Autonomous vision based docking of roombots

Häfliger Adan

June 12, 2016

Contents

1 Project Goal

The main purpose of this project is to use a camera and markers to make a Roombot metamodule dock autonomously to a passive part. Using markers on the passive part, we can figure out the position of the camera by doing extrinsic camera calibration.
Then we know where the Roombot metamodule is so we can try docking manoeuvres. Repeating multiple times the calibration and manoeuvres can lead to docking without needing to be on grid.

2 Introduction

2.1 General Motivation

Autonomous docking is basically a way to make Roombots modules able to grip to each other regardless of position and grid-alignment.
Docking of modular robot is essential to achieve the level of adaptability that we expect from them. My project looks at a camera and marker set up to see how it performs and if it could be used for this task.

2.2 Unexpected troubles

Initially, we used an independent small computer to process the image but we had unexpected incompatibility between the computer's processor and the camera drivers (Raspberrypi Zero's ARMv6 isn't compatible but ARMv7 is). We found no other small enough computer to replace the original one so we only use an operator PC right now. The delay from trying to fix this issue (and others issues . . .) made me not have enough time to try docking maneuvers with the metamodule.

2.3 Tasks abstract

Some of the work done isn't used in the end because of the mentioned troubles above but here is a tentative plan of what was done.

1. Hardware setup and integration
2. Tools for future users
3. Network Architecture
4. Homography based extrinsic camera calibration
5. Experiment
6. Performance review

2.4 Project Setup

Before talking about the project requirements I want to show the initial project setup.

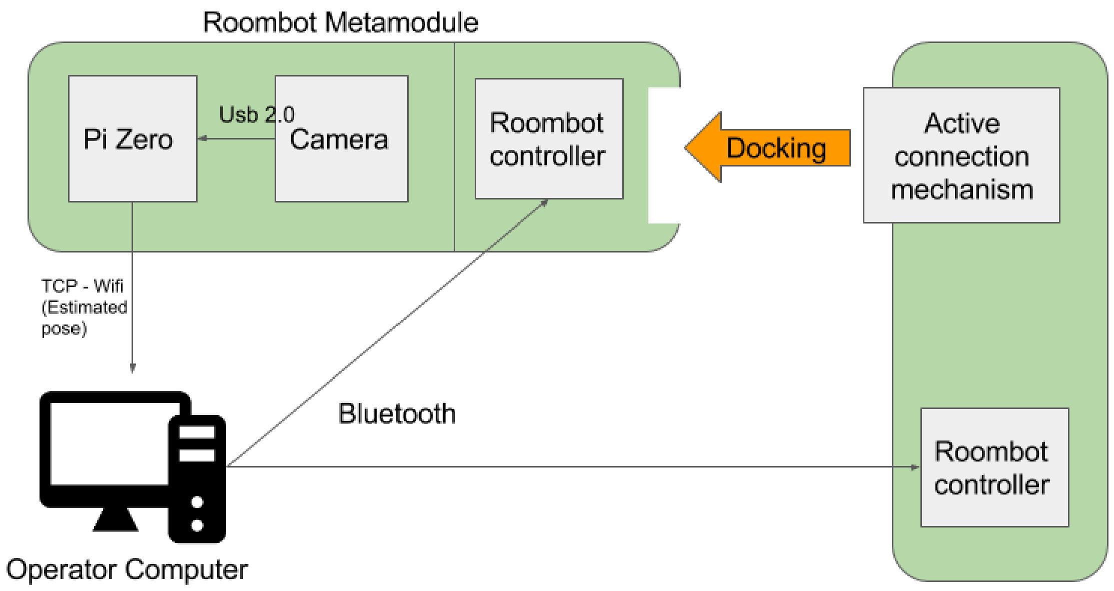

Figure 1: Roombot Setup

Because of unexpected hardware incompatibility between the camera and the RaspberryPi Zero's processor, we are not using an independent computer inside the roombot but the camera is connected to the operator computer through usb but this set up is still valid for future compatible mini-computer.

3 Project Requirements and Specifications

3.1 Visual Range

The visual range is a constraint for the docking servoing range. Of course the longer the range, the more flexible our docking system is. For example, if we could make two Roombots meta-module dock within a distance of 1 meter, it's better than 50 centimetre.
But there are many challenges with cameras. Since we do not have a "dynamic lens" system, the settings for optimal vision needs to target the closer range since it is the hardest and important part. Therefore we set up nice settings for closer range (10 cm to 1cm).
Another point is that we know the "rough" coordinates of the meta-modules in the simulation, so we can safely get relatively close before needing precise coordinates for docking.

