, (A, B, C), in the graph returned by the algorithm *A1*. (A, B, C) was already in the Delaunay triangulation because the graph returned by A1 is a sub-graph of the triangulation. Hence, the algorithm *A1* have removed an edge from (A, B, C). Here is the contradiction. We proved that it is impossible that a triangle remains in the graph returned by the algorithm *A1*.

2) **The resulting graph respects the relation (1)**. We can prove it by reductio ad absurdum.
Assuming that it does not respect the relation (1), it means that $(i,\ j) \in \mathbf{U}$ and $\exists k \in \mathbf{X}\ such\ that\ (dist(k,j) < dist(i,j)\ \textbf{and}\ dist(i,k) < dist(i,j)$. Hence, the graph contains the triangle (i, j, k), thus contradicting 1) where we proved that there was no more triangle in the graph. We proved that the graph respects the relation (1)

We are also sure that the resulting graph is connected. In fact, the Delaunay triangulation produces a connected graph by definition, and the algorithm *A1* does not disconnect the graph. In fact, this algorithm deletes an edge if and only if there exists another way that is shorter. Hence there is always a route between two nodes.

In conclusion, we used the OpenCV algorithm that solves the Delaunay triangulation to connect our points into a graph, and we developed the algorithm in figure 24 to make our graph respect the relation (1).

The relation is obtained with a complexity $\in O(n \cdot log(n) + t)$, where t is the number of triangles in the Delaunay triangulation, so the complexity $\in O(n \cdot log(n) + n)$.

At this point, we can see in the figure 26 that it is possible that this technique connects circles that are outside the game grid. Hence, we have to filter the arcs to delete the noise.

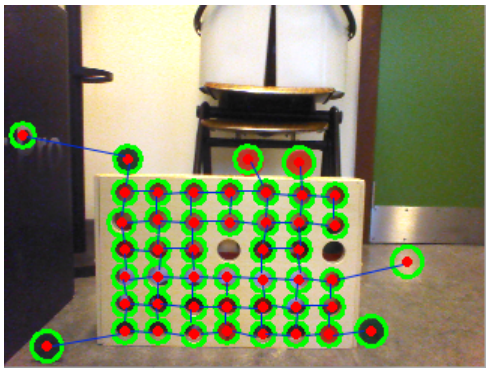


Figure 26: The graph connection connects a circle with its nearest neighbours.
Each undirected arc represents two directed arcs in opposite directions.

3.3 Connection filtering

We would like to remove the noise from the graph. To do so, we developed an algorithm that works this way : for each arc, we count how many other similar arcs there are in the graph. If the number is bigger than a certain threshold that we will call `min_similar_vectors`, it means that the arc is not a result from the noise and we keep it. Otherwise we delete it.

`min_similar_vectors` will depend on `min_circles` that was introduced in the previous section. The minimal set of detected circles is 12 (see figure 37) and, in this case, the maximal number of horizontal similar vectors is 6 and the maximal number of vertical similar vectors is 5. In this minimal case, the maximum number of similar vectors is also the minimal number, because, as explained in the section 3.4, we need all the arcs illustrated in the figure 37 to detect the grid. In general, for X detected circles in the grid, the maximal number of similar horizontal vectors is X-6 and the maximal number of vertical similar vectors is X-7. Hence, in order to admit an error (when the detected circles are not neighbours), we will define `min_similar_vectors` with the following formula:

$$\texttt{min\_similar\_vectors} = \frac{12}{\texttt{min\_circles}} \cdot (\texttt{min\_circles} - 7)$$

To compare the arcs, each arc will be considered as a vector (the vector connecting the two circle centres connected by the arc in the picture) and a small error margin will be allowed to counter the camera projection.

This technique filters the arcs because the grid is made of lines and columns, hence, if we

connect each circle with its horizontal and vertical nearest neighbours (which has already been done in the section 3.2), we will have nearly equals vectors horizontally and nearly equals vectors vertically. They are **nearly** equals because of the error margin that we allowed earlier. An example is shown in figure 27

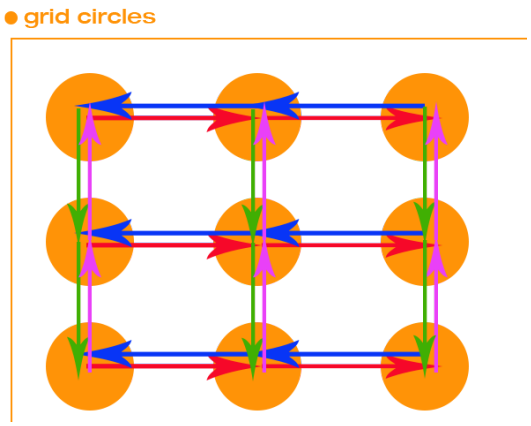


Figure 27: Similar vectors are in the same colour

Now we have to set the value of the error margin. Since the circles detection, we know the parameters `minRadius` and `maxRadius`. To set the error margin, we must consider the maximum error to allow, illustrated on figure 28. If an error margin that is bigger than the hypotenuse of the triangle formed by three neighbouring circles with minimum radius, we would allow diagonal vectors, and as a result, we would not have a grid at the end of the detection.

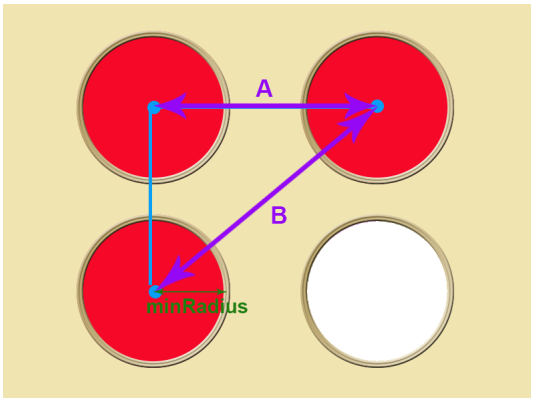


Figure 28: B must not be allowed as an arc in the graph

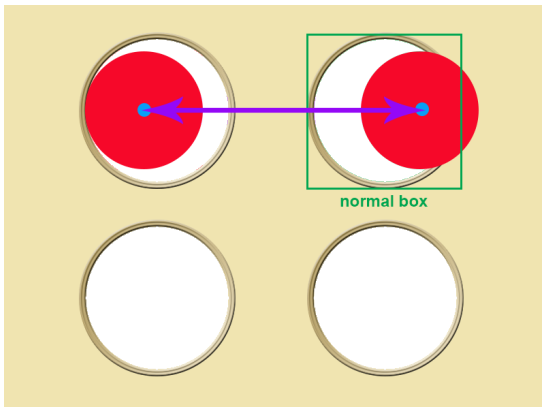


Figure 29: The `HoughCircles` method can detect circles slightly outside its "normal box".

Hence, the maximum error will be $0.9(dist(A, B))$. The 0.9 factor stands to avoid diagonal arcs if the error illustrated on the figure 29 occurs.

Nevertheless, this filter can isolate non-noise circles as we can see in figures 30 to 32. To counter this problem, we can filter, return to the graph connection without considering circles detected as noise and re-filter, see figure 32. We can also note that, even with this double filter,

grid circles can be isolated. It is not a problem as the next steps do not need every grid circle to detect the game board.

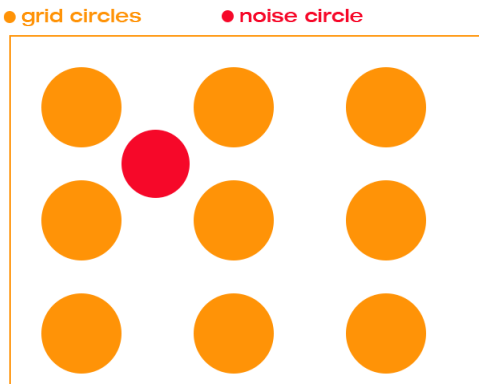


Figure 30: The circle detection can detect noise.

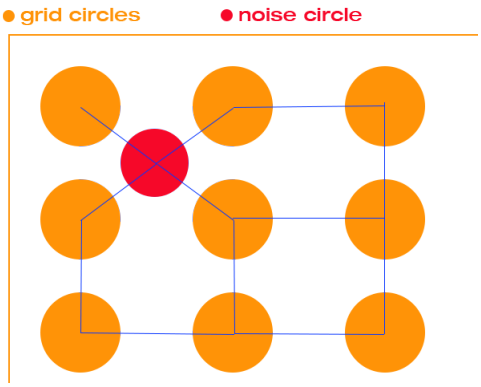


Figure 31: The graph connection could take a noisy circle for a grid circle.

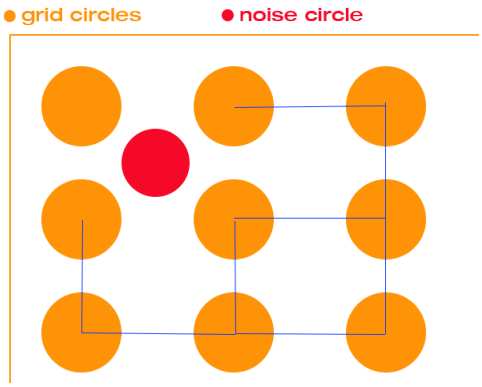


Figure 32: The connection filter, in this case with `min_similar_vectors` = 4 deletes the noise from the graph by removing the arcs that were connected to the noise

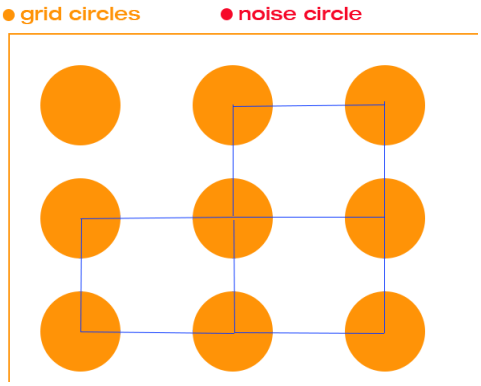


Figure 33: The second graph connection do not consider the noise that was detected by the filter and produces more legit connections than the first graph.

