Simultaneous Localization and Mapping (SLAM)
Towards an autonomous search and rescue aiding drone

Frank Heukels
University of Twente, The Netherlands

January 16, 2015

Samenvatting

Wanneer overlevenden moeten worden gevonden in een beschadigd gebouw is het veiliger voor de hulpverleners om van te voren een 3D kaart te hebben van de binnenkant van het gebouw dat is gemaakt door een robot. Deze kaart kan dan worden gebruikt om overlevenden op te sporen en ze te redden. Deze thesis stelt een lichtgewicht algoritme voor dat op een robot kan worden uitgevoerd om een 3D kaart te genereren. Deze kaart wordt gemaakt door meerdere RGB- en diepte beelden samen te voegen in 1 groot 3D model. Om dit te realiseren wordt de beweging van de drone tussen twee opeenvolgende frames berekend. Deze beweging wordt berekend door naar de beweging van herkenningspunten te kijken die voorkomen in twee opeenvolgende RGB-beelden. Deze beweging wordt vervolgens geoptimaliseerd door het gebruik van de "point to projection scan matching" methode. Wanneer de beweging van de drone groot genoeg is, wordt het nieuwe beeld toegevoegd aan de 3D kaart.

Om het algoritme te kunnen testen is er gebruik gemaakt van een dataset waar de camera door een hal wordt bewogen. Het algoritme is in staat om een 3D kaart te maken met een gemidelde van 17 beelden per seconde (op een HP 8570w). Er is echter wel een oplopende fout in het model. Deze fout zou kunnen worden verkleind door het gebruik van "Inertial Measurment Unit" of door het implementeren van "loop closing" of graaf-optimalisatie methoden.

Abstract

When having to find survivors in a heavily damaged building it is saver to send a drone that generates a 3D map before sending in emergence personnel. The generated map can then be used to locate the survivors and rescue them. This thesis proposes a light weight implementation of a 3D mapping algorithm that is designed to be run onboard of a drone. The map is generated by stitching together multiple images from a RGBD (RGB and depth image) camera by looking at the movement of the drone between two consecutive frames. This is accomplished by calculating an initial transformation by using ORB features and then optimizing this transformation using the point to projection scan matching method. If the found transformation indicates that a large enough distance is traveled then the next frame is added to the feature map. This feature map is used later to generate a 3D model.

To be able to test the implementation a dataset is recorded by moving the camera through a hallway. The implementation is able to stitch together all the images into a single 3D model with an average frame rate of 17 fps on a HP 8570w. There is however an ever-increasing error that can be reduced by using an Inertial Measurement Unit (IMU) or by implementing a loop closing algorithm or by using graph optimization techniques.

Contents

1 Introduction

Due to recent advances in electronics and computer science autonomous, drones are becoming a more and more viable solution to multiple different problems. Examples of this are drones that carry life saving equipment like defibrillators or drones that can follow the user and capture video footage.

This thesis proposes a way of using a drone (hexacopter) for emergency search and rescue inside of buildings. After a disaster it can be too dangerous for emergency personnel to enter heavily damaged buildings to look for survivors. Before sending in emergency personnel, a drone could be send in that creates a 3D map of the inside of the building. This map can then be used to locate survivors and organize a rescue operation.

There are multiple challenges that need to be overcome for this to work. The first of which is that in order to create a map the drone needs to know where it is, but to be able to know where it is it needs a map. This problem is known as the Simultaneous Localization and Mapping (SLAM) problem. The solution for this problem is to try to calculate the movement of the drone between sensor measurements. For example, lets assume that a camera is used as a sensor. Then the movement is determined by: capturing an image, moving the drone and capturing a second image. By looking at the differences between the images it is possible to calculate the movement of the drone.

The second problem that needs to be overcome is the limited amount of available processing power on board of the drone. SLAM implementations tend to required large amounts of processing power and memory to be able to generate accurate maps. Because of this, the algorithms are either performed offline or use a wireless link between the drone and a base station. Because there is limited processing power and memory onboard of a drone it will be necessary to create a scaled down version of existing SLAM methods.

The goal of this project is to find a sensor that can be used to generate a 3D map, scale down the existing SLAM methods so that they can be run onboard of a drone and to combine the two to generate a 3D map.

2 Related work

Over the years allot of research has been done towards trying to build robot that can autonomously create a map of the environment and use this map for navigation. One of the problems with this being that in order to build a map, first the location needs to be known, but in order to know the location a map is required. This problem is referred to as SLAM (Simultaneous Localization And Mapping) and has been solved in different ways.

One solution to this problem is by using Extended Kalman Filters (EKF) to track the location and uncertainty of each feature that was extracted from the sensor data [6, 42]. These systems try to estimate the uncertainty of the features by comparing the odometry sensors (accelerometer, gyroscope and magnetometer) with the observations that where made by the sensor. One of the main problems with using an EKF is the amount of processing that is required when calculating the inverse of the covariance matrix when there are a large amount of features. In an effort of reducing the computational complexity the Sparse Extended Information Filter (SEIF) [10] was introduced. This method tries to optimize the calculations that are required for the EKF by taking advantage of sparse matrices. The downside of this method is however, that a higher error is created and a calculation is introduced in the prediction step that requires a high amount of processing.

A common cause of errors in these kinds of systems is the drift of the odometry sensor. In an effort to deal with this problem a Rao-Blackwellised particle filter based method was introduced called FASTSLAM [18]. This method tries do deal with the odometry uncertainties by adding random offsets to the odometry data for each particle.

The methods mentioned above are commonly used in combination with a lidar. This is due to its high accuracy and its ease of use. The downsides are however that they are relatively expensive ($1000+) and that they measure in a 2D plane. A less expensive option is to use a ring of sonars.

Instead of using lidar or sonar, some implementation are based on using camera systems. These systems either use a single camera (monocular vision) or a dual camera (stereo vision) setup. One of the main challenges of using camera systems for SLAM is accurately determining depth in the image. Monocular vision systems achieve this by finding features in consecutive images and determine the depth based on the estimated movement of the camera. Because these systems require a movement estimate, it is difficult to accurately determine the depth due to errors in the movement calculation. Stereo vision systems have the advantage of being able to determine the depth from a single frame. Here the depth is determined by finding corresponding features in the left and right image. Based on the feature location and the corresponding depth, methods like iterative closest point (ICP) can be used to find the 6DOF rigid body transformation that resembles the camera movement between the frames.

A relatively new sensor that is used for solving the SLAM problem is the cheap RGBD sensor ($180). This sensor output two images, one RGB image and one depth image. The depth image is created by projecting a pattern of infrared dots onto the

environment. This pattern is then captured by and infrared camera and the depth is determined by looking at the distortion of the pattern.

Implementations that use the RGBD sensor for solving the SLAM problem usually use the same basic method. First, features are found in the RGB image and they are matched to features in previous images by using feature descriptors. By adding a depth value to the matched features, a transformation is calculated. Next an Iterative Closest Point (ICP) method is used to merge together all the images into a single large 3D map. A property of these implementations is that the frame rate is relatively low on desktop pc's, only a few frames per second (< 4 fps).

Another approach is proposed by Yuichi Taguchi et al. [40] where they use planes in the image instead of feature points. The advantage of using planes is that they are more robust to noise and there are usually fewer planes in a depth image than there are features in the RGB image. The downside of this method is that extracting the planes requires a large amount of processing power, which results in a low frame rate on desktop pc's (2 fps).

3 Sensors and feature extraction

In order for a robot to create a map of the environment and localize itself in it, it has to be able to determine the distance to certain objects. This is usually achieved by using a lidar (light radar), RGB camera (mono or stereo), RGB-D camera or sonar. To be able to efficiently use the sensor data it might be necessary to extract features from it. The basic operation and the feature extracting methods of each sensor are described in the following paragraphs.

3.1 Laser range finder (Lidar)

The most commonly used sensor for sensing the environment in 2D indoor SLAM is the laser range finder. It works by sending a laser pulse towards an object and measuring the time it takes to reflect from the object and return to the sensor. By repeating this for multiple different angles it is possible to measure a 2D slice of the environment. Because of its high accuracy and ease of use this is a very suitable sensor for SLAM. It is however, relatively expensive ($1000+). In the literature it is usually implemented in two different ways: by using scan matching or by extracting corners.

3.1.1 Scan matching

The main principle of scan matching is trying to find an optimal rotation and translation of the new sensor endpoints so that they fit the previously made scans (fig.1). Here the left image shows the initial orientation of the new scan (blue) and the previously made reference scan (red). By finding corresponding points between the red and the blue points it is possible to find a rigid body transformation that minimizes the distance between the two scans (right image).

In order to reduce the processing time, the scans can be downsampled using gridmaps as is done in [43]. Gridmaps are discrete representations of the map where the map is divided into equally sized cells. Each of these cells contains a probability of being occupied, empty or unexplored. Because the grid map is a discrete representation, bilinear interpolation can be used to increase the accuracy of the scan matching algorithm [36].

3.1.2 Corner extraction

Another way of using the laser range finder data is to extract corners and centers of walls as is done in [13]. Here they make a distinction between two kinds of features: fixed features and moving features. Fixed features (the filled black circles in fig. 2) are the corners of walls and can be found by looking for non differentiable local minimum or maximum (point A and B in fig. 2) or jump discontinuities (point C in fig. 2). Moving features (open circles) are the centers of walls (point D in fig. 2), which can be found by looking for local minimums or maximums. Scan matching of the features is used to merge the new features with the feature map.

