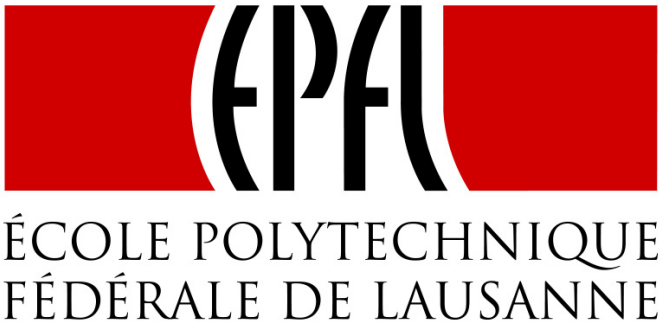
Biorobotics Laboratory
Ecole Polytechnique fédérale de Lausanne

Semester Project Report

Localization of an underwater swimming robot

Matthieu Dujany
Supervisor : Mehmet Mutlu

Spring 2018



Contents

Acknowledgement

I would like to thank my supervisor Mehmet Mutlu for his advice and precious help during this project. I would also like to thank Bezad Bayat and all the researchers from BioRob lab who helped during this semester.

1 Introduction

1.1 Scope of the project : conditions and requirements

The aim of this project is to optimize and compare two indoor localization techniques for an underwater swimming robot. The computation of the localization will be done onboard, in an embarked microcontroller (a nanoPi).

The localization has to be effective indoors (for demonstrations in pools), so GPS cannot be used. Only global localization has to be achieved : there is no obstacle detection and avoidance layer which would have to run at high frequency. As a consequence, the computational efficiency of the localization program is not a constraint or a main objective.

There is no mapping involved. The pool would have been previously "prepared" for the robot : the patterns used or the Aruco markers would be put on the floor of the pool, and the precise map of their positions loaded on to the board.

We can aim at a localization precision of the order of magnitude of the centimeter, with an update frequency of 2 Hz. This update rate could be increased if the precision is lower or if the robot's speed is increased.

The robot used is a lamprey swimming robot. We want to achieve only 2D localization in the pool : the robot would be swimming at the surface of the pool, at a height of 10 to 20 cms.

A camera is located below the head of the robot and filming downwards. Using the video stream of this camera, the localization will be performed using 2 different techniques which we are going to present in the next sections : Aruco markers and Dot Pattern localization.

1.2 Camera : Field of view and calibration

The camera we will use is a Ximea MU9PC-MH : it has a native resolution of 2592x1944 pixels (pixel size 2.2 μm). We will activate 4 pixel binning to filter the noise and reduce the size of the images to be analyzed : the effective resolution is then 648x486 with a pixel size of 8.8 μm. The objective mounted on it gives the camera a focal length of 2.92 mm, an horizontal Field of View angle of 88.6 deg and a vertical Field of View angle of 72 deg. If the camera is 15 cms above the floor of the pool, this would result in a field of view on the floor of 293 mm x 220 mm and a projected pixel size in the field of view of 0.45 mm.

Since we are using the whole field of view of the camera (and not just its centre), we need to take into account the distortion effect. It's important to correct it as precisely as possible, in particular with the second localization method where we will need a submillimetric precision. For this project, we used the standard camera calibration program offered in openCV[1] with a 9x6 chessboard pattern (square size 24.7mm).

The calibration results of this program includes a camera matrix (a 3x3 matrix containing the focal length of the camera and its center) and distortion coefficients.

[1] available here: https://github.com/opencv/opencv/blob/master/samples/cpp/calibration.cpp

These results are essential for the Aruco detection. We will use the openCV undistort() function to correct the distorsion in our images using these calibration results.

1.3 Finding the right paper

As was mentioned previously, the pool will be prepared with the markers or the dot pattern placed on the pool's ground. The support for the printed markers or dot patterns has to meet a few important criteria. First, it has to drown and remain perfectly flat on the floor of the ground. The water must not distort the paper, because it would ruin the markers or the pattern. The second important issue is that, in addition to being waterproof, the plastic should not reflect light : light reflections would totally hide the marker or the pattern.

For these reasons, we selected a waterproof, matt, printable paper: the Xerox Premium Never Tear Matt White Cling Film (007R92059). This paper proved to resist correctly to long term immersion in the pool of the lab (2 days and 2 nights in a row) without ink running or support warping and to be sufficiently matt. I have not done multiple waterproof fatigue tests (multiple immersions and drying) but it should

2 Localization using Aruco markers

2.1 Introducing Aruco markers

Aruco markers are basically QR codes which enable to perform pose estimation. They were developped by Garrido-Jurado et al. in [1].

One marker consist in a small square, composed of black and white little squares. These black and white little squares encode bits which themselves encode a given marker ID. We can generate markers of different size (see figure 1). With the generation tools of Aruco markers, the size of the marker can be changed and also the size of its border (black bits surrounding the marker). In standard detection conditions, a one black bit border size is enough but the marker size is an important parameter.

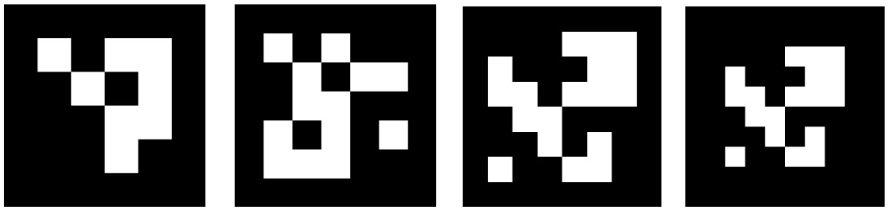


Figure 1: Aruco Markers of respective sizes 4, 5 and 6 (the fourth is the same as the third but with a border size of 2 black bits instead of 1).

Their detection is also achieved using built in functions of the Aruco module of openCV, an extensive open-source library of computer vision programs. The Aruco module of openCV provides all the necessary functions to generate and detect Aruco markers.

The localization and pose estimation is performed using the position of the marker's four corners in the image (knowing the real world length of the markers in and the camera calibration results): the inside bits only serve to encode the marker ID. The marker must remain square.

Our method will be the following. Before the run, we place markers of different IDs at precise positions in the pool and we load in the robot a map containing the markers IDs and positions in a user defined world frame of reference. The video stream of the camera will then be analysed, looking for these markers, using the Aruco module of openCV. For each marker detected in an image, a translation vector ($\vec{t}$) and a rotation vector (Rodrigues notation, which can be transformed into a rotation matrix R) are computed : they correspond to the position and attitude of the marker in the frame of reference of the camera. From these, we can compute the position and attitude of the camera in the frame of reference of the marker and use the markers map to compute the camera position in the user defined world frame ($\vec{pos\_wf}$). We suppose here that the markers are already aligned with the natural pool reference frame, since they are on its ground, so the last change is only a translation ($\vec{pos\_marker}$).

$$\vec{pos\_wf} = \vec{pos\_marker} - R^{-1} * \vec{t}. \tag{1}$$

To be useful, markers must then have different IDs and their IDs must be properly decoded. A higher marker size will allow a higher number of bits so a higher number

of marker IDs but also a higher distance between markers in terms of bit flips and so an improved robustness. Bigger markers will however occupy more space in the field of view.