The filter we developed uses a K-Dimensional Tree, built in $O(n \cdot log(n))$ with the technique detailed in [9]. A KDTree is a tree that retains the distance between the elements. It is then really easy and fast to query the x elements that are the closest to a certain element. Notice that, because of the *double filter*, in the worst case, we connect and filter two times. So, the complexity at this point $\in 2 \cdot O(2n \cdot log(n) + n) \in O(n \cdot log(n) + n)$.

Now we are almost sure that there is no more noise inside the grid. If such noise is still in the grid after the two passes of the filter, then the parameter `param2` of the method `houghCircles` should be higher, to reduce the risk of detecting false circles and/or the parameter `minDist` should also be bigger so that there is no noise between two circles in the grid. But outside noise is still possible, some examples are the orange circles in figure 34. This residual noise resists to the filter because the vector that connects it to a grid circle is very similar to vectors between

two grid circles.

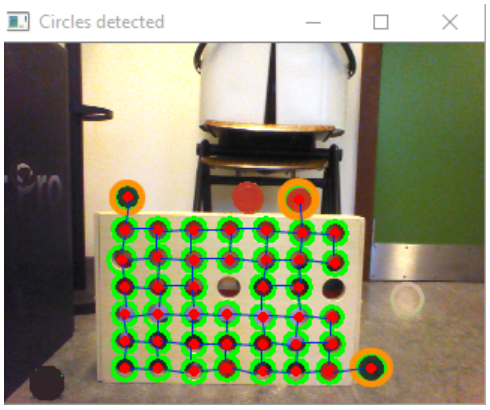


Figure 34: Result of step 3

3.4 Graph exploration

Now, we have a connected graph that tends to represent our game board, but we are not sure that every circle on the game board has been detected at step 1. Therefore, we cannot just expect to have a clear 6x7 grid of detected and filtered circles.

Therefore, we will develop an exploration algorithm that searches for possible grids inside the filtered graph. To explore, we will organize our vectors in four clusters using the `kmeans` method implemented in `SciPy`. Each cluster will contain vectors going in the same direction. Now we are able to distinguish "up" vectors from "down" vectors and "left" vectors from "right" vectors. We will keep only the "up" and "right" vector as a basis for the exploration.

During this exploration, we will allow to go through a vector in both its direction and its opposite direction.

For every node in the graph (sorted randomly), we will start an exploration to mark each node as (i, j), which means the circle is at the i^{th} line and the j^{th} column.
The first node is marked `(0, 0)`. The figure 35 explains how to mark each node.

This exploration is a breadth-first search (BFS), it means that the exploration is based on a queue, and each time that an element that has not already been explored is encountered, we add it to the queue. We repeat until the queue is empty.

Once each node is marked, we normalize the marks by ensuring that each mark has no negative component, as in the figure 36.
The graph is connected, hence, if the biggest line component in the marks is under 6 or if the biggest column component in the marks is under 7, we are sure that there is no possible 6x7 grid in the detected circles and also that the board is not detectable by our analysis.

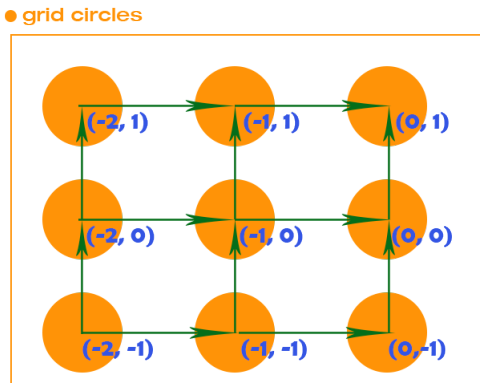


Figure 35: The graph marking starts from the (0, 0) node and marks every other node with a position that is relative to the start node.

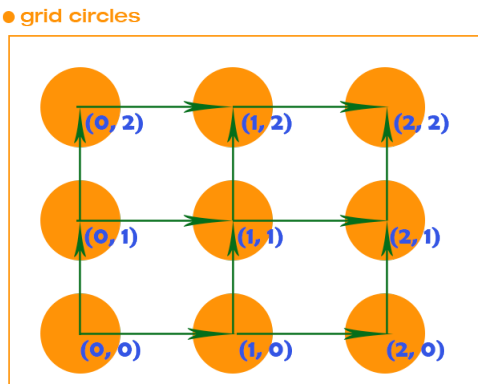


Figure 36: It is easier to consider the (0, 0) node as the lowest leftmost node.

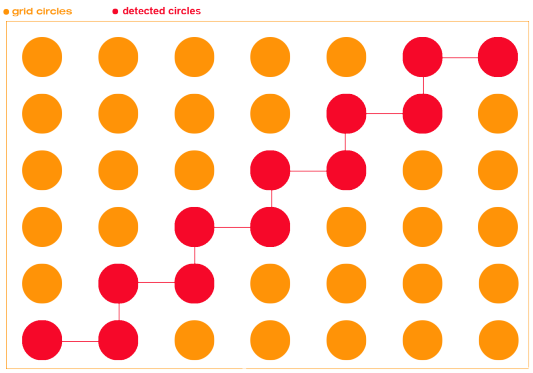


Figure 37: The minimum set of detected circles to detect a 6x7 grid of circles

Complexity

The complexity of the BFS is $\in O(n^2)$ because we explore n nodes using a memory to avoid to explore two times the same node. The memory used is a list where the slot i contains True if the i^{th} node has already been explored, hence, its access has a complexity $\in O(1)$. Hence we explore n nodes and check if its neighbours have already been explored in $O(n)$ because in a graph with n nodes, the maximum number of neighbours for a node is n. As a result we have in the worst case a complexity $\in O(n^2)$ to explore our graph. But in reality, a node in the graph have maximum 4 neighbours by its construction. So, we can say that the complexity, in fact, $\in O(n)$.

Handling too large grids

It can happen that bigger grids than 6x7 grids are detected. In this case, we will separate our big grid into 6x7 inner grids. The inner grid with the biggest amount of vectors is taken as the game board. If there is a tie in this amount, we can decide to drop the picture to restart the analysis on another picture (and thus with other detected circles) in order to be more confident on the position of the game grid.
For example, in figure 38, 3 inner 6x7 grids can be distinguished. The number of inner grids $\in O(n)$.

In practice, multiple detections will be launched, and the results will be compared. As point of comparison, we take the pixel coordinates of the (0, 0) hole. if the distance between two results is bigger than the distance between a hole and its vertical nearest neighbour, we consider the detection as unstable. This allows to have a trustful detection.

This achieves the step 1 of the figure 5.

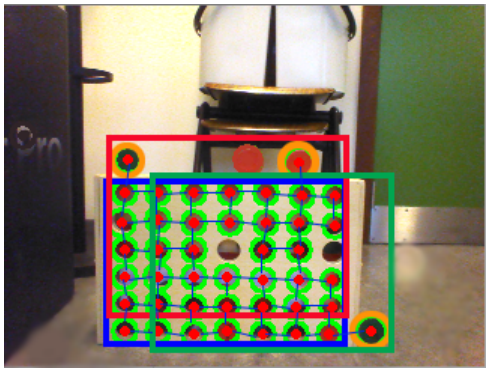


Figure 38: The result of step 4 is the blue rectangle. The two others are noisy.

As a result, the algorithm that finds the correct grid has a complexity
$\in O(n \cdot log(n) + n + n \cdot n) \in O(n^2)$.

3.5 Finding a homography

Now that we know where our grid is and, more importantly, where some circles of our grid are in the picture, we can search for a homography matrix to crop and convert our picture, distorted by the camera, into a distortion-less "perfect" picture. To do that, we need a reference image, in which we know the exact position of each circle of the grid. We use the detected circles of the grid in the original picture and match them with the theoretical one in the reference image, see figure 39, using OpenCV's `findHomography` [14] method.

3.5.1 Reference image

With the homography matrix, we can now reshape and refocus our image so it matches the reference image. OpenCV provides the method that we need: `warpPerspective` along with the `WARP_INVERSE_MAP` flag, see [18] to see the details. An example of transformed picture is the figure 40.

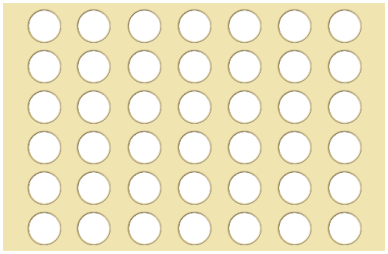


Figure 39: The reference image is an image that respects the board measurements.

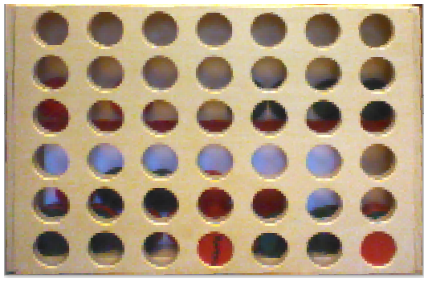


Figure 40: The transformed picture is shaped as the reference image.

Working with this transformed picture is very helpful as it provides the actual pixel coordinates of the grid circles in the reference image. Thanks to the transformation by the homography matrix, the pixel coordinates of the circles in the reference image are the same as the pixel coordinates of the circles in the transformed picture. Hence, even if a grid circle was not detected or isolated by the filter, we can still infer it by the coordinates of the reference image. This will be used in the next section.

3.5.2 Game state analysis

The homography can also be used to analyse the current game state. Has NAO will play autonomously, its artificial intelligence must know the current state of the game. To do so, we can just make NAO walk back so it has the game board in its vision field, and then, using the homography, we can get the current state of the game by looking which color dominates inside each front hole.

To do so, we will take a circular area of 3 pixels around the center of the front hole and take the average color of these pixels. At this point, the color of the pixels is defined by the RGB method. Each pixel has three color values; a red value, a green value and a blue value. The mix of the three values gives the color of the pixel.

The weakness of this method is that it is difficult to define exactly the color of a pixel. For example, our discs are green and red. Hence, we would like to detect which pixels are red, which pixels are green, and the others are considered as empty. We can imagine that a red pixel will have an RGB value of (100%, 0%, 0%), but this value represents a perfect red, and it is very unlikely for a pixel in our picture to have exactly this value. Hence, we would have to define boundaries with RGB values to define what we consider as red and what we consider as green.

