Distance and Movement Measurement of an Object based on Stereo Images

Lennard Rose, 5110000, FHWS Moritz Zeitler, 5118094, FHWS

Abstract

This paper explains how to compute distance to, and movement of an object within vision of a stereo camera. Included is the whole workflow consisting of all steps that have to be made to get such a recognition to work. These steps are the hardware setup, all calibration steps, object detection, object tracking and the measurements of the values themselves.

I. Introduction

From a high level standpoint the human eye is a very simple complex. It enables us to see things, even in color. But if we go deeper there is much more to our sight. For example we can see depth, which is very complex as a concept. This is based on the human eye having to different point-of-views. In this project the so called Stereo-Vision, will be translated to a computational level. To establish this we use a stereo camera setup to take photos/videos, create a depth map and try to compute the distance and movement of an object. The following chapters will describe this in a step by step fashion, starting with the theory behind it and the implementation afterwards.

II. Prerequisites

The initial idea is to run this Stereo-Vision on some kind of self driving vehicle, in our case a JetBot AI Kit[1]. This JetBot AI Kit is based on a NVIDIA Jetson Nano Developer Kit[2]. Mounted on this Bot is a the IMX219-83 Stereo Camera[3] of waveshare. As already stated the initial idea was to develop a system that's running directly on the bot. After assessment of the provided resources and some in depth testing the decision was made to not use the bot for running our applications since it would impede development tremendously. This means all data used in the project was recorded using the bot, but all computation and programming was done on other machines. Obviously this means that the resulting pipeline should be deployable to the bot but there are some severe driver problems that stop this from happening.

III. Stereoscopy

When capturing a 3D object in an image, the objects gets projected from a 3D space to a 2D (planar) projection space, losing all its depth information. This is called the Planar Projection. The question now is, how to get back the depth information from this 2D space. With 2 Cameras, this is possible by comparing the left and the right image, called Triangulation. On a abstract level, if one takes a picture with two cameras both of the x axes are exactly on the same level (depending on the mounting). For points on the y axes this isn't certain. The solve this problem there is the possibility to use epipolar geometry and stereoscopy. Basically the search space gets reduced. This is done by projecting the line LL between the points OL(left camera), XL(projection of point X in the left frame) and X(being the target 3D point) as ECR on our right picture, crossing the point XR(projection of point X in the right frame). The same on the left picture, project the line LR between the points OR(right camera), XR and X, crossing the point XL as ECL. The centers of both cameras get also project to there significant other frame. With this, the searchspace got reduced of a corresponding point to compute our disparity from to a single line, ECL respectively ECR (this is called an epipolar constraint).EL and ER is the projection of one camera onto its significant other cameras pov. These points are called epipoles. Together with LL and LR this baseline forms the epipolar plane (blue in figure 5). All this information can be represented by the Fundamental matrix of the image, which can be calculated with the projected points and the projection matrices of the cameras. Further mathematical description can be found in the book *Multiple View Geometry in Computer Vision*[4], but won't be discussed any further at this point.

IV. Workflow

This chapter contains a detailed theoretical and practical description of each individual step in the workflow.

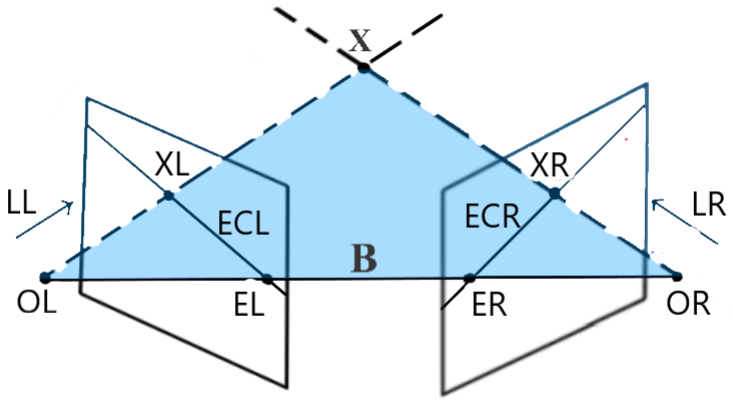

Fig. 1. Epipolar geometry

A. Stereo Camera Calibration

Cameras are projecting a three dimensional object to a two dimensional image. This projection is never alone determined by the original object, but also by all parameters of the used camera. These you can separate in two groups, external and internal. External parameters are the camera orientation in different planes. Internal parameters are determined by the cameras internal characteristics.

$$\mathbf{K} = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Fig. 2. Intrinsic Matrix

