Computer Vision Project Report
Robot Navigation

Riccardo Mengozzi

November 14, 2023

1 Introduction

The aim of the project is to simulate a real-time processing of a stereo video captured by a mobile robot camera. In particular:

1. Compute distance from the object
2. Estimate the size of the object

To do that, one should be able to compute the disparity map, isolate the disparity values of the object and compute a main disparity which will be used to compute the distance. Then, the corners of the chessboard should be detected to estimate the size of the object.

2 Detecting the chessboard

Since only the disparity values of the object are important, it is useful to keep track of its center. To do that, the detection of the chessboard corners is necessary.

2.1 Preprocessing

To detect the chessboard one can simply use a OpenCv function called *cv2.findChessboardCorners* that, given an image and the pattern size of the chessboard, gives the coordinates of the corners with respect to the image reference frame. If one tries to use this function directly on the grayscale version of the original frames of the videos, in a lot of frames it wont be able to detect anything. It is obvious that the frames have to be preprocessed for the function to detect better the chessboard. Intuitively, this preprocessing should enhance the contrast between the white background of the chessboards and its black squares, this means blurring with linear filters will probably not be a good solution as they will smooth the edges of the chessboard leading to a even worse result. It is possible to improve the situation by stretching the pixels intensity with a dynamic range based on percentile. But what range one should use? In this case the optimal range has been found based on maximizing the number of frames in which the chessboard could be detected. The optimal range has been found to be: [25, 88].

(a) $min = 5$, $max = 95$ (b) $min = 25$, $max = 88$ (c) $min = 40$, $max = 60$

Figure 1: Stretching with different ranges, frame 140 of left video

2.2 Keeping track of the object

Using the OpenCV function, this is how the chessboard is detected in various frames:

(a) Corner detection in left frame n°0 (b) Corner detection in left frame n°70 (c) Corner detection in left frame n°110 (d) Corner detection in left frame n°140

Figure 2: Corner detection in left video

Generally, in the right video less chessboards can be detected, probably because the right camera is more angled with respect to the chessboard leading to a more difficult detection. It is not necessary to detect the chessboard in every frame: as long as the number of consecutive frames in which the chessboard is not detected is not too large. In this project, when the chessboard could not be found, the corners coordinates of the previous frame have been used. For better precision, when the chessboard is not detected, one could think to predict the coordinates of the corners using a sort of derivative: summing to the last coordinates the difference between the last and second last coordinates.

Now that it is possible to know the position of the corners of the chessboard, its center can be computed as the mean of the x coordinate and y coordinate of all the corners, since they are equally spaced.

This is the result across various frames:

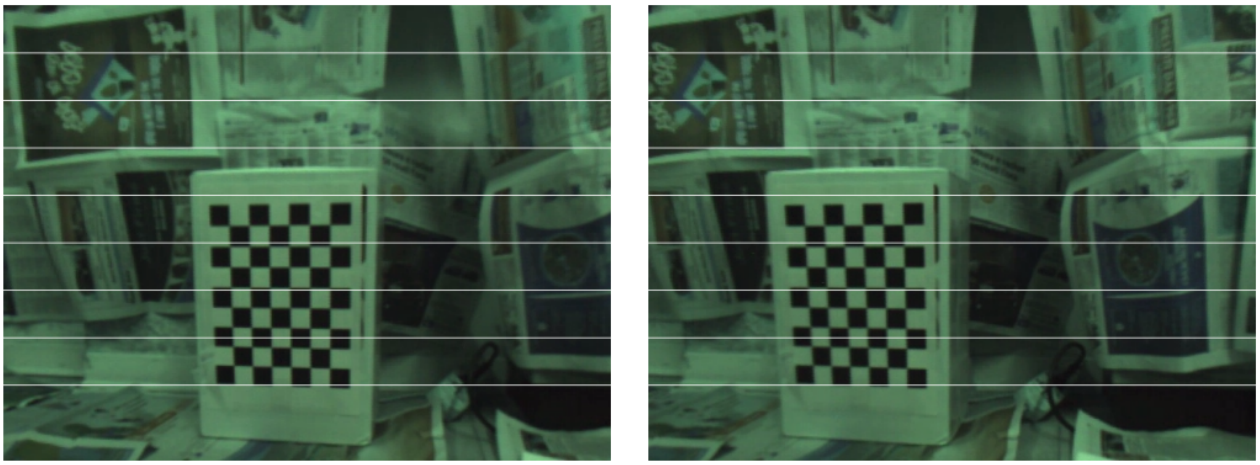
(a) Center of the chessboard in frame n°0 (b) Center of the chessboard in frame n°70 (c) Center of the chessboard in frame n°110 (d) Center of the chessboard in frame n°140