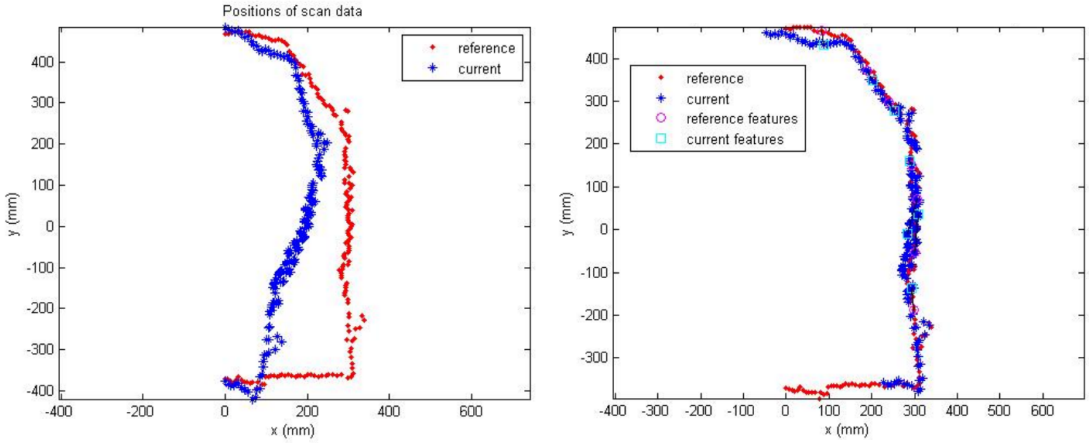


Figure 1: Effect of scan matching [27]. Left: before scan matching. Right: after scan matching.

The downside of using a lidar for 3D mapping is that it only captures 2D slices of the environment. A straightforward way of dealing with this is to try to combine multiple 2D slices together by using the onboard odometry sensor. Another way is to generate 3D point clouds with a lidar by mounting it on a pivoting platform as is done in [8]. Here a 3D scan is made by changing the pitch of the lidar and merging multiple scans together.

3.2 Vision RGB

Instead of using an expensive laser range finder to sense the world, implementations have been made that use a relatively cheap mono or stereo vision [15, 16, 20] system. Because the images don't directly contain any depth information, some form of image processing is required. The general principle behind determining depth from images is by finding corresponding features in multiple images and determining the depth by looking at the change in position. For mono vision implementations, this comes down to using successive images and using the movement of the robot and the change in position in the images to determine the depth. Stereo vision methods have the advantage of being able to determine the depth from a single frame. This is done by looking for corresponding features in the left and right image and determining the depth by looking at the difference in location. The advantage of the stereo system is that no relatively noise movement information is required, which results in a more accurate depth determination.

A way of finding corresponding points in both images is by using the sum of absolute distances. Sum of absolute distances is a method that tries to find correspondences between images for each pixel. The general principle is that it takes the sum of a pixel and its neighbors in one image and it tries to find a pixel in approximately the same position in the other image that when summed with its neighbors gives the same value.

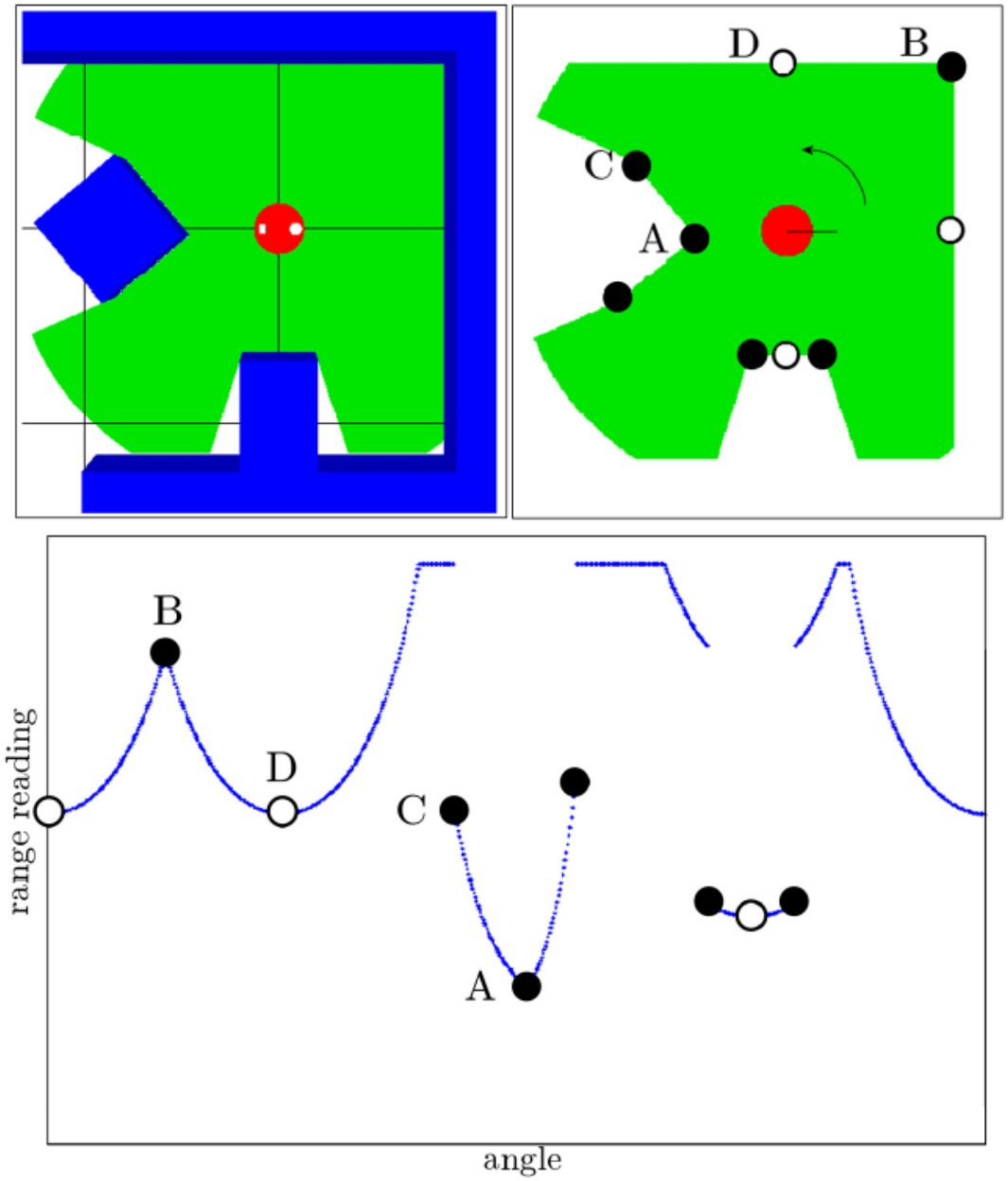


Figure 2: Top left shows an environment and the resulting laser scan. Top right shows the location of the extracted features. The bottom image shows the lidar data and the extracted features [13].

If the difference between the sums of each pixel is below a threshold, then they are seen as corresponding pixels. An efficient implementation of this is described in [4]. It is however difficult to determine if a certain area has been seen before because no features

are extracted.

Instead of trying to find correspondences for each pixel in every frame methods like Scale Invariant Feature Transform (SIFT) and saliency operators try to find features in textured areas of the image. Because a set of features is extracted from the image it is possible to determine if a certain area has been seen before.

Another method for extracting features from images is by extracting edges [20], which can be accomplished by using a Canny edge detector. The advantage of using an edge detector is that it works well in areas where there is very little texture available (empty hallway).

Cameras have also been used as a way of increasing the accuracy of other sensors. An example of this is [24], where corners are extracted from the camera image and compared to lidar data.

3.3 Vision RGBD

Multiple papers have presented ways of using both the depth image and the camera image for mapping of an environment [22, 34, 39]. The process of merging multiple scans together is similar to monocular systems. It works by finding features in the new RGB image and matching these features to features in previous RGB images. The difference with monocular systems is that these systems don't need to calculate the depth of the features because the depth image is all ready available.

This depth image is generated by using the integrated (infrared) IR pattern projector. Instead of measuring the time it takes for IR light to travel from the camera to an object and back to the camera again, like a lidar, the RGBD sensor projects a pseudo random uncorrelated pattern (fig. 3) onto the object. This pattern consists out of multiple IR dots that are projected by the IR projector. The reflection of the pattern off the objects is then captured by the IR camera. By comparing clusters of dots to hard coded patterns the distance is determined.

Besides looking at the relative distance between dots, also an astigmatic lens is used. This lens distorts the dots and results in an elongation of the dot, which is based on the direction of the depth variation [31]. By looking at the size and orientation of the elongated dots it is possible to refine the depth measurement.

Another method [41] proposes to improve the accuracy by also matching planes,instead of only using features, that are extracted from the point cloud, which are found by applying RANSAC. The reason for using planes is that they are more robust to noise and there are fewer planes than features in a frame. If, however, there are fewer than 3 planes (minimal amount to be able to perform 3D matching) present in the new scan, then the algorithm will use 2 planes and 1 feature point. The points are extracted from the RGB image using SURF and they are back projected onto the depth image. If there are less than 2 planes available, then the algorithm will try to perform the registration in the following order: 1 plane and 2 points or 3 points. After the registration, an interpretation tree is used to prune false correspondences and bundle adjustment is used to optimize the global map.

