Laboration 2: RGBD-cameras

Sensors and Sensing

Marek Bečica, Tom Olsson

December 7, 2015

All code for this exercise can be found at
`https://github.com/tgolsson/sensors-laboration2-xtion`

In some of images and plots the "variance" is wrongly labeled as "standard deviation". This is just a typesetting error in the images, and not the actual values. All values are variances.

Contents

Code listings

List of Figures

List of Tables

1 Theory and motivation

Robot vision is an important subject in the development of autonomous robots. However, as with all sensors, cameras suffer from noise which makes interpretation of the images hard, which in turn reduces the success of processing such as image identification.

1.1 RGBD-cameras

RGBD-cameras, short for *Red-Green-Blue-Depth*-camera, is a type of low-cost camera commonly used for robot vision. The concept became widely popular with the release of the Microsoft Kinect in late 2010.

These cameras consist of two separate parts: one normal color-based camera, and one infrared sensor with accompanying projector. The sensing consists of projecting a deterministic pattern onto the scene using an infrared emitter, and then unprojecting by comparing the image to previously captured patterns at known depths. By interpolating through these patterns throughout the full scene, a depth-image is generated.

1.2 Noise

A common problem in any type of sensing is the introduction of noise into the system. This noise can come from many sources, and be predictable or unpredictable. Examples of noise sources could be frequency hum from electric circuits, flickering lights, air pollution or pure inaccuracy. This noise can skew the results of sensors and make execution more error prone.

There are many approaches to reduce noise. Proper calibration is a good start, but this can only reduce some types external noise. Internal noise of the sensor needs to be analyzed and minimized on a much lower-level such as by using specially constructed algorithms. For sensors that generate some sort of sequence, one very naive (but nonetheless effective) approach is the use of smoothing.

2 Implementation

The purpose of this exercise is to calibrate an RGBD-camera and investigate its characteristics. Then, several smoothing algorithms shall be evaluated to reduce noise in the depth sensor.

2.1 Hardware and environment

This exercise was performed using an *ASUS Xtion Pro*. The camera was connected over *USB2* to a laptop running Linux kernel 4.2.5. The communication to the camera is done using the *Robot Operating System* [ROS] version *Indigo Igloo*. All ROS packages used are compiled directly from GitHub development branch for Indigo Igloo.

Other software used includes the OpenCV libraries, version 2.4.12.2-1.

2.2 Camera setup

As a first step we installed *openni2* package for ROS Indigo Igloo. This package contains drivers for the ASUS camera. After installing the package we connected the camera to the USB 2.0 port on the laptop, and sure that the system can recognize the camera through command `lsusb`. After that we ran the openni2 package using `roslaunch openni2_launch openni2.launch`.

After starting `openni2` we ran the command `rosrun rviz rviz`, which we used to visually inspect the topics published by the camera through *openni2*. In the RVIZ window we needed to set the global option **Fixed Frame** to **camera_link** in order to see the camera data.

We created several visualizations to see what camera publishes on each topic. There are four major types of topics with different subtopics that can be visualized using RVIZ. There are topics for **depth image**, **depth registered image**, **RGB image** and **infrared image**. The depth and depth registered data can also be used as pointclouds. The topics can be seen in table 1. The subtopics are omitted, because they contain the same data but filtered.

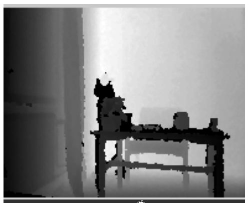
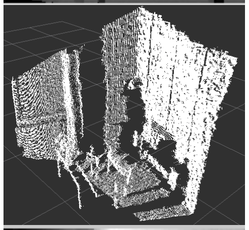
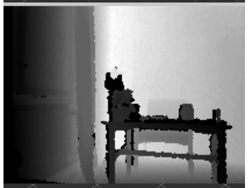
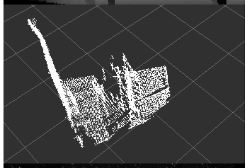
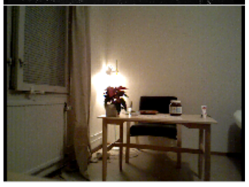
Topic	Description	Data preview
`/camera/depth/image_raw`	Depth image	
`/camera/depth/points`	Point cloud	
`/camera/depth_registered/image_raw`	Combination of depth and RGB images	
`/camera/depth_registered/points`	Same as depth registered, but as a pointcloud	
`/camera/ir/image`	The infrared pattern captured by the sensor	
`/camera/rgb/image_raw`	RGB image	