We can avoid this issue by converting the pixels we want to classify into another color space. We used the HSV color space. The HSV also have three values, but this time, the values are :

- **Hue**: value between 0 and 360 degrees, it defines the color of the pixel.
- **Saturation**: a percentage that defines the color saturation of the pixel.
- **Value**: a percentage that represents the brightness of the pixel.

We will assume that we can trust the Hue value if the Saturation value is bigger that 65%. Hence, if the Saturation value exceed 65%, we can easily determine if the pixel is red or green, using the Hue value.

Our tests shown that the green discs are mostly detected as dark, nearly black pixels[6] and that the red discs are always detected with a Saturation value that exceed 65%. So, if the Saturation value is under 35%, we check if the Value value is under 65%. If it is, the disc is considered as green. Otherwise, the hole is considered as empty.

This color classification ends the step 2 of the figure 5 as we can detect if the other player has cheated by comparing the current state with the last state in memory. The step 3 is covered in the section 7.

This analysis requires a lot of lighting, but if the lighting is too high (or too low), the results might be biased... In fact, the background of the front holes is of light blue. And it is hard to draw the line between the red discs, the green discs, and the light blue empty slots. We chose a blue background in order to ease the circle detection, as it enhance the contrast to detect the circles. A white background was considiered at start, but the contrast was not enough and the circle detection was not able to detect enough circles. The section on the future works will explain a method that could solve this problem.

3.6 Finding 3D coordinates

Now that we saw how NAO could see the board, we must not forget that the final goal of the project is to make NAO play the game. Hence, the robot will need to know where the board is in 3D. To achieve this, OpenCV provides a method named `solvePnP` [16] that works in the same way than `findHomography` previously mentioned. This corresponds to the step 4(a) of the figure 5.

To map pixels to 3D coordinates, we need a 3D model of our board. The model provides the measurements of the game board in centimetres, (cf. figure 3). The model consists of relative positions, starting with the (0, 0, 0) coordinate, located at the bottom-left of the front of our board as shown on figure 41. `SolvePnP` also needs the camera's intrinsic parameters and the camera distortion coefficients obtained in the section 1.2.2.

[6]The black pixel has a special HSV value; (0°, 0%, 0%)

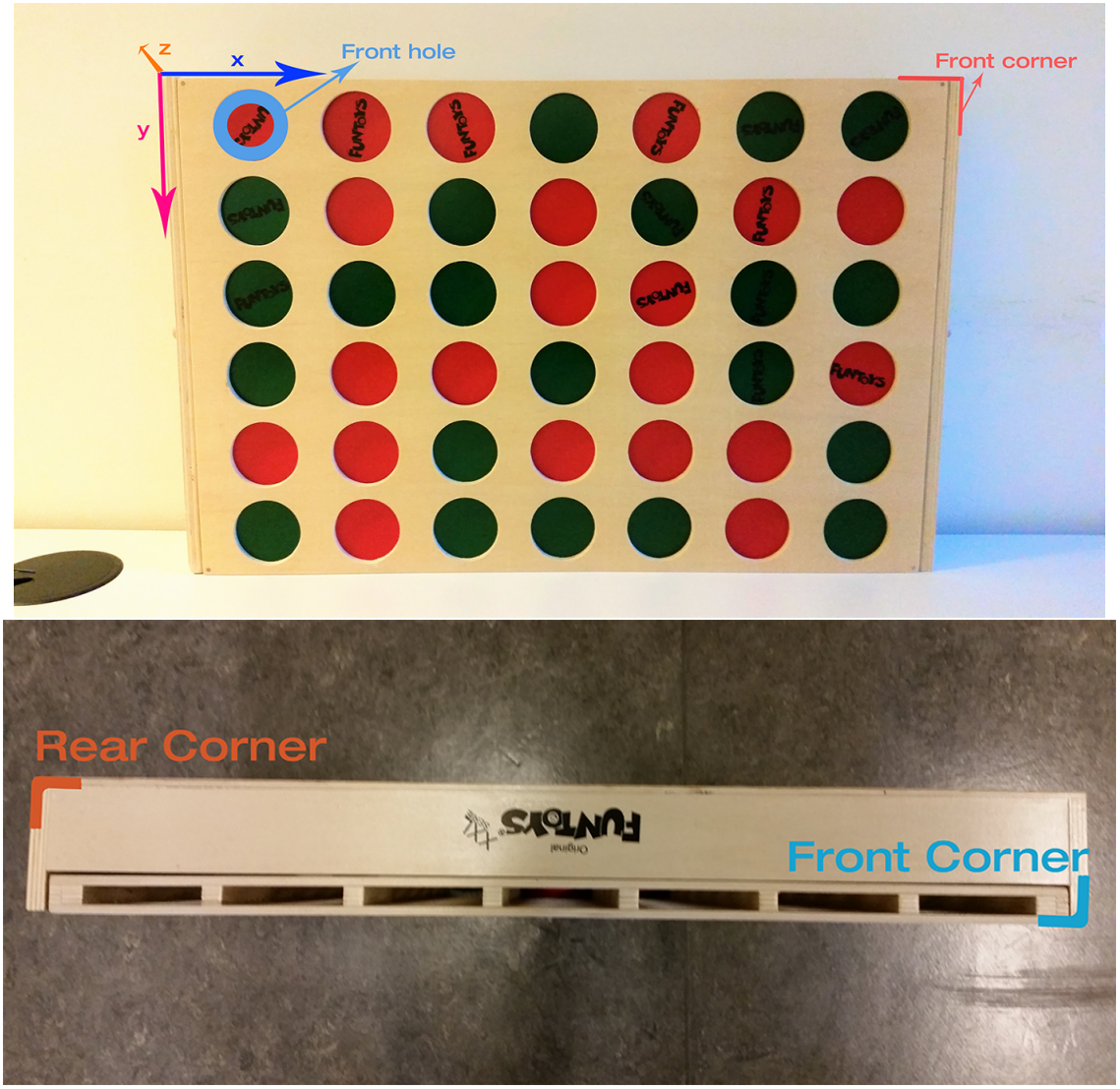


Figure 41: The 3D model of the game board starts with an axis, front/rear corners and front holes

Now to get the 3D coordinates, we must have 2D coordinates that match some of the coordinates of the 3D model. In fact, since we found the homography matrix, we know the pixel coordinates of the front holes, but also the corner coordinates (by the transformed picture). The section 5 will show how we can find the upper hole coordinates.

4 Walking towards the board

Now that NAO can detect the game board, it must walk towards it so it can play the game. For that, the 3D coordinates that we found in the previous section will be used, but we need to transform them to make it understandable by NAO in order to achieve the step 4(b) of the figure 5.

4.1 Axes systems

The 3D coordinates were found using NAO's top camera and `solvePnP`. Hence, the coordinates have the camera as origin. Unfortunately, NAO does not provide any method that takes coordinates that use the camera coordinate system. But it does provide methods to make it walk to coordinates relative to its torso, its feet or its starting point. These coordinate systems are shown in the figure 42.

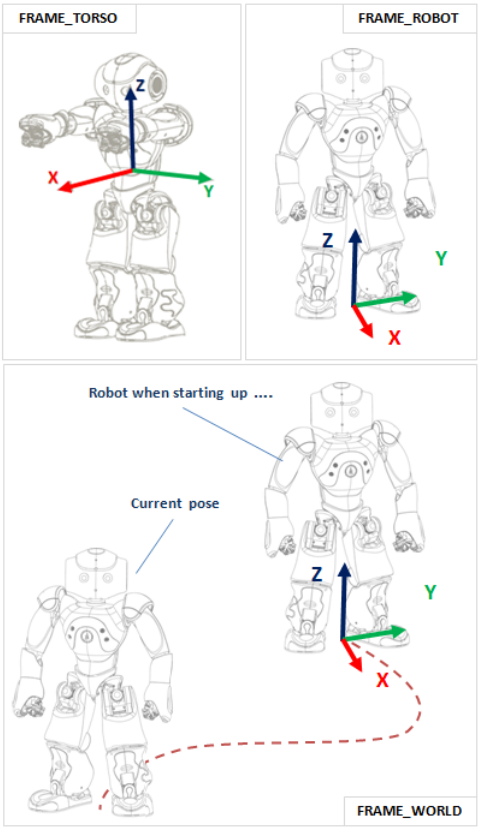


Figure 42: NAO provides three coordinate systems to express coordinates

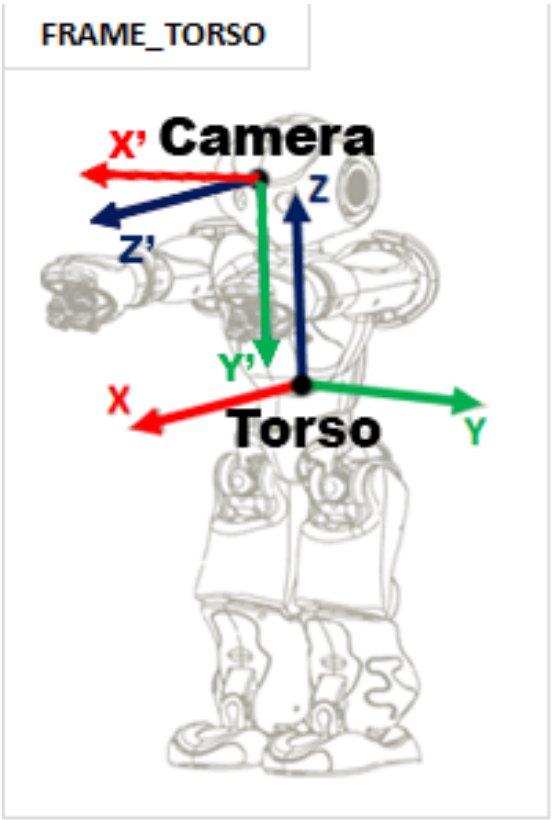


Figure 43: The coordinate systems of the camera and the torso frame

Hence, the coordinates relative to the camera must be transformed into coordinates relative to one of its frames. The difference between the coordinate system of the camera and the coordinate system of the robot torso is shown in figure 43.

The coordinate system can be easily transformed into one of the robot's coordinate systems. We will take the torso frame, as it is illustrated in the figure 43, of the robot to explain how we do this.