IThe Aruco markers generation tool developped by Garrido-Juradoe et al. ([2]) provides an interesting feature. Custom "dictionaries" of Aruco markers can be generated, optimizing the distance in terms of bit flips between all the couples of markers included in this dictionary. A dictionary is a set of Aruco markers and is passed as input to the detection program which only looks for these particular markers. As a consequence, knowing how many markers one needs, one can build a custom dictionary of markers which perfectly suits the situation.

The detection parameters can also be adapted in order to improve the detection but also its computational efficiency. The black and white threshold to discriminate between black and white markers is quite a straightforward parameter. But there are also other parameters, such as the minimal and maximal size of one marker candidate : this helps to reduce the number of potential marker candidates which are then further analysed to determine if they are markers or not. In our case, the markers should more or less have the same size in the camera image (except if the camera has an important pitch or yaw).

2.2 From markers to boards

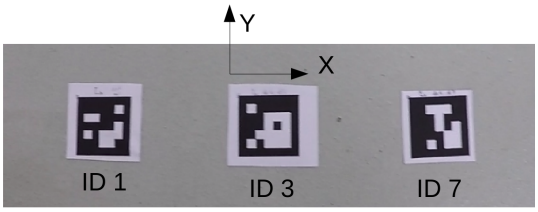
In theory, one Aruco marker is enough to compute the position and attitude of the camera, thanks to its 4 angles. However, experience showed that this gave unstable results. In particular, some Euler angles would repeatedly shift from one side to another. A stable computation of the attitude is required even if we only want to compute the camera's position in the world frame because the rotation matrix R is used to change frames (see equation 1).

As figure 2, the instability of the Euler angles computation has a significant impact on the computation of the position. For this simple test, the camera was moving above the markers (aligned along the X axis) with a rather stable Y and Z. On figure 2c, the yellow trajectory is the closest to the reality. The Y axis trajectory on figure 2d should be a flat line.

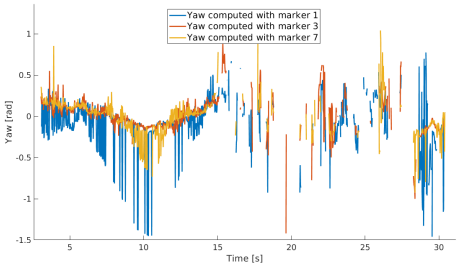
As figure 3 shows, changing the spatial organisation of the markers does not solve the issue.

Having more Aruco markers per frame is not a solution because the pose estimation provided in the Aruco module of openCV is performed for each marker, using only the information of this marker. To fuse the informations of multiple markers, the program would need to know the relative positions of these markers (even if in our application we could provide a marker's map to the program, but this is not a standard case). We could have modified the programs to implement it but fortunately, the Aruco module offers a native solution to this problem : the use of boards of Aruco markers (see Figure 4).

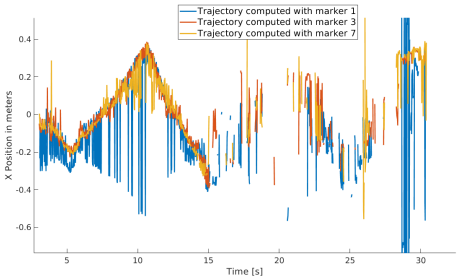
Aruco boards are just a set of markers, regularly arranged in a matrix shape. These boards are generated with a precise spacing between markers and marker side length, with markers from the same dictionary. The consequence is that the relative positions of the markers in real world units is known and sent to the program (as



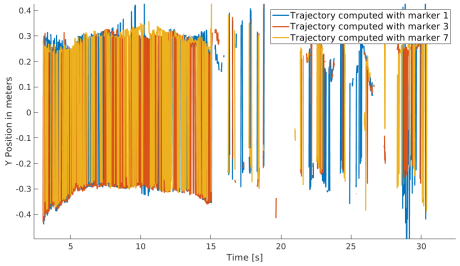
(a) Markers (side length (5cms), spacing between them 15 cm.



(b) Yaw of the camera according to each marker

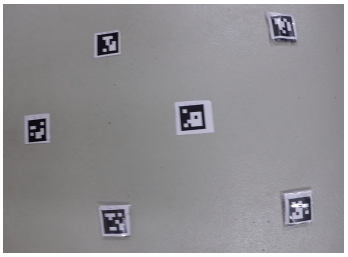


(c) X Position of the camera computed according to each marker

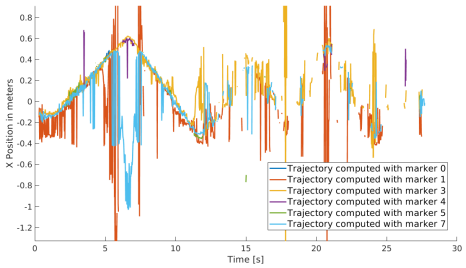


(d) Y Position of the camera computed according to each marker

Figure 2: Results of localization with single markers. The camera was held as stable as possible in Y and Z, and moving above the markers along the X axis.



(a) Marker configuration.



(b) X Position of the camera

Figure 3: Results of localization with single markers in a different configuration (the image is distorted because a GoPro was used before the Ximea camera at the beginning of the project but the distorsion is corrected by openCV). The camera was held as stable as possible in Y and Z, and moving above the markers along the X axis.

well as the mxn shape of the board and its first marker ID), which can fuse the informations from the detection of all the markers of the same board to compute the position and localization of the camera.

A nice feature of these boards is that the localization can also be done if one part of the board is occluded (even if the localization is better performed if all the markers of the board are detected). As long as at least one marker of the board is detected, a localization estimation can be done (see figure 5a with the board on the

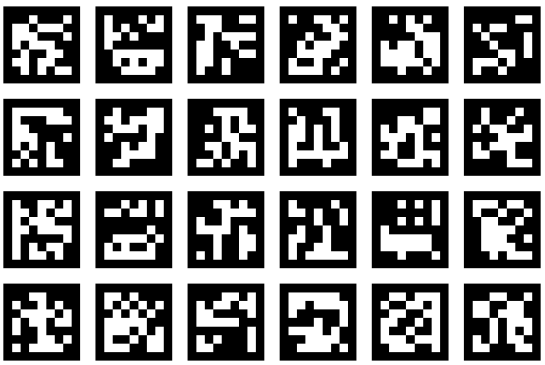


Figure 4: A board of 6x4 Aruco markers, generated with a custom 7x7 dictionary containing 10000 markers. The spacing between markers is 40 pixels and the marker side length is 200 pixels.

right).

Our localization principle will remain the same. We place the boards on the floor and load the map of the boards in the robot. Each frame is analyzed, looking for markers. Detected markers are regrouped into boards according to their IDs : board 0 would contain marker IDs from 0 to 23, board 1 markerIDs from 24 to 47 ... A board is considered detected if at least one marker belonging to it has been detected in this frame. Then, for each board detected, we can compute the position of the camera. For each frame, the position of the camera will be the average of the positions computed according to each detected board, weighted by the number of markers detected in this board (the more markers, the more reliable the computation).

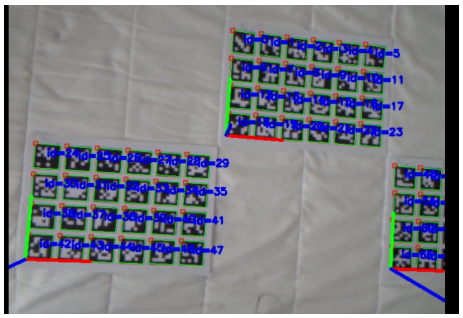
We verified with a few demo runs that using these boards effectively corrected the instability we had obtained with single markers.

