# 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***

## 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, and then unprojecting them by comparing 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 for the depth data.

### 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 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

### 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 pointcloud, 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.

Pointcloud file RGB-image file

### 2.4 Camera calibration

```
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 before being able to reduce it, measurements need to be made. 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 measurements are made against the same baseline.

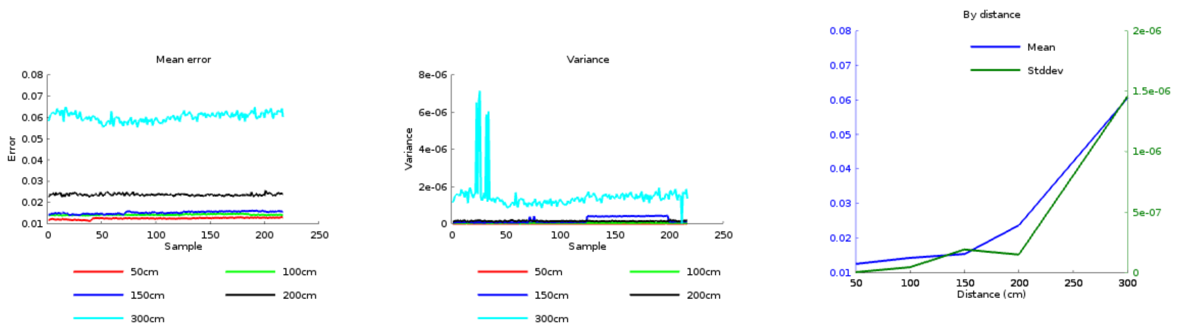

Figure 1: The error and variance for a 20x20 pixel window at the center.

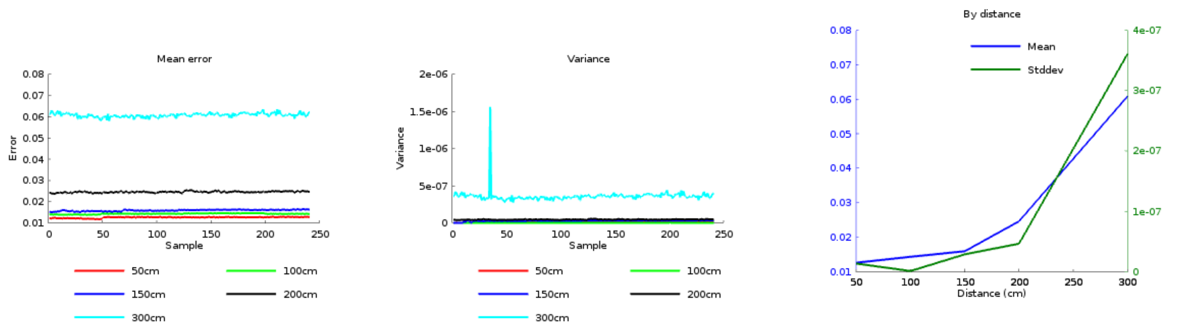

Figure 2: The error and variance for a 40x40 pixel window at the center.

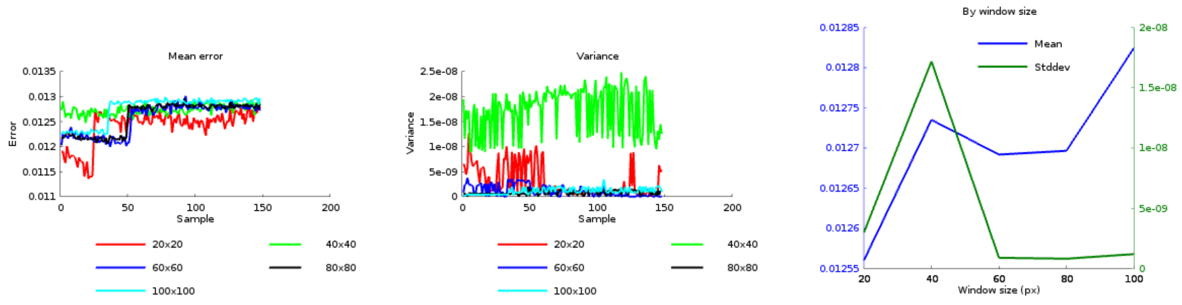

Figure 3: The error and variance at 50cm for various window sizes.

### 2.6 Noise filtering

A large issue when implementing 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. The *gaussian* and *median* filters from **OpenCV** work fine without any modification. The bilateral filter however suffers a segmentation fault when there are 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 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.

## 3 Results

## References