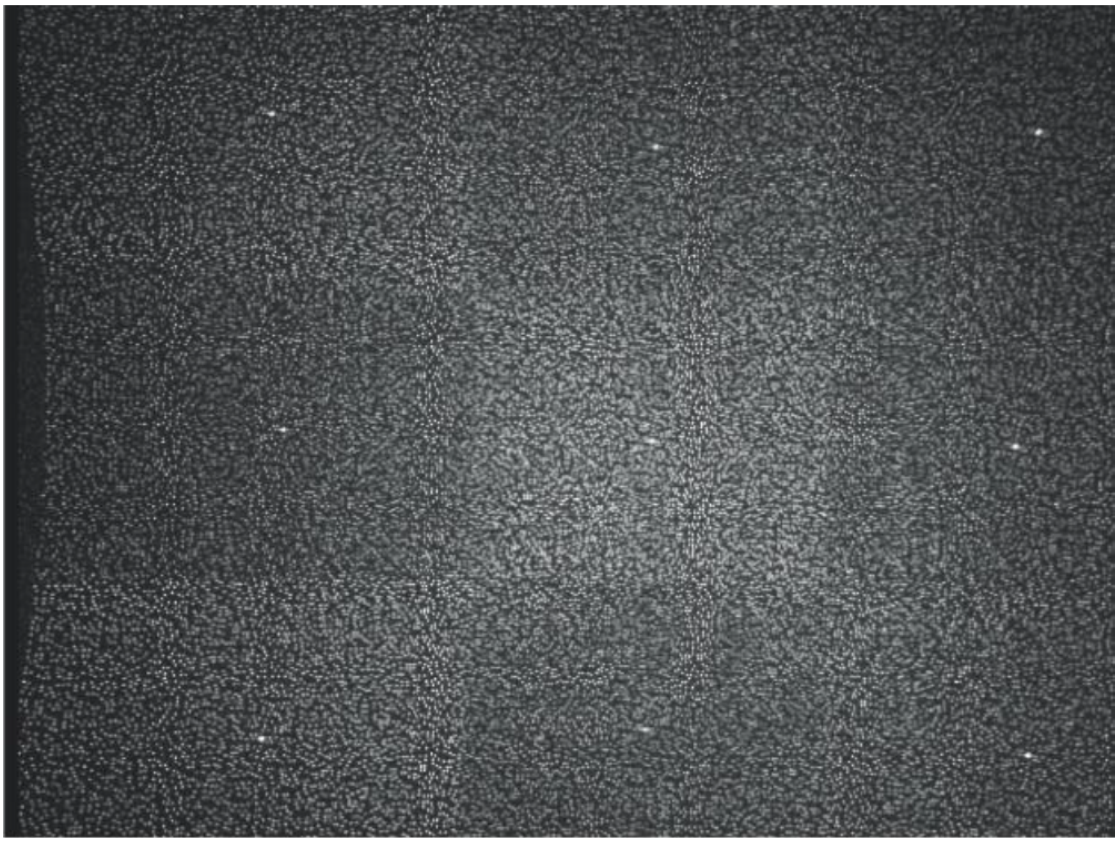


Figure 3: Light coding pattern [31].

3.4 Sonar

A less commonly used sensor for SLAM is sonar [3, 30]. The main principle behind using sonar for detecting objects is that the sonar sends out a high frequency sound wave in a three dimensional cone. When this wave hits an object it gets reflected off it and the reflection is then detected by the sonar. By measuring the time that it takes for the wave to come back to the sensor it is possible to detect the distance to objects. The advantage of using sonar over using lidar is that it detects objects in a 3D cone instead of a 2D plane. It does however, have a large drawback. The sound wave can be reflected away from the sensor, which either results in the sensor sensing nothing or sensing a wave that has been reflected by multiple objects. A way of overcoming this problem is by combining sonar with lidar as is done in [3]. When sonar indicates a distance that is larger than the distance indicated by the lidar (the sound wave reflected off multiple objects), the distance indicated by the lidar is used. Another way of dealing with this problem is by using multiple digital signal processors to remove the echo's as is done in [17].

3.5 Comparison

In order to be able to choose between the sensors a comparison is made. This comparison will grade each sensor on several aspects (table 1) and discuss the reasoning behind the grade and the chosen sensor.

	Accuracy	Computational complexity	Price	Field of view	Depth of view	Output dimensionality
Lidar (Standard)	++	+	- -	++	++	2D
Lidar (pivoting)	++	+-	- -	++	++	3D
RGB Camera (Mono)	+	-	++	+	+	3D
RGB Camera (Stereo)	+	- -	++	+	+	3D
RGBD Camera	++	-	+	+	+	3D
Sonar	-	+	++	-	+	2D

Table 1: Comparison of different environment sensors (- - = horrible, - = bad, +/- = alright, + = good, ++ = excellent)

3.5.1 Lidar

The main advantage of the lidar systems is that they are very accurate and have a high field of view. The SICK LMS-1xx rangefinder [44] for example, has a resolution of 0.5 cm, a rotational resolution of 1 degree, a field of view of 270 degrees and a depth of view of 20 meters.

Extracting features as is done in [13] will result in having to find local minimum and local maximum and sudden jumps in distance in the laser data, which is not very computational expensive. The scan matching algorithm however, is more computational expensive with a complexity of On^2 [5].

The main disadvantage of the lidar is that it only takes a 2D slice of the environment. When this is used on a hexacopter then it will be very difficult to generate a map because almost all the movements will result in losing sight of all the previously seen features. The only movement that is available is moving in the same plane as the lidar. Other movements will need to use the odometry sensors to try and predict where the hexacopter has moved to, which will slowly accumulate errors. Another issue is that it can be difficult to determine the height of the hexacopter based on the lidar data, especially when flying in an empty corridor.

A way around this would be to use the pivoting system. The main problem with this method is that the robot will have to stay stationary while the lidar is pivoting in order to get a good result, which is very difficult to do on a hexacopter system.

3.5.2 RGB Camera

The advantage of using a camera is that they are relatively cheap and can create a 3D map. These kinds of systems are however more computationally expensive than the laser

sensors and have a lower field of view. Another problem with the camera systems is that the accuracy of determining the depth is based on the used resolution and the distance of an object to the camera. If an object is close the the camera it will occupy more pixels in the final image, which results in a better depth determination. When an objects is further away, it will occupy less pixels in the final image and the depth calculation will be less accurate. This could be improved by using a higher resolution camera but this will result in longer processing. The difference in computational complexity, between monocular and stereo systems, comes from the fact that when using a stereo system, two images need to be processed where for a monocular system only one image has to be processed.

3.5.3 RBGD camera

The main advantage that the RGBD camera system has over the normal RGB camera system is that it has onboard depth image calculation, which will reduce the required processing per frame. A disadvantage of using a RGB-D camera is that it consumes more power than a RGB stereo system. The higher power consumption is caused by the projection of the IR dots and the required processing to generate the depth image.

3.5.4 Sonar

The advantage of using a sonar is that it measures in a 3D cone instead of a 2D slice. This enables the system to better detect the environment and avoid collisions with for instance overhangs. The field of view of a single sonar module is relatively small (30 degrees [11]) but this can be overcome by using multiple sonars in a circle.

The downsides of using a sonar is that reflections have a large influence on the accuracy of a sonar and that it can be difficult to match the new scan to reference scans just like with the lidar.

3.6 Choice

Based on the discussed sensing methods it seems best to use the RGB-D camera for the generation of the 3D map. The lidar and sonar methods have the problem of having to rely to much on the odometry sensors to determine the height of the robot and that they lose their reference points while moving. Even though the camera systems are very similar to the RGBD system they are not chosen because of the extra depth calculations that are required. Especially with the stereo system, because then the feature extraction methods, which take up most of the processing time, will have to be run twice (once for each image). This leaves the RGB-D camera of which there are two ways of implementing it, the feature matching method or the plane matching method. Due to the high execution time and the lower flexibility (there are few methods of extracting planes) of the plane matching method, it seems best to use the feature matching method.

4 Minimization methods

In order to create a map, multiple consecutive RGB and depth images have to be aligned. This is achieved by trying to minimize the error between points in an earlier scan (reference) and the most recent scan (query). Minimizing the error between two scans is done by trying to find a rigid body transformation that when applied to one of the two scans results in a minimal distance between the two scans. This transformation will contain a rotation R and a translation T. The optimal transformation is the transformation that minimizes e in eq. 1 where N is the number of points, a_i is a point in point cloud A (reference) and b_i is a point in point cloud B (query). Here point cloud A and B have the same amount of points.

$$e(R,T) = argmin \sum_{i=1}^{N} ||(a_i - Rb_i - T)||^2 \tag{1}$$

4.1 Brute force

There are multiple different methods to find the minimal value of e. The easiest method is by applying a brute force algorithm. This method will try every possible translation and rotation within a certain window of possibilities. Even though this method results in the optimal translation and rotation it is almost never used because of the high amount of processing to try every possibility. A way of increasing the speed of this method is by using a coarse to fine grain search method as is done in [25]. Here the search space is divided in a fixed amount of cells. By iteratively finding the best match and reducing the area of the search space the optimal solution is found.

4.2 Singular Value Decomposition (SVD)

A more efficient solution is by using Singular Value Decomposition (SVD) [1]. Using SVD it is possible to rewrite a matrix A to a multiplication of the matrices U, Σ and V (eq. 2). Here U contains the eigenvectors of AA^T, Σ contains the eigenvalues and V contains the eigenvectors of A^TA. By using SVD the translation and the rotation can be calculated by using eq. 3 and 4, respectively. Here μ_r and μ_n represent the mean of the reference scan and the new scan, respectively.

$$A = U\Sigma V^T \tag{2}$$

$$T = \mu_r - R\mu_n^T \tag{3}$$

$$R = UV^T \tag{4}$$

4.3 Newton method

Another possible method for finding the minimum of a function is the Newton method. This method uses the gradient to incrementally minimize a function. The first order Newton method (eq. 5) uses the gradient of the function to determine the next value of x.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{5}$$

The second order Newton method (eq. 6) also uses the second order derivative (the curviness). This results in making large steps when the function is far away from the minimum (large gradient, small curviness) and making small steps when the function is close to the minimum (small gradient, high curviness).

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{6}$$

4.4 RANSAC

False matches can have a large influence on the accuracy of the match. A way to avoid including these matches for the transformation calculation is by using RANSAC (RANdom SAmple Consensus). RANSAC works by selecting 3 random features that all have a match and calculating the transformation that aligns them. Next the found transformation is applied to all the features in the dataset and the distance between each feature and its match is calculated. If this distance is less than a threshold then the feature is counted as an inlier. The number of inliers is then used to calculated the minimum number of iterations (R) according to eq. 7 where P represents the probability of getting the optimal solution, W represents the probability of choosing an inlier and N is the number of features.

$$R = \frac{log(1 - P(success))}{log(1 - W^N)} \tag{7}$$

5 Feature extraction

The way multiple RGBD frames are matched is by finding features that occur in both the query and the reference frames. A way of doing this is by extracting features from the grayscale images of both the query and reference frames and trying to match them. This chapter will discuss multiple different feature extractors that are commonly used in computer vision applications.