Taking the difference shown in figure 43:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} Z' \\ -X' \\ -Y' \end{pmatrix}$$

And we can easily find M such that

$$\begin{pmatrix} X' \\ Y' \\ Z' \end{pmatrix} \cdot M = \begin{pmatrix} Z' \\ -X' \\ -Y' \end{pmatrix}$$

$$M = \begin{pmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}$$

But the camera is not located in the torso and does not look forward all the time. Hence, we have to transform the points coordinates using the rotation matrix R and the translation vector T that represent NAO's camera position from its torso. T and R can be obtained using NAO's motion system's API.

Finally, we can use NAO's API to walk to certain coordinates relative to its torso frame.

4.2 World Representation

Now it would be useful to track the board's coordinates while the robot is walking and moving in order to walk towards it. Using our 3D model of the entire board, we can also track the upper holes coordinates, which will be useful when we will begin to search the hole in which the robot will drop its disc after the robot finished walking.

To do so, the `ALWorldRepresentation` module of the robot was used to stock 3D coordinates of our objects. The objective of this module is to update the coordinates of objects when NAO is moving. Hence, if the board center is located three meters away from the robot and the robot walks one meter toward the board, the module is supposed to update the board coordinates so the robot knows that the board center is now two meters away from it.

The goal of `ALWorldRepresentation` is illustrated in the figure 44.

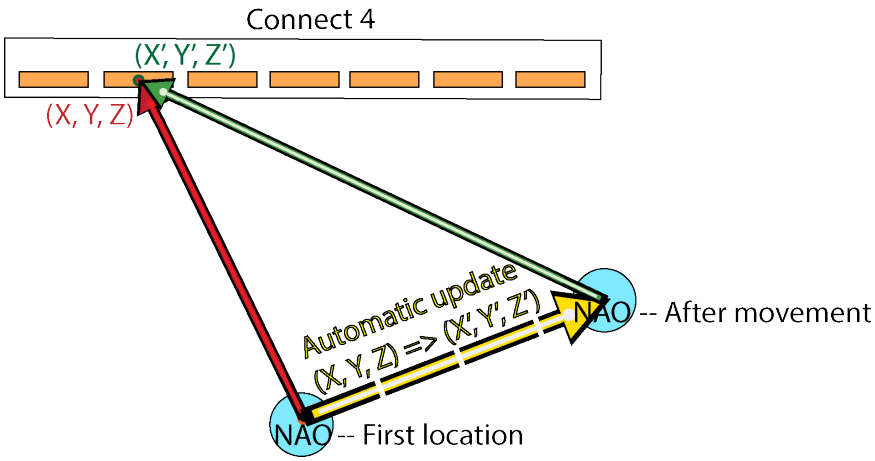


Figure 44: The `ALWorldRepresentation` module is designed to automatically update objects' coordinates.

In order to make NAO drop a disc in a hole of the board, the hole coordinates must be very accurate. We developed a module in which the coordinates of the board could be refreshed at any moment to prevent odometry errors to accumulate during the robot's movements.

The idea is that when a part of the game board can be detected, we map its pixel coordinates with our 3D model and `SolvePnP`, and the tracked coordinates can be refreshed for every part of our 3D model.

4.3 Accuracy

Unfortunately, as explained in sections 9.1 and 9.2 NAO tends to slip when it walks. Hence, an accurate localization system would be needed to trust NAO's walk to arrive at the given coordinates, and to trust the coordinates updated by its world representation. See section 10 on future works to explore the different ways to solve this problem.

As a result, our development on this point is not accurate at all. Even with our methods to refresh the board's coordinates, the new coordinates are almost immediately outdated. As an alternative, we used NAO's odometry information to update the assumed position of the game board, and due to time restriction, we assumed that the robot walks correctly to the board and that, if it slips, a human would replace it to the correct position.

5 Upper holes

The upper holes detection is crucial to make NAO play the game. It needs to see, as precisely as possible, where to place its hand to drop the disc.

As the robot can now recognise the board, we can assume that the robot can walk towards it[7]. As the robot is standing in front of the board, it must now detect the 3D coordinates of the hole in which it will drop the disc.

5.1 Rectangle detection

The holes on the top of the board will approximately appear as rectangles on the pictures taken by NAO. Below is explained the method we developed to detect such rectangles in a picture.

5.1.1 Polygon approximation

First, the `findContours` method of `OpenCV` is used. It takes a binary image as input and gives an array of contours detected in the image as output. These contours are the clusters of white pixels in the binary image. An example of binary image is given on figure 45. A binary image can be obtained using the `Canny` method implemented in `OpenCV`. This method uses the color differences and the gradient intensity to detect what could be a contour in the image.

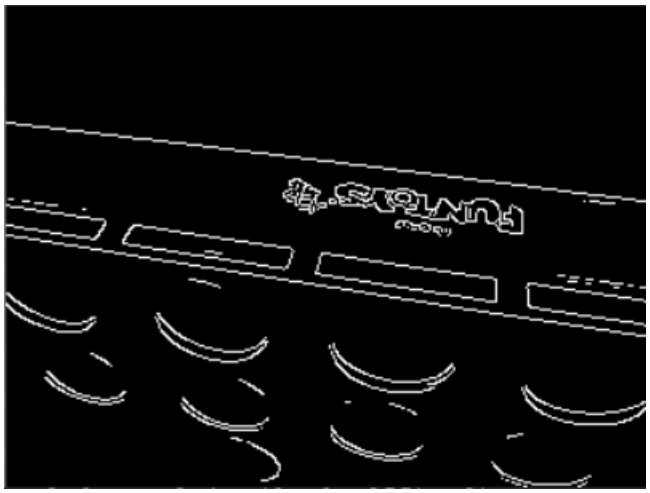


Figure 45: The `Canny` method transforms an image into a binary image as this one.

Next, we will use the `approxPolyDP` method of `OpenCV`, which gives a polygon approximation using a contour. The approximation accuracy is given by a parameter, which defines the minimum length of a side of the polygon. The maximum number of sides in the polygon was set to 100 as it worked well in practice. To do this, the minimum length of a side of the polygon was set to $0.01 \cdot arcLength(contour)$. This method is then applied to each contour that was detected previously in the binary image.

[7] We will see in the section 9 why this step is not that easy.

As we know that we are looking for rectangles in the image, the area of the minimum enclosing rectangle of each polygon will be compared with the area of the polygon itself. If they are close enough, it means that the polygon approximation is an approximation of a rectangle, and hence, it is kept for further analysis. An example of this detection is visible on figure 46

33% of error between the two areas is allowed as the tests showed that this percentage of error gave the best results in practice in different lighting conditions.

Figure 46: The noise is visible as the blue rectangles
that are not the upper holes of the board

5.1.2 Rectangles filtering

As we saw in the figure 46, there is some noise that must be filtered to keep only the rectangles that match the upper holes of the game board. We detail below the different filtering techniques that we developed.

Included rectangles

During the tests, we saw multiple times that shadows and light effects resulted in the detection of false included rectangles. Two types of included rectangles were found, they are shown in figure 47.

Figure 47: Light effects and shadows result in false included rectangle detection.

To handle this case, we decided to test for each rectangle if it was included in another one. we keep both if their common area is less than 33% of the smaller rectangle's area, otherwise, we only keep the bigger rectangle.

This operation has a complexity $\in O(n^2)$, n being the number of detected rectangles, but this is acceptable because a small amount of rectangles is detected by the initial detection.

Length/Width ratio

By looking at the model measurements that are represented in the figure 3, we can see that the upper holes are rectangles with a length/width ratio of $\frac{5.85cm}{0.85cm} = 6.8824$. Hence, we can just keep the rectangles that approximately match this ratio. The approximation comes from the perspective. The rectangles detected are not exactly the rectangles of the model. Indeed, to get the exact rectangles of the model, the game board would need to be looked from the top, without any distortion. The allowed error is 33% of the ratio.

This operation has a complexity $\in O(n)$, as we only check once for each rectangle if it fits the model ratio.

Rectangles similarity

When NAO is close enough to the game board to drop a disc in one of the upper holes, it can see between 2,5 and 3,5 holes in its vision angle following its head yaw angle[8]. Hence, we can use it to filter more falsely detected rectangles. The small amount of holes that can be seen by NAO can be explained by the short field of view of its cameras.

Hence, we will take each rectangle and compare it to every other remaining rectangle to check if the two rectangles are similar in length and width.

We will also check if the two rectangles are aligned. To explain how we do this, we will take the figure 48. If the dot product between the vector A and the vector B is close to 1, it means that the vectors are parallel. and if the vectors are parallel, the rectangles are aligned.

Figure 48: Two rectangles are aligned if the vectors A and B are parallel

5.1.3 Rectangles coordinates

The coordinates of the detected rectangles can found the same way as for the circle grid of the game board: a 3D model of the Connect 4 board is used, in which we described the upper hole measurements, and the `SolvePnP` method is applied.

[8]The yaw angle of NAO's head determine whether it looks to the left, to the right or forward.

5.2 Hole recognition

Now that the holes in the picture can be detected, the problem is to know which one of the seven holes NAO is looking at. This is a difficult problem, because each upper hole looks like any other upper hole. We need to find a particularity for each upper hole in order to distinguish them from one another. To do so, we have used identifiers above the holes.

We first tried to make each upper hole identifiable with NAO landmarks[9], which are circular landmarks that can be used by NAO to localize itself. Nevertheless, their ID was not well recognized by NAO's cameras. In fact, NAO could not detect the ID of the landmark once it was sloped to the camera. And as the landmarks were located on the top side of the board, NAO could not see them otherwise than sloped. In the tests, the ID of a landmarks was detected correctly only once in 25 tries.

Then we tried QR Codes[10], but once again, the QR codes were tilted from the camera, and the code could not be detected by NAO's cameras. In practice, the codes were detected 15 times on 25 tries.

Finally, we tried identifiers that are often used for augmented reality, as in [25] and [11]. They are called *Hamming Markers*. An example of the detection of such markers is shown in the figure 49. The markers are almost always recognized when using a resolution slightly higher than usual (640x480), which implies a slower frame rate, but it does not matter because we only need one image, and the detection is fast. They were recognized 23 times on 25 tries.