Table 1: Camera ROS topics

2.3 ROS setup

Instead of using RVIZ to view the data as before, a custom ROS-node can be used. As there are three types of data, color image, depth image, and point cloud, there are three listeners setup to receive this data. An example of the data on these topics can be found in the embedded files below[1]. The `depth image` and `color image` are stored in the OpenCV `y[a]ml` format, and the

[1]If these embedded files do not work, such as if you use Adobe Acrobat, please download them from the `report` folder in the GitHub repository. Verified to work with Okular and Evince on Linux.

point cloud is stored in the *Point Cloud Library* [PCL] `pcd` format.

Point cloud file RGB-image file Depth-image file

2.4 Camera calibration

Next, the camera was calibrated by setting its internal parameters in openni. This is required to get an undistorted image from the camera and therefore the right distance measurement.

To calibrate the camera we used the ROS *camera_ calibration* package. Before running the calibration process, we measured the number of squares on the checkerboard and their size. The checkerboard used had 11 squares on one axis, and 7 on the other; and each side was 6.5 cm long. Using the command `rosrun camera_calibration cameracalibrator.py -size=6x10 -square=0.065 image:=/camera/rgb/image_raw camera:=/camera/rgb -approximate=0.1`, we could then start the calibration process. As shown in the command, we used the `rgb/image_raw` topic to do our calibration.

In the calibration window we could see image from the selected topic with the pattern highlighted. We set the camera be parallel with the floor and pointed it at the wall. After that continuously moved and tilted the pattern inside the camera field of view until X, Y and Size progress bar filled up and, and the camera could be calibrated from the captured data. The data collection took about 2-3 minutes to complete. The software is supposedly able to store the calibration directly to the camera; but this did not work. Instead, we later passed the calibration file to the openni software as a command line argument.

The calibration file was saved into *out.txt* file in the **tmp** directory before being converted into a YAML file. To do that, we used the command `rosrun camera_calibration_parsers convert /tmp/out.txt camera.yaml`, which converted the calibration file into a YAML file in the format used by openni. The calibration file is shown in listing 1.

To check the calibration results we pointed the camera at nearby straight lines (corners, tables etc.) and visually checked if they got distorted around the edges (fisheye lens effect). That showed that the straight lines stayed as straight lines, so the calibration was successful. To quantitatively verify the calibration, we took a screenshot of the camera image, and fitted a line onto one of the straight lines in the picture. As this was possible to do without errors, it shows that the line is not only visually straight but also perfectly straight. Had this not been the case, it would have been possible to measure the error between the perfect line and the camera line but this was found to be unnecessary.

We also noticed that there are some objects, such as lights and reflective surfaces, that ruin the measurements and return wrong distance data.

Listing 1: `The calibration file for the camera`

```
image_width: 640
image_height: 480
camera_name: rgb_PS1080_PrimeSense
camera_matrix:
  rows: 3
  cols: 3
  data: [546.589906, 0, 319.850785, 0, 545.054549, 240.484512, 0, 0, 1]
distortion_model: plumb_bob
distortion_coefficients:
  rows: 1
  cols: 5
  data: [0.073113, -0.09862, 0.002178, 0.001978, 0]
rectification_matrix:
  rows: 3
  cols: 3
  data: [1, 0, 0, 0, 1, 0, 0, 0, 1]
```

```
projection_matrix:
  rows: 3
  cols: 4
  data: [546.70282, 0, 320.499416, 0, 0, 547.7890619999999, 240.692826,
    0, 0, 0, 1, 0]
```

2.5 Noise characterization

The sensors used in the camera suffer from noise, and the first requirement for improvement is to objectively quantify the error. First, 5 measurements of 10 seconds each were recorded with `rosbag`. The measurements were taken *50*, *100*, *150*, *200*, and *300* cm from a white wall. The camera was placed on a table, so that the camera normal was perpendicular to the wall. This ensures that all later noise and smoothing comparisons are made on the same data.

The recorded data can be downloaded here: `https://drive.google.com/folderview?id=0B4HtjhS_-Y3ZZEN4NGctMk9DSDg&usp=sharing`.

