Laboration 2: RGBD-cameras

Sensors and Sensing

Marek Bečica, Tom Olsson

December 4, 2015

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

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

Contents

Code listings

List of Figures

List of Tables

1 Theory and motivation

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, a full 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 that make algorithm much more error prone.

There are many approaches to reduce noise. Proper calibration and good testing environments is a good start, but this can only reduce 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 had to install *openni2* package for the corresponding version of ROS. 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. We made sure that our system can recognize the camera through command `lsusb` and after that we run the openni2 package using `roslaunch openni2_launch openni2.launch`.

During the first launch there was a warning about no calibration file found, but we didn't need it yet for the camera testing. After running openni2 we run in the new terminal window command `rosrun rviz rviz`, which we used for inspecting the topics where camera publishes its data. In the RVIZ window we needed to set global option **Fixed Frame** to **camera_link** in order to see the camera data.

We need to create 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**. From the depth and depth registered topic we can also get point cloud. The topics can be seen in table 1 (we omitted the subtopics, because they contains the same data, but with different compress method, which is not relevant for this task).

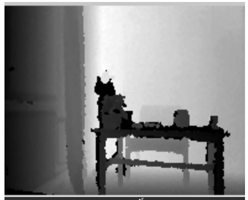
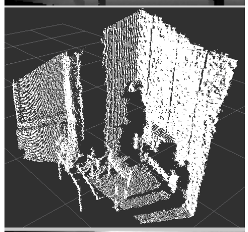
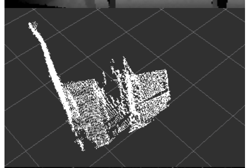
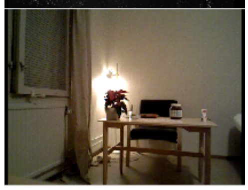
Topic	Description	Data preview
`/camera/depth/image_raw`	Contains depth image	
`/camera/depth/points`	Contains point cloud of the point distances	
`/camera/depth_registered/image_raw`	Contains depth image projected from both sensors	
`/camera/depth_registered/points`	Contains point cloud of the point distances projected from both sensors	
`/camera/ir/image`	Contains infrared image	
`/camera/rgb/image_raw`	Contains 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

In this part we were supposed to calibrate the camera which requires to setup internal parameters. After that we should be able to get not distorted image from the camera and right distance measurement.

As a first step we had to install *camera_calibration* package. After that we listed all topics where camera publishes its data and we chose topic `/camera/rgb/image_raw`. Before running the calibration process, we measured the number of squares on the checkerboard and their size. 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 run the calibration application with correctly setup parameters according to our measurements of the calibrating pattern.

In the calibration window we could see image from the selected topic with detection of the pattern. We set the camera to the parallel position with the ground and pointed it to the wall. After that we kept moving and tilting the pattern all over the camera field of view until X, Y and Size progress bar showed long line and button Calibrate light up. This process took up about 2 minutes until we got good results and Calibration button lighted up. We pressed the button and then waited for few minutes until the application was able to compute the right parameters for our camera. After that buttons Save and Upload lighted up. We saved the calibration file and tried to upload it to the camera firmware, but for some reason that function didn't worked properly and we had to set the calibration file as a parameter of the openni2 package.

The calibration file was saved into *out.txt* file in the **tmp** directory and we needed to convert it to YAML file. In order to do that, we used command `rosrun camera_calibration_parsers convert out.txt camera.yaml` which converted the calibration file into yaml file which we could use for the openni2 package. The calibration file is shown in listing 1.

To check the calibration results we pointed the camera at some straight lines (wall corners, table etc.) and checked if they got distorted around the edges. That showed that the straight lines stayed as straight lines, so the calibration was successful. To quantitatively verify if the calibration worked we run the code from the Task 4: Noise characterization, that measured average and standard deviation for the distance in the small window. Then we pointed the camera to the objects in different distances and checked if the mean of the distance is the same as the reality, which mostly was within the range of the camera.

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 against the same baseline.

First, the errors were quantified at varying distance with set window sizes. Fig. 1 shows the error and variances for a 20x20 window, and fig. 2 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 by other researchers, and is logical: the further the distance, the more things can interfere with measurements such as light conditions, dust particles, and so on.

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 $\sum_{X_1}^{X_\infty} = \bar{X}$, i.e., the static error. From this it follows that the variance will then also 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.