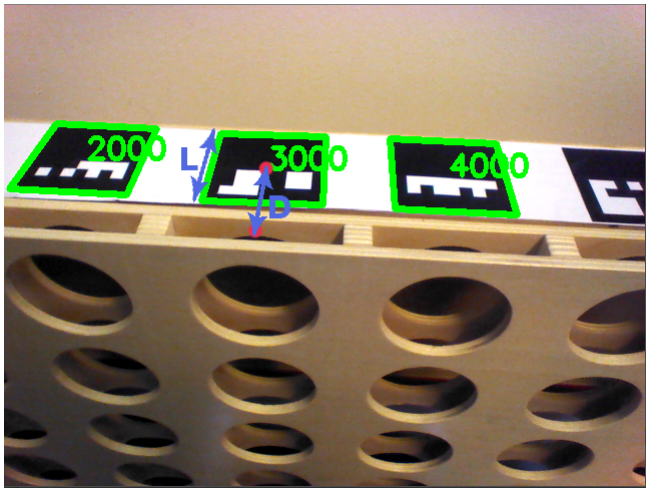


Figure 49: The Hamming marker have a unique ID that allows to distinguish them from one another.

[9] http://doc.aldebaran.com/2-1/naoqi/vision/allandmarkdetection.html?highlight=landmark
[10] http://www.qrcode.com/en/about/

The figure 49 shows that each marker has a unique ID, so, each hole can be distinguished. These codes rely on the Hamming(7, 4) theory, explained in detail in [5]. Each line of the marker encodes 4 bits of data on 7 bits, using 3 bits of parity to correct possible errors of detection This means that the codes detection can be slightly altered (e.g. detection in a blurred or distorted image) and still be able to detect the marker's ID correctly.

Using such markers simplifies the hole recognition and we can use the coordinates of their corners to match the 3D model of the game board to know the 3D coordinates of the upper holes.

5.3 Model matching

Using the Hamming codes, the detected rectangles can be filtered even more precisely than before. As we know the code position on the board, we can expect their respective hole to be below them, at a distance D from their center. Hence we can check if the ratio between D and L (that can be seen in the figure 49) is respected for the rectangles. To get the rectangle that matches this ratio, we use the `KDTree` structure that was already used previously in the circle grid detection.

Finally, when there is nothing left but holes and hamming codes, `SolvePnP` can be applied to match the 3D model previously defined to find the 3D coordinates of the upper holes.

In practice, the more matching points, the better the results of `SolvePnP`. Hence, the rectangles previously detected can be used along with the markers.

6 Manipulating objects

We will now discuss the steps 5 and 6(b) of the figure 5. Let us assume that NAO is near the game board by the work introduced in the previous sections.

6.1 Grab the disc

The step 5 can be achieved by NAO asking a disc and the person giving it, but as it was assumed that NAO is close to the game board, it must not make too big movements so the board doest not fall. Hence, NAO must move to an intermediate position before it can raise its arm.

The intermediate position will be a position where NAO just have its arm perpendicular to its body. This position is supposed without risks for the game board as NAO is supposed to face the game board, but in practice, NAO's odometry is not accurate (see section 9.1), and it is likely that NAO arrives sloped to the game board. This case will not be handled, and we will assume that NAO is facing the game board when it arrives in front of it. To solve that problem, NAO could ask for the disc before walking towards the game board. Doing this, it would avoid to hit the board with its arm.

Now that NAO's arm is in a safe position for the game board, it can raise its hand to ask the other player for a disc. This technique we developed will now make NAO wait until the other players bump its head. In fact, the three touch sensors on the head of NAO are waiting a touch, and once the event "head touched" is triggered, NAO close its hand, grabbing the disc placed by the other player. This way, the interaction with the robot is enhanced and we do not need to interact with the computer in order to close NAO's hand.

Other techniques would allow NAO to grab the disc autonomously. But developing such techniques is a difficult task. In fact, a hand of NAO consists of three little fingers that can be closed to grab things as a clamp, using the opposable thumb. Such hands cannot grab large discs lying on the ground. Hence, discs dispenser would have been needed to allow NAO to grab the discs autonomously. We did not develop such techniques because of the time constraints. Some are discussed in the future work section (10).

6.2 Drop the disc

The step 6(b) is crucial for NAO to play the game. Indeed, dropping a disc in the game board is what makes the Connect 4 game. This step is particularly difficult, as the robot image analyses must be very accurate. In fact, NAO must have a precision of 0.15cm, which is the difference between an upper hole in the board and the disc dimensions (visible in the figure 3).

6.2.1 Funnels

The techniques discussed earlier in this report do not provide such an accuracy. This comes from the conversion between the 2D image analysis into 3D coordinates, for which the calibration of the section 1.2.2 provided an accuracy of 0.51cm on average.

To fix the lack of accuracy and to remain within the time constraints, we chose to add a

funnel for every upper hole of the game board. In fact, to avoid using funnels, we would have to find the coordinates of the holes with a better accuracy, which mean that the methods previously mentioned should be deeply reworked in order to get the needed accuracy.

We would like to thank Duncan De Weireld that modelled the funnels in 3D, using `OpenSCAD`[11], an Open-Source software that allows to model 3D objects using lines of code.

Figure 50: Here is an example of the funnels used to allow NAO to drop a disc into the board.

The funnels were printed in 3D, using an `Ultimaker 2`[a],

[a] `https://ultimaker.com/en/products/`

These funnels added a little difficulty for the 3D model of the Connect 4. In fact, where the hamming markers were on the upper side of the game board, they are now on the funnels (as it can be seen in the figure 50). This means that the 3D coordinates of the four corners of the markers is harder to define, as the side of the funnel on which they are placed is tilted.

Furthermore, the rectangle detection can not be performed any more as the rectangle that could be seen on the hole is now hidden with the funnel. However, the accuracy gain when using the rectangle detection was not significant. In fact, we gained 2 millimetres of accuracy, which was not enough to avoid the funnels.

6.2.2 Inverse kinematics

Knowing the upper holes coordinates, NAO must now place its hand above the hole, in the correct position, then drop the disc. NAO has sensors in his joints that allows it to know exactly the position of each of its joints. That position can be expressed in NAO's whichever coordinate system. In practice, NAO's left hand and its torso frame will be used. The left hand of NAO was used because the other hand of the robot used during the development was broken.

To allow NAO to place its hand above the hole and its funnel, NAO's inverse kinematics module is used. As a reminder, the inverse kinematics module indicates how to move each joint of the robot in order to place one of its joints to given coordinates. As we can detect the coordinates of each holes by the techniques developed in the section 5, we would like to place NAO's hand to these coodinates.

[11] `http://www.openscad.org/`

Position of NAO's body First, we manually placed NAO's hand above a hole and we captured the coordinates of the centre of the hand. This position was considered as the "perfect position", relative to the considered hole, for NAO to drop the disc. To drop a disc, the developed technique will make NAO go to a position that approaches the "perfect position". The perfect position accuracy, `ppA`, is left as a user-defined parameter. As NAO is not accurate in its movement, getting to the exact "perfect position" can take a very long time. The different values of the parameters are compared later in this section.

Position of NAO's hand

Once the robot has reached the approximated position, it will try to place its hand in the correct position to drop the disc. Indeed, the hand must be in a certain position for it to drop the disc inside the hole. NAO gives 6D vectors to express its joints position.
A position vector consists of:

$$[x_{coord}, y_{coord}, z_{coord}, x_{rot}, y_{rot}, z_{rot}]$$

$$rot = rotation\ angle \quad | \quad coord = coordinates$$

$$\text{perfect position} = [x_{perf},\ y_{perf},\ z_{perf},\ Rx_{perf},\ Ry_{perf},\ Rz_{perf}]$$

The coordinates give the position of the centre of NAO's hand, and the rotation angles define the way the hand is rotated from the torso (as it is the torso frame that is considered in this part). Hence, the "perfect position" captured earlier also defines the rotation angles of the hand by Rx_{perf}, Ry_{perf} and Rz_{perf}.

Looking at the picture in the figure 51, it can be seen that the rotation around the X and the Y axes does not have to change with the robot's position. In fact, if the robot inclination is changed (if the robot is rotated around the Z-axis), the rotation of its hand around the Z-axis must change too.

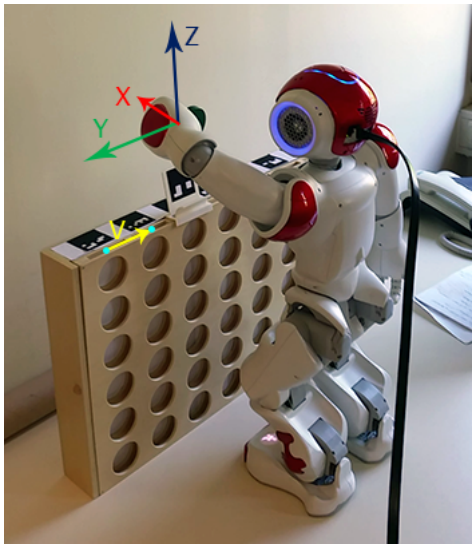


Figure 51: The perfect position of NAO for it to drop a disc in the game board

Hence, if the robot is not at the "perfect position", because of the perfect position accuracy defined earlier, and if NAO is sloped to the board, the Z-axis rotation angle of the hand must change according to NAO's rotation. That rotation can be obtained using vectors. If we take the vector (1, 0, 0) from NAO's torso, and a vector that binds two upper holes of the board, represented by V in the figure 51, an angle of $\frac{\pi}{2}$ indicates that NAO is facing the board.

This is how we indicates the Z-axis rotation angle of NAO's hand, the angle between V and X is computed, we will call it α. But $\frac{\pi}{2}$ is subtracted from α to retain the rotation angle from

the "perfect position", where NAO was facing the board. $\beta = \alpha - \frac{\pi}{2}$. Now, we have to add the perfect angle captured earlier, and it gives $\gamma = \beta + Rz_{perf}$. Doing this, if the robot is facing the game board, $\beta = 0$, and $\gamma = Rz_{perf}$, and if it is sloped, the hand is rotated accordingly to β.

The tests showed that NAO struggled to put his hand in the good position when $\beta > \frac{\pi}{4}$ or $\beta < -\frac{\pi}{4}$. Therefore, if this case happens, our technique will rotate NAO so it faces the board before it tries to put his hand in position.