3.2 Mini Computer

The Raspberry Pi Zero was a nice trade-off between size and performance. It has a mini-usb port that we can use for our usb camera and an usb wifi dongle. Sadly, the ARMv6 processor isn't compatible with the camera linux drivers that need ARMv7. So we didn't use the board in the end. We propose alternatives later.

3.3 Camera

The camera used is a MU9PC-MH USB 2.0 Camera by Ximea which is 5 x 15 x 8 millimeters. It is a scientific grade camera that can take variable lenses which makes it easier to get to our application range and given its size, it can be used inside roombots. It is also has modifiable camera parameters (exposure time, gain, ...) which can help getting good images for certain application and distance.
The camera lens is fixed with a lock ring.

3.4 Markers

For markers detection we want to try different markers and see which one gives us the best results.
We will test three different Marker configuration with 16 coordinates. 4 Aruco Markers, 16 Blob and a checkerboard.
Here is the layout defining object points:

We need to have small markers that can fit above the docking region of the ACM.

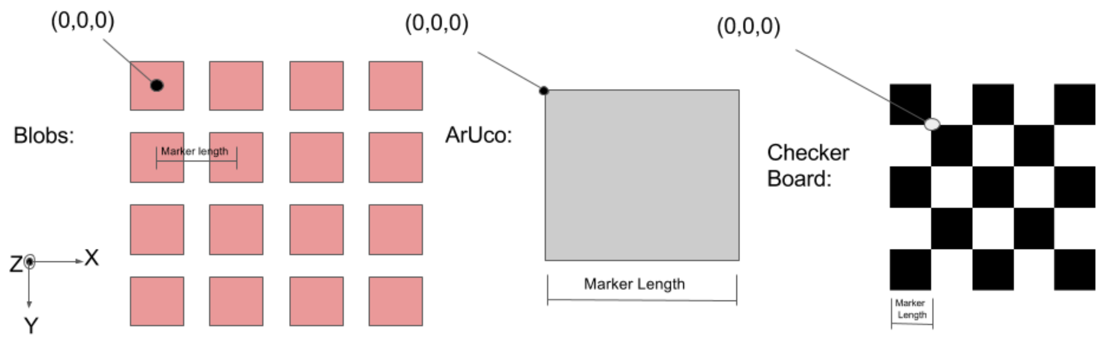

Figure 2: Marker configuration

3.5 Roombot Metamodule

We wanted to use two roombot module connected to a passive part. The metamodule formed is 44 cm long if straight, and will be able to move far enough for our application range. We want to dock within 20 cm of a passive part with markers. It also possesses 6 degree of freedom . . .

4 Main Tasks

4.1 Yocto Project Kernel

4.1.1 Purpose

Since the mini computer inside Roombot is susceptible to power shortage, we want a kernel that doesn't get corrupted if the power goes down during processing. One solution is to build a minimal Linux system compatible with the mini computer using the Yocto Project. Some key features are read only system, fast boot time and easy installation and set up.

4.1.2 Implementation

Since there are good resources and documentation it wasn't to hard to build a Linux based kernel. Using the meta-rpi layer and tutorials, we built a working kernel for the Raspberry Pi Zero. It is built using bitbake tool from the console-image recipe from meta-rpi. The only changes made to the image was to add OpenCV dependencies that are already built within Yocto. The other libraries were to be manually installed through the network using SSH.

After building and installing the kernel into the SD card, we can set up the board to automatically connect to a WIFI network. We don't have to plug it in into a screen to set up the wifi configuration, since we can directly change the files. Here is a sample of the changes for wireless connection: In wpa_supplicant.conf:

```
network={
ssid="mutlunet"
proto=WPA
scan_ssid=1
         key_mgmt=WPA-PSK
psk=cc1dab3bfdf30a24c8d48334e4d2c971e83f67b30a27831820b7b99f55201c90
}
```

In interfaces:

```
iface wlan0 inet static
wireless_mode managed
wireless_essid any
wpa-driver wext
address 192.168.1.110 #Fixed ip for each metamodule
netmask 255.255.255.0
gateway 192.168.1.1
wpa-conf /etc/wpa_sup
```

Using an USB dongle, we can SHH into the board and further set it up. More details of how to build the image won't be reported here but this tutorial covers the subject: Jumpnowtek Tutorial

4.1.3 Conclusion

In the end it was working quite well but the incompatibility with the camera made us leave this approach although the things learnt can be used for another mini-computer later on. Also quite some time was lost trying to make the camera drivers work. We thought that it was a problem with my image of Linux at first, and we only confirmed the problem when I tried the official RasbperryPi Zero Image. Then we tried to contact Ximea support to see if they could assist in anyway or what we could do to solve our issues, but support for ARMv6 is not planned and they said to try different board. Since the drivers files are pre-compiled, we can't really see if it was possible for us to fix it.