2.3 Demo runs with Aruco Boards

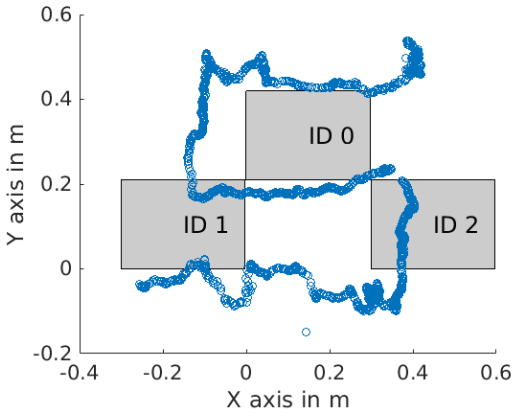
We printed three Aruco boards (6x4 markers) on A4 sheet (marker side length 39 mm, separation 7.8 mm, dictionary used : Aruco predefined dictionary of ID 6, which contains 250 markers of size 6). The first board contained marker IDs from 0 to 23, the second from 24 to 47 and the third from 48 to 71.

For the first run (see Figure 5, we made a skiw S-shaped trajectory above the boards, trying to be as stable as possible in Z and in terms of rotation : the camera was facing the boards with as little pitch, yaw and roll as possible. The results show that using the informations from all the boards we can reconstruct nicely the trajectory of the camera (5b), even if each board has its dead spots (5c. On Figure 5d, we can see that the boards "agree" on the camera's position and attitude and that the computation results are particularly stable.

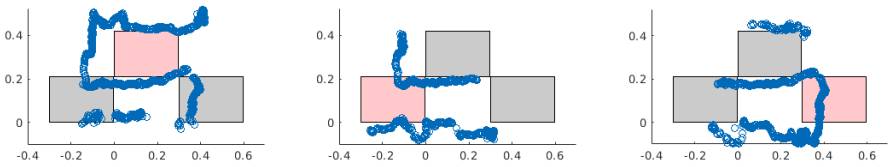
We then did a second run, with the same board disposition but going forward along Y, the results are reported on figure 6. As the Y position graph confirms (6c), we went much in the second part (0.12 m/s along the Y axis according to the detection results) than in the first part (0.044 m/s), yet the detection results



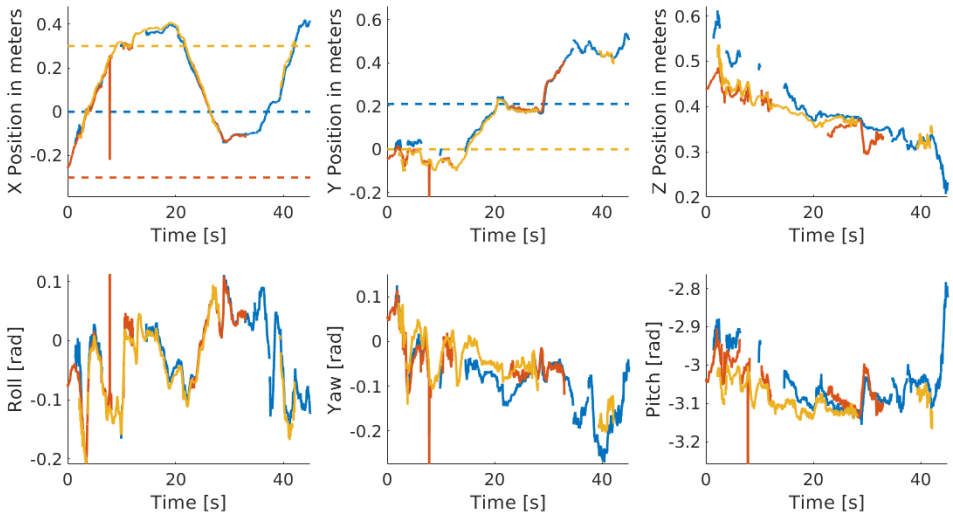
(a) Extract from the detection.



(b) Computed trajectory.



(c) Computed trajectory according to only one board (in red).

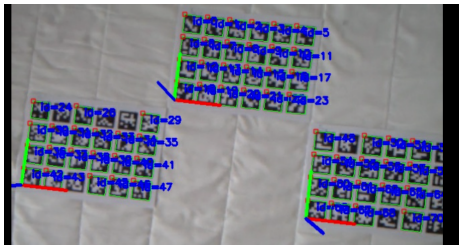


(d) Position and attitude : blue is according to board 0, red board 1 and yellow board 2 (the dashed lines are the board first marker's positions).

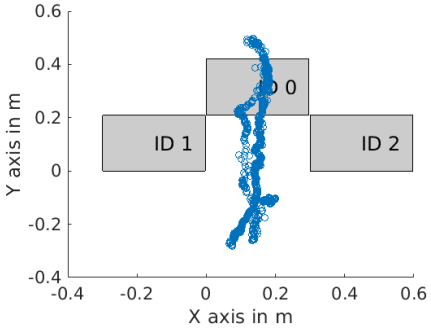
Figure 5: Results of localization with Aruco boards. The camera used is now the Ximea camera (30 fps, exposition time 30 ms), the mean detection time per frame (on my computer) is 7.8 ms.

remained clean, in terms of position and also of attitude.

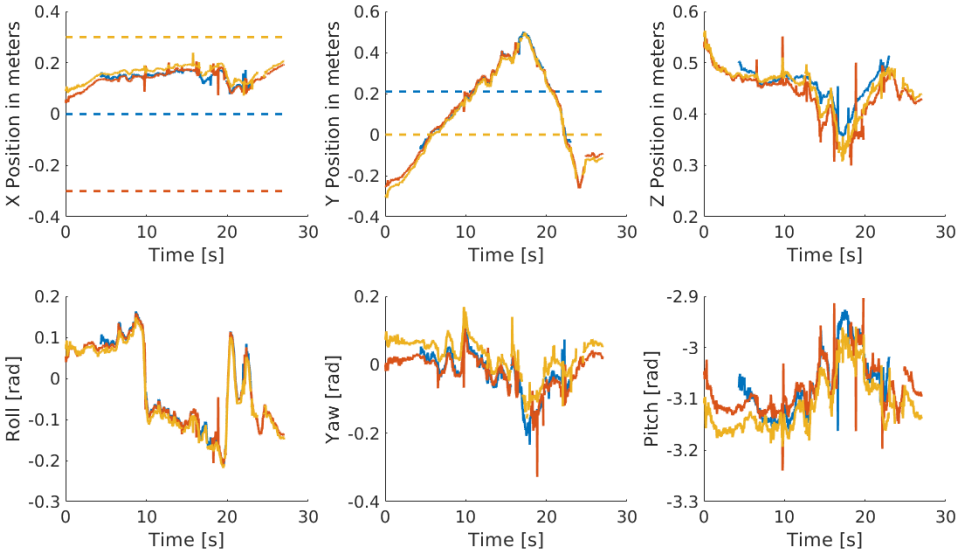
To test the system's robustness to perturbations in terms of rotation, we did a third demo run. The trajectory was the same S shaped trajectory as in 5b but we kept moving the camera around, adding rotation perturbations, in order to imitate what could possibly be the movements of the head of the lamprey robot where the camera will ultimately be. The results are reported Figure 7. We can see that there



(a) Extract from detection.