5.1 SIFT

SIFT (Scale-Invariant Feature Transform) [9] is a scale and rotation invariant feature extraction method that locates features based on a Difference of Gaussian(DOG) function.

In order to make the features scale invariant an image pyramid is used (fig. 4 a). This pyramid is constructed by first generating S scaled images by calculating the convolution of a Guassian kernel(G) and the original image(I) (eq. 9). The amount of scaling of each scaled image is determined by eq. 8 where $k = 2^{\frac{1}{S}}$. These scaled images are combined into the first octave. The next step is to generate the second octave which contains another S scaled images. These scaled images are calculated by the convolution of a Guassian kernel and a sub sampled version of the original image. The dimensions of this sub sampled image are half of the dimensions of the original image. This process is repeated O times where O represents the number of octaves.

$$\sigma_s = k \cdot \sigma_{s-1} \tag{8}$$

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{9}$$

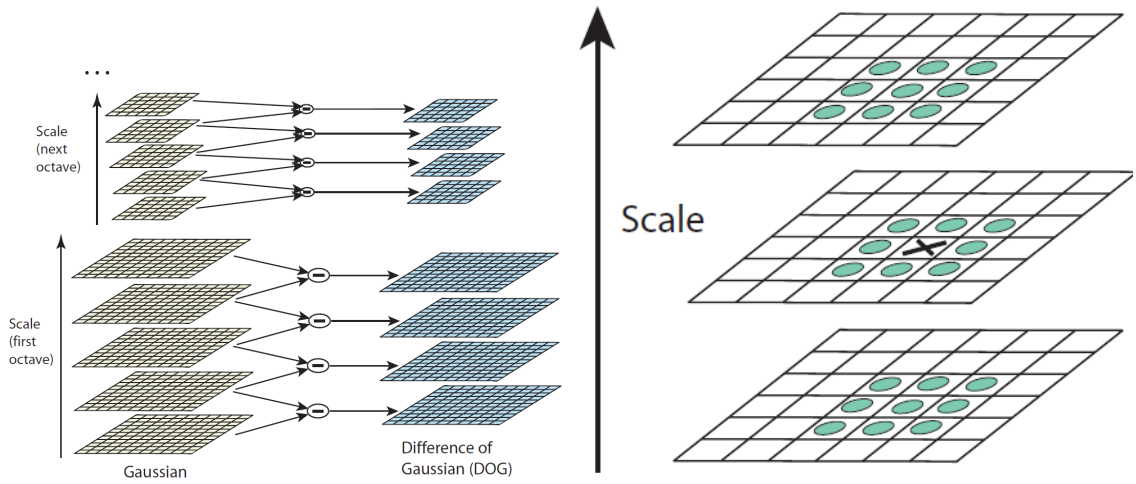
$$G(x, y, \sigma) = \frac{1}{2 * \pi * \sigma^2} e^{(-\frac{x^2+y^2}{2*\sigma^2})} \tag{10}$$

After the image pyramid is constructed the features are located by looking at the difference between scaled images. For scaled image I_S this is done by comparing every pixel with its 8 neighbors in I_s and its 9 neighbors in I_{s-1} and I_{s+1} (fig. 4 b). The pixel is registered as a feature if its pixel value is a the minimum or maximum of all the 26 neighboring pixels.

After the keypoints are located, the exact keypoint position is calculated by setting the derivative of a Taylor expansion to zero to find the exact minimum or maximum. In order to make the keypoints less susceptible to noise, the keypoints that are on an edge or have a low contrast are removed.

To make the found keypoints rotation invariant their orientation (θ) and magnitude (M) are calculated. This is done by applying eq. 11 and eq. 12 where L is the image closest in scale to the scale of the keypoint and x and y are the location of the keypoint.

$$M(x, y) = sqrt((L(x + 1, y) - L(x - 1))^2 + (L(x, y + 1) - L(x, y - 1))^2) \tag{11}$$



(a) Difference of Guassian operation (b) Feature detection

Figure 4: SIFT DOG and feature detection

$$\theta(x,y) = tan^{-1}(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}) \tag{12}$$

5.2 SURF

The SURF (Speeded up robust features) [12] feature extractor works by looking for maximum values of the determinant of the Hessian matrix. First the original image is converted to an integral image. In an integral image the value of a pixel represents the sum of the intensities of all the pixels in the rectangle between the current point p and the origin (eq. 13). Next every pixel is evaluated by calculating the Hessian matrix at that point (eq. 14). Here L represents the convolution of the second order derivative of a Gaussian. To make the process more computational efficient the second order Gaussian matrices are first discretized and cropped. In order to make the feature detector scale invariant this processed is repeated with different sized second order Gaussians. Octaves are used to be able to find the features in multiple scales. The SURF feature extractor uses multiple octaves (3 or 4) each of which contains three images that where processed at different scales. The features are found by looking at the maximum of the determinant of the Hessian and this position is refined by using interpolation techniques.

$$I_\Sigma(x) = \sum_{i=0}^{i<x} \sum_{j=0}^{j<y} I(i,j) \tag{13}$$

$$\mathcal{H}(x,\sigma) = \begin{bmatrix} L_{xx}(x,\sigma) & L_{xy}(x,\sigma) \\ L_{xy}(x,\sigma) & L_{yy}(x,\sigma) \end{bmatrix} \tag{14}$$

5.3 Harris corner detector

The Harris corner detector [2] works based on the pixel intensities inside a window. If the window is on a flat region (all pixels have the same color) and it is moved, then the difference between the pixel intensities stays the same. Here the difference between pixel intensities is determined by eq. 15 where w is the window function, x and y are pixel coordinates inside the window and u and v are the shifting values in x and y respectively. An example of a window function is a Guassian, here pixel differences near the center of the window get a larger weight than pixel differences near the edge of the window. When the window is over an edge, then movements along the edge will not have an effect on the difference between pixel intensities but movements perpendicular to the edge will have an effect. Finally if the window is over a corner then movements in any direction will have an effect on the difference between the pixel values.

$$E(u,v) = \sum_{x,y} w(x,y)(I(x+u, y+v) - I(x,y))^2 \tag{15}$$

By applying a Taylor expansion to eq. 15 it is possible to reduce the equation to eq. 16 where I_x and I_y are the gradients in x and y respectively.

$$E(u,v) = \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_{xy} \\ I_{xy} & I_y^2 \end{bmatrix} \tag{16}$$

By looking at the eigenvalues of eq. 16 it is possible to determine if the point is a corner. When this is the case both eigenvalues will be large. If one of the two eigenvalues is large and the other is small, then the window is on an edge and if both eigenvalues are small, then the window is on a flat region. The shape in the window can be calculated by eq. 17 where λ_1 and λ_2 are the two eigenvalues and k is a constant. If R is positive then it is a corner, if R is negative then it is an edge and if R is small then it is a flat area.

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \tag{17}$$

5.4 FAST

The FAST (Features from Accelerated Segment Test) [14] algorithm is a corner detection algorithm that determines if a pixel is a corner by looking at its 16 neighboring pixels (see fig. 5). The neighboring pixels are orientated in a circle and are all assigned an index (1..16). Next the intensity of each pixel on the circle is compared to the intensity of the center pixel. If there are k consecutive pixels that all have an intensity higher than the center pixel or that all have an intensity that is lower than the central pixel, then the center pixel is considered a corner. In fig. 5 point p is a corner pixel if $k \leq 12$ because the pixels with indexes 12-7 are all brighter than pixel p. In order to get better results a threshold t is introduced. With this threshold only pixels that have an intensity(I) that is higher than I(p)+t or lower than I(p)-t are taken into account.

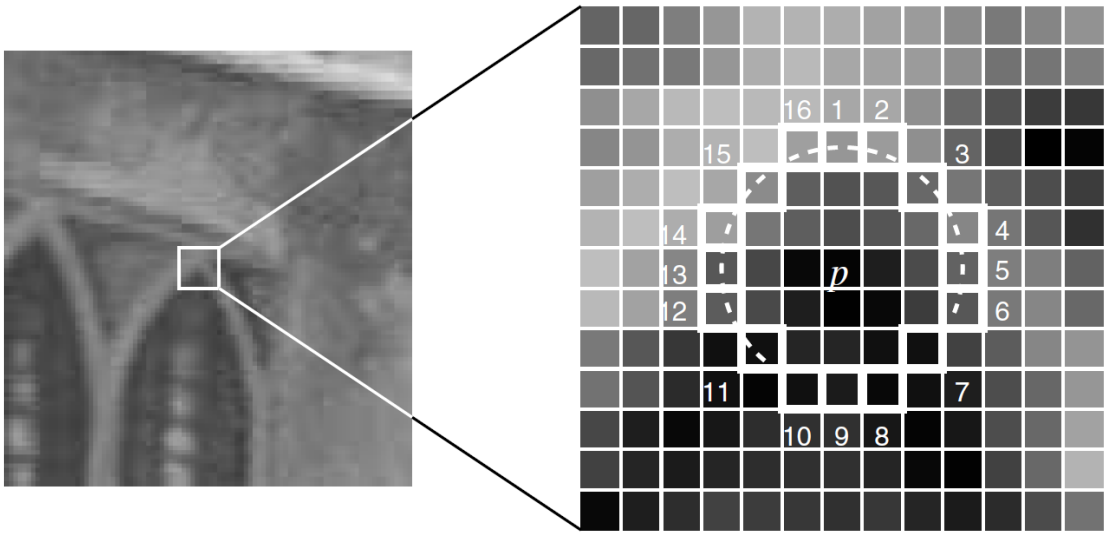


Figure 5: FAST sampling pattern

If k≥12 then the processing speed can be increased by first looking at the pixels 1,5,9 and 13. If three of them all have an intensity higher than I(p)+t or all of them have intensities lower than I(p)-t, then p can be a corner. When this is not the case then there is no point in processing the other 12 pixels and the algorithm can move on to the next pixel.