4.2 Tools for future users

We provide a Virtual Machine set up with howto's and scripts such that future user of the Pi Zero can modify and make new images with the needed packages and can easily understand how it works. (Although since we don't rely on the Raspberry Pi Zero it is not useful anymore).

4.3 Network Architecture

We used static IP and client server architecture to send message through TCP between the mini computer and the operator PC. This was running well in the Pi Zero, but it is not used in the end. The client/server code was based on this tutorial.

4.4 Camera Extrinsic Calibration

4.4.1 Camera Calibration Theory

The purpose of camera calibration is to find the relationship between spatial coordinate of a point in the world and the associated point in the image. Since our camera follows the pin-hole image formation model we have a simple and linear relationship (in homogeneous coordinate): $\begin{pmatrix} su \\ sv \\ s \end{pmatrix} =$

$$\begin{bmatrix} k_u & 0 & c_u \\ 0 & k_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} & \mathbb{R}^{3\times3} & & \begin{matrix} t_x \\ t_y \\ t_z \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

s_u, s_v, s : image plan coordinate
X, Y, Z : object world coordinate
Intrinsic parameters:
k_u and k_c : image enlargement factor
f : focal length
c_u and c_v : optical centre coordinate
Extrinsic parameters:
$\mathbb{R}_{3\times3}$: Rotation matrix from world coordinate to camera coordinate
t_x, t_y, t_z : Translation vector from world coordinate to camera coordinate

Thereby by detecting specific point in the image that we laid out at known position, we can retrieve the position of the camera, relative to the object coordinate of those points. In our case, knowing the position of the camera implies that we know the position of the robot and that can help for docking.

4.4.2 Intrinsic calibration

We first use a checker board pattern to figure out the intrinsic parameters. We use OpenCv's camera calibration sample to generate an YML file. This also helps against lens distortion based on the checker board distortion in the image, OpenCv can figure out the intrinsic parameter mentioned above.

4.4.3 Marker detection

To find specific point in the image, we use markers, We wanted to try different markers and see how good they perform.
To detect the blob Marker, we use the OpenCV SimpleBlobDetector class. To detect the ArUco marker, we use ArUco marker detection and to detect the chessboard pattern, we use OpenCv findChessboardCorners function. We hope to have good performance with every marker. Also the nice thing about ArUco marker over the other two is that it provides useful functionally integrated with the library, like id per marker, detection of point is in order, etc... Also we think that the blob library should be more robust overall.

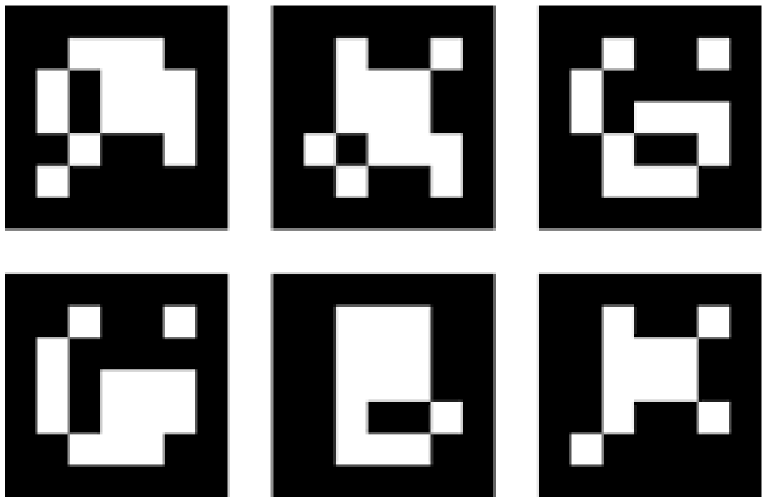

Figure 3: ArUco Marker Examples

4.4.4 OpenCv C++ implementation

Here I want to give some glimpse of the OpenCv code.

```
undistort(imageIn, imageOut, TheCameraParameters.CameraMatrix,
                  TheCameraParameters.Distorsion);
```