We can represent these internal parameters as an upper triangular matrix K. This matrix has the values fx and fy as the x and y focal lengths. cx and cy as the x and y coordinates of the optical center in the image plane. Gamma is the distortion between the axes, this value is replaced by a ”0” in OpenCV´s calibration function.

$$\begin{bmatrix} R \mid t \end{bmatrix} = \left[\begin{array}{ccc|c} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \end{array}\right]$$

Fig. 3. Extrinsic Matrix

The external parameters are represented by an 3x3 rotation matrix, that indicates the rotation bias of the camera, combined with an 3x1 translation vector.

$$\mathbf{P} = \overbrace{\mathbf{K}}^{\text{Intrinsic Matrix}} \times \overbrace{[\mathbf{R} \mid \mathbf{t}]}^{\text{Extrinsic Matrix}}$$

Fig. 4. Projection Matrix

Together these form the 3x4 Projection Matrix. This matrix and the 3D-coordinates of our point is used to calculate the image coordinates of the 2D projection. Calibration does nothing else than alter these matrices so our 2D projection matches the 3D

object. This is usually done by giving the calibrating program/ function a set of 2D images with known 3D coordinates. This Calibration consists of two steps removing of Distortion as well as rectifying the images[5].

1) Distortion: A camera lens always has some kind of curvature. This curvature is also visible in an image taken with this lens. This is called distortion. There are many different types of distortion which can be seen in figure 4.

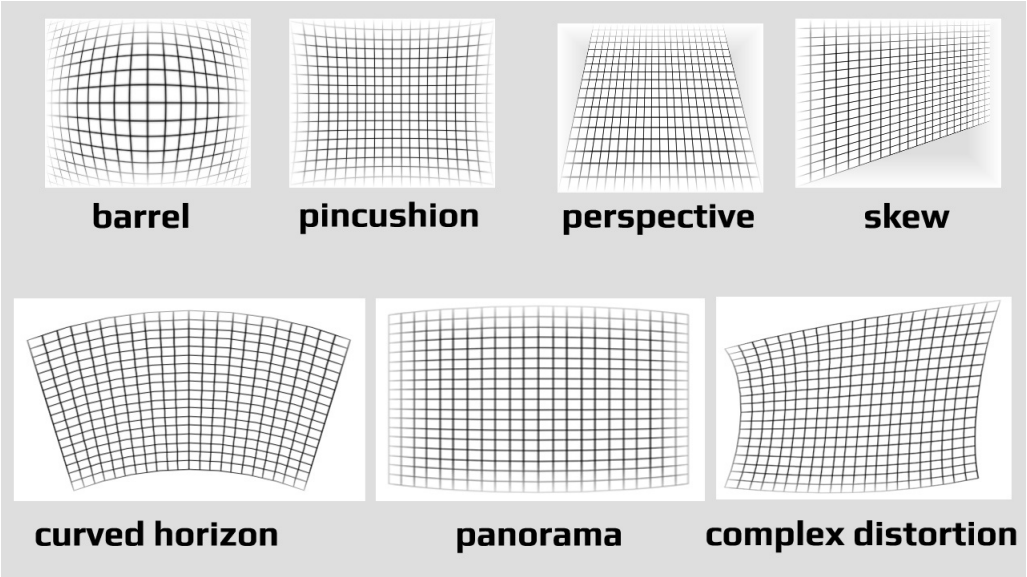

Fig. 5. Distortion types

To remove this distortion we take pictures of a chessboard. OpenCV, the package we use to do all calibration steps, can easily detect key points using this chessboard pattern. This is based on the big difference in pixel values between white and black. In combination with the size of the squares as well as the amount of the squares, OpenCV is able to create a calibration matrix removing the distortion.
A very important part is to be very precise while taking the calibration pictures. To get a good result you need to take calibration photos, basically of all parts of the pov of the camera.
The python implementation looks like the following: First of all we convert the input picture to grayscale and feed it into a detection function for chessboard corners.

```
ret, corners = cv2.findCirclesGrid(inputImage,
                                   (chessboardColumns, chessboardRows),
                                   None)
```

As input we have the chessboard picture, columns of the chessboard, rows of the chessboard and an output array we don't use. This function results in a return value whether a chessboard got detected and if so the 2D points of the chessboard corners. If the detection was successful the returned points get even more refined using:

```
cv2.cornerSubPix(inputImage,
                 corners,
                 (11, 11),
                 (-1, -1),
                 criteria)
```