Figure 3: Center of the chessboard

3 Computation of the disparity map

3.1 Rectification

Firstly, it is necessary to rectify the image. In this particular case the frames seem to be well rectified from the start:



(a) left frame n°140 (b) right frame n°140

Figure 4: Proving rectification between left and right frames n°140

However, if the image were not rectified, one should find the rototraslation between the left and right cameras and the cameras intrinsics. The functions that allow that in OpenCV also undistort the left and right image along with the rectification:

1. *cv2.stereoCalibrate*: return the camera matrices and distortion coefficients for both cameras, the rototranslation between the two (R, T), the essential matrix (E) and the fundamental matrix (F).

2. *cv2.steroRectify*: given the camera matrices, distortion coefficients and the rototranslation found before, it returns the rectification transforms and projection matrices for both cameras, as well as the two regions of interest.

3. *cv2.initUndistortRectifyMap*: given the previous outputs, it returns the vertical and horizontal maps for each image to actually rectify and undistort them.

4. *cv2.remap*: given the maps found before it applies the warping to the left and right images.

3.2 Disparity map

OpenCV offers different ways to compute a disparity map, these vary in speed and precision. In this project have been tried:

1. *cv2.stereoBM_Create*: this is the simplest algorithm, based on SAD. It performs block matching. It is certainly the fastest, but also the least accurate and robust. It is composed of the parameters:

    - *numDisparities*: range of disparity values,
    - *blockSize*: size of the matching blocks.

2. *cv2.stereoSGBM_Create*: more complex method based on a modification of the SGM algorithm, making it a block matching instead of pixel wise matching algorithm. This method has different modes that changes its tradeoff in speed/accuracy based on the number of direction in which the block matching is performed, such as:

   (a) "3WAY" mode: faster but less accurate

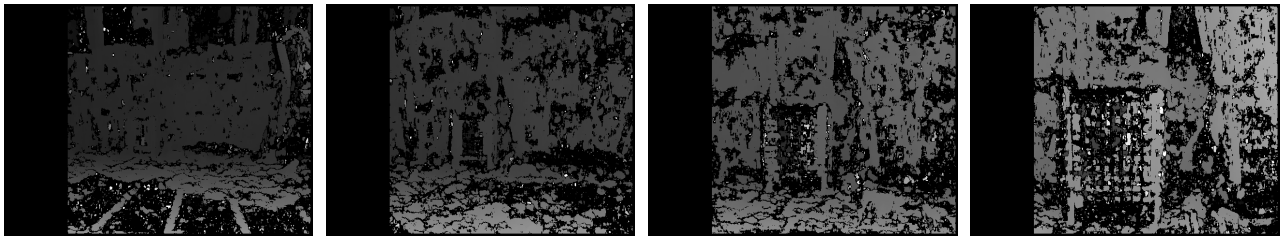
   (b) "HH" mode: slower but more accurate

   This method has more parameters that can be set:

   - *minDisparities*: offset that allows the disparity values to not saturate,
   - *numDisparities*: as before,
   - *blockSize*: as before,
   - *P1, P2*: parameters that enhance the smoothness of the map.