Is used to distort the image based on the image distortion coefficient found in the intrinsic calibration process before.

```
solvePnP(modelPoints, corners, _cameraMatrix, _distCoeffs, rvecs, tvecs);
```

Is the OpenCv function that return the rotation and transformation extrinsic parameters mentioned before. It is the pose of the object relative to the camera coordinates. Note that the rvecs variable only has the 3 polar angle which we can use to find the $\mathbb{R}_{3\times3}$ matrix. Also *modelPoints* contains the real world coordinate that we know of the points. Corners is the measured point coordinate in the undistorted image. We need to figure out the camera pose from these vectors.
To convert from rvecs to yaw pitch roll angles, we use

```
DecomposeProjectionMatrix(...);
```

For blobs detection we do some color filtering of the image in the hsv domain and some image processing, erosion and dilation, here are key samples:

```
cv::cvtColor(undistortedImage, hsv, cv::COLOR_BGR2HSV);
// Color filtering for binary mask
inRange(hsv, cv::Scalar(0, 100, 20), cv::Scalar(15, 255, 255), mask);
...
int morph_size = 3;
Mat element = cv::getStructuringElement( MORPH_RECT,
        Size( 2*morph_size + 1, 2*morph_size+1 ),
                Point( morph_size, morph_size ) );
cv::erode(...);
cv:dilate(...);
//Inverse black and white
cv::bitwise_not(...);
```

Also with the blobs, we need to associate each marker to each corresponding object points. In order to solve this problem, we first find the corners by measuring the distance in pixel of each marker to the corner of the image, then when we found the corners, we can linearly interpolate between two markers to find the markers within the corners and we use an incrementing error such that we always find corresponding marker, the shorthand of this method is that it only works at angles below 90 degrees. In our case since the ACM is symmetrical around both axis, we don't have to test these rotated angles anyway.

4.5 Experiment

What we want to do is to characterize the camera/marker system and see how well it estimates the camera pose at certain distances and angles.
For the experiment, we use the optical motion capture system with the Motive software. We set up 3 optical marker for the camera and for the marker, to detect their position and orientation. We will consider these as groundtruth and compare the estimated values from our application.

5 Results

5.1 Initial bias

Because we have to manually set up the real position of the camera and markers relative to the optical markers, there is already, at the start, an error that is not accounted for in the groundtruth. Careful positioning was tried, but it's not perfect. Also the yaw, pitch and roll angles measured for each optical marker is relative to the initial position when we declare the markers as rigidbodies. Therefore the initial rotation is considered to be (0,0,0) but in reality, there are a few degrees of misalignment because of how our camera and markers are mounted.

5.2 Motion capture system issues

To detect the optical marker, the system uses LED that emits invisible light but this emission affects the camera greatly. To overcome this, we turn off

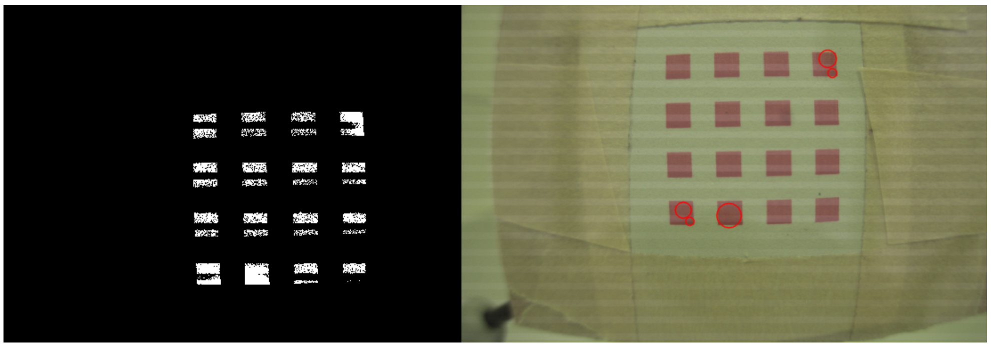

Figure 4: LED interference

or reduce the LED power in the software. Sometimes, after losing track of object and re-tracking it, the software is tricked into thinking there are new optical markers exactly on top of old ones and it can also further reduce down the groundtruth precision. Mean marker error can go up to 0.60mm/marker so it has an impact.

Also the position from which we do measurement isn't fixed for each marker so there is non-characterized unfairness between markers based on where I decided to the measurement. This could have been better by using indicators on the ground.

5.3 Configuration and Example

For each marker, we decided to test 3 coordinates with 2 different roll and pitch which gives us 12 points. The general pattern is: Front - Front rolled - Left - Left rolled - Right - Right rolled - Front pitched - Front pitched rolled - Left pitched - Left pitched rolled - Right pitched - Right pitched rolled.