Inverse kinematics convergence If the perfect position accuracy is too blunt, it is likely that NAO has not the possibility to place its hand at the given coordinates. Indeed, NAO's motion is limited by its degrees of freedom. In fact, where a human being has three rotation axes for each of its joints, NAO has only two.

The problem is that NAO's inverse kinematics module does not indicate whether the given position is reachable or not, it just tries to make the appropriate movement and ends in the closest reachable position. As a result, the only way to determine if the position was well reached is to try the movement, retrieve the actual position of the hand using NAO's sensors, and compare it with the targeted position. Here is the technique we developed to get NAO's hand in the good position.

Let $real\_pos$ be the real position of NAO's hand, and $target$ the hand position for NAO to drop the disc into the targeted hole.

$$real\_pos = [x_0, y_0, z_0, Rx_0, Ry_0, Rz_0]$$

$$target = [x, y, z, Rx, Ry, Rz]$$

To compare these positions, two parameters will be introduced. We called the first one `cA` for coordinates accuracy. This parameter defines the maximum difference between the coordinates of $real\_pos$ and the coordinates of $target$. More precisely, if $|x - x_0| > cA$ or $|y - y_0| > 2 \cdot cA$ or $|z - z_0| > 2 \cdot cA$, then the robot moves of $x - x_0$ meters along the X-axis and $y - y_0$ meters along the Y-axis. The factor 2 that appears for the Y-axis and the Z-axis appears because the robot needs to be more accurate following the X-axis rather than following the Y-axis or the Z-axis (because of the funnels topology).

The second parameter is called `rA` for rotation accuracy. This parameter defines if the rotation angles of the actual position are accurate enough. In fact, if $|Rx - Rx_0| > rA$ or $|Ry - Ry_0| > rA$ or $|Rz - Rz_0| > rA$, the robot is considered in a bad position, as it could not place its hand following the good angles. To solve this problem, the robot will try to reach the "perfect position" and will try again until the `ppA` is satisfied, and will finally try again to put his hand in the good position.

We called this the inverse kinematics convergence because the robot will eventually stop moving and play the disc once it has reached the accuracy given with the parameters. The sharper the accuracy, the slower it will converge, and it can take a very long time.

Parameters comparison In the table below, we rapidly compared the convergence time and the success rate of the technique with different values for the parameters. For each parameters sequence, we tested two scenarios, and for each scenario, we tried two times to put the disc in the hole. We tried to reproduce the scenarios as sharply as possible between the different tests. The different scenarios can be seen in the figure 52. The first is a situation in which NAO is sloped to the board and located at its left, while the second scenario is the "perfect scenario", in which the robot is facing the board, just a little farther than the "perfect position".

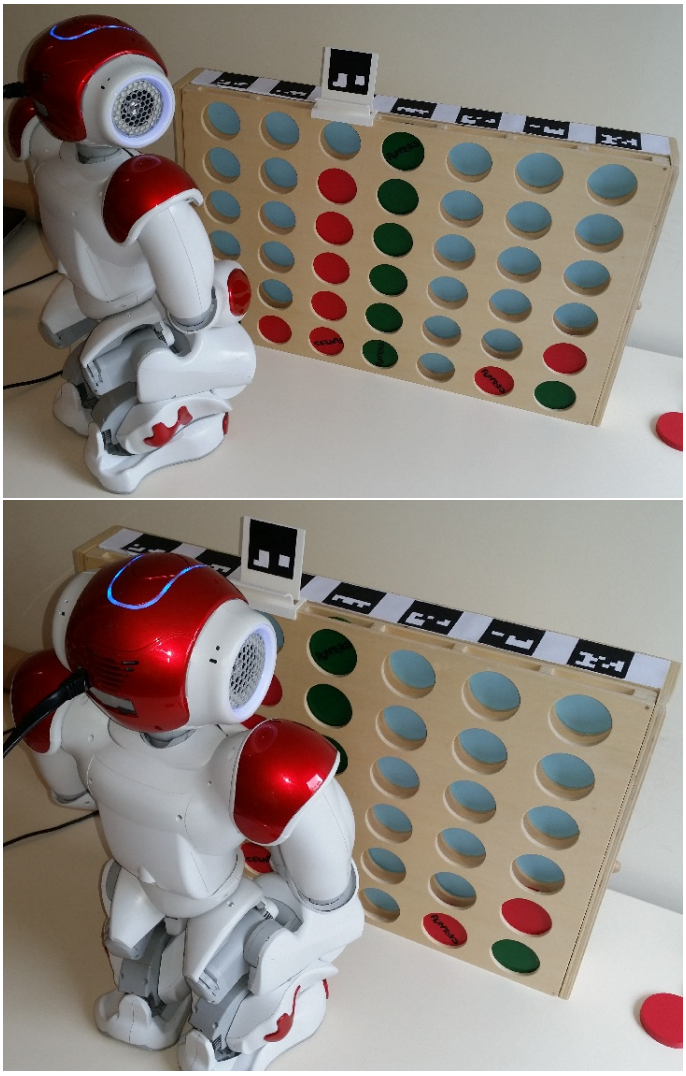


Figure 52: The two scenarios that were used for the tests

ppA (m)	**cA** (m)	**rA** (rad.)	**Success rate**	**Avg. convergence time** (s)
0.025	0.0025	$\pi/12$	75%	49.2
0.025	0.0025	$\pi/6$	25%	35.7
0.025	0.005	$\pi/12$	100%	29.5
0.025	0.005	$\pi/6$	25%	27.0
0.025	0.0075	$\pi/12$	50%	36.2
0.025	0.0075	$\pi/6$	0%	22.3
0.05	0.0025	$\pi/12$	100%	38.9
0.05	0.0025	$\pi/6$	25%	29.5
0.05	0.005	$\pi/12$	100%	23.4
0.05	0.005	$\pi/6$	50%	20.4
0.05	0.0075	$\pi/12$	75%	17.4
0.05	0.0075	$\pi/6$	0%	19.3
0.075	0.0025	$\pi/12$	100%	26.6
0.075	0.0025	$\pi/6$	25%	32.4
0.075	0.005	$\pi/12$	50%	21.3
0.075	0.005	$\pi/6$	0.0%	17.5
0.075	0.0075	$\pi/12$	25%	20.5
0.075	0.0075	$\pi/6$	0.0%	17.5

`ppA` = perfect position accuracy
`cA` = coordinates accuracy
`rA` = rotation accuracy

We can see that `rA` must be sharp to get a high success rate. We will take `ppA` = 0.05, `cA` = 0.005 and `rA` = $\pi/12$ as it is fast and successful. The value of each parameter can be changed when launching the solution.

6.3 Summary

In a nutshell, NAO can drop a disc inside the game board using the following steps. First, NAO finds the coordinates of its targeted hole using the 3D model matching of the section 3 and 5. After that, NAO tries to move to a "perfect position", relative to the coordinates of the hole, to play its disc.

But as NAO's odometry cannot be trusted, NAO will try to go to that position until it reaches the position with an accuracy defined by the user-defined parameter `ppA`. In fact, the coordinates of the hole is computed each time NAO moved, so the algorithm can check if NAO is in the good position or not.

Once the wanted accuracy is reached, NAO will try to place its hand in position to drop the disc. If the robot cannot place its hand correctly, it moves accordingly to the difference between the ideal position for its hand and its actual position.

This is repeated until NAO's hand position is accurate enough, depending on the parameters `cA` and `rA`, and finally, NAO can drop its disc.

7 Game strategy

Now that we discussed how NAO could detect the board and put a disc in it, we will talk about how NAO chooses the hole to play for each turn. The **A**rtificial **I**ntelligence implementation in our project is discussed below, this corresponds to the step 3 of the figure 5

7.1 Game framework

We developed a game framework for the Connect 4 game. This framework was conceived to ease the development of future strategies.

The principle of this framework is simple: a game takes two players, each one with a strategy object. When a player must choose an action, the strategy of the player chooses the appropriate action following the current state of the board.
When the strategy of the current player returns the chosen action, the game will check if the action is feasible (i.e. the column corresponding to the chosen hole is not full).

Once the action is performed, the game checks if the action led to a terminal state , i.e. one of the players won or there is a draw. The naive method is heavy in computations, as we would check if there are four pieces of the same colour in a row on every possible row. These rows are shown on figure 53. Hence we would take every possible sequence of four holes in every possible row. There is 69 possible combinations and, without assumptions concerning the discs, we cannot reduce the number of test. As some strategies explained in the section 7.3 simulate several sequences of actions for a given state, the terminal state test must be as lightweight as possible.

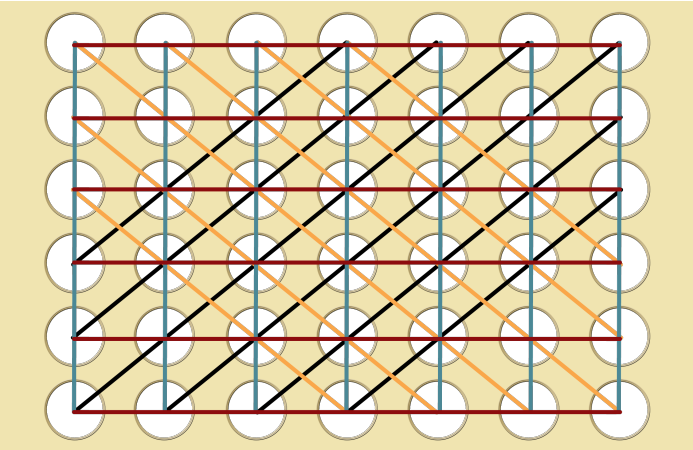


Figure 53: There are many rows in which 4 discs of the same colours in a row can be dropped.

Hence, we make the assumption that if an action is performed on the game, the state before this action was not terminal. We can make sure of this by blocking every action once a terminal state is reached.

If the state was not terminal before, the only way the state is terminal after the action is performed is that the played disc ends the game. There are three separate cases.

1. The player that played the disc wins: the played disc finished a row of discs belonging to him.

2. The result is a draw: the played disc filled the last available hole without managing to have four discs of the same colour in a row → there is no more valid action after this one.
3. No player won this turn.

As we can focus on the last disc played, checking if one player won is easier. Instead of exploring each possible row, we can focus on the rows containing the disc as they are the only rows that could have been filled with this disc.