(b) Computed trajectory.



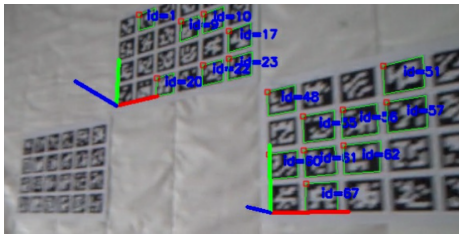
(c) Position attitude.

Figure 6: We made a forward backward trajectory along Y axis (as stable as possible in X and Z and also in terms of rotation), Ximea camera used (30 fps, exposition time 30 ms), mean detection time per frame 8.9 ms.

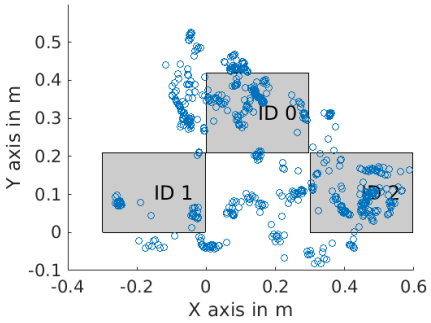
are many holes in the detection : frames where no markers were identified. With these perturbations, the motion blur (7a) is really an issue.

To confirm that the issue was the motion blur and not poor detection results when the camera is tilted, i did an ultimate demo run (see figure 8). The board organisation was the same, the camera was placed at the center, slowly rotating along the X axis (pitch angle, 8a) and then the Y axis (yaw angle, 8b).

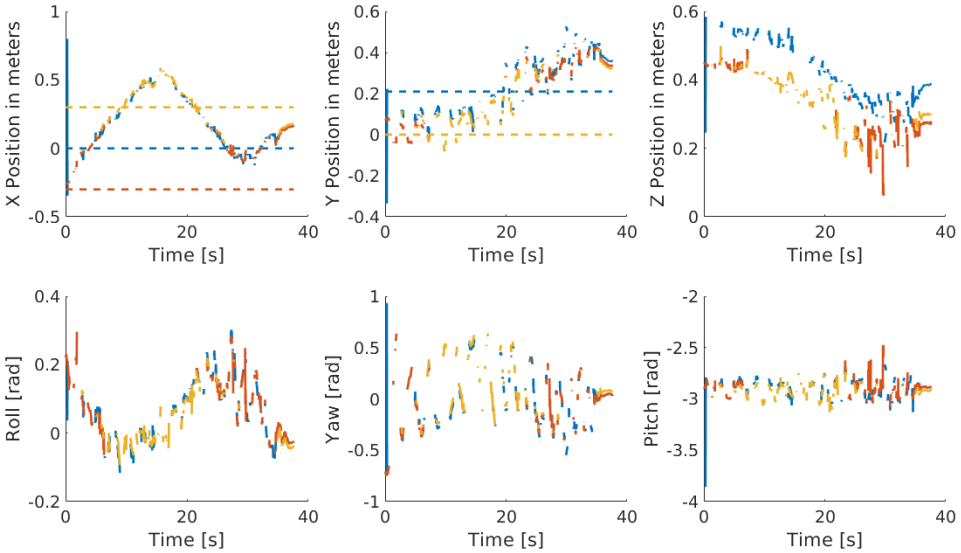
With these satisfying results, we propose to cover the floor of the pool with these boards of makers. The important parameters which have to be determined is the number of markers per board and the size of each marker.



(a) Extract from detection.

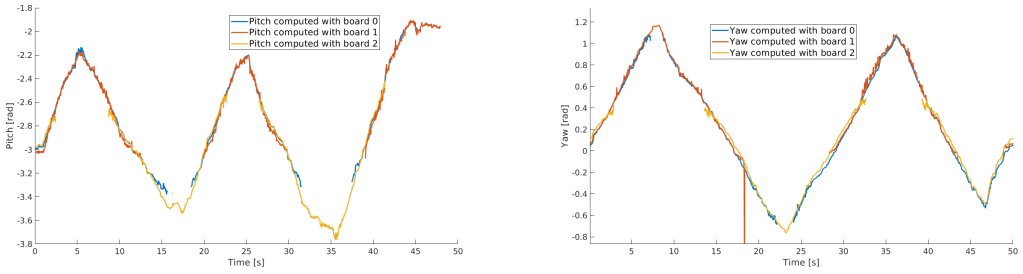


(b) Computed trajectory.



(c) Position attitude.

Figure 7: We made the same S shaped trajectory as in 5 but with important perturbations on the camera angles, Ximea camera used (30 fps, exposition time 30 ms), mean detection time per frame 7.4 ms.



(a) Pitch run.

(b) Yaw run.

Figure 8: Rotation runs. Ximea camera used (30 fps, exposition time 30 ms), mean detection time per frame 6.8 ms (pitch) and 6.9 ms (yaw).

2.4 Proposed parameters for the pool

We have to find a tradeoff on the number of markers per board and their size. The more markers per board, the more accurate and stable the localization obtained will be, but only if they are accurately detected (the marker extremities positions in the frame). The bigger the markers, the more accurate the detection will be. However, we have a limited field of view and resolution so we cannot use huge markers.

At a height of 15 cms, the field of view of our camera is 293mm x 220 mm : we will then use Aruco boards printed on A4 paper. The 6x4 Aruco marker disposition corresponds well to the dimensions of the paper and enables to make use of the most part of the paper (the markers have to remain square).

The boards would be placed next to each other, living no side

In the best positions, the camera will be centered right above one board and the program will use the informations from about 24 markers fused in one board to compute the localization. In the worst positions, the camera will be right above the angle : it could also detect up to 24 markers [2] from four different boards (top left, top right, bottom left, bottom right) so their information will be used separately.

The built-in Aruco dictionaries have a maximal size of 1000 markers, which would enable to make only 41 board and cover only $2,55m^2$. I generated a custom dictionary of 10000 markers (7x7 markers for improved robustness) which enables to generate 416 boards to cover $25,9m^2$.

2.5 Testing with Motion Capture

I tested this configuration using the Motion Capture system of the laboratory. I printed four boards (6x4 markers) from my custom dictionary (a dictionary cointaining 10 000 markers of size 7) on A4 sheets (marker side length 3.9 cm, separation 7.8mm). Board 0 contained markers from 0 to 23, board 1 from 24 to 47, board 2 from 48 to 71, board 3 from 72 to 95.

I tried to hold the camera 15 cms above the ground (water height) and did different runs. I placed four markers on the camera as shown on figure 9.