The 'cornerSubPix' of openCV refines the corner locations [6] of the checkboard. All results of the previous steps are saved in a list to cache them for now. With these lists we can calibrate a single camera:

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(object_pts,
                                                   img_pts,
                                                   img.shape[::-1], None, None)
```

Feeding these points as well as the shape of the image into the calibrate Camera function delivers the intrinsic and extrinsic parameter. These parameters are stored inside the mtx variable. Also we get the overall RMS re-projection error in ret and the distortion coefficients. Rvecs[7] and tvecs are rotational and translation vectors, which aren't used in this system and won't be described further. On a high level openCV executes the global Levenberg-Marquardt optimization algorithm to minimize the reprojection error, that is, the total sum of squared distances between the observed feature points imagePoints and the projected

(using the current estimates for camera parameters and the poses) object points objectPoints[8]. With these matrices we can now compute a new optimal matrix based on these parameters.

```
h, w = img.shape[:2]
new_mtx, roi = cv2.getOptimalNewCameraMatrix(mtx,
                                             dist,
                                             (w, h), 1, (w, h))
```

As a result the function returns a new optimal intrinsic camera matrix. The function also returns a valid pixel region which outlines a region of all-good-pixels in the undistorted image. These matrices and results will be later used to undistort and rectify images coming from our camera for better results while computing the disparity and depth map.

2) Rectification: With image rectification, the goal is to project the two different images taken by the camera onto a common image plane. Through this process we make sure that the individual lines of both images are on the same level, speaking on a very high level. This means the epipolarlines are calibrated to be exactly horizontal, which helps finding the correspondence of a point in both of the stereo pictures. For this step we need the previous calibration matrices for both of our cameras as well as pictures from both cameras at a similar time containing the chessboard to compute the rectification. Through this computation we get:

1) essential matrix
2) rotational matrix
3) translation vector (previously described as extrinsic matrix)

```
retS, new_mtxL, distL, new_mtxR, distR, Rot, Trns, Emat, Fmat = cv2.stereoCalibrate(obj_pts,
                                                                                    img_ptsL,
                                                                                    img_ptsR,
                                                                                    mtxL,
                                                                                    distL,
                                                                                    mtxR,
                                                                                    distR,
                                                                                    imgR.shape[::-1],
                                                                                    criteria_stereo,
                                                                                    flags)
```

The function can also compute the full calibration for each of the cameras but isn't recommended to do so due to the high dimensionality of the parameter space and noise in the input data[9]. Because of this the individual calibration is done beforehand and gets supplied as an input. With these matrices we are able to compute a coordinate representation of a 3D point in one of the cameras using the result matrices as well as the one of the individual camera matrices. Similarly to calibrateCamera, the function minimizes the total re-projection error for all the points in all the available views from both cameras. The function returns the final value of the re-projection error. Now the goal is to rectify these cameras using stereoRectify:

```
rect_l, rect_r, proj_mat_l, proj_mat_r, Q, roiL, roiR = cv2.stereoRectify(new_mtxL,
                                                                          distL,
                                                                          new_mtxR,
                                                                          distR,
                                                                          imgL.shape[::-1],
                                                                          Rot,
                                                                          Trns,
                                                                          rectify_scale, (0, 0))
```

Once again all previous results are input for the function. The function results in two rectification transform matrices, which brings points given in the unrectified camera's coordinate system to points in the rectified camera's coordinate system. Also the projection matrix, in the now recitfied coordinate system of the camera, which projects points into the camera image.

B. Disparity Map

Disparity is usually computed as a shift of an image feature when viewed in the right/left image. For example, a single point that appears at the x coordinate k (measured in pixels) in the left image may be present at the k coordinate k minus 3 in the right image. This would result in a disparity value of 3 in pixels. These disparity values are the base for the computation of the depth map. This can be done using specific algorithms, in this case the Block Matching Algorithm is used.

Fig. 6. Visualization of the Block Matching Algorithm working on input and output image

Figure 6 displays the method the algorithm implements. Input is one pixel, or in this case a small window of pixels, which makes the algorithm more resilient. With the input the algorithm searches the most similar neighbor in the corresponding stereo picture. In this case the right image is the input, with the green square the search window. The algorithm searches for its corresponding window on the left picture. The best match is visualized by the green window the second best by the red window. On a theoretical level the algorithm searches through the image line by line to find the best match. This can be done in with different metrics, for example absolute or normalized differences. The algorithm searches for the least differences to determine correspondence. However, the algorithm tends to find multiple pairs of corresponding points due to repeating textures or similar occurrences. In python the algorithm is implemented as a class which needs to be instantiated. To be more precise the implementation used in this project uses a slightly more advanced[10] but follows the same principle.