If there is no victory, the second case can be easily tested by exploring the top line of the board. If every hole is filled, no more disc can be played and the game finishes in a draw.

Finally, if there is no victory and no draw for this turn, the next turn can take place, and we repeat until the case 1 or 2 is reached.

This strategy assumes that no player cheats. It would be easy to detect that a player cheats by the technique discussed in the section 3.5.2. Hence, if a player cheats, NAO can tell that players that it saw him cheat, and consider that the cheater has lost.

7.2 Strategy types

There are multiple types of players that could use the framework. Each strategy defines a choice-making algorithm, which can wait for a valid input to make its choice. We listed some examples below.

Choice-making AI

This type includes every strategy that chooses an action for a given game state and that returns the best action according to its evaluation method.

This will be the type of strategy that NAO will use to play. Once the choice-making algorithm chose the hole for this turn, the robot will walk to the board to drop the disc.

Input strategy

This type will just wait for a valid keyboard input to play. It can be used by a person to play against another console-based AI, or against NAO

Visual-based AI

For a human-being to play against NAO, there is two way to make the computer and NAO aware of the last action of the person: we can use an input-based AI, in which one can indicate the actions of the other player, or NAO can wait to see the discs dropped by the person. The input-based AI is not very convenient as it requires a lot of interaction with the computer. This is why a visual-based AI would be great to select the next action by analysing the current board state, using the technique introduced in the section 3.5.2 and comparing it to the last one. Once a person, or another independent robot, dropped a disc in the board, the intelligence will take the new disc as an action, so NAO can be aware of its opponent's actions.

This type of strategy basically replace any independent player that is not directly playing on the computer. Hence, a person or another robot can use such strategy to be part of NAO's game.

7.3 Implemented AI

We implemented a choice-making algorithm for the Connect 4 game. The algorithm consists of an Alpha-Beta tree exploration. This type of algorithm simulates X levels of actions, X being a parameter. At the first level, the algorithm will simulate every possible action for the current state of the game. Then, for each possible action, the algorithm will evaluate the possibilities of the opponent for the next turn. The action returned is the action that led to the most favourable / least unfavourable situation, assuming that each player chooses the best action (according to the evaluation of the algorithm) at each turn (see [4] for the details).

The evaluation function of our algorithm is very simple:

- If the player has won $\rightarrow$ 1000 points.
- If the opponent has won $\rightarrow$ -1000 points.
- Otherwise $\rightarrow$ 0 point.

This function is very simple but is sufficient to play against any casual Connect 4 players. In fact, this AI will simply avoid to loose and try to win. However, the framework developed makes it very easy to implement other strategies. The only thing to implement is the evaluation function, as the Alpha-Beta mechanism is already implemented in the framework.

8 Logical loop

The final challenge of this project was to link up all the parts. We lacked time to test this loop correctly, and it is therefore a prototype. Furthermore, the 3D printers failed to print the rest of the funnels. Hence, all the tests have been performed with a single funnel as there was no time left to try to print them again To describe the loop, we will use the steps introduced in the figure 5.

8.1 Step 1 - Find the game board

At first, NAO has no idea where the game board is and this step is to find the game board. Hence, it must wander randomly in the room. We did not implement the random part, in fact, we assumed that, if the board is not in front of NAO, it can see the board just by turning around itself. Hence, NAO will turn around, looking for the board until it detects it.

Nevertheless, the game board detection introduced in the section 3 requires a distance. As the distance is unknown, the algorithm will try to detect a game board using multiple distances, and with the `sloped` flag set to true in order to allow the board to be farther or closer than the tested distance. The technique developed in the section 1.3 is also used to place NAO's head in the good position to see the board if it is located at the assumed distance.

Once the board is detected, NAO stops turning around and will walk to be placed half of a meter away from the board so it can take the next step. This first movement can be failed if NAO slips too much. In this case, we assume that a person would just replace the robot in the correct position.

8.2 Step 2 - Analyse the game state

When NAO is in place, it analyses the game board using the technique of the section 3.5.2. Then if it is its turn, the next step can begin, otherwise, the robot waits until the other player has played. In practice, this step is a problem as our technique is extremely light-dependant. Hence, the action of the other player is given by entering it on the computer, using an *input-based* strategy, introduced in 7.2.

8.3 Step 3 - Choose the best action

The strategy used by the robot can now take a decision using the current game state and its choice-making algorithm. If the strategy used is an input-based strategy, the robot will wait until the choice has been entered in the computer.

8.4 Step 4 - Walk towards the board

NAO must now walk to the board, but this time, it will try to reach the perfect position for it to play the disc. Once again, NAO can slips on its way to the board. A person would have to replace the robot in front of the game board, at least so it can see two markers of any upper hole.

8.5 Step 5 - Grab the disc

To grab the disc, NAO just raises its hand and wait for a person to bump its head once the disc is in its hand.

8.6 Step 6 - Drop the disc

To place its hand in the good position, NAO will use the technique developed in the section 6.2, but to detect the coordinates of the targeted upper hole, the markers on the top of the board must be detected. To detect them, NAO's head must be in the good position. Without a trustful localization module, there is no way to determine the position of NAO from the board because of its blunt odometry when it walks.

For this reason, if NAO does not find the markers in its filed of view, it will perform a spiral movement with its head until it detects two markers. If it reaches the end of its spiral without detecting the markers, NAO ask the persons around him to place it so it can see the markers.

8.7 Step 7 - Walk back

This step is optional if a *vision-based* strategy is not used. Indeed, it is useless to NAO to walk back if it does not need to look at the board. It can just wait until the action is performed and entered on the computer. However, if a *vision-based* strategy is used, NAO must walk back so it can analyse the game board to prepare its next action.

8.8 Step 8 - Repeat

Ideally, NAO must repeat from step 2 until the game reaches an end, but in practice, the motors of the robot overheat after 5 or 6 discs played. The only way to solve this problem would be to find a solution to effectively cool its joints.

9 Encountered problems

During the development, we have encountered some problems that slowed us down. Some of them are explained below.

9.1 Localization module

Robot localization is a common problem that consists in giving enough information to the robot so it knows where it stands in a room in relation to its starting point. There are many scientific references on the subject, as [22] and [8].

NAO's official documentation[12] gives information on a module called `ALLocalization` that is supposed to allow NAO to know where it stands in a room, using its odometry and its cameras. Unfortunately, the main methods, `learnHome` and `goToHome`, do not work correctly. After many tests, the robot never returned well to its initial location.

But `ALLocalization` uses another module, `ALVisualCompass`, that allows NAO to navigate in a room using its cameras. We tried it many times, but without success.

In fact, `ALVisualCompass`[13] gives a method, called `moveTo`, that is supposed to move the robot to a given position, using its cameras to approximate the distances, and this method was ineffective. These bugs did not allow us to move the robot precisely in 3D. All we could do was using its motion methods, based on odometry only, that are not precise at all as the robot tends to slip on the ground when it walks. Hence, for this project, every time the robot walks to see the game board, we assume that a person is there to replace the robot in the correct position if needed.

9.2 World representation

NAO comes with a World Representation module, called `ALWorldRepresentation`[a]
This module is designed to automatically update object positions relatively to NAO's frames using the odometry of the robot.

But once again, the odometry of the robot is not trustworthy once the robot walks. Hence, the world representation slowly slips as the robot itself slips, and the information automatically updated is rapidly obsolete. This is illustrated on figure 54

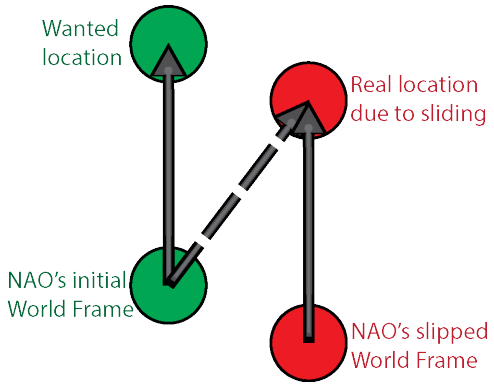


Figure 54: NAO's world representation rapidly becomes obsolete.

[a] `http://doc.aldebaran.com/2-1/naoqi/core/worldrepresentation.html#alworldrepresentation`

[12] `http://doc.aldebaran.com/2-1/naoqi/vision/allocalization-api.html`
[13] `http://doc.aldebaran.com/2-1/naoqi/vision/alvisualcompass-api.html`

10 Future work

The result of our work is not perfect and would need some improvements in order to be released and maintained. We thought about some topics and techniques that could improve the way that NAO plays, but due to time restrictions, we did not implement these.

10.1 Game state analysis

The step 2 of the figure 5 was covered in this report as a colour analysis in an image. This technique can be a problem when the lighting is low or the colour of the discs slightly change with the time.

A more robust technique would be to simulate the human mind. This can be done by machine-learning algorithms, as in [23], that will, as a human-being would do, learn over time the differences between a red disc, a green disc and an empty space.

Such algorithms require training data, in which the developer gives the real colors of the discs along with the raw data. Doing so, the algorithm can progressively learn what characterize the red discs, the green discs and the empty spaces, but we have to define what is the raw data that the algorithm will classify into colors or empty spaces. An idea could be to give the color of the pixels in the area of the slot we want to classify alongside the previous state of this slot. Indeed, it is unlikely that a slot previously classified as red suddenly change into a green one, except for a cheat attempt.

That technique would make NAO even more autonomous than before, as it will learn on its own to analyse the slots of the game.

10.2 Artificial intelligence

The step 3 of the figure 5 was not the main goal of this project. In fact, this part was the easiest because there is a lot of work, as `VICTOR` [1], that covers how to implement a good AI to play the Connect 4, but the other steps of the project took a very large percentage of the time span allowed. This is why we only developed a basic AI for this project.

The artificial intelligence explained in the section 7.3 is very basic and it would be great to implement multiple AIs with different difficulties, so NAO can play with everyone and adapt itself to the level of the person it plays with.

Random AI A very easy AI would be a random one. The column it would play would be a random column among the available ones. Another, a little harder to beat, would be to mix the AI we developed with the random AI. It could randomly play using the random AI, and randomly play using the AI we developed, therefore simulating a person's distracted mind, sometimes playing normally, sometimes playing randomly.