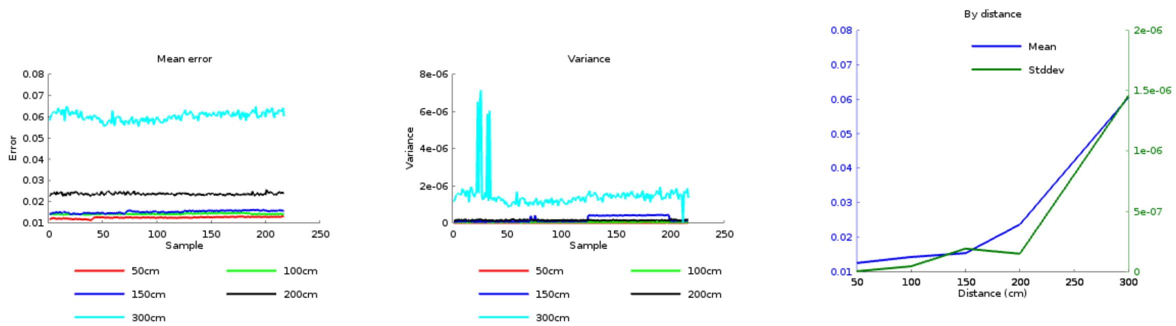


Figure 1: The error and variance for a 20x20 pixel window at the center.

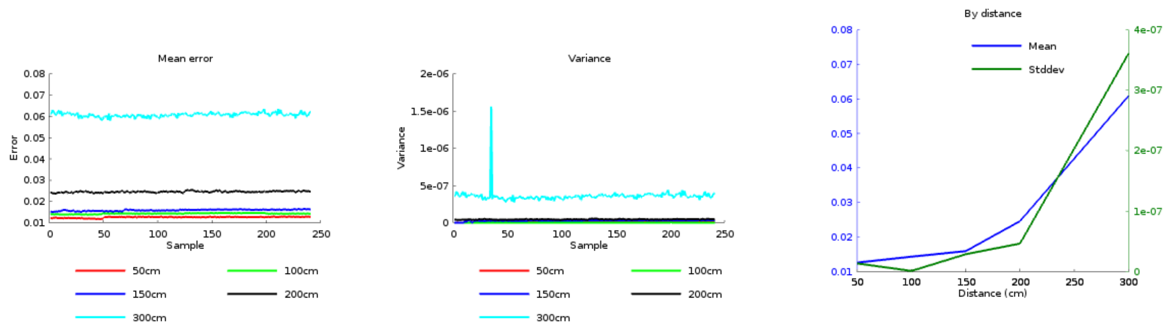


Figure 2: The error and variance for a 40x40 pixel window at the center.

Moving on, the distance can instead be kept constant while the window size is varied. This is shown for 50 cm in fig. 3. 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.

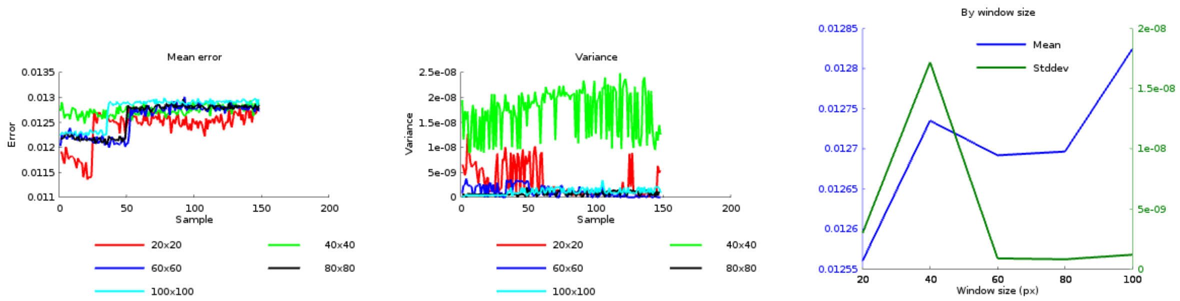


Figure 3: The error and variance at 50 cm for various window sizes.

The cleaned data shown in fig. 4 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.

One last conclusion 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.