The authors also propose another way of increasing the processing speed of the algorithm that is based on machine learning. Here a decision tree is generated by using training data and a specified k and t. This decision tree contains pixel indexes of the neighboring pixels and based on whether or not the intensity of that pixel is higher or lower than I(p)+t or I(p)-t respectively a decision is made to either check another pixel index or to make the decision if p is a corner point or not.

5.5 BRISK

The BRISK (Binary Robust Invariant Scalable Keypoints) [28] feature detector is based on a combination of an image pyramid and the FAST corner detector. It works by looking for corners using the FAST corner detector across multiple different scaled images in the image pyramid. If a corner p is found, then a score s is assigned to it and compared to the scores of the same point in the next and previous scaled images s_n and s_p respectively. Here s is the number of consecutive pixels around the feature that all have a higher or lower pixel intensity than the feature pixel. When s> both s_n and s_p, then point p is considered as a corner. Next the optimal scaling value of the feature is determined by using interpolation techniques.

5.6 ORB

The ORB (Oriented FAST and Rotated BRIEF) [29] feature detector is a corner detector that applies FAST to each scaled image in an image pyramid. Bad corners are then filtered out by using a Harris measure (eq. 15). Finally an orientation (θ) is calculated for the remaining corners with eq. 18 where atan2 is the quadrant-aware version of arctan.

$$m_{i,j} = \sum_x \sum_y x^i y^j I(x,y)\theta = atan2(m_{01}, m_{10}). \tag{18}$$

6 Binary descriptors

In order to be able to match features from multiple images a descriptor has to be generated for each feature. Based on the feature descriptions two features can be matched to each other if the distance between them is below a certain threshold. This distance can be an Euclidean distance or a Hamming distance when dealing with binary descriptors. Because a low execution time is required for the final implementation only binary descriptors are investigated. Binary descriptor matching is the fastest option because of the low execution time of the calculation of the Hamming distance. Calculating the Hamming distance involves applying an XOR between the two descriptors and calculating the number of binary 1's in the resulting value.

A common method among the binary descriptors is the use of sampling patterns. These patterns contain pixel pairs that are located in an area around the feature. The descriptor is generated by comparing the pixel intensities of the pixel pairs. If the intensity of the first pixel of the pixel pair (a) is less then the intensity of the second pixel in the pixel pair (b) then a 1 is added to the descriptor, otherwise a 0 is added (eq. 19). In order to reduce the influence of noise, the patch around the pixel value is smoothed by using a Gaussian kernel. The variance of the Gaussian kernel differs between the implementations.

$$f_{nd} = \sum_{1<i<n_d} 2^{i-1} \tau(p, a_i, b_i) \tag{19}$$

$$\tau(p, a, b) = \begin{cases} 1 & \text{if p(a)<p(b)} \\ 0 & \text{otherwise} \end{cases}$$

Commonly used binary descriptors are BRIEF,BRISK,FREAK and ORB. The main principle of each of these descriptors is discussed in the following paragraphs.

6.1 BRIEF

The BRIEF (Binary Robust Independent Elementary Features) descriptor [21] uses a Gaussian based sampling pattern (fig. 6). Here the x component of one of the pixel points is sampled from a Gaussian distribution that is centered around 0 and has a variance (σ^2) of $\frac{1}{25}S^2$. Next the y component is sampled from a Gaussian distribution that is centered around the x component and has a variance of $\frac{1}{100}S^2$. The variances that are used where experimentally found to perform best.

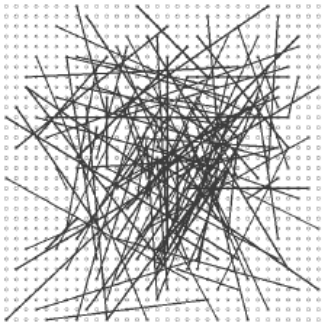


Figure 6: BRIEF sampling pattern[21]

6.2 BRISK

The BRISK descriptor [28] uses a sampling pattern that consists out of evenly spaced circles concentric with the keypoint (fig. 7). Instead of using a fixed variance for the Guassian (red circles in fig), the BRIEF descriptor uses variances based on the distance to the keypoint (located in the origin in fig. 7). Based on the sampling points(blue circles in fig. 7) a set of sampling pairs ($\mathcal{A}$ eq. 20) is generated. This set is then split up into two subsets: the short distance ($\mathcal{S}$ eq. 21) and the long distance ($\mathcal{L}$ eq. 22) sets.

$$\mathcal{A} = \{(p_i, p_j) \in \mathbb{R}^2 \times \mathbb{R}^2 \mid i<N \wedge j>i \wedge i, j \in \mathbb{N}\} \tag{20}$$

$$\mathcal{S} = \{(p_i, p_j) \in \mathcal{A} \mid ||p_j - p_i||<\delta_{max}\} \subseteq \mathcal{A} \tag{21}$$

$$\mathcal{L} = \{(p_i, p_j) \in \mathcal{A} \mid ||p_j - p_i||>\delta_{min}\} \subseteq \mathcal{A} \tag{22}$$

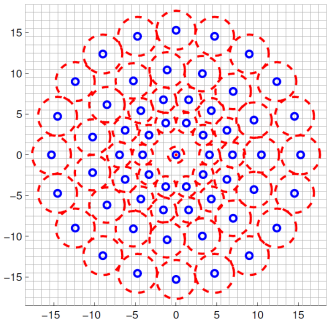


Figure 7: BRISK sampling pattern[28]

The orientation of the keypoint is determined by taking the average of the gradients of the long distance sample points. This orientation is then used in combination with the keypoint scale to rotate and scale the sampling points. Next the descriptor is determined by comparing the pixel intensities of all the short distance sampling points.

6.3 ORB

The ORB descriptor [29] is based on the BRIEF descriptor but also incorporates the orientation of the keypoint (steered BRIEF). A way of accomplishing this is by rotating the pixel pairs in BRIEF according to the orientation of the keypoint. In order to evaluate this method PCA (Principle Component Analysis) was applied to the pixel pairs and the highest 40 eigenvalues where studied (fig. 8). The optimal feature descriptor has a low correlation and a high variance. As can be seen, the eigenvalues of steered BRIEF are lower than the eigenvalues of normal BRIEF. This means that the steered BRIEF pixels pairs are less correlated than the normal BRIEF pixel pairs, which is preferred. The figure also shows that the eigenvalues of steered BRIEF decrease more rapidly than the eigenvalues of normal BRIEF, which means that there is a lower variance in the data.

In order to increase the performance of steered BRIEF the authors propose rBRIEF. The pixel pairs of rBRIEF are chosen so that the correlation is low and the variance is high. This was accomplished by trying to find pixels pairs that when applied to multiple different features that where extracted from multiple different types of images (images of cars, animals, etc.) result in a low correlation and a high variance. As can be seen in figure 8 rBRIEF has lower eigenvalues than BRIEF and the eigenvalues decrease less rapidly than the eigenvalues of BRIEF.

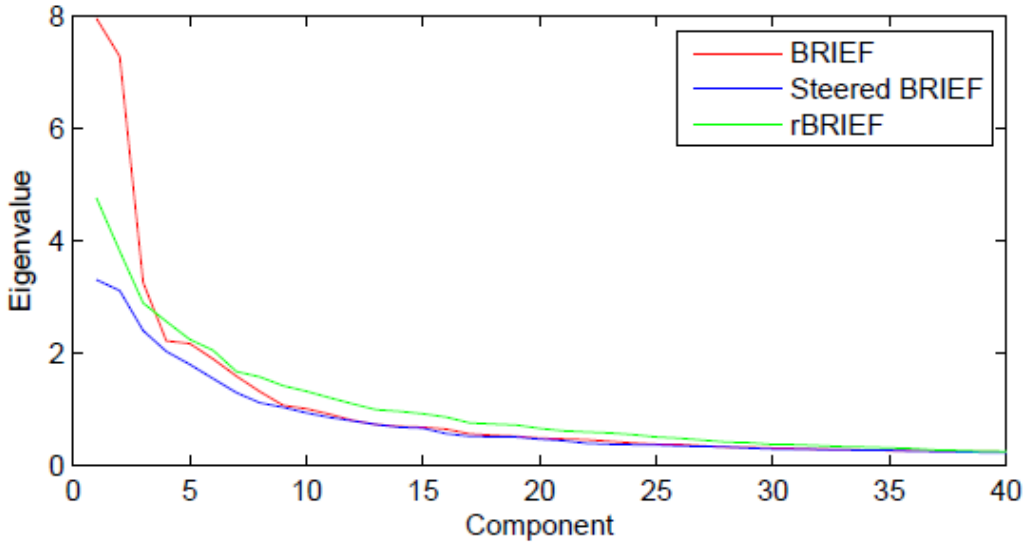


Figure 8: ORB sampling pattern Principle Component Analysis(PCA)[29]

6.4 FREAK

The FREAK(Fast Retina Keypoint) [32] descriptor uses a sampling pattern (fig. 9) that is based on the human retina. Because the human eye is more sensitive in the center of the retina, the sampling points are close together and use a small Gaussian variance for smoothing. As the sampling points get further away from the center, the density of sampling points reduces and the variance increases. Because of the large number of possible sampling pairs, the FREAK descriptor uses a reduction method similar to the ORB descriptor. Here only the sampling points with a low correlation and high variance are found.

FREAK also incorporates the orientation of the keypoint. The orientation is calculated according to eq. 23 where M is the number of pairs that are used in the orientation calculation and P_O^{ri} is the 2D vector of the spatial coordinates of the center of the receptive field, which is similar to the BRISK descriptor. Instead of using the long distance samples, FREAK uses pairs with symmetric fields with respect to the center (fig. 9).