First, the errors were measured at varying distance with set window sizes. Fig. 1 shows the error and variances for a 20x20 window, and fig. 2 on the next page shows the same for a 40x40 window. The most obvious pattern can be seen in the right-most plot in the two figures: there is a very strong correlation between distance, and both variance and error. This has been shown before, and is logical: the further the distance, the more things can affect the measurements.

Another interesting pattern is that the mean error barely changes between the two windows; however, a larger window reduces the variance significantly. This is not so much related to reduced errors, but a result of the definition of a variance. If the errors are assumed to adhere to some sort of distribution $X \in (-\infty, \infty)$, then $\frac{1}{|X|}\sum_{X_1}^{X_\infty} = \bar{X}$, i.e., the static error. From this it follows that the variance will be on same order of magnitude as the noise in the signal. However, if too few points are sampled the sum will only approximate the mean, and the variance will become unstable. There is therefore a trade-off between window size and computational cost; i.e. while a larger window may be preferable for preciseness this may be computationally expensive, especially for more complex filters.

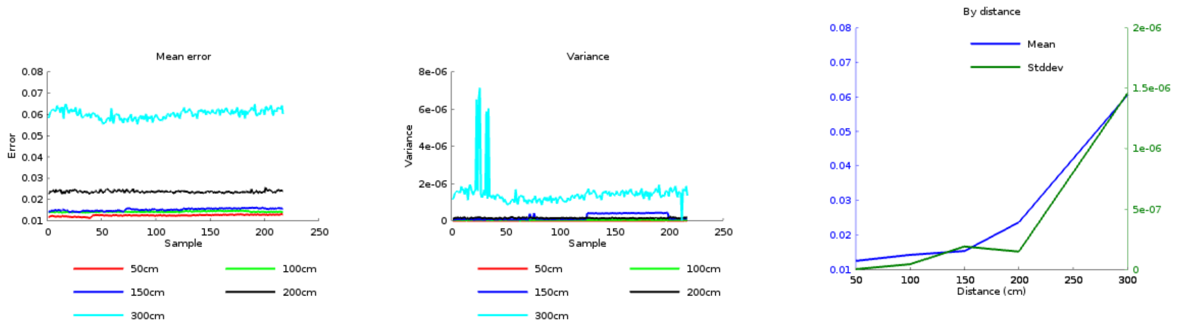


Figure 1: The error and variance for a 20x20 pixel window at the center.

Moving on, the distance can instead be kept constant while the window size is varied. This is shown for 50 cm in fig. 3 on the following page. Here, the effect mentioned above with variance is amplified a lot; and error and variance become very stable for the large window sizes; and in the case of variance also very small. There is a visible artifact between samples 0-60 which has much smaller values than the other values. As most of the values are higher, these smaller values can be discarded.

The cleaned data shown in fig. 4 on the next page highlights how stable the mean is for the larger window sizes. It also shows that something is interfering in the 40x40 measurement, though it is hard to pinpoint what. One (hypothetical) possibility is that there is a small patch of very noisy values that get introduced when the window size increases, and as the window size is still quite small it has a large impact on the variance.

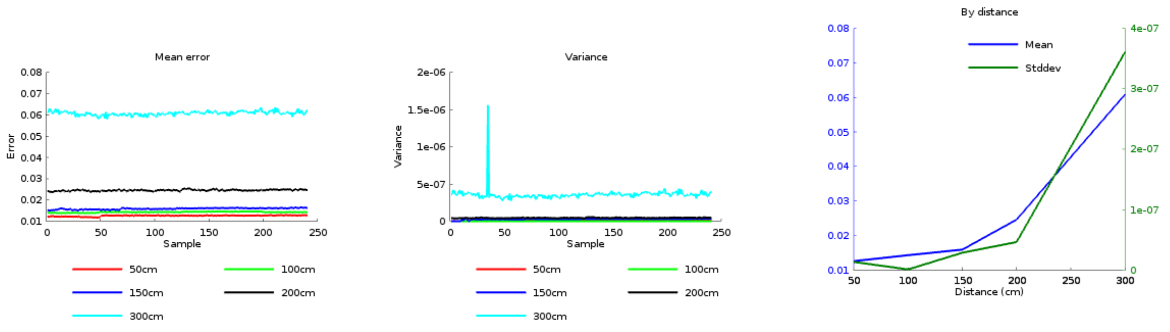


Figure 2: The error and variance for a 40x40 pixel window at the center.