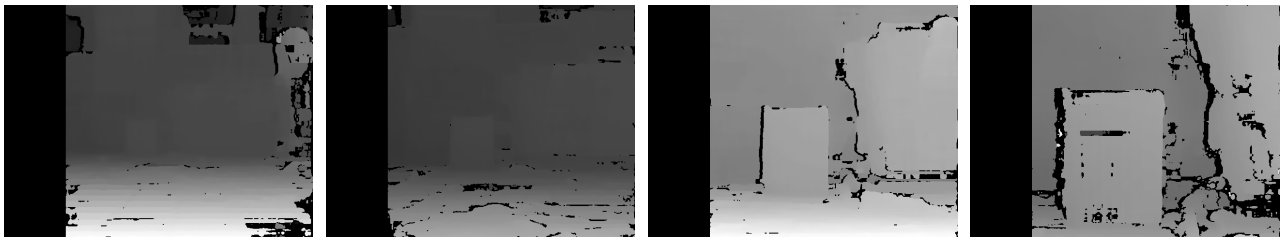
3. *cv2.WLSFilter*: post filtering algorithm that allows to have a smoother disparity map. It fill the gaps of the disparity map created by the pixels that couldn't be "seen" by both camera based on the original image.

The result using these different algorithms are the following:



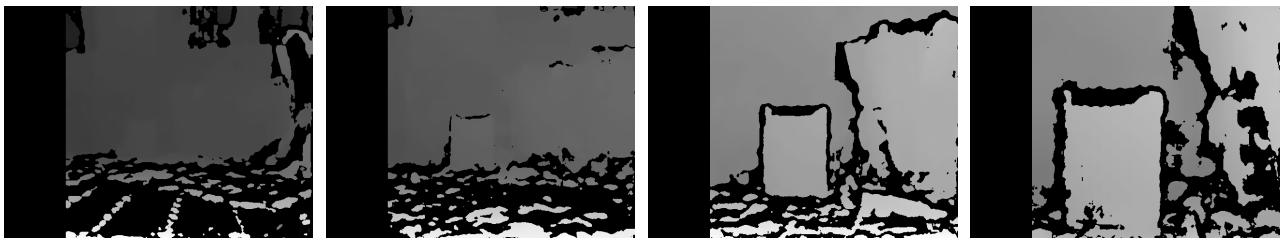
(a) Frame n°0 (b) Frame n°70 (c) Frame n°110 (d) Frame n°140

Figure 5: Disparity map with cv2.stereoBM. Execution time: 32 ms



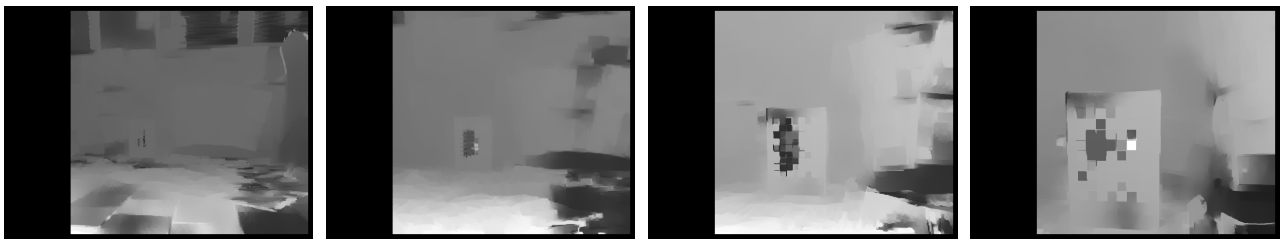
(a) Frame n°0 (b) Frame n°70 (c) Frame n°110 (d) Frame n°140