```
stereo = cv2.StereoSGBM_create(minDisparity=min_disp,
                               numDisparities=num_disp,
                               blockSize=window_size,
                               uniquenessRatio=10,
                               speckleWindowSize=100,
                               speckleRange=32,
                               disp12MaxDiff=5,
                               P1=8*3*window_size**2,
                               P2=32*3*window_size**2)
```

Important are the different options we supply to the SGBM class. The block size parameter supplies the window size for the algorithm. The disparity amount, which has a minimum and a maximum, specifies at which range a disparity match is acceptable. Speckleand P1 and P2 are filtering algorithms to smooth the output[11], again we skip further explanation. To further improve those parameters, an user interface was build to change the values on the fly.

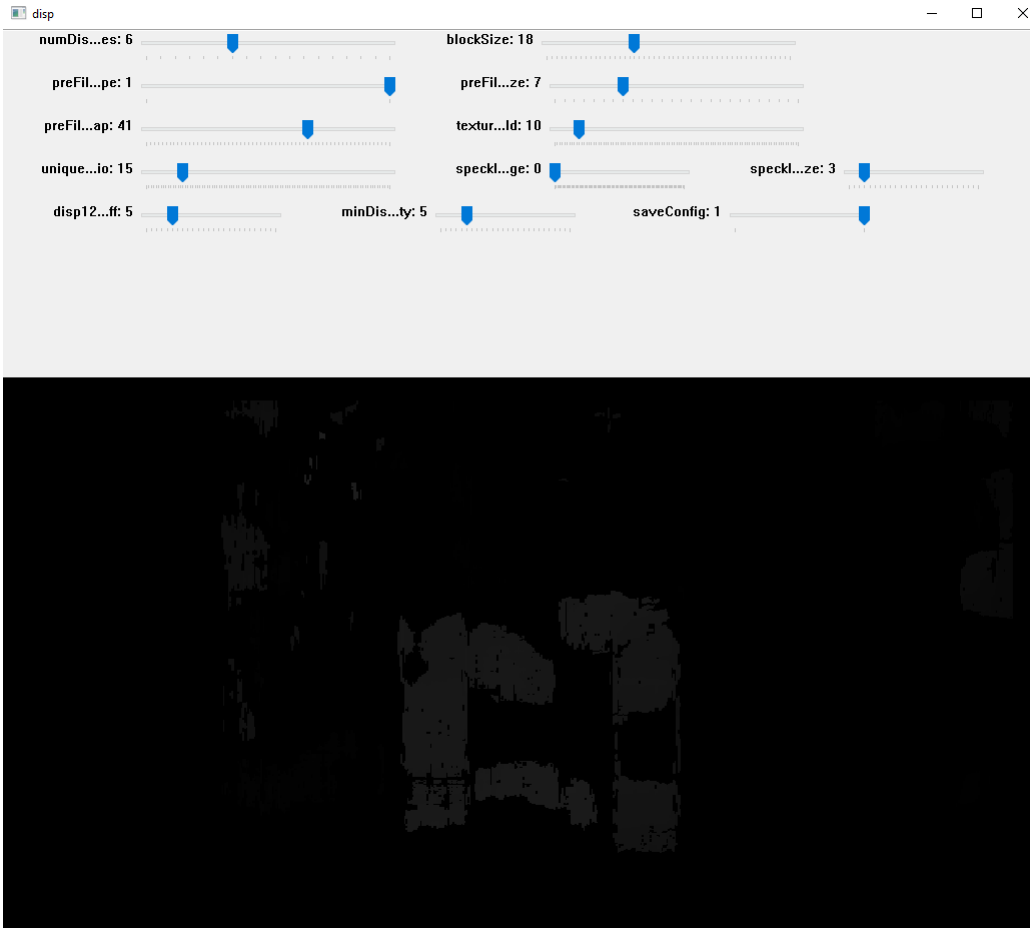

Fig. 7. UI for calibrating all parameters of the Block Matching Algorithm

In the UI are many different parameters to tune. Also there is the option to save the parameters, it isn't currently possible to use a button in openCV thus there is a slider which produces a config.json.The result can look something like Figure 8.

Fig. 8. Resulting Disparity Map after calibration without filtering

On the left the disparity map and on the right the original left side image.

C. Depth Map

At this point there are two different options to compute a depth value from the previously created disparity map. The first one being the usage of the openCV function 'reprojectImageTo3D', which takes the disparity map as well as the Q matrix we computed during the rectification process to compute a depth map. Alternatively there is the possibility to sort of calibrate

the disparity-depth mapping. This means taking a video/ picture of an object at a known distance, creating the corresponding disparity map and extract the value. With both of these values a regression line gets computed. With the formular of the regression line we can now average the distance of an object based on the its disparity value.

Fig. 9. Depth map calculated with blue color filter and a computed distance based on a regression function

The resulting regression is extremely basic:

$$distance = \frac{average\_disparity}{0.34} \tag{1}$$

On top of this the result will get rounded to two decimals. In Figure 9 we can see this working on a sample video. The box is placed at distance of 1.50m and the output of the whole function is 1.49m. This is a pretty good result.

D. Object Detection

E. Object Tracking

V. Measurements

In this section some measurements will be presented and discussed.

A. Distance

For testing purposes the initial idea is an event based system. On double-click an openCV event gets triggered, containing a x and y position of the mouse click. These values get enhanced to a window, all disparity values in this window get added up to compute an average. With this basic function we can encapsulate computing the distance. This results in following precision:

actual distance	computed distance
0.5	0.5
0.5	0.5
0.6	0.61
0.6	0.6
1.2	1.18
1.2	1.18
1.3	1.28
1.3	1.31
1.4	1.44
1.4	1.42

The mean squared error of these observations is 0.00034. The closer the value to 0 the better the result, meaning that the computed distance is extremely good. But this needs to be put into perspective, meaning this test was performed under laboratory conditions which should result in this kinda of perfection.