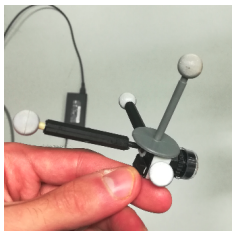


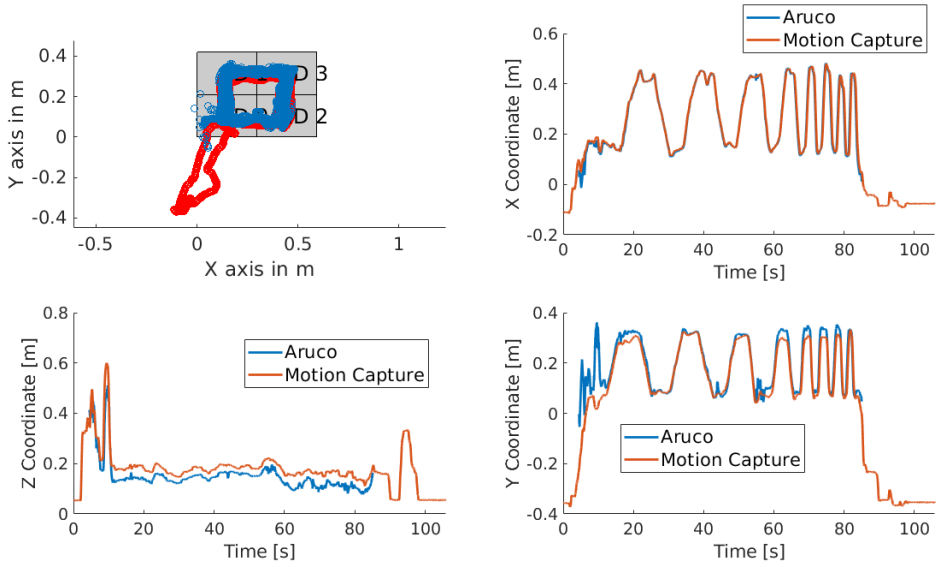
Figure 9: 4 markers for motion capture placed on the Ximea camera.

The Motion capture system ran at 120 fps and the Ximea camera was set at 30 fps (with an exposition time 30 ms). I synchronized the two by moving abruptly the camera along the Z axis twice at the beginning of the run (see figure 10a, Z

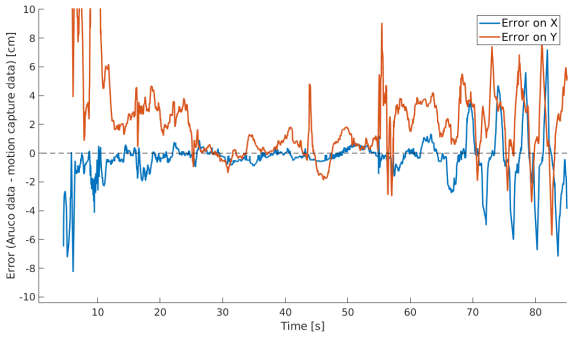
[2]Due to the margin of the printing and because the 6/4 ratio is not equal to the 29.7/21 ratio, their would be a bit more white space at the limits of the board than between markers from the same board.

Coordinate). I used the vertical peaks of the Z position on the records from the two sources to set the time shift between the Aruco estimated position and the one from the motion Capture.

I tried to align as much as possible the X,Y,Z axis of the motion capture system with the boards. The origins of the two coordinate systems were different : I just translated by (-5.5 cm; -6 cm) the X,Y position extracted from the motion capture record for it to match with the center of the Aruco boards. This is not the perfect transformation, so to get an estimate of the error, we can look at the amplitude of the error fluctuations rather than the mean value. Since I kept the boards at the same location for all the runs, I kept the same translation for all the runs.



(a) Stable run.



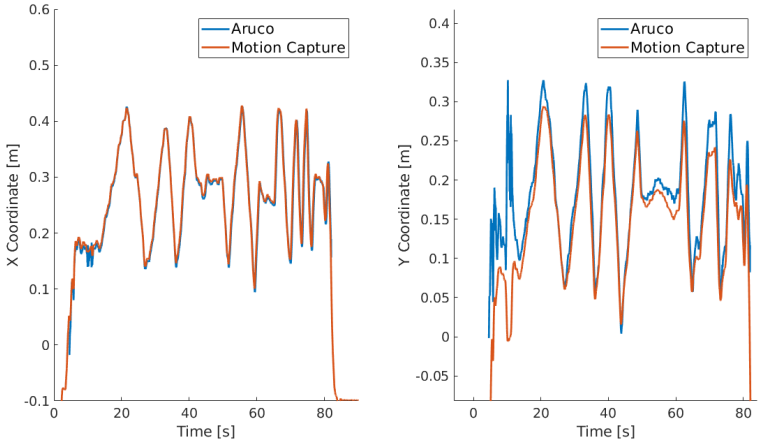
(b) Errors.

Figure 10: Run 1. Camera was as stable as possible in Z and in rotations, doing squares above the boards. Ximea camera used (30 fps, exposition time 30 ms). Mean detection time per frame 11.8 ms.

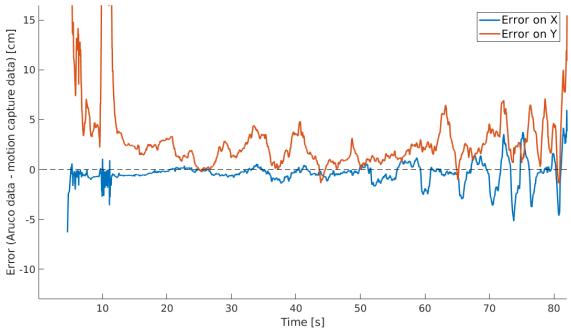
The first run was done in the easiest conditions possible for the Aruco detections : the camera was stable in Z, held just face to the Aruco boards, stable in terms of perturbations and additional rotations. The results are reported on Figure 10. From figure 10b, we see that the typical error on X when the camera is moving slowly

(about 2cms/sec) is less than one cm. The error on the Y axis is more important and this is a results that we also found in the other runs. It could be due to a misalignment of the Y axis of the board with respect to the Y axis of the motion capture system.

We also see that as the speed increases in the second part of the run (after 60 ms), the error increases. This could be due to a synchronization issue between the two systems. The motion capture system delivers timestamps but for the Aruco boards, I estimated time using only the framerate of the camera. The camera is not recording at a stable 30 fps. With the following runs, I had better synchronization results using a framerate of 29.8 fps to compute the timestamps.



(a) Stable run, moving along diagonal.

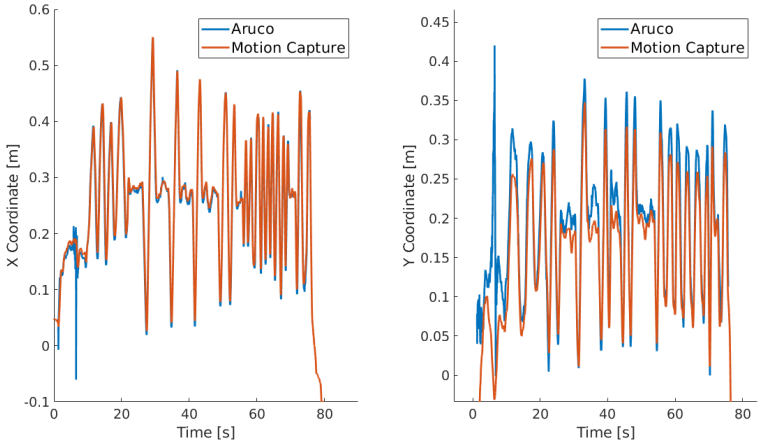


(b) Errors.