Figure 6: Disparity map with cv2.stereoSGBM and 3WAY mode. Execution time: 76 ms



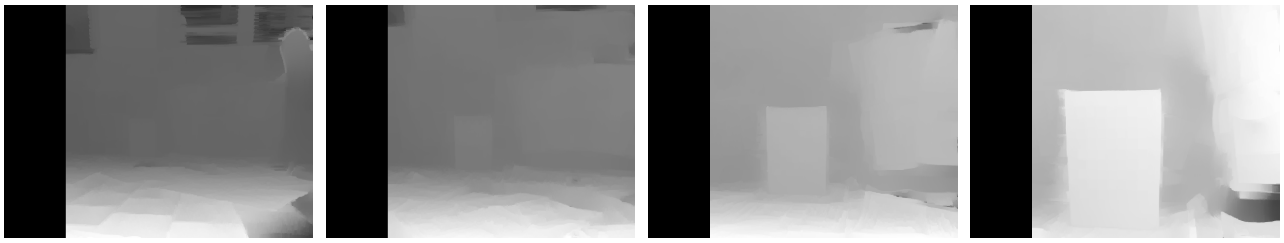
(a) Frame n°0 (b) Frame n°70 (c) Frame n°110 (d) Frame n°140

Figure 7: Disparity map with cv2.stereoSGBM and HH mode. Execution time: 350 ms



(a) Frame n°0 (b) Frame n°70 (c) Frame n°110 (d) Frame n°140

Figure 8: Disparity map with cv2.stereoBM and WLS filter. Execution time: 35ms



(a) Frame n°0 (b) Frame n°70 (c) Frame n°110 (d) Frame n°140

Figure 9: Disparity map with *cv2.stereoSGBM*, 3WAY mode and WLS filter. Execution time: 75 ms

From these results, it is possible to see that:

1. *stereoBM* is certainly the fastest method, but it creates a poor disparity map. Also the use of the WLS filter cannot correct all the artifacts created by the algorithm.

2. *stereoSGBM* with "HH" mode is certainly the one that creates the most uniform regions in the disparity map, even if the objects are very close, but it is obviously not suited for real-time application due to its computation complexity.

3. For real-time applications, *stereoSGBM* with "3WAY" mode seems to be the best trade-off between speed and accuracy, even more if combined with the WLS filter, which do not increase a lot its computation time even if it improves greatly the result.

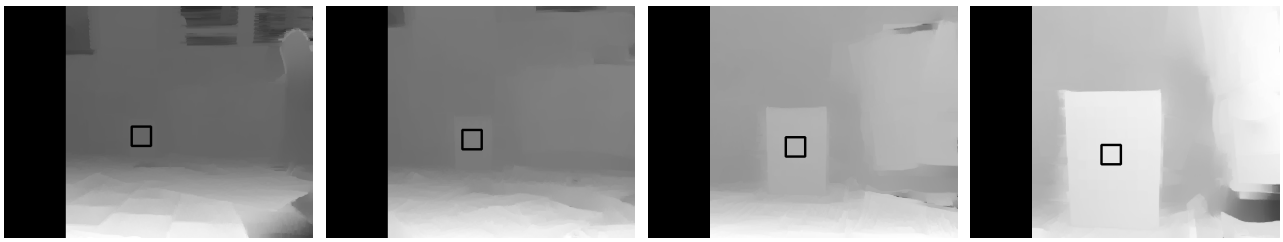
4 Computation of the distance

To compute the distance, apart from the focal length and the baseline which are given, the main disparity (d_{main}) of the chessboard is necessary to use the formula:

$$dist = \frac{f \cdot b}{d_{main}} \tag{1}$$

d_{main} can be computed in many different ways, in this project the best result has been given by the mean of the disparity values of a window centered in the center of the chessboard.

The size of this window cannot be too small because outliers will have an important influence, and it cannot be too big, otherwise pixel outside the object will be taken in consideration. For this reason, the size used has been 41x41 pixels. The next pictures show the area taken in consideration to compute the d_{main}.