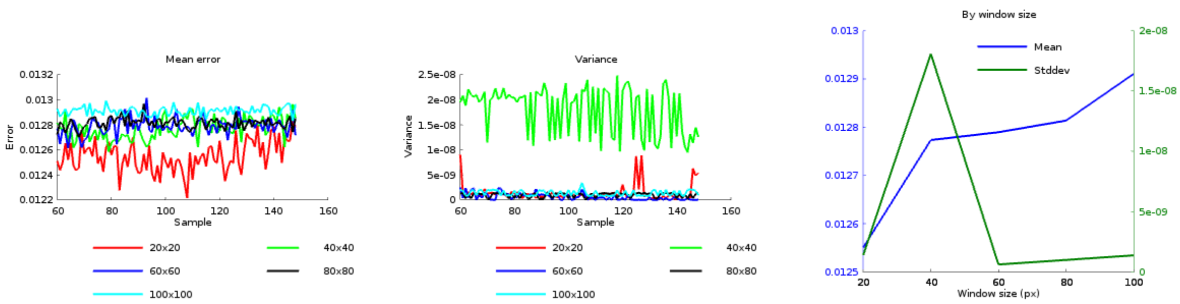


Figure 4: The error and variance at 50 cm for various window sizes with faulty data removed.

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.

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

Listing 2: `The temporal smoothing algorithm`

```cpp
        // Set the asdlasdom
        buffer[buffPoint] = submatrix;
        buffPoint = (buffPoint + 1) % 10;
        if (numElements < 10){
            numElements++;
        }
        for (int y=0; y < Y_SIZE;y++){
            for (int x = 0; x  < X_SIZE;  x++){

                std::vector<float> numbers;
                for (int i=0; i<numElements; i++) {
                    float val = buffer[i].at<float>(x,y);
                    if (!std::isnan(val) && val < 10 && val != 0){
                        numbers.push_back(val);
                    }
                }
                int length = numbers.size();
                std::sort(numbers.begin(), numbers.end());
                if (length > 1)
                {
                    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];
                    }
                    float sum = 0;
                    for (int i=0; i<length; i++){
                        sum += numbers[i];
                    }
                    outputA.at<float>(x,y) = sum / (float)length;
                }
                else{
                    outputA.at<float>(x,y) = 0;
                    outputM.at<float>(x,y) = 0;
                }
            }
        }
```

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.

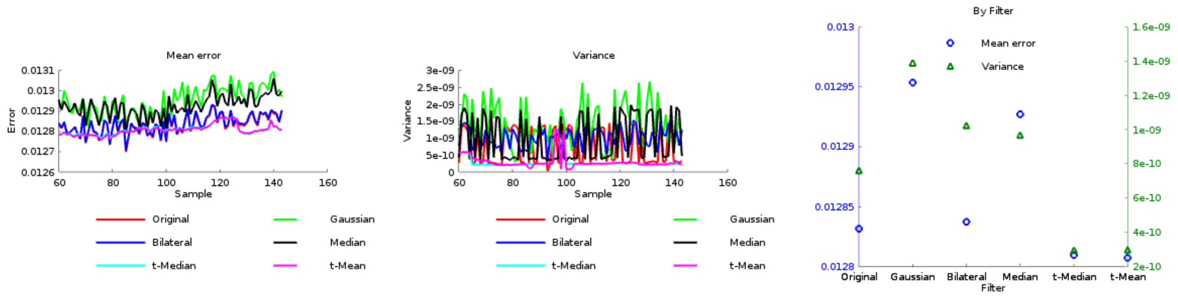


Figure 5: Mean and variance for the filters at 50 cm with 20x20 window size.

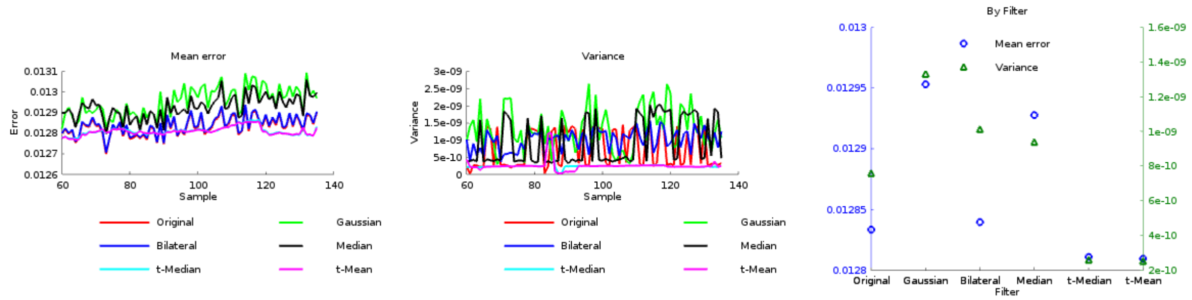


Figure 6: Mean and variance for the filters at 50 cm with 40x40 window size.