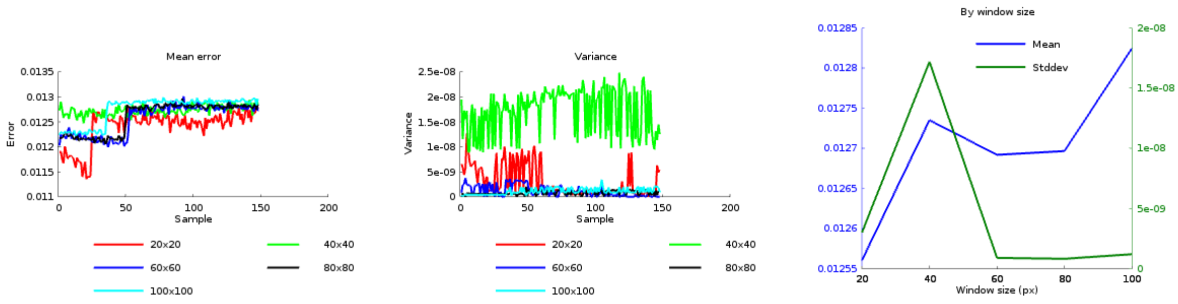


Figure 3: The error and variance at 50 cm for various window sizes.

One last phenomenon can be seen, and this is that the mean increases with window sizes. This is caused by the increase in angle towards the edges of the window. As all distances are measured from the sensor, this causes them to increase slightly. The increase is approximately $\frac{1}{\cos\alpha}$, where α is the angle relative to the camera normal.

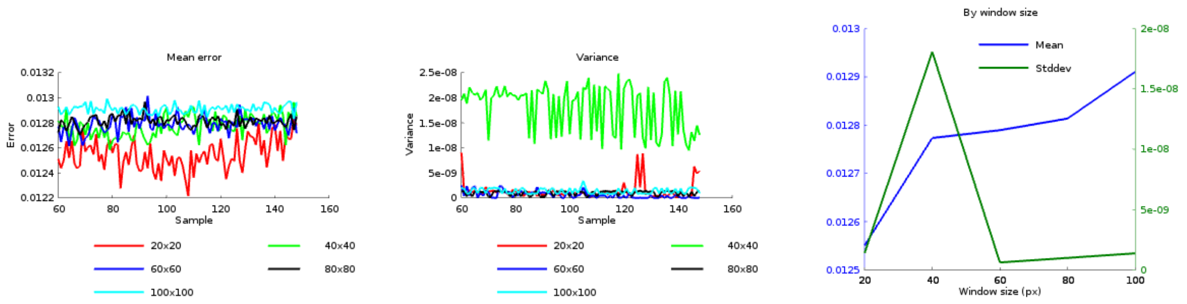


Figure 4: The error and variance at 50 cm for various window sizes with faulty data removed.

2.6 Noise filtering

To improve the input depth image five different filters were implemented for smoothing. The first three filters; *median*, *gaussian*, and *bilateral*, come from the OpenCV library. The last two filters, in the temporal domain, were implemented manually in C++ and are shown in listing 2 on the following page. A large issue when using the filters was handling the large number of invalid values in the input. In some situations, a majority of the elements were NaN, which obviously makes it hard to draw any conclusions from the data. This caused particular problems with the bilateral filter, which suffers a segmentation fault when there are too many NaN values in the input.

Listing 2: `The temporal smoothing algorithm`

```cpp
        // Median and average matrices
        cv::Mat outputM(X_SIZE,Y_SIZE,CV_32F, cv::Scalar(0.0f));
        cv::Mat outputA(X_SIZE,Y_SIZE,CV_32F, cv::Scalar(0.0f));

        // Calculate temporal and median
        for (int y = 0; y < Y_SIZE; y++)
        {
            for (int x = 0; x < X_SIZE;  x++)
            {
                // Separate all the good values from bad ones
                std::vector<float> numbers;
                for (int i = 0; i < numElements; i++)
                {
                    float val = buffer[i].at<float>(x,y);
                    if (val < CUTOFF_DISTANCE && val > 0)
                    {
                        numbers.push_back(val);
                    }
                }

                int length = numbers.size();

                // Sort for median
                std::sort(numbers.begin(), numbers.end());
                if (length > 1)
                {
                    // Median
                    if (numbers.size() % 2 == 0)
                    {
                        outputM.at<float>(x,y) = (numbers[(int)length/2 - 1]+numbers
    [(int)length/2])/2.0f;
                    }
                    else
                    {
                        outputM.at<float>(x,y) = numbers[(int)length/2];
                    }

                    // Mean
                    float sum = 0;
                    for (int i=0; i < length; i++)
                    {
                        sum += numbers[i];
                    }
                    outputA.at<float>(x,y) = sum / (float)length;
                }
                else
                {
                    // NOTE: Setting to 0 might not be the best approach, though we
    know it's an invalid value (because 0 distance makes no sense)
                    outputA.at<float>(x,y) = 0;
                    outputM.at<float>(x,y) = 0;
                }
            }
        }
```