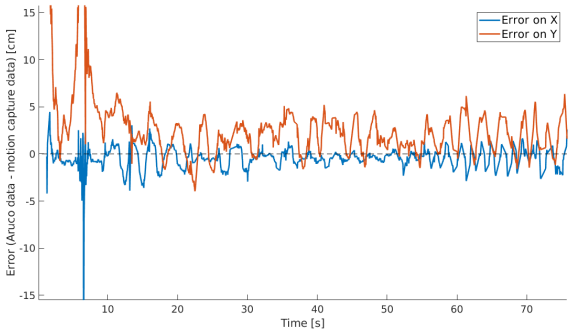
Figure 11: Run 2. Camera was as stable as possible, moving along the board edges and the diagonal. Ximea camera used (30 fps, exposition time 30 ms, but framerate set at 29.8 fps for timestamps estimation). Mean detection time per frame 13.3 ms.

I did a second run in a bit less favorable conditions (see results on Figure 11). The camera was not above the centers of the boards, but on the edges and on the diagonal. This run was to make sure that the position computed would still be accurate even if the camera is filming between boards and so using less markers per board (but more boards) to find its positions. The results are quite satisfying and resemble a lot to the previous ones (comparing visually figures 11b and 10b). We

still have this high speed effect, the error computed increases but it might be just a synchronization error.



(a) Slightly unstable run, moving along diagonal.



(b) Errors.

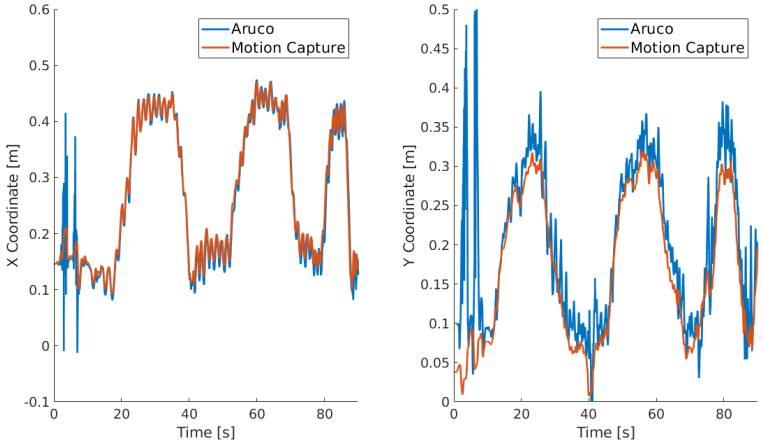
Figure 12: Run 3. Camera was stable in Z, slightly tilting along pitch and yaw, moving along the board edges and the diagonal. Ximea camera used (30 fps, exposition time 30 ms, but framerate set at 29.8 fps for timestamps estimation). Mean detection time per frame 11.8 ms.

Indeed, on figure 12b, the error is less important, whereas the speed is at least as important. For this run (see figure 12), the detection's robustness to slow perturbations on yaw and pitch was tested. The results are satisfying, there is no dramatic increase of the error on figure 12b.

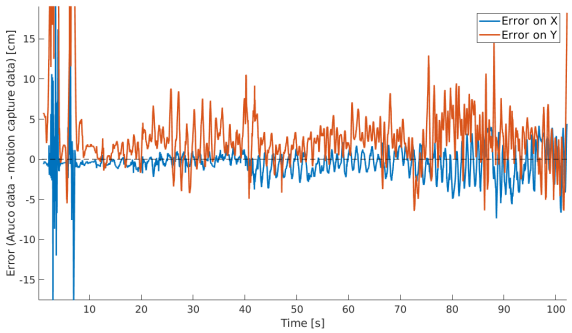
The last run (see figure 13) was the closest to the lamprey swimming conditions. The camera was moving along the X and Y axis at normal speed (not necessarily centered on the boards) and with fast tilting on yaw and pitch. The performances on the system (see figure 13b) are surprisingly good, even more if we take into account the fact that the time synchronization was not perfectly accurate.

2.6 Discussion

The Motion Capture results systematically showed an error more important on Y than on X but there is no immediate explanation for it. It could just be that the Y



(a) Slightly unstable run, moving along diagonal.



(b) Errors.

Figure 13: Run 4. Camera was stable in Z, fastly tilting along pitch and yaw, moving along the board edges and the diagonal. Ximea camera used (30 fps, exposition time 30 ms, but framerate set at 29.8 fps for timestamps estimation). Mean detection time per frame 11.5 ms.

axis of the motion capture was not perfectly aligned with the Y axis of the board. However, the fact that there is a very low error on X prove that the X axis were properly aligned, and given that we are using orthogonal axis and This could be looked more into depth. Likewise, I could not compare the Euler angles computed by the Motion Capture system and by the Aruco boards. This would have required me to precisely put the markers on the camera, to define properly the center of the camera body localization with respect to the markers in the motion capture system. I tried to do it, but the results are quite disappointing and the aim of this project is to perform only 2D localization of a robot and not its whole attitude.

The detection time per frames were in general slightly above 10 ms, but this was with my computer and not on board (NanoPi). However, given the good precision of this method and the update rate we want, we do not have to decode 30 frames per second. Furthermore, this detection time could be reduced by tuning more precisely the detector parameters of the Aruco markers (already implemented in the library) : we could for example set constraints on the maximal and minimal sizes of potential candidates for Aruco markers, knowing at what height will be the camera and the

size of the printed markers.

Overall, the results I obtained with the Motion Capture are quite promising. With the appropriate number of boards, we could easily scale up this method to cover a whole pool. If less precision is required, a sparser version could be implemented, with only a few boards at the pool's extremities.

3 Localization using dot patterns

3.1 Introducing dot pattern localization

The second localization technique (dot pattern localization) which we have investigated is less straightforward. It is commercially exploited by the Anoto company and is still patented in the US and in the UK. Anoto sells it as a solution to instantaneously stream on a numeric device what is written with a special pen on this type of paper. The paper has this dot pattern grid printed on it and the pen sold has a camera which enables localization of the tip of the pen.

For this project, we will use the libdots library developped by the Computer-Human Interaction in Learning and Instruction (CHILI) group of EPFL ([3]) to generate and decode this pattern. It can be used for academic purposes, is written in C++ and provides a dot pattern generation tool and a dot pattern detection program.

As the name suggests, it uses a pattern of black dots on a white background. These dots are organized in a grid and each dot is shifted in one direction with respect to the closest center (see Figure 14) Each dot is either shifted up, down, left or right : there are 4 positions possible, so each dot can encode two bits.

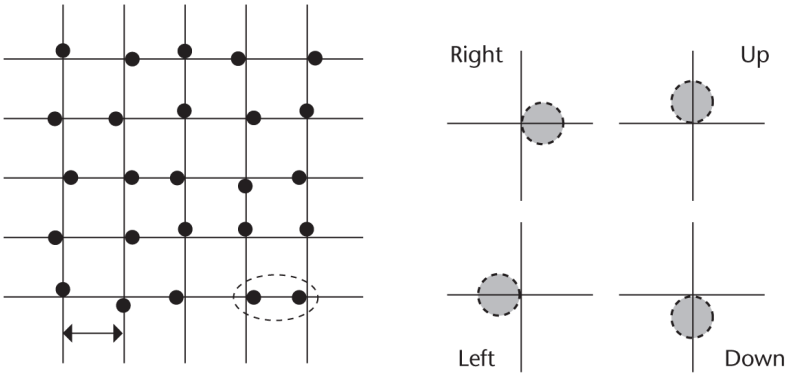


Figure 14: Figure taken from [4]. In [3], the grid spacing is 0.508 mm but we will scale this up for our application (roughly ten times, so 5.08 mm grid spacing).

