Overview

The task in question is as follows:

- We are given 2 perfectly rectified stereo images of a scene, along with the setup parameters of the cameras which produced them. The given parameters that are employed in my method are:
    - Focal length (f)
    - Baseline length (b)
    - Distance between camera principal points (doff).
- We are to produce a real-world 3D point cloud that corresponds to the scene depicted in the 2 images.

The steps undertaken can be summarized as follows:

1. Produce a disparity map for the set of images. A disparity is a data structure which contained a disparity value for each pixel of the images.
2. Compute the real-world (x, y, z) coordinates of each pixel using geometric relationships between the disparity value, the camera parameters, and the point's coordinates.
3. Plot all of the points in 3D space to produce the point cloud visualization.

Disparity Map

Producing the disparity map mainly centers around solving the correspondence problem. In this case, Sum of Absolute Differences (SAD) Block Matching was employed to calculate the most appropriate disparity value for each pixel. Essentially, one window of pixels is compared at a time. We shift the window from one image by an offset amount, then superimpose this window upon the corresponding window in the reference image on the same epipolar line. The sum of the absolute differences between each corresponding pixel intensity in the superimposed windows is calculated. The process is repeated for every possible offset value, and the best disparity estimate for that pixel is the one which gives the smallest sum of absolute differences. The algorithm performed was generally based upon the method described in the following source, which constitutes a common stereo matching technique:

http://www.csd.uwo.ca/~olga/Courses/Winter2016/CS4442_9542b/L11-CV-stereo.pdf

Figure 1 shows a simple illustration of the block matching technique measuring cost between the window in the reference left image and the shifted window in the right image.

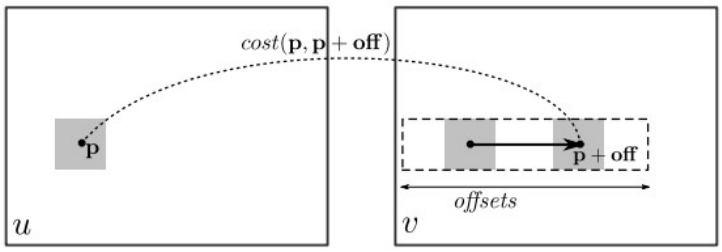


Fig. 1 Stereo Block Matching Technique for Rectified Images

1. Choose an odd window size for the comparison windows.
2. Iterate over every pixel of the reference image (right image was chosen). Row first, then column. Every time you iterate a row or column, set up max and min row/column indices for the windows to ensure you do not go over the image edges.
3. For each pixel, iterate over every possible offset value we can check for (from 0 to the max dispersion value we can check for that doesn't go over the image edge).
4. Crop out a window in the reference image. Crop out a corresponding image in the left image that has the same min and max row coordinates but is shifted rightwards by the current disparity iteration value to account for the disparity shift.
5. Compute the sum of the absolute differences between corresponding pixels in the 2 windows, and store it in an array.
6. At the end of the offset iterations, we sort the SAD array to find the index (essentially the offset value plus 1) which gave the smallest SAD and set that as the disparity value of for that pixel. The disparities values are stored in a 2D array.

The algorithm can be seen from my uploaded code files. It forms the basic core of the process; however, several points can be made about available programmer choices during the algorithm's actual execution.

Choice of Window Size

Most sources I found indicate window sizes between 7x7 to 15x15. Smaller windows work better around object boundaries while larger windows better preserve flat, low texture areas[1]. The images I chose[2] were relatively varied in depth, and I also wished to speed up the matching process as much as possible. Hence, I chose 7x7 pixels windows. It has to be noted by numerous sources that there is no perfect window size.

[1] https://www.chromasens.de/sites/default/files/whitepaper_-_3dpixa_block_matching_artefacts_on_depth_estimation.pdf

[2] http://vision.middlebury.edu/stereo/data/scenes2014/zip/Vintage-perfect.zip

Choice of Window Cost Evaluation Method

There are many methods of evaluating the cost of a chosen offset value for a given window, including: SAD, SSD, ZDAD, and ZSSD among numerous others[3]. SAD was chosen because it was the simplest calculation to perform, and tests have shown that it produces the same result in shorter time[4]. It is the most widely used measure for reasons of computational simplicity[5].