In order to solve this issue, all NaN-values in the input are set to 10000 before being passed to the bilateral filter. By the edge preserving property of bilateral filtering, this should reduce the impact they have on the actual filtering process. After the filtering is done, every cell which was set to 10 000 is set to 0; and the mean and variance is calculated on all non-zero elements.

Another solution we considered - but did not implement - was to use local median-filtering at each `NaN`. We deemed this to be a very computationally expensive process, but could possibly yield better results.

The result of filtering our depth-images is seen below in fig. 5 and 6. Just as in the previous section, the unreliable data at the start is removed to clean up the results. As can be seen, the two temporal smoothing algorithms far outperform the other algorithms at 50 cm for both window sizes, though the difference is very small. However, when increasing the distance up to 300 cm this changes as shown in fig. 7. In this case, the gaussian and bilateral filter have a better performance on both variance and mean error.

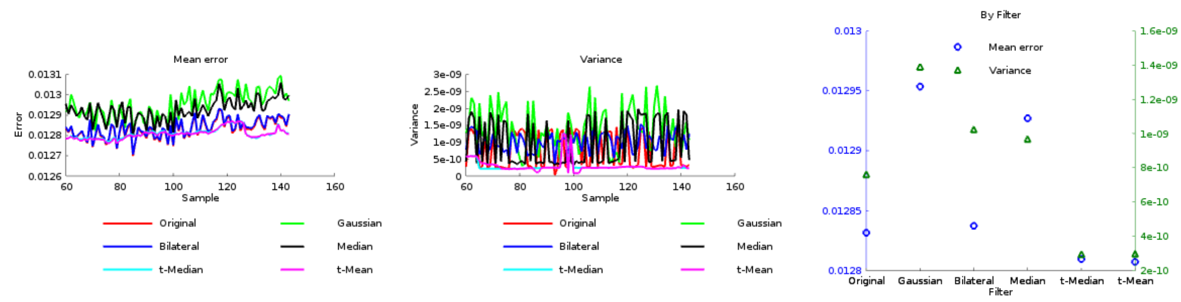


Figure 5: Mean and variance for the filters at 50 cm with 20x20 window size.

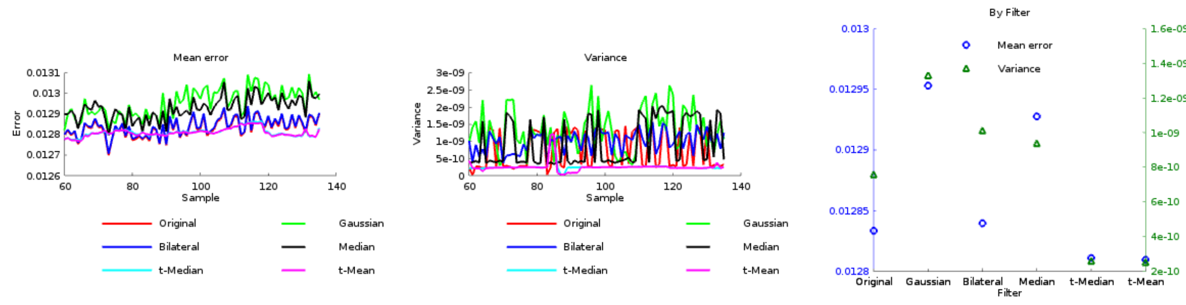


Figure 6: Mean and variance for the filters at 50 cm with 40x40 window size.