In order to increase the execution speed of the matching, the keypoint is processed in 4 steps, which are sorted from course to fine grain information. If the distance after a step is above a threshold, then the keypoint is discarded as a match.

$$O = \frac{1}{M} \sum_{P_O \in G} (I(P_O^{r1}) - I(P_O^{r2})) \frac{P_O^{r1} - P_O^{r2}}{||P_O^{r1} - P_O^{r2}||} \tag{23}$$

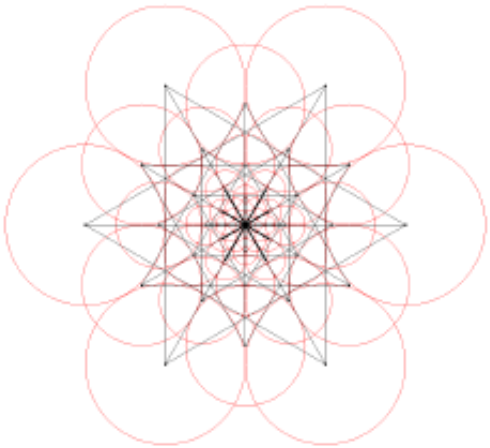


Figure 9: FREAK sampling pattern[32]

7 Feature matching

In order to efficiently and accurately match the features of multiple scans, the features have to be matched to each other. This involves comparing the feature descriptors of previously acquired features (reference features) to the feature descriptors of the newly acquired features (query features). To be able to compare feature descriptors some kind of distance measure is required. Because binary features are generated, a suitable distance measure is the Hamming distance. The Hamming distance between two binary descriptors is the number of bits that are different, which is calculated by applying an XOR operation and counting the number of bits that are 1 (eq. 24).

$$\begin{array}{r} (10110) \\ XOR\,(00101) \\ \hline (10011) \end{array} Hammingdistance = 3 \tag{24}$$

7.1 Brute force

A straight forward way of implementing feature matching is by applying a brute force algorithm. Here the Hamming distance is calculated between each descriptor in the query set and each descriptor in the reference set. If the minimum distance is below a certain threshold, then the descriptors are seen as a match. The problem with using a brute force matching algorithm is that it has an complexity of O(rq) where r is the number of reference features and q is the number of query features. This results in long execution times when there are a large amount of features. A way of reducing the execution time is by matching multiple features in parallel.

7.2 Hash table

Another way of matching binary descriptors is by interpreting the binary descriptor as an index for a hash table. Here each entry in the hash table contains an index of the matching descriptor. For this to work two problems have to be overcome: the hash table size and the number of Hamming distance permutations.

The first problem comes from the fact that if the entire binary descriptor would be used as an index in the hash table, then the hash table would need 2^b entries where b is the number of bits in the binary string of the descriptor. Here an entry consists of the index of the reference descriptor.

The second problem is related to the way of dealing with a maximum distances (t) > 0. When t=0 then finding the match just involves looking at the table entry that corresponds to the descriptor. However when t=1 then besides the original binary string also all permutations of the binary string where 1 bit is flipped have to be processed. This results in having to process L(b,t) (eq. 25) possibilities as a worst case for a given

b and t.

$$L(b,t) = \sum_{k=0}^{t} \binom{b}{k} \tag{25}$$

A solution for both problems is to use s disjoint binary sub strings instead of the entire binary string [35]. Here the entire binary string is divided into m parts, which results in sub strings with l ($\frac{b}{m}$) bits. This results in having to use s hash tables that each have 2^l entries. As a result of splitting the hash table up into multiple smaller ones each entry can contains more than 1 index of possible matching descriptors.

Another advantage of using sub string is that the number of permutations that are required to find matches with a distance $<$ t is greatly reduced. This is because in eq. 25 besides the number of bits, the threshold t is also reduced. The reason for this is that the maximum distance is distributed across the sub strings (eq. 26). For example if there are 30 sub strings and the maximum distance is 29 then there must be at least 1 sub string that perfectly matches a sub string of a reference descriptor. This results in only having to look for sub strings that perfectly match. When t$>$s it is still necessary to check every permutation.

$$t_{sub} = \lfloor \frac{s}{t} \rfloor \tag{26}$$

After finding all possible matching options, the distance between the entire descriptors are calculated to find the best match.

8 Scan matching

Instead of trying to find a transformation based on the features that are found in the RGB image, it is also possible to find a transformation based on the depth image. These methods work based on trying to find a transformation that minimizes the distance between all the points in 2 point clouds. Usually this is achieved by using either an Iterative Closest Point (ICP) or a Normal Distribution Transform (NDT) method.

8.1 Iterative Closest Point (ICP)

The ICP method works by first trying to find a matching point in the first point cloud (A) for each point in the second point cloud (B). Once each point in A has a corresponding point in B, a transformation can be calculated that minimizes the distance between all the matching points. For this to work several problems have to be overcome.

8.1.1 Closest point search

The first and most expensive step of the ICP algorithm is trying to find points in the new scan that correspond to points in the reference scan. A point in the new scan corresponds to a point in the reference scan when there is no point in the reference scan that is closer to that point in the new scan.

One way of finding the correspondences is by applying a brute force method. Here the distance from each point in A is calculated to each point in B. This results in a complexity of $\mathcal{O}(N \cdot R)$ where N are the number of points in A and R are the amount of points in B.

A more efficient way is by using a kd-tree to find the correspondences between points. The structure of the tree is that the root represents the entire point cloud and it has two successor nodes. Each successor node represents a sub point cloud that also have two successor nodes. The leafs of the tree are buckets that contain the size of the sub cloud and all the points that fall into the sub cloud.

When the corresponding point of point p needs to be found in the kd tree, first the tree needs to be traversed until the leaf is found that represents the sub point cloud in which p falls. Next the distances from point p to all the points in the bucket are calculated and the shortest distance is selected. If the shortest distance is smaller than the distance from point p to the edge of the sub point cloud, then the point corresponding to the shortest distance is chosen as the corresponding point of point p. If the shortest distance is larger than the distance from point p to the edge of the sub point cloud, then it is possible that the closets point to point p is in the neighboring sub point cloud. When this is the case, backtracking is used to go to the neighboring sub point cloud. This process is called Ball Withing Bounds (BWB) and is illustrated in figure 10.

8.1.2 Distance functions

In order to be able to find corresponding points some kind of distance function is required. The distance function that was used in the original ICP algorithm is the point to point

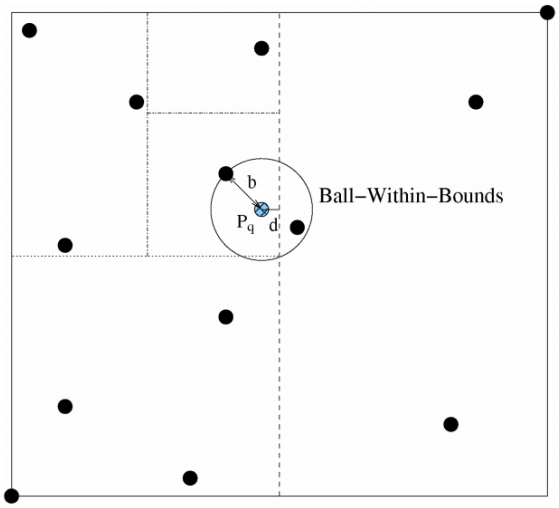


Figure 10: Ball Within Bounds (BWB) [19]

distance [7]. The point to point distance is calculated by determining the Euclidean distance (eq. 27) between two points. Here R represents the rotation matrix and T represents the translation matrix.

$$d(x, y) = \|(x - Ry - T)\|^2 \tag{27}$$

The downside of using the point to point distance is that it assumes that the corresponding points are in exactly the same position in the environment. Because the RGBD camera takes a discrete sample of the environment this is not necessarily the case. In order to try and overcome this problem a new distance measure was introduce: the point to plane distance [7].

This distance is based on the projection of point x on the tangent plane of point y. Here point x is a point in A and y is a point in B. The point to plane distance between point x and y is calculated by eq. 28. Here R is the rotation matrix, T is the translation vector and n is the normal at point p.

$$d(x, y) = \|((x - Ry - T)) \cdot n\|^2 \tag{28}$$

The downside of both the point to point and the point to plane distance functions is that they need matching points. A method that overcomes this problem is the point to projection distance measure [7]. This method only works when dealing with range images like the depth image of the RGBD camera. The idea is to project one image onto the other by using a rotation matrix R and translation matrix T. Finding point u in B that is a match for point v in A works as follows:

- Point u is transformed to a 3D point
- The transformation using R and T is applied to u

- The transformed u position is back projected to a 2D xy position
- Point v is located at the found xy position in the depth image of A

The downside of this method is that it is less accurate than using the point to point or point to plane methods.

8.1.3 Outlier removal

During the closest point search it can happen that wrong matches are generated. This can be caused by for instance: a wrong initial transformation or measurement noise. If these wrongly matched points would be used for finding the transformation, then they will introduce an error. A way of dealing with wrong matches (outliers) is by using Random Sample Consensus (RANSAC), which is explained in section 4.4.

8.2 Normal Distribution Transform(NDT)

The Normal Distribution Transform (NDT) tries to find a rigid body transformation that maximizes the probability of the new scan matching the reference scan. This is done by first dividing the reference point cloud into fixed size voxels. For each voxel k that has N (N>3) points ($p = (x, y, z)^T$) in it a mean (μ_k) and covariance matrix (Σ_k) is calculated using eq. 29 and eq. 30 respectively.

$$\mu_k = \frac{1}{N} \sum_{v=1}^{N} p_v \tag{29}$$

$$\Sigma_k = \frac{1}{N} \sum_{v=1}^{N} (p_v - \mu_k)(p_v - \mu_k)^T \tag{30}$$

Because of the discretization, the normal distribution voxel map contains discontinuities in the surface at the edges of the voxels, which can cause matching errors. A way of overcoming this problem is by using multiple overlapping voxels. These overlapping voxels are generated by defining the start of the successive voxel 1/2 of the voxel size later instead of starting the next voxel a full voxel size later (fig. 11). Here the green point falls within the four overlapping rectangles (red, orange, black and blue) that are shifted by half of the rectangle size. When dealing with a 3D case, 8 overlapping voxels are used.