That means that the first measure is in front of the camera, the second one is in front but with Z angle to the left of the camera and so on.

For each orientation that we want to test, the system does 20 measurement and uses the mean values for comparison. Example:

```
MEASURE1 Point 0 :
yaw = -7.66421 pitch = -13.7201 roll = 1.86153
tx = -2.49576 ty = -3.12575 tz = 56.8859
. . .
. . .
Point 19 :
yaw = -7.68444 pitch = -13.6558 roll = 1.85858
tx = -2.49445 ty = -3.12686 tz = 56.885
==============
pitch mean = -13.2663 std = 0.633149
yaw mean = -7.33589 std = 0.583569
roll mean = 1.77032 std = 0.139898
tx mean = -2.49367 std = 0.00346351
ty mean = -3.12675 std = 0.00138608
tz mean = 56.8875 std = 0.0345895
```

Also the lighting was adjusted for certain angles to make sure detection was good, this is not really life robustness but we are comparing marker precision not robustness.

I'm also showing an example of image detection to better show what these measurements are based off.

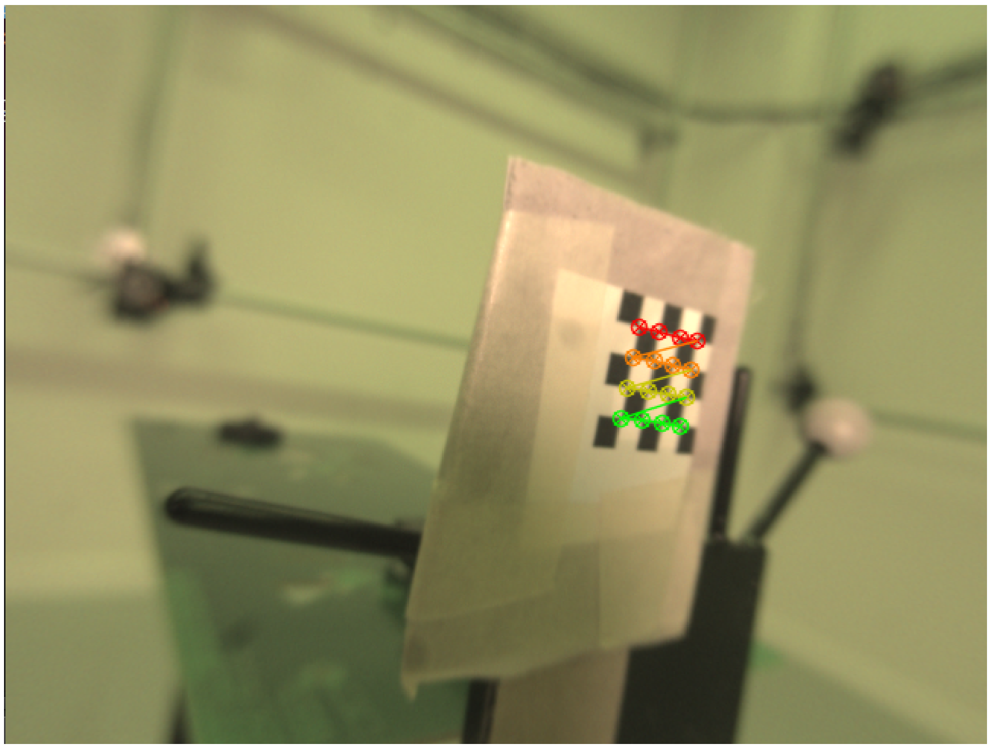

Figure 5: Detection example

5.4 Standard deviation of measurement

To check for problem in the captured image we can check the s.t.d of the calculated position and rotation. This tell us, for a specific orientation, if the difference between different captured image is big or not. Usually this happened when the light was not perfect, or something is moving.
This is shown in my dataset for each value. This stdev gives us confidence that we have (almost) no noise in the images.
There are a few outliers in the dataset but it is insignificant compared to the stdev of the error and the measurement are good overall with regards to that. I don't see the use of showing these values here but it can be used to check if one measure seems wrong.