(a) Frame n°0 (b) Frame n°70 (c) Frame n°110 (d) Frame n°140

Figure 10: Area for d_{main} computation

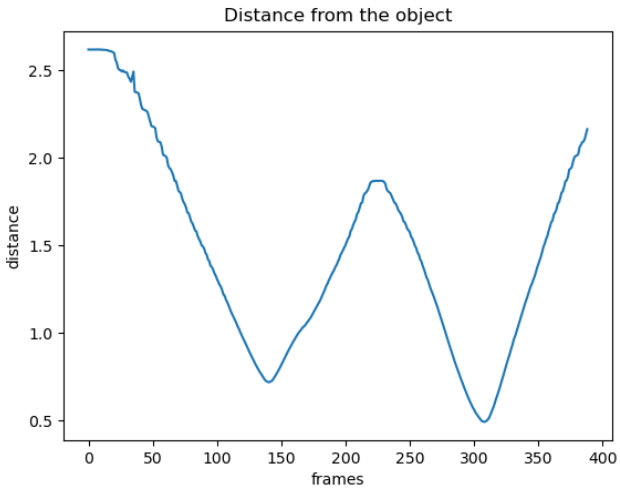
Combining the computation of the disparity map and the detection of the chessboard, the execution time of the program for each frame is starting to get too big, 144ms to be precise. Since the camera records in 15 fps, one would want the execution time per frame to be less than 60 ms for the program to be able to keep up with the video. Of course everything depends on the available hardware of the robot. One thing that can be done to reduce the execution without reducing the accuracy is to compute the disparity map in a cropped version of the original image. The image will be cropped only vertically to prevent the extension of the left "black pixels area" of the disparity map: this happens because, the more the image is cropped horizontally, the more are the pixels in the left image that cannot find its correspondent pixel in the right image. In this project the object is assumed to not shift vertically in the image, so it is possible to crop with a constant value. If this assumption were not true, one should crop the image making sure the centre of the object, which is known, stays always on the centre of the cropped image. Once the image is cropped, the coordinates of the chessboard corners need to be updated accordingly. Cropping the image vertically in the range [200:400], this is the result:

(a) Frame n°0 (b) Frame n°70 (c) Frame n°110 (d) Frame n°140

Figure 11: Area for d_{main} computation with cropping

The execution time per frame is now: 90 ms. To achieve the goal of 60ms, one could use better hardware than the one used for this project, switch to a faster programming language (C++) or simply trade some accuracy for speed.

Now that disparity is known, the distance can be computed.



(a) Distance from the object per frame.

```
[frame_125] distance = 0.90
[frame_126] distance = 0.88
[frame_127] distance = 0.86
[frame_128] distance = 0.85
[frame_129] distance = 0.84
[frame_130] distance = 0.82
[frame_131] distance = 0.81
[frame_132] distance = 0.79  TOO CLOSE!!!
[frame_133] distance = 0.78  TOO CLOSE!!!
[frame_134] distance = 0.77  TOO CLOSE!!!
[frame_135] distance = 0.76  TOO CLOSE!!!
[frame_136] distance = 0.75  TOO CLOSE!!!
[frame_137] distance = 0.74  TOO CLOSE!!!
```

(b) Alarm when the object is too close

Figure 12: Distance plot and output

5 Computation of the size of the chessboard

For the final step everything is already known apart from the pixel-wise size of the chessboard in the frames, as the formula used to compute the size of the object in mm is:

$$W(mm) = \frac{w(px) * f(px)}{distance(mm)} \quad H(mm) = \frac{h(px) * f(px)}{distance(mm)} \tag{2}$$

The height and width of the chessboard in pixels has been computed in 2 different ways:

1. **Indirectly**: with the diagonal and the angle between the diagonal and the horizontal. The angle is known because the pattern size is known.
2. **Directly**: computing the euclidean distance between the 4 vertices of the chessboard:
   - Height: distance between 1st and 2nd corner
   - Width: distance between 1st and 3rd corner

Comparing the results with the real size of the object, these are the final RMS errors for both methods:

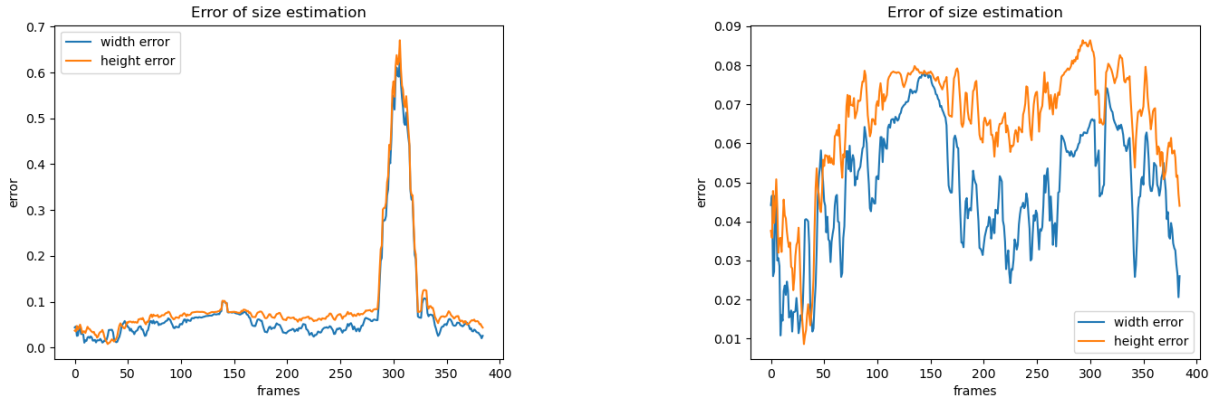
```
RMS error diagonal method =  [0.09069923 0.0243419 ]
RMS error sides method =  [0.04883805 0.06521594]
```

Figure 13: RMS error for width and height estimation for both methods

As we can see from the results, the total RMS error is similar in both methods. The second one will be used to have a more uniform error in width and height.

6 Conclusions

In this final chapter, some consideration about the results of the project will be made.



(a) Error plot without WLS filter

(b) Error plot with WLS filter

Figure 14: Error plot with and without WLS filter

From the results it is possible to see that without WLS filter a huge spike is present between frame 280 and 320, during the second approach maneuver of the robot to the object. This is probably caused by the d_{main} approaching more and more the bounds of the disparity range, due to the fact that the robot is too close to the object. This can be seen in the following plot.

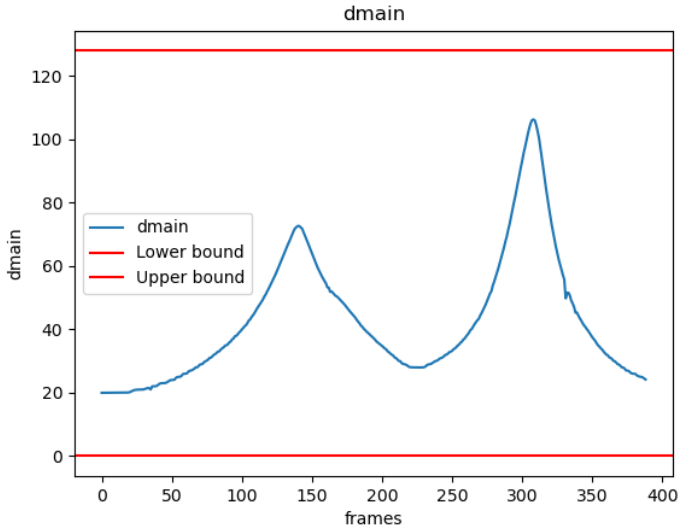


Figure 15: Main disparity (d_{main}) plot

It is also possible to see from previous plots that the further the robot is from the object the more the error oscillates, leasing to a less stable measurement.