The rigid body transformation that matches the new scan with the reference scan is found my minimizing e in eq. 31. Here p_v is point v in the new scan that falls within voxel k1 until kM (for 2D M=4,for 3D M=8) of the discretized reference scan.

$$e(p) = -\sum_{kt=1}^{M} e^{\frac{(p_v - \mu_k)^T \Sigma_{kt} (p_v - \mu_k)}{2}} \tag{31}$$

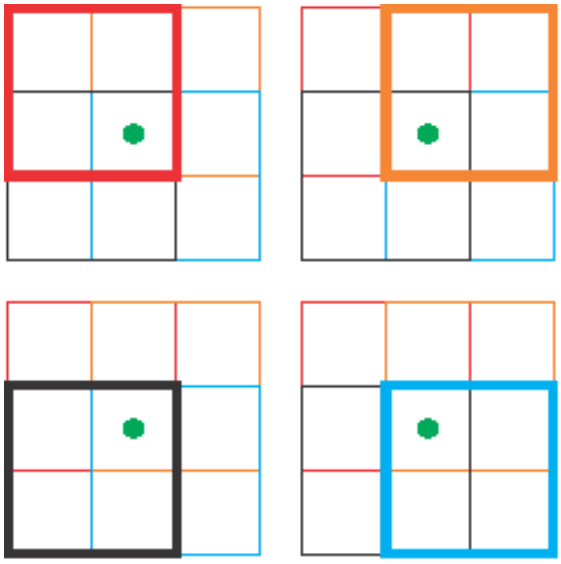


Figure 11: NDT cell overlap

The transformation that minimizes eq. 31 is calculated by applying Newtons method eq. 32 where t is the previously calculated translation or in the case of the first iteration the initial transformation, H is the Hessian of e and g is the gradient of e.

$$t_{new} = t - \frac{g}{H} \tag{32}$$

8.3 Choice

Because an important factor in the implementation is the execution time it seems best to use the ICP point to projection method. This will insure a fast execution time but it will also require a good initial transformation.

9 Implementation

This chapter will cover the choices that where made and the solutions to problems that where encountered during the implementation. Figure 12 shows a flowchart of the steps that are taken during the SLAM process.

First a frame of images (RGB and depth) is read from the dataset. Next a grayscale image is generated from the RGB image, which will be used for the feature extraction. Because large parts of the SLAM implementation use the open computer vision (OpenCV) library, first the RGB and depth image from the sensor are converted to OpenCV matrices. The RGB matrix is then converted to grayscale by using an OpenCV function. After generating the grayscale image it is passed to the feature extraction and description functions of openCV. The found features are then matched to the features in a previously processed frame and a rigid body transformation is calculated based on the found matches. This transformation is then used as a starting point in the scan matching procedure. The scan matching procedure updates the transformation and passes the result to the feature map, which acts as a database. If the transformation indicates that the robot has traveled a large enough distance, then the frame and its features are added to the feature map to serve as reference frames for later frames. The following subsections will discuss each part in more detail.

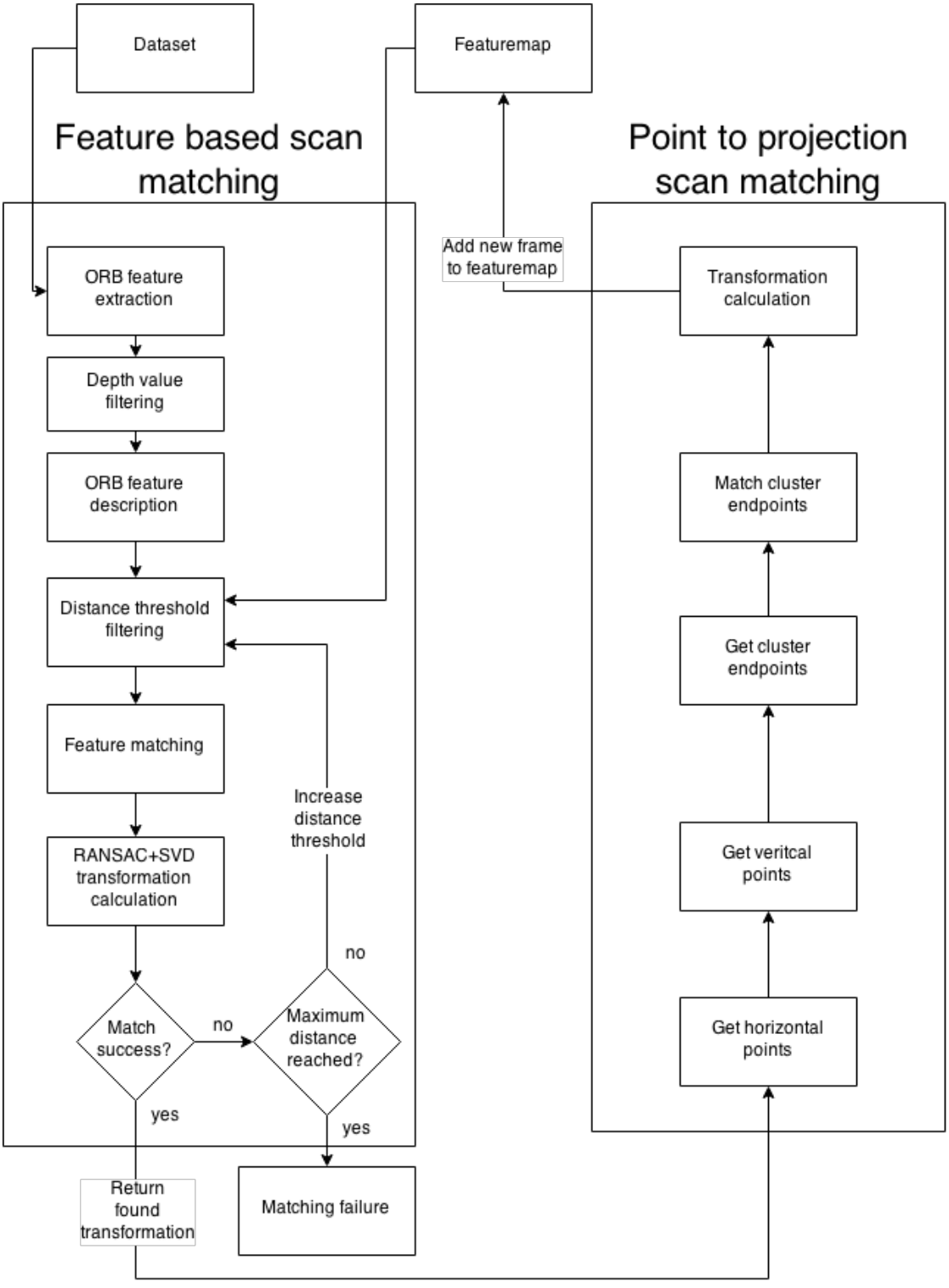


Figure 12: Flowchart of the implementation

9.1 Sensor choice

There are two different RGBD sensors available: the Microsoft Kinect and the Asus Xtion PRO LIVE. The specifications of both sensors are shown in table 2.

Figure 13: Size comparison between the Microsoft Kinect and the Asus Xtion [37]

As can be seen most of the specifications are similar. There are however some important differences between the two sensors: the height, the weight and the power consumption. The final goal is to be able to mount the sensor to a hexacopter platform in order to autonomously map the environment. This results in a sensor that needs to be as small, light and power efficient as possible. Because of these restraints the Microsoft Kinect is less suitable due to the significantly higher size, weight and power consumption.

9.2 Dataset generation

During the testing phase, instead of directly using images from the Xtion, a previously recorded dataset is used. This has the advantage of not having to move the camera around for every test and ensures that experiments can be repeated with the same input image. The first step in generating the dataset is by manually moving the camera

Point of comparison	Microsoft Kinect	Asus Xtion
Hardware Compatibility	USB 2.0	USB 2.0
	USB 3.0	USB 3.0 (using a hotfix)
View Adjustment	Has motor that can be controlled remotely by the application	No motor, only manual positioning
Size	12” x 3” x 2.5”	7” x 2” x 1.5”
Weight	3.0 lb	0.5 lb
Power Supply	USB + ACDC power supply	USB
Power Consumption	12 watts	<2.5 watts
Distance of Use	between 800mm and 4000mm	between 800mm and 3500mm
Field of View	57 °horizontal, 43 °vertical	58°horizontal, 45°vertical
Vertical tilt range	27°	Not applicable
Frames per second (FPS)	30	30
Depth Image Size - Resolution	640x480 pixels	640x480 pixels
OS Platform Support	Microsoft Windows	Microsoft Windows
	Linux	Linux
	MacOS	MacOS
	Xbox 360	
Programming Language	C++	C++
	C# (Windows)	C# (Windows)
	Java	Java
Libraries	OpenNI	OpenNI
	OpenKinect	
	Microsoft SDK	

Table 2: Microsoft Kinect and Asus Xtion specifications [38]

through the environment and generating a recording. This recording contains all the depth and RGB images that are made.

To gain access to the sensor and to generate the recordings, the open source OpenNI 2.0 library is used. This library outputs the RGB image with 24 bits per pixel (8 bits per color) and the Depth image with 11 bits per pixel. The values of the depth image represent the distance in mm from the object to the camera plane. Here the maximum value is around 9000mm and out of range measurement are stored with a distance of 0mm. The precision of the depth values is however not constant. When objects are close (between 0.5 and 4 meters) then the step size of the depth values will be relatively small (1-50mm) but when objects are further away (7 meters +) then the step size can grow to (150mm +) (fig.14).