The tricky part of the detection is to determine accurately this grid, because it is not printed and it is only the position of the dots with respect to this grid which encodes the information. If the grid estimation is wrong, even with properly detected dots, the localization will be wrong. In the article [3], they use a grid spacing of 0.508 mm (distance between two centers of the grid) and the dots are shifted by $0.508/6 = 0.085$ mm from these grid's centers. Even if we will use a bigger grid spacing, this already gives an order of the magnitude of the precision which needs to be achieved for proper localization.

A complete grid estimation means determining its two generating vectors $\vec{u}$ and $\vec{v}$ but also an origin for the grid. With the orthogonal dot pattern we use, $\vec{u}$ and $\vec{v}$ are orthogonal and their length is equal to the grid spacing (Anoto has also developped triangular and hexagonal dot pattern grids).

3.2 Detection using the libdots library : Principle

In the libdots library, the detection is performed as follows (explained in details in [3]).

The image is thresholded (the threshold can be adapted to the conditions) to get a binary black and white image and detect the dots. Then each dot is connected to its four closest neighbours : if both dots agree to be connected, their connection is called a symmetric edge.

A good first estimation for $\vec{u}$ and $\vec{v}$ is obtained by clustering these symmetric edges into 4 clusters of vectors constrained to be symmetric around the origin (because the symmetric edges can be in opposite directions). $\vec{u}$ and $\vec{v}$ are then two orthogonal vectors chosen among the means of each cluster obtained.

The first estimation of the origin is done by looking a suitable 3x3 dot region. This region must contain a center dot (starting dot) whose 4 neighbors (called side dots) are symmetrically connected to it while the 4 other dots (called corner dots) must be symmetrically connected to 2 side dots : this is a 3x3 region where all dots are symmetrically connected with their closest neighbours of the region. The first guess of the origin $\vec{o}$ is the weighted average of the starting dot and the side dot position.

Then all the detected dots positions $\vec{p}$ are transformed to grid coordinates $\vec{g}$, rounding their to the closest integer.

$$\vec{c} = \begin{pmatrix} \vec{u} \cdot (\vec{p} - \vec{o})/(\vec{u} \cdot \vec{u}) \\ \vec{v} \cdot (\vec{p} - \vec{o})/(\vec{v} \cdot \vec{v}) \end{pmatrix}$$

$$\vec{g} = \lfloor \vec{c} \rceil$$

$$\vec{\delta} = \vec{c} - \vec{g}$$

Then the grid origin can be refined $\vec{o}$ by substracting it the median value of $\vec{\delta}$.

For each detected dot, a likelihood to be up, down, left or right is computed given the detected position of the dot and the grid estimation. To decode the X and Y position of the camera, the library needs an 8x8 dot region. The best region (which will be used for decoding the position) is selected by finding the 8x8 area where these likelihoods are maximal. The program knows then the position of the upper angle of this 8x8 region in grid coordinates.

It is worth noting that the program can reach subgrid precision, as long as the camera's resolution is higher than the grid spacing, which is a required condition for the detection to work. The program computes the position in grid units of the upper angle of the best 8x8 decoding region found in the frame. Then it adds to this position the translation vector to the central pixel of the image (which corresponds to the position of the camera). This translation vector is known in pixel units, which has a higher precison than grid unit.

3.3 Detection using the libdots library : Results

I had a few troubles installing and compiling the libdots library and tool. I eventually managed using Ubuntu 16.04.4 LTS (64 bits), libdots-1.12 (last version available) and podofo-0.9.5 and freetype-2.5.2 for the dependencies. To compile the generate

pattern, the dependency flags have to be put in this order -lfreetype -lpodofo -lfontconfig -ljpeg -lz -lpthread -lssl -lcrypto (the order mattered on my computer, I had errors with different orders).

On figure 15, I represented the results of the decoding performed with the libdots tool on a pattern generated with it. The blue square is the 8x8 dot area used for decoding. The color of the printed dots are scaled from green to red according to the likelihood of the decoded position (green means that the program is quite confident on the detected position of the dot, red means it is highly unlikely but still the best choise).

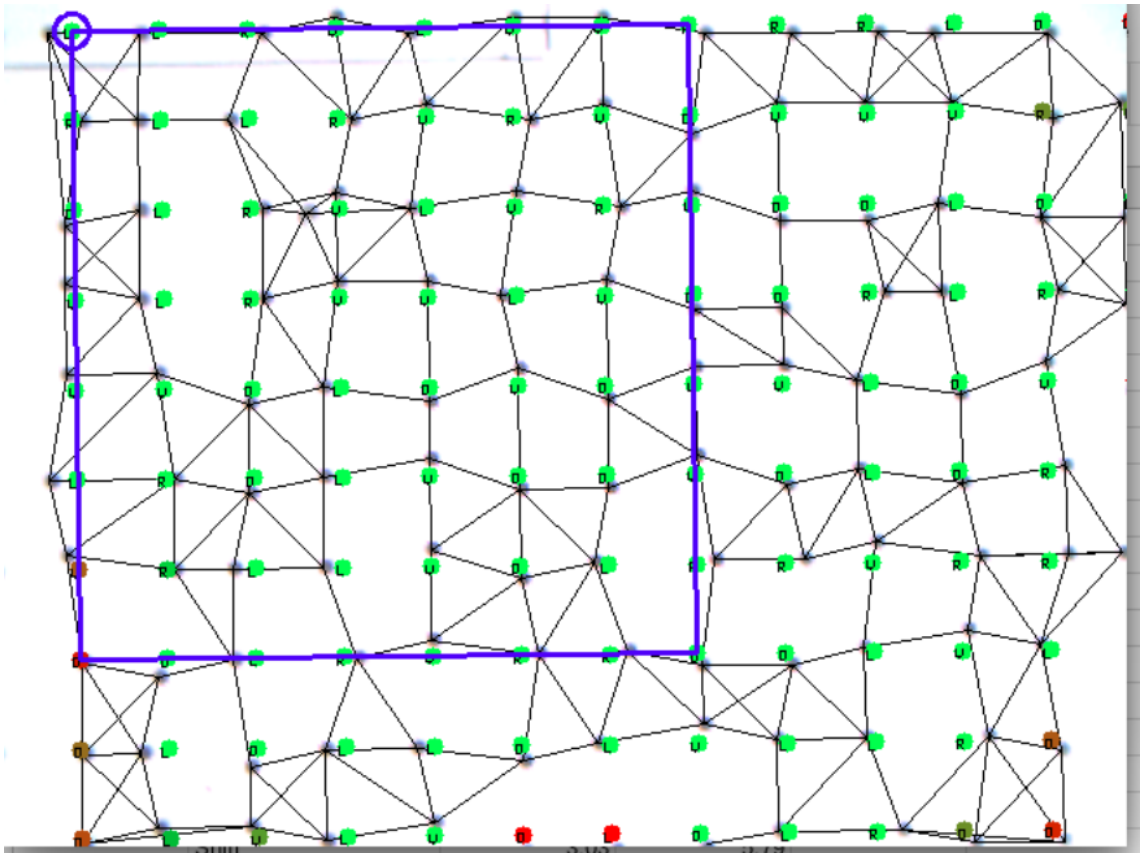


Figure 15: Results of dot pattern detection. The grid spacing used here is 5.08 mm and the camera was held 10 cms above the pattern, stable and parallel to the sheet.