7 Optional tasks

7.1 Variable disparity range

Focusing on the result without WLS filter, one solution to get rid of the spike is to use a variable range for the computation of the disparity map. To do that, one can use the *minDisparity* parameter of OpenCV *stereoSGBM* class. This parameter acts as an offset in the range if the *numDisparities* parameter is then assigned as the sum of the upper bound and the previous *minDisparity* value. To be sure the d_{main} never saturates the range of the disparity, one should use as offset a value such that d_{main} stays always in the centre of the range, so:

$$range = [offset,\ max\ +\ offset]$$

$$d_{main} = \frac{(offset + (max + offset))}{2} = offset + \frac{max}{2} \tag{3}$$

$$offset = \left(d_{main} - \frac{max}{2}\right)$$

Using this solution the result is the following:

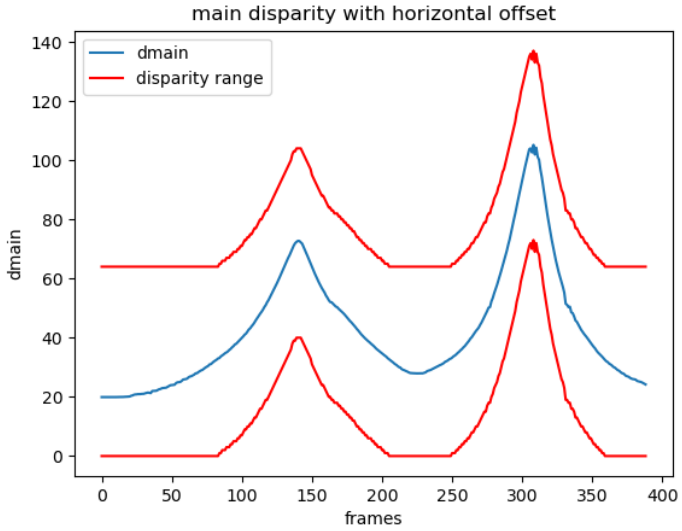
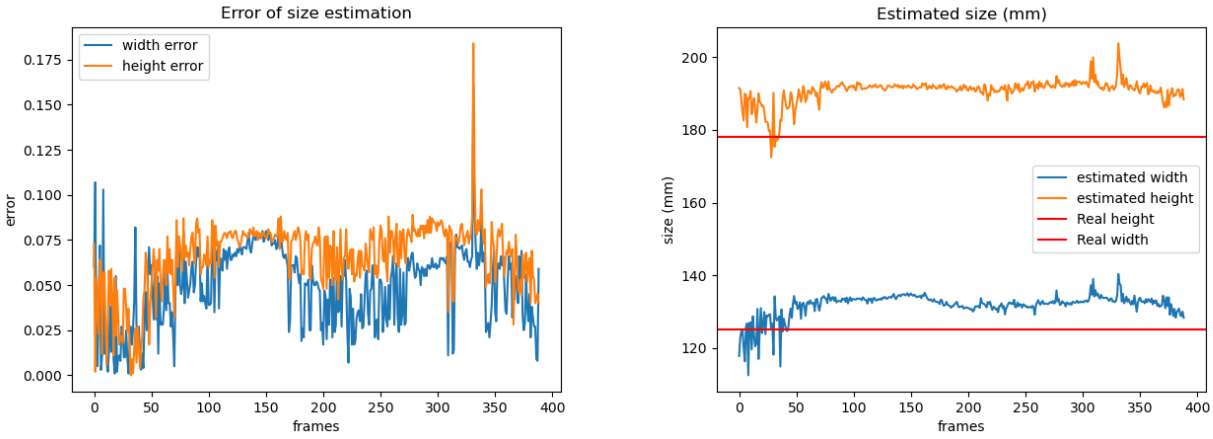


Figure 16: Main disparity with horizontal offset



(a) Error plot

(b) Size estimation plot vs real size

Figure 17: Error plots with variable disparity range

It is possible to note that now the spike is disappeared and the plot is really similar to the one showed with the use of WLS filter.