Advanced Alpha-Beta Another intelligence could be an Alpha-Beta using an advanced evaluation function. Rather than just checking if the player has lost or won, the following rules could be implemented additionally to the score of a victory and a defeat :

- 10 points for each disc in a row that can be completed and that belongs to the player. Hence, two discs in a row that can be completed would give 20 points.
- -10 points for each disc in a row that can be completed and that belongs to the other player.
- 10 points for every row that the player fills with his discs.
- 20 points for every series of disc belonging to the other player that the player blocked.

VICTOR Finally, an AI that implements `VICTOR`, the unbeatable strategy introduced earlier in this report, could be developed.

Levels of difficulty The levels of difficulty would then be the following :

Level	Artificial Intelligence
1	The Random AI
2	The mix between the random AI and the basic Alpha-Beta AI
3	The basic Alpha-Beta AI of section 7.3
4	The advanced Alpha-Beta AI
5	The AI that implements `VICTOR`

10.3 Robot localization

The steps 4(b) and 7 of the figure 5 could be improved. As we could not trust NAO's API, it was difficult to develop a robust technique to avoid the robot to slip away from its path. Such a technique could be a real-time localization system.

We could develop an advanced localization technique, as covered in [22], or use another robot API than Aldebaran's. There exists ROS[14], which is an operating system specially made for the robots. Advanced localization algorithms are already developed in ROS. There is a ROS binding that allows to control NAO through ROS, and we could install it on NAO.

We did not install it at first on the robot because the majority of the ROS packages are controlled through `C++`, and we were not familiar enough with the language to consider it within the time span of the project.

[14] `http://www.ros.org/`

10.4 Disc manipulation

Multiple points can be improved for the disc manipulation.

Disc acquiring

Currently, the robot waits for a person to put a disc in its hand, but an autonomous technique could be developed for the robot to pick up its next disc itself. But the discs used in this project are too big for NAO to pick them up when they are lying on the ground. Hence, a disc dispenser could be designed to help NAO to grab the disc.

Funnels

The funnels were placed to ease the disc manipulation by the robot, because developing a technique providing the needed accuracy for a funnel-less solution was too time-consuming for this project. To acquire such accuracy, one technique would be to improve the camera calibration or the method to transform pixels into 3D coordinates.

We started to develop a machine-learning technique that uses a neural network to allow NAO to learn the ideal position of its hand, given the 3D coordinates of the upper holes. Doing so, the robot could mimic the learning routines of human beings and learn to place discs by itself, using the visual landmarks acquired by its cameras. We did not have the time to feed the algorithm with training data.

Hamming markers

The Hamming markers above the upper holes were placed to ease their identification, but other techniques could be designed to avoid the use of such markers. For example, the rectangle detection developed in the section 5.1 could be used along with a hole tracker. The tracker would try to detect a landmark, such as an edge hole, for which we know the position (first or last hole), and to estimate the position of the targeted hole from this landmark. Obviously, to use the rectangle detection, we would have to get rid of the funnels.

10.5 Graphical user interface

Currently, the project is controlled through a command line interface, but in order to make the software usable by anyone, it would be interesting to develop a graphical interface where the user could see everything that NAO sees.

10.6 Code quality

Due to the time restrictions of the project and the complexity of the different tasks to implement, the quality of the code is lacking. The code would need some improvements to be maintained.

In order to be correctly tested, the parts of the code that control NAO should be abstracted to work without any robot connected. For example, if a module requires an image of the board captured by NAO, the abstract robot would give a random image that would match the needs for the test.

The problem is that for a smoke test of the entire software, the tests would barely need a simulation of the robot. Indeed, the robot's hand movement cannot be tested otherwise.

Part IV
Conclusion

This project gave us the opportunity to learn thrilling subjects in computer sciences such as computer vision techniques, human-robot interactions, artificial intelligence, inverse kinematics, and many more. It was an exceptional experience and a great occasion to work with a humanoid robot.

The final result of the project is still a prototype, and each step could be improved. In order to release the application to anyone, a lot of improvement, detailed in the future work section of the previous part, would be needed. The main part that needs improvements is the 3D coordinates acquisition that would need to provide a precision of 0.0015m to avoid the use of the funnels and a tracking system for the upper holes would be needed to get rid of the markers that identify each upper hole. Nevertheless, the result is quite satisfying as the robot is able to play the game, even if it is a bit slow and that the board requires some material, as funnels and markers. The main goal of the project has therefore been reached.

Acknowledgement

We would like to thank the directors of this project: Pierre Hauweele, that helped many times to find new techniques and to overcome the robot limitations, and Tom Mens that helped to make this report as complete as possible and that gave us the opportunity to work with NAO.

References

[1] Victor Allis. *A Knowledge-based Approach of Connect-Four – The Game is Solved: White Wins*. 1988. URL: http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf (visited on 08/10/2015).

[2] Esubalew Bekele et al. "Pilot clinical application of an adaptive robotic system for young children with autism". In: *Autism* 18.5 (July 2014). URL: https://asknao.aldebaran.com/sites/default/files/publications/autism-2013-bekele-1362361313479454.pdf (visited on 10/10/2015).

[3] Jaroslav Boroviccka. *Circle Detection Using Hough Transforms Documentation*. URL: http://www.borovicka.org/files/research/bristol/hough-report.pdf (visited on 19/11/2015).

[4] Samuel H. Fuller and John G. Gaschnig. *Analysis of the alpha-beta pruning algorithm*. 1973. URL: http://repository.cmu.edu/cgi/viewcontent.cgi?article=2700&context=compsci (visited on 09/05/2016).

[5] Jonathan I. Hall. URL: http://users.math.msu.edu/users/jhall/classes/codenotes/hamming.pdf (visited on 09/05/2016).

[6] HumaRobotics. *NAO plays Connect 4*. URL: http://www.humarobotics.com/en/robotics-lab/nao-humanoid-robot-plays-connect-4/ (visited on 11/10/2015).

[7] HumaRobotics. *NAO plays poker*. URL: http://www.humarobotics.com/en/robotics-lab/nao-plays-poker/ (visited on 11/10/2015).

[8] Patric Jensfelt. "Approaches to Mobile Robot Localization in Indoor Environments". doctoral. Royal Institute of Technology, Stockholm, 2001. URL: https://www.diva-portal.org/smash/get/diva2:8964/FULLTEXT01.pdf (visited on 09/05/2016).

[9] S. Maneewongvatana and D. M. Mount. "It's okay to be skinny, if your friends are fat". In: *4th Annual CGC Workshop on Computational Geometry*. 1999. URL: http://www.cs.umd.edu/~mount/Papers/cgc99-smpack.pdf (visited on 11/10/2015).

[10] D. Marquart. "An Algorithm for Least-squares Estimation of Nonlinear Parameters". In: *SIAM Journal Applied Mathematics* 11 (1963), 431–441.

[11] Jeremiah J. Neubert and Tom Drummond. *Using Backlight Intensity for Device Identification*. URL: http://www.und.nodak.edu/instruct/jneubert/papers/pda_id.pdf (visited on 09/05/2016).

[12] OpenCV. *Canny*. URL: http://docs.opencv.org/3.0-last-rst/modules/imgproc/doc/feature_detection.html?highlight=Canny#canny (visited on 22/11/2015).

[13] OpenCV. *findCirclesGrid*. URL: http://docs.opencv.org/3.0-last-rst/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=findcirclesgrid#findcirclesgrid (visited on 17/11/2015).

[14] OpenCV. *findHomography*. URL: docs.opencv.org/3.0-last-rst/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=findhomography#findhomography (visited on 24/11/2015).

[15] OpenCV. *houghCircles*. URL: http://docs.opencv.org/3.0-last-rst/modules/imgproc/doc/feature_detection.html?highlight=houghCircles#houghcircles (visited on 17/11/2015).

[16] OpenCV. *solvePnP*. URL: docs.opencv.org/3.0-last-rst/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=solvepnp#solvepnp (visited on 22/11/2015).

[17] OpenCV. *Subdiv2D*. URL: http://docs.opencv.org/master/df/dbf/classcv_1_1Subdiv2D.html#aed58be264a17cdbe712b6a35036d13cb (visited on 02/12/2015).

[18] OpenCV. *warpPerspective*. URL: http://docs.opencv.org/3.0-last-rst/modules/imgproc/doc/geometric_transformations.html?highlight=warpperspective#warpperspective (visited on 24/11/2015).

[19] Renzo Poddighe and Nico Roos. "A NAO robot playing tic-tac-toe – Comparing alternative methods for Inverse Kinematics". In: *25th Belgium-Netherlands Artificial Intelligence Conference (BNAICS)*. 2013. URL: http://bnaic2013.tudelft.nl/proceedings/papers/paper_40.pdf.

[20] Aldebaran Robotics. *ASK NAO : Autism Solutions for Kids*. URL: https://asknao.aldebaran.com/.

[21] Brian Thorne and Raphaël Grasset. *Python for Prototyping Computer Vision Applications*. URL: http://ir.canterbury.ac.nz/bitstream/handle/10092/5430/12630971_2010-Python_for_Prototyping_Computer_Vision_Applications.pdf (visited on 07/05/2016).

[22] Sebastian Thrun, Wolfram Burgard and Dieter Fox. *Probabilistic robotics*. Ed. by MIT Press. Intelligent Robotics and Autonomous Agents series. 2005.

[23] Edward Tolson. *Machine Learning in the Area of Image Analysis and Pattern Recognition*. URL: http://web.mit.edu/profit/PDFS/EdwardTolson.pdf (visited on 09/05/2016).

[24] Z. Zhang. "Camera Calibration". In: G. Medioni and S.B. Kang. *Emerging Topics in Computer Vision*. Ed. by eds. Prentice Hall Profressional Technical Reference, 2004. Chap. 2, pp. 4–43. URL: http://research.microsoft.com/en-us/um/people/zhang/Papers/Camera%20Calibration%20-%20book%20chapter.pdf (visited on 28/04/2016).

[25] Xiaowei Zhong et al. "Designing a Vision-based Collaborative Augmented Reality Application for Industrial Training". In: *it - information technology* 45.1 (2003). Ed. by Oldenbourg Verlag, 7–19.