5.5 Mean and standard deviation of difference

An important result is the mean error of orientation per marker in millimetre and degree (Lower score is better):

Mean	translation X	Y	Z	Yaw	Pitch	Roll	Score
Blob:	19.1	4.8	11.3	3.2	2.7	2.7	7.3
Checkerboard:	25.1	8.6	7.6	4.0	4.2	0.8	11.7
ArUco:	18.3	11.9	12.7	27.9	11.9	6.1	14.8

stdev	translation X	Y	Z	Yaw	Pitch	Roll
Blob:	21.3	5.2	7.4	4.5	3.6	0.8
Checkerboard:	31.3	9.2	9.5	7.1	5.0	1.0
ArUco:	20.3	8.2	6.4	42.9	17.8	3.8

According to my results, Blob markers are the best, and ArUco is the worst while checkerboard was in the middle but it seems like the ArUco measures were not really good so I'll talk about them separately and try to figure out what went wrong.

5.6 Analysis

In general, the result aren't extraordinary. The Y and Z translation gives result that are good enough if we take into account the initial error, it satisfies me. The X translation is bad, being 2 cm wrong on average is not good, we can see from its stdev that the measured value for this translation is varying a lot, so our system is probably not doing a good job for this axis, with every Marker.
The measured angles are about 4-5 degree wrong on average but the stdev is pretty high so it is probably good at certain orientation and worse at others.
Also the Z distance from the camera to the marker impacts heavily the performance of the system because the current lens is more precise at closer distance, for this reason we should look at it:

	Mean Z distance
Blob:	37.2
Checkerboard:	39.7

Turns out I was indeed more diligent towards blobs, but only by 2.5 millimetre on average so this wouldn't explain the better score of blobs.
Another important thought is that the calculated translations from the front facing measurement is way better than with angles:

1st Point Translation Error	X	Y	Z
Blob:	-6	-5.7	2.3
Checkerboard:	0	-3.7	13.4
7th Point Translation Error			
Blob:	-6.5	2.8	5.9
Checkerboard:	1.3	9.5	20.7

The checkerboard Z translation error is weird and should be re-verified, this is most likely due to initial set up bias. Overall it seems to be coherent with what was just said. (7th point is also front facing but pinched a bit so I included it too).

5.7 ArUco Result Failure

To try and figure out what went wrong, I looked at each point separately with respect to the groundtruth and found a few points that made almost no sense. For example, the yaw value is reversed and what not. Then I checked the result value of application and realised the problem. There was an error in the initial marker length assumed. It would be nice to redo measurement, but at the moment of realising this mistake, there was no time left.

5.8 Conclusion

In conclusion, we would need to do more measurement with better precision to truly asses the potential of the system.
What we can say now is that the system seems to be good at finding orientation more than translation within steeper angles. From a docking perspective, we should align first rotationally and then do translations. In fact, according to our data, the system is better at calculating the translation when there is less angle in the image.

These result would tell us that the system alone is not good enough for autonomous docking, we can probably get really close in terms of rotation(4-5 degrees off) also in translation but the mean error tells us that it can be off up to 1cm even when front facing, which is probably too bad for docking. Combining this approach with Stephane Bussier's work on hall sensors could give good results.

6 Alternatives and Future work

6.1 Hardware

It is hard to find mini computers that are small enough to fit inside Roombot modules. Another option could be the Banana Pi BPI-D1 Which has built in WIFI and camera support. We tried it but we were not satisfied with the delay from image capture to transmission. There was about 3 second of delays with the default set up.
There are many mini-computer coming into the market and there will probably be new ones that are as small as the Pi Zero and that have more processing power and more recent processors for better compatibility. We should be positive this is going to be the case.

6.2 Markers

The other thing to try next is ArUco board detection, which can lead to more robust pose estimation since it can, based on the predefined board layout, know where it should detect Marker, so in case of occlusion it should still work. It would be more robust, but would it be more precise in perfect conditions?

6.3 Docking Experiment

The next thing would be to try and use this system to do docking manoeuvres. This would be a better way to see how good this system is and if it should be used, we didn't have the time to do it but it would be the obvious next step in this project's direction.

7 Credit

7.1 Take Home Message

With this project I learnt the hard way that hardware incompatibility is primordial. I worked for several weeks for almost no concrete outcome because of bad drivers (pre-compiled files, there was no way to fix them).
Also I think that I didn't go deep enough into some tasks that I was given, I was only scratching the surface and not checking enough.
Given the experiment settings, the results aren't perfect, but we can still reason about the system.
Overall I think it went well even after problems we had.

7.2 Software and tools

Software used:

1. The Yocto Project
2. OpenCV
3. Meta-rpi console-image layer
4. ArUco Library
5. Motive
6. Webots
7. WMware

7.3 Bibliography

Automatic generation and detection of highly reliable fiducial markers under occlusion.