Choice of Using Integral Images for Sum Calculations

The integral image representation is an image processing technique which allows the evaluation of sums over rectangular regions of an image with only four operations regardless of size[3], and numerous sources describes its use as a way of speeding up the block matching process. Essentially, an algorithm[3] is used to produce an integral map over the image awaiting summation. After computing the absolute difference value for each pixel within the corresponding windows, one would then use a simple 4 step calculation to obtain the sum of all the differences. This was explained as more favorable than iterating through each pixel within the window. The sum calculation over the integral image is described below:

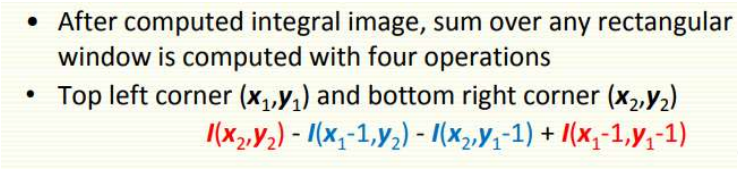


Fig. 2 Integral Map Sum Calculation

I implemented both the simple block matching algorithm and the integral images method in Python, but unfortunately was not able to find a noticeable improvement in speed. This may have been due to the lack of optimization in the code I used to produce the integral images. I employed the *scalar accumulator algorithm* for integral image calculation which was proven to be the fastest on "different compilers and optimized compilation settings[3]".

Choice of Coding Language

I implemented the described algorithm first in Python but noticed that the process was taking too long to finish. It took around 20 minutes for a 300x300 pair, and the 2900x1900 image I had was set to take hours. The issue most likely relates to the lack of optimization in my code as I just implemented the core algorithm. I tried to find the Python source code

[3] https://www.ipol.im/pub/art/2014/57/article_lr.pdf
[4] https://robotics.ee.uwa.edu.au/theses/2005-Stereo-Kuhl.pdf
[5] https://doc.lagout.org/network/H.264%20and%20MPEG4%20Video%20Compression.pdf
[6] http://www.csd.uwo.ca/~olga/Courses/Winter2016/CS4442_9542b/L11-CV-stereo.pdf

of OpenCV's StereoSBGM and StereoBM functions but were unable to do so, while the C++ source code was not readily understandable due to a lack of documentation. OpenCV's documentation merely describes the functionalities and general principle of these functions but did not delve into their specific make up. I tested my image with these functions and found them to be much faster than my Python code. I switched to MATLAB, implemented the exact same simple block matching algorithm and produced a disparity map in about 20 minutes. This is likely due to MATLAB's vectorization methods which optimized summing and sorting calculations on 2D window arrays and 1D offset lists.

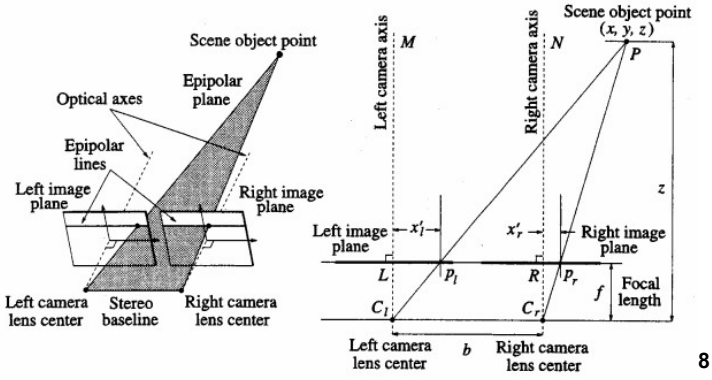
3D (x, y, z) Coordinates

Each pixel's x and y coordinates remained the same as the pixel's column and row coordinates. The z (depth) coordinate was the one which needed to be triangulated in order to obtain the 3D point cloud. For z, I implemented the same formula that was indicated in the source of the provided images, which is as follows:

> To convert from the floating-point disparity value d [pixels] in the .pfm file to depth Z [mm] the following equation can be used:
>
> Z = baseline * f / (d + doffs)

[7]

Numerous sources indicate the z value as inversely proportional to the disparity value of the corresponding pixel. However, the exact formula employed differed as a result of differences in the choice of origins and the locations of the image and camera planes with respect to each other. Most sources have an epipolar geometry setup similar to the following:



[8]

Fig. 3 A Common Epipolar Geometry

In this case the formula for z is: $z = \frac{bf}{(xl' - xr')}$,

xl' – xr' being the disparity value of the pixel corresponding to that location.

[7] http://vision.middlebury.edu/stereo/data/scenes2014/

[8] https://www.cse.usf.edu/~r1k/MachineVisionBook/MachineVision.files/MachineVision_Chapter11.pdf

If I were to base the geometry off the diagram of the reference document, assuming that the origin lies at the P1 point, I would give the following formulas for the coordinates:

$$Py = \frac{fb}{b-(c2x-c1x)} \qquad Px = x1 \times \frac{Py}{f} \qquad Pz = y1 \times \frac{Py}{f}$$

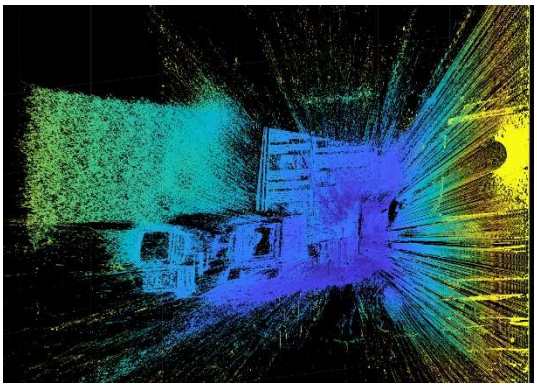


Fig. 4: Physical setup of the lab apparatus with all major components connected

I tested out different geometric equations for the coordinates and found that keeping the x and y coordinates as the unaltered pixel values and using the formula given by the Middlebury stereo image source for z produced the best results, as shown in the uploaded video and MATLAB figure. When I used the 3 formulas above, a prism-shaped cloud was produced instead, shown on the right. The discrepancy may have been due to wrong assumptions about the location of the origin and the specific epipolar geometry employed to produce this pair of rectified images. Deducing the epipolar geometry for the given images was difficult because no clear representation was given by the image source for the actual geometric relationships of the parameters provided.

Mobile Robots

Depth estimation for mobile robots involves the processing of real-time images. Every image captured by the rover must be immediately rectified to allow accurate robot maneuvers. Hence, the rectification process must be executed as quickly as possible. Numerous algorithms have been researched and tested. Efficiency has been improved through the use of fast GPUs to take advantage of parallelization to optimize the computation[9]. Rectification is possible only after the cameras have been calibrated to obtain their intrinsic and extrinsic parameters, which are then organized in transformation matrices[10]. MATLAB provides a Stereo Camera Calibrator App which performs this process easily on a stereo camera[11]. Images must then be undistorted according to the acquired parameters. To perform this task on a rover, it would be best to mount a calibrated stereo camera at its front. To help with the task, ultrasonic and IR sensors relaying spatial distance may provide additional information about the robot's distance relations to its surroundings to refine the rectification process.

[9] C. D. Pantilie, I. Haller, M. Drulea and S. Nedevschi, "Real-Time Image Rectification and Stereo Reconstruction System on the GPU," 2011 10th International Symposium on Parallel and Distributed Computing, Cluj Napoca, 2011, pp. 79-85.

[10] https://robotics.ee.uwa.edu.au/theses/2005-Stereo-Kuhl.pdf

[11] https://www.mathworks.com/help/vision/ug/stereo-camera-calibrator-app.html