B. Direction

VI. Conclusion

The most important conclusion of this study is quite sobering. With the experience gained from working with the system, there are way too many factors that influence the result. Even more important those factors or variables are not always controllable. This means from time to time, results from some functions are more or less arbitrary. Especially the calibration of the stereo cameras is quite a difficult task. Most of the time the result of openCV calibration, meaning removing of Distortion as well as rectifying images, isn't comprehensible. A big variable are the pictures, and their quality, as a consequence this means the camera used is quite important. In our case the cameras are designed for this purpose, but this doesn't mean the cameras don't have problems. Basically the calibration process was done by try and error. This means the calibration is not as easy as most documentations suggest. By this a the following algorithms, for example computation of distance, are inferred by this, lets call it blur.

If those problems are left aside the result for distance as well as speed are quite satisfying. With a better setup that doesn't depend as much on calibration, for example usage of a Intel RealSense camera could enhance the result enormously. And almost as important, would ease up development since the camera modul for JetBot in combination with the nVidia Jetson Nano development board does also provide some problems which won't be discussed at this point.

As a final conclusion, on a theoretical level, the initial problem doesn't sound really super complex. But once we factor in the practical part the difficulty level rises significantly. All in all this means the result of this study is quite satisfying once it's viewed from a objective standpoint.

References

[1] Waveshare. (2020) Jetbot ai kit. [Online]. Available: https://www.waveshare.com/wiki/JetBot_AI_Kit

[2] NVIDIA. (2020) Nvidia jetson nano. [Online]. Available: https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-nano/

[3] Waveshare. (2021) Imx219-83 stereo camera. [Online]. Available: https://www.waveshare.com/wiki/IMX219-83_Stereo_Camera

[4] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, 2004.

[5] C. Loop and Z. Zhang, "Computing rectifying homographies for stereo vision," in *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, vol. 1, 1999, pp. 125–131 Vol. 1.

[6] W. Förstner and E. Gülch, "A fast operator for detection and precise location of distinct points, corners and centres of circular features."

[7] G. Gallego and A. J. Yezzi, "A compact formula for the derivative of a 3-d rotation in exponential coordinates," *CoRR*, vol. abs/1312.0788, 2013. [Online]. Available: http://arxiv.org/abs/1312.0788

[8] openCV. (2021) Camera calibration and 3d reconstruction - calibratecamera. [Online]. Available: https://docs.opencv.org/master/d9/d0c/group__calib3d.html#ga3207604e4b1a1758aa66acb6ed5aa65d

[9] ——. (2021) Camera calibration and 3d reconstruction - stereocalibrate. [Online]. Available: https://docs.opencv.org/master/d2/d85/classcv_1_1StereoSGBM.html

[10] H. Hirschmuller, "Stereo processing by semiglobal matching and mutual information," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 2, pp. 328–341, 2008.

[11] openCV. (2021) Stereosgbm class reference. [Online]. Available: https://docs.opencv.org/master/d9/d0c/group__calib3d.html#ga91018d80e2a93ade37539f01e6f07de5