7.2 Multiple main disparities

Until now, it has been supposed the object to be planar with respect to the camera plane. This of course is not true: the object is clearly angled horizontally with respect to the camera and this lead to an error in particular in the width estimation. However, it is possible to compute the orientation of the object relative to the camera by considering multiple main disparities in different regions of the object. For this project, two regions have been considered, one in the first square column from the left and one in the last, as the next figure shows:

Figure 18: Regions for multiple main disparities computation

Thanks to this method, it is possible to estimate the difference of the distance between the two regions that allows for the calculation of the angle by this formula:

$$\theta = \sin^{-1} \frac{\Delta d}{W} \tag{4}$$

Being, Δd the distance difference and W the previously estimated width. Once the angle θ is computed, it is possible to correct the width estimation:

$$W' = W \cdot \cos \theta \tag{5}$$

The result on the error is the following:

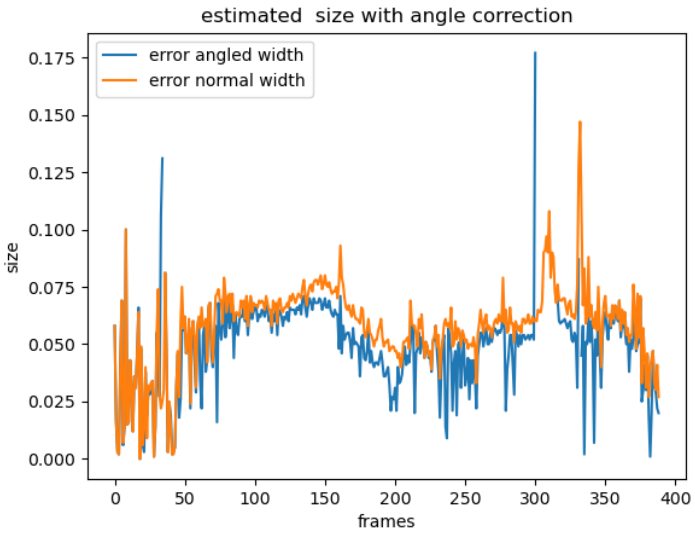


Figure 19: Error plot pre and post angle correction

It is possible to see how the blue curve (width estimation after the correction) is slightly better than the previous estimation. A problem encountered during the second approach maneuver is that, due to the variable disparity range added before, the closer the robot is to the object, the higher is the horizontal offset in the disparity range and the bigger becomes the unusable left part of the disparity map. This leads the "angle correction" to not work in that phase:

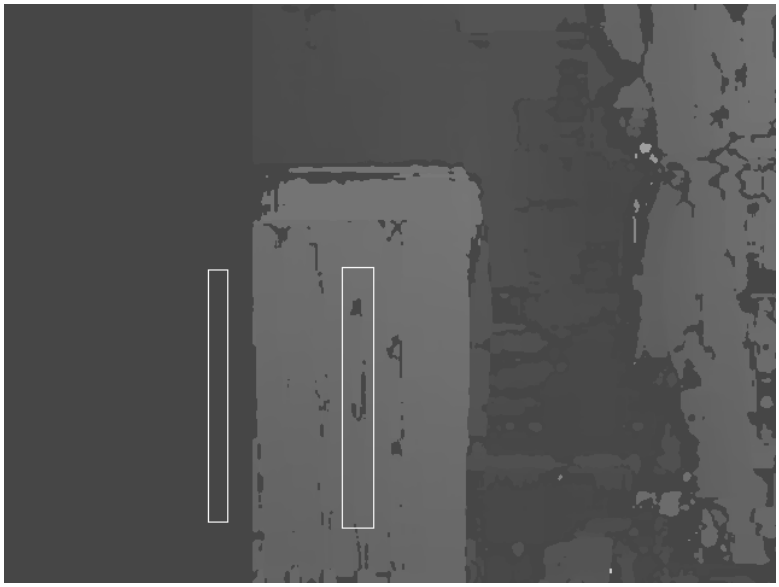


Figure 20: Problem of horizontal offset in disparity range