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. 7 on the following page) there is a lot of noise and the edges are not as clear as they should be. The used filters should smooth out the noise to be able to get better measurements. The resulting image 8 on page 10 shows all filters applied to the center of the image.

The first image is the original window 80x80 px without any filters applied but we removed NaN values and replaced them with high number. Second image is original with applied gaussian blur filter. This filter blurs the image and reduce details, which results in smoother edges and reduced noise. Third picture is image with applied median filter. We can see that median filter preserves the edges while removing some noise.

First picture in the second row is original image with applied bilateral filter. This filter also preserves sharp edges and removes some noise by smoothing the image. The second picture in the same line is the average over 10 images. That method can reduce some noise, which appears only in few pictures or reduce effect of moving objects, that could affect the measurements. Last picture is the median over 10 images. Effect of that method is similar to the previous one, but it preserves more details.

Figure 7: Reference image for demonstrating effects of the filters

3 Results

References

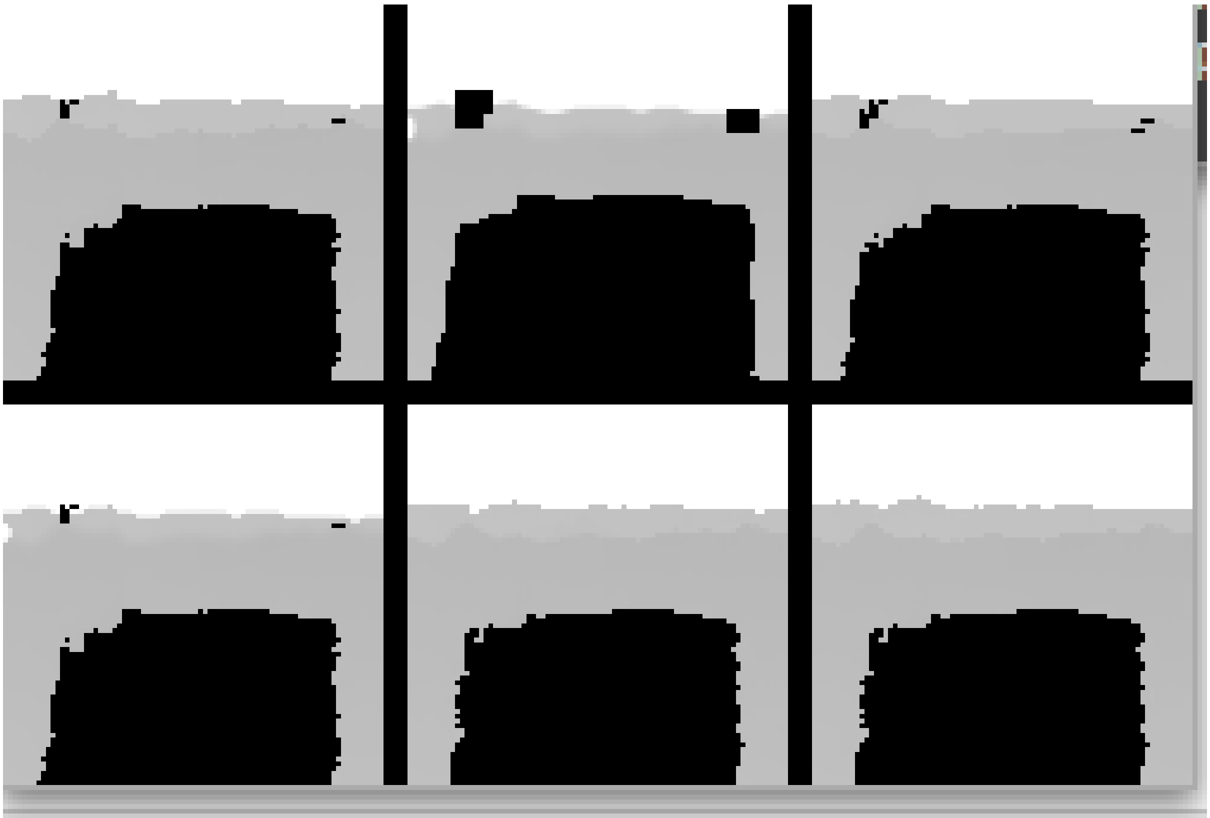


Figure 8: This image shows different filters applied to the center of the window 80x80 px. From the top left corner - original image, gaussian, median, bilateral, temporal average, temporal median