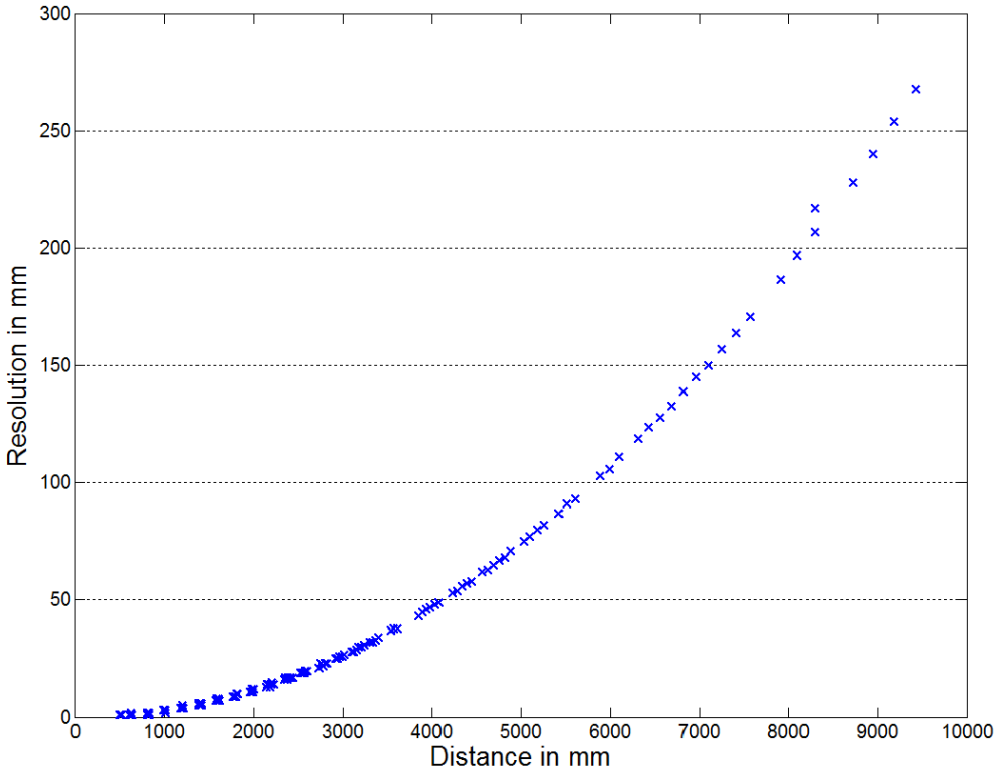


Figure 14: Depth resolution as a function of distance [33].

Because the depth and the RGB images are captured using two different cameras, some kind of registration is required. If this step is not performed then a pixel at X,Y in the RGB image will not correspond with the pixel at the same coordinate in the depth image. The registration of the image is done by the openNI library, which uses the calibration values of the sensor that where determined by the manufacturer.

Besides setting the registration mode the resolution is also set. As is described in the specification (table 2), the maximum resolution is 640x480 but the implementation uses

a resolution of 320x240 and a frame rate of 30fps. The main reason for this is that this will reduce the processing time of the feature extraction method. Besides the reduction in processing time this resolution is also chosen because when both the RGB and depth image have a resolution of 640x480 the frame rate is very unstable. This is tested using the example code that is supplied with openNI 2.0 on both a laptop (HP 8570w) and the Raspberry Pi.

9.3 Reading dataset images

Initial versions of the implementation read the images directly from the recording. These images would then be converted to openCV matrices by using a memcopy operation and converted to grayscale so that they can be used by the rest of the implementation.

The problem with this method is that the openNI function that reads the images from the recording is very unstable. This function would at random moments skip X amount of frames from the recording where X is roughly between 1 and 20. The problem with this is that the resulting 3D model shows significant differences when generated based on the same dataset.

The openNI library also supports a seek function. This function enables the user to look for a frame with a specific index. The problem with this function is that this results in a loss of synchronisation between the RGB images and the depth image. Besides the loss of synchronisation, this function also randomly reads the wrong images. This problem is overcome by rereading the last correct image and trying the failed image again.

To overcome the problem of having to reread images, all the images of the dataset are converted to images that are stored in the hard drive. When an image from the dataset is required, the image indexes are manually resynced and the correct images are read from the hard drive.

9.4 Feature extractor and descriptor comparison

Because execution time is of a large importance to the implementation, OpenCV was used. This has as an advantage that it supports multiple different feature extraction, description and matching methods that are all highly optimized. In order to find the optimal feature extractor and descriptor a measurement was performed, which compared three feature extractors (ORB,BRISK and FAST) and four feature descriptors(ORB, BRIEF, BRISK and FREAK). The SIFT and SURF methods where not considered due to their long processing times. Because the extractors take multiple different input arguments the tests where run multiple times with different input arguments. The arguments that where altered between runs and their minimum and maximum values are shown in table 3.

Besides adjusting the arguments the distance between the images was also adjusted. Here 4 different distances where used: 2cm, 10.7cm, 20.3cm and 26.3cm. Distances much larger than 25cm where not included into the test because the feature map distance threshold is set to 25cm. These distances where determined by the mapping algorithm.

ORB			
Argument	minimum value	step size	Maximum value
Scale factor	1.0	0.1	1.9
Number of levels	1	1	10
BRISK			
Argument	minimum value	step size	Maximum value
Number of octaves	1	1	10
Threshold	10	10	100
FAST			
Argument	minimum value	step size	Maximum value
Threshold	10	10	100

Table 3: Minimum, maximum and step size of the tested arguments for each feature extraction method

The optimal extractor should be able to find a large amount of correct features (inliers) in a short amount of time. First a comparison is made between the feature extractors by looking at the number of inliers they found (fig. 15). Here it can be seen that the number of inliers of the BRISK and FAST feature extractors are significantly lower than the number of inliers when using the ORB extractor. For this reason only the ORB extractor will be considered.

To be able to find the optimal settings for the extractor and descriptor a cost function was created (eq. 33). This function takes into account the normalized number of inliers (n_{in}), the normalized number of RANSAC iterations (n_{it}) (section 9.8) and the normalized processing time (pt). Because of the large importance of having a fast algorithm the processing time is given a higher weight.

$$c = 2 * pt + n_{in} + n_{it} \tag{33}$$

The results showed that the optimal feature descriptor is the ORB descriptor for each of the distances. There is however a difference in the optimal settings of the parameters. The scale ranged between 1.1 and 1.2 and the number of levels ranged from 5 to 10. Through performing multiple experiments the optimal parameters of 1.2 for the scale factor and 5 for the number of levels where determined.

9.5 Feature extraction and description implementation

The input to the ORB feature extractor is the grayscale image of the current frame. After processing the image an output is generated, which is a vector of coordinates of where features are located. Because the features are extracted by just using the grayscale image it can happen that there is no depth value available (out of range) for that feature. The features without a depth value are removed because they are not usable in the following steps.

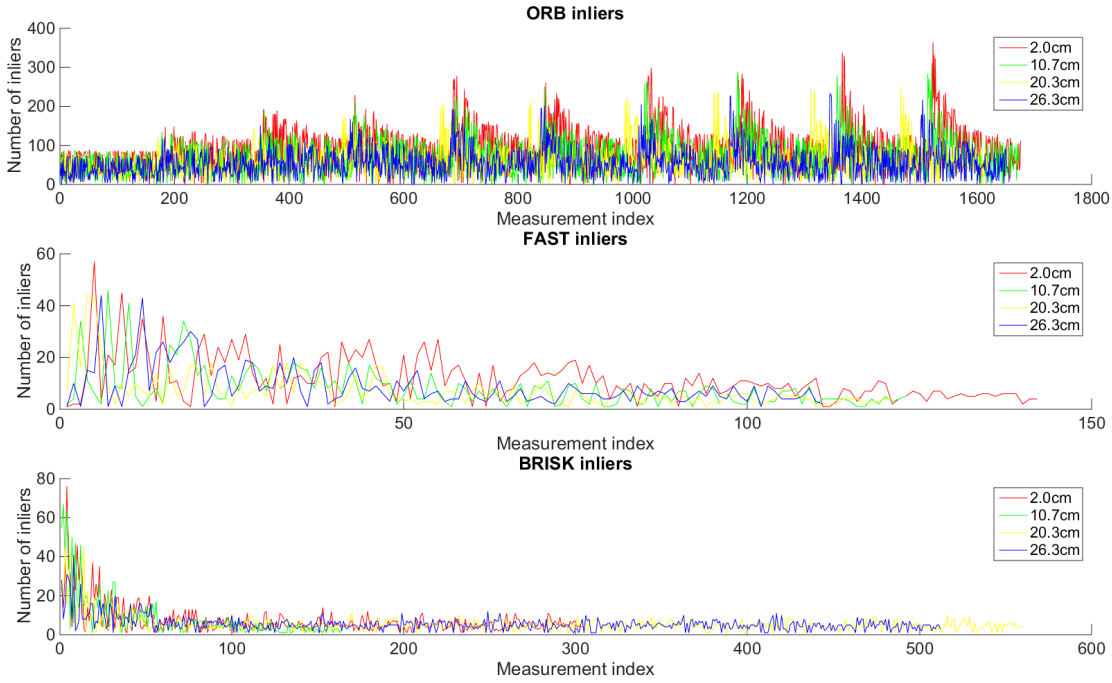


Figure 15: Plots showing the number of inliers per feature extractor

The coordinates of the feature are then passed on to the ORB feature descriptor which generates a 256 bit (32 independently accessible 8 bit values) feature descriptor for each feature.

9.6 Maximum distance

As can be seen in the specifications (table 2) the maximum recommended distance is 3.5 meters. If there are enough matched features inside the 3.5 meter range then the algorithm will be able to find a transformation. However, when this is not the case then the maximum allowed distance is increased so that hopefully there will be more matches. The downside of this is that the accuracy of the feature locations will decrease, which will in turn decrease the accuracy of the found transformation. Because of this, the maximum allowed distance is increased by 1 meter every time the algorithm fails to find a valid transformation. This way the algorithm starts with accurate features and slowly adds