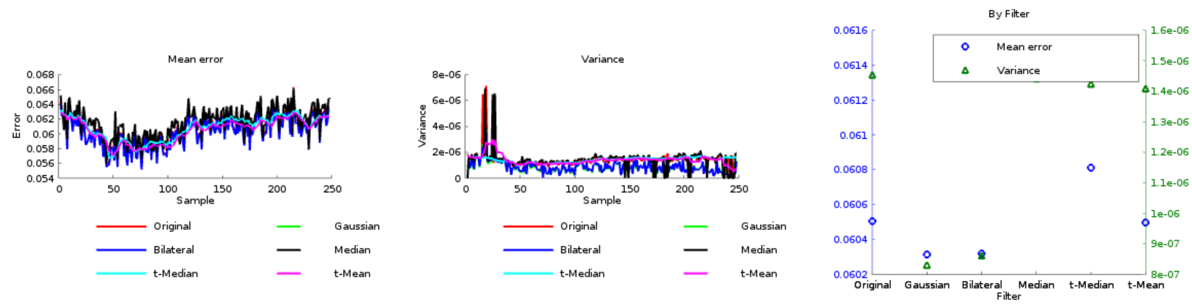


Figure 7: Mean and variance for the filters at 300 cm with 20x20 window size.

To visualize the smoothing algorithms we set up a scene with a small box in the middle, and a bigger box behind it. This allowed us to clearly see the edges of the smaller box through the camera. As we can see in the original image (fig. 8) there is a lot of noise and the edges are not as clear as they should be. The used filters should smooth out the noisy edges to be able to get better measurements. The result of filtering the area around the smaller boxis shown in fig. 9 on the following page.

The filtered area is 80x80 px in the middle of the camera data. The first image is the original without any filters applied, but with NaN values set to 0. The second image is filtered with a gaussian kernel. This filter blurs the image and reduce details, which results in smoother edges and reduced noise. The last picture on the upper row is median filtered. We can see that median filter preserves the edges while removing some noise.

The first picture in the second row has been filtered using a bilateral filter. This filter also smooths, while preserving large intensity differences. The second picture in the same row is the average over 10 sequential images. This method reduces noise which appears only in few pictures or reduce effect of moving objects, that could affect the measurements, as well as oscillations in measurements. Last picture is the median of 10 images applied per pixel. Effect of that method is similar to the previous one, but it preserves more details. The effect of temporal smoothing was so strong that a hand could be repeatedly moved quickly in and out of the camera field, without affecting the measurements.

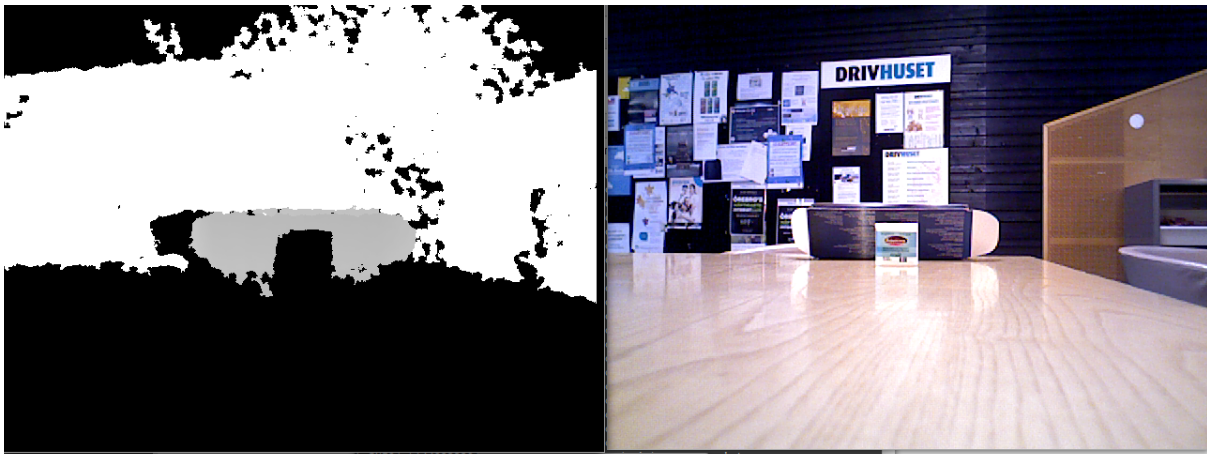


Figure 8: Reference image for demonstrating filtering effects.

3 Results

During this exercise we could see that data received from this ASUS camera is very noisy and because of this they might not be completely reliable. On top of that we found that when viewing surfaces that emit or absorb light, we get wrong distance values.

Before we the calibration we could see that camera suffers from the fisheye lens effect, that distorted objects near the edges of the field of view. The calibration process reduced this effect to minimum and resulted in straight undistorted lines.