Once I corrected the position shift that was added in libdots (a residual shift from the camera position to the center of the robot they used), I could check that the decoded position corresponded to the position of the camera. I estimated the position of the camera using the center of the image used for detection and measured on the pattern that it corresponded to the decoded position.

However, this detection is still subject to a considerable limit : the camera has to be perfectly parallel to the dot pattern (details are given in section 3.6).

3.4 Optimal parameters selection: Grid size

With the libdots library, a minimal 8x8 dot pattern size has to be in the field of view of the camera.

A tradeoff has to be found between a larger grid spacing (which means bigger dots, easier to detect) and enough dots in the field of view of the camera. The libdots generation tool offers to set any grid spacing.

In the worst case, the diagonal of the 8X8 dot pattern area required to appear fully in the field of view of the camera is in the vertical direction of the field of view.

	Article	Scaling x10	Scaling x20	Scaling x30
Camera Resolution	188 x 120		648 x 486	
Camera distance (mm)	12,7		150	
Camera vertical angle °	30		72	
Distance between 2 dots (mm)	0,508	5,1	10,2	15,2
Dot size (diameter in mm)	0,085	0,85	1,69	2,54
Projected Pixel physical size (mm)	0,057		0,448	
Max distance (8x8 dot matrix) (mm)	5,39	53,9	107,8	161,6
FOV vertical size (mm)	6,81		218,0	

Figure 16: The article referred to is [3].

From Figure 16, it seems that the best trade-off is using a grid spacing of about 10 mm. There is a 4 pixel binning activated on the Ximea camera (the projected pixel physical size is actually 4x4 pixels of the sensor), which should reduce the noise. The field of view given does not take into account the distorsion effects which reduce the useful field of view, even if the distorsion on the Ximea camera is low and can be reduced with openCV's undistort().

3.5 Detection using the libdots library : Scaling up to multiple papers

The first issue with this method is the scaling to multiple papers : how can we cover an area bigger than just an A4 or A3 sheet ?

The libdots pattern generator offers different solutions. The first one is breaking a bigger pattern into many pages. We can generate a mxn matrix of A4 pages with matching patterns. If it is then printed on different A4 sheets which are stuck next to each other properly, this could be a solution. However, I had resizing and then margin issues with the printers that I did not manage to solve. Some lines of the pattern can be cut by the printer but it must be printed perfectly centered on the page. If there is a slight shift, then the grid of two different sheets will not match (even if the pattern on each sheet taken separately can be decoded). The issue then is that when the camera is above the edge of two sheets, the program will have only half the image to decode a position, using alternatively the grid of one sheet or the other, and most likely not enough dots to decode.

The libdots generating pattern tool also offers to set an offset on the starting position (X and Y position of the point at the top left of the page). The hard way would be to set the offset so that it would match perfectly the real world's position of the pattern.

A more flexible solution could be similar to what was done with the Aruco boards. The dot pattern encode so many different positions that we could set a mapping between a given position in terms of grid unit with a real world position. The map would define the real world position and the grid units position of the topleft dot of each A4 sheet pattern. Of course the offset (on X and Y) in terms of grid unit between each sheet would have to be bigger than the number of positions (on X and Y) encoded in each sheet.

Then when a position in terms of grid units is decoded, we look for the A4 sheet to which it corresponds ; it is unique as long as the offset has been properly set. Then we substract from the detected position (grid unit) the position of this A4 sheet origin (grid unit). This translation from the origin of the paper can be converted into real world units knowing the grid spacing. Thanks to the map, we can add to this the position in real world units of the origin of the A4 sheet.

3.6 Future work on dot pattern localization : Investigating a new grid decoding technique

The main limit to the detection as it is implemented by libdots is that the camera has to be perfectly parallel above the dot pattern printed sheet. With any roll angle the detection will work, but small angles of pitch or yaw prevent any successful detection.

This is because there is no perspective correction in their method. With slight angles of pitch and yaw, the projected image of the dot pattern grid on the camera's sensor is no longer an orthogonal grid and the projection fails.

The method used by Anoto is described in their patent WO2001075783 (see https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2001075783¿ for the integral publication of the patent in English[3]).

The dots are detected the same way as in libdots (threshold and then binary image). However, the grid estimation is done differently. The dots are not connected to each other to cluster the main directions. They use a 2D spatial Fourier transform along the two dimensions of the image pixels. The two main peak positions in the resulting Fourier transform function give the length and the orientation of the two vectors generating the grid deformed by perspective. As a consequence, these vectors are not necesarily orthogonal and of the same length. Then an image transformation which transform these two vectors into orthogonal vectors of the same length has to be found (the condition on the output vectors can be different if a not orthogonal pattern is used). The patent does not mention how it is found.

We could use the getPerspectiveTransform() function of openCV to find this correct image transformation. Once found, it could be applied to the whole image to correct the effect of perspective and the obtained image could be decoded by the libdots program.

This function computes the perspective transformation transforming 4 points of a source image into 4 points of a destination image. In our case, the 4 points of the source image would be the 4 angles of the parallelogram defined by the two vectors

[3]Google Patents also has it, but the character recognition performed on the patent text has a few misleading errors

of the perspective deformed grid. The difficulty lies in finding the 4 points of the destination image. Since the output vectors have to be orthogonal and of the same length, they will be the 4 angles of a square whose side length remains unknown because we do not know the required length of the output vectors.

Their length could be estimated using the know grid spacing of the printed pattern and an estimate of the camera's distance to it. We want to use this localization technique at a known height (the water level) and we could neglect the contribution to distance of the yaw and pitch rotations. However, this would mean that the relative fluctuations of the robot's height have to remain low.

I did not have the time to implement and test these methods. openCV also offers other functions to find the right homography (transformation between two planes, in our case the perspective deformed grid and the original grid). This remains very delicate because any slight error on the perspective transformation will give ill-positioned points and a bad decoding.

Conclusion

I have not had the opportunity to test the Aruco method underwater because the waterproof head for the Ximea camera was not ready. Given the results of the motion capture, it should work nicely given the needs on precision and update rate, even if the computation on the nanoPi could be much slower.

The dot pattern localization of libdots is way less robust and cannot be used for the moment in swimming robots with an unstable camera in terms of pitch and yaw. Perspective transformation's functions from openCV could correct this.

In both case, motion blur compensation could help improve the detection but I have not investigated it.

References

[1] S. Garrido-Jurado, R. Mu noz Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280 – 2292, 2014.

[2] S. Garrido-Jurado, R. Mu noz Salinas, F.J. Madrid-Cuevas, and R. Medina-Carnicer. Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recognition*, 51:481 – 491, 2016.

[3] Lukas Oliver Hostettler, Ayberk Ozgur, Séverin Lemaignan, Pierre Dillenbourg, and Francesco Mondada. Real-time high-accuracy 2d localization with structured patterns. *Proceedings of the 2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8. 4536 – 4543, 2016.

[4] K. Schreiner. Uniting the paper and digital worlds. *IEEE Computer Graphics and Applications*, 28:6–10, 11 2008.