In order to reduce noise we tried several methods. The first method was to increase the window from which we computed the mean distance and variance. Our results showed that bigger window give better results, but in the real time usage this can be computationally expensive. We also verified previous results that long distances can be detrimental to the camera performance.

The second method we tried was to use filters to smooth the image and as a result reduce the noise. As our results showed, the best performing filter were temporal filters on the 50 cm distance, but gaussian and bilateral filter at 300 cm distance. This could be related to the increasing variance and mean at high distances, which causes self-similarity to become stronger than sequence-similarity of the images.

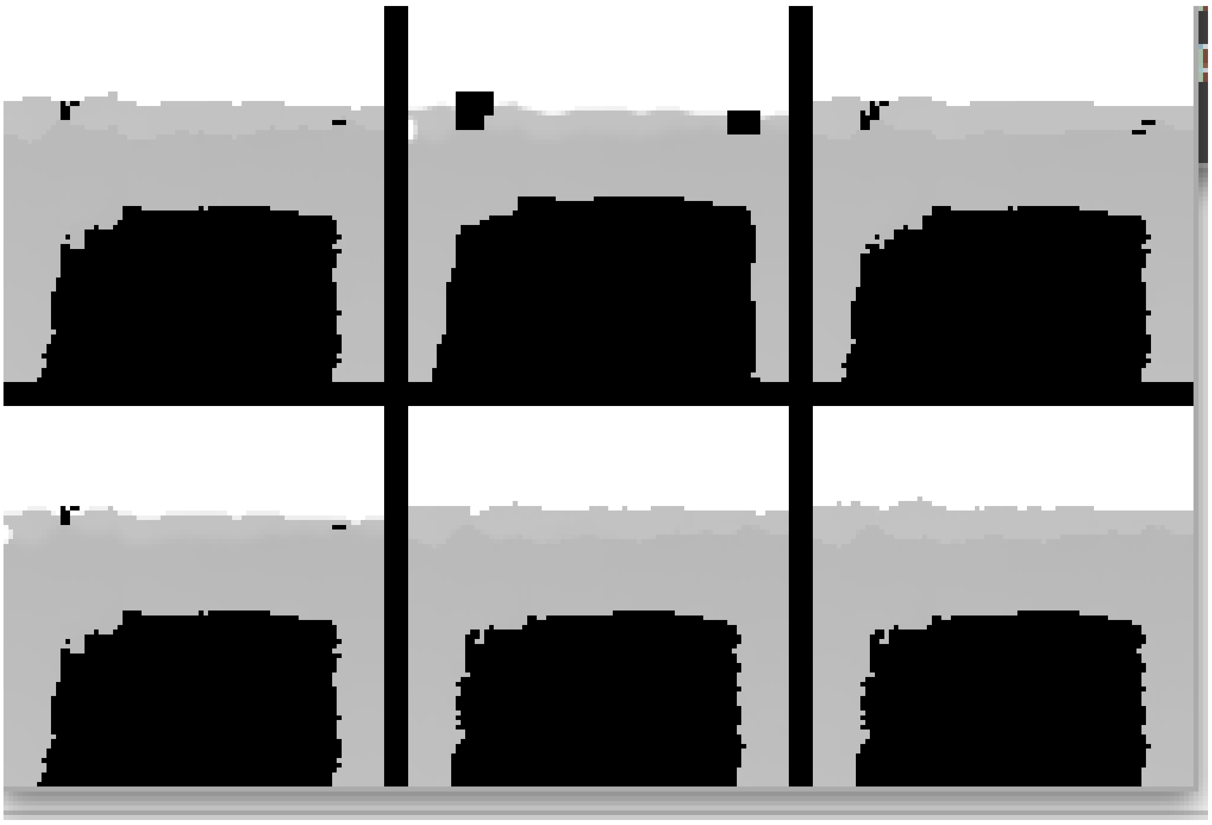


Figure 9: Different filters applied to the center 80x80 px of the reference image. From the top left corner - original image, gaussian, median, bilateral, temporal average, temporal median.

The results are not conclusive for other situations than a white wall, because the camera, and by extent the filters, will behave very differently in different environments. We have noted that different surfaces will drastically change the performance, and we noticed that flickering lights interfered with our measurements. Because of this, we can't say that using these filters are the best for every situation, despite the fact that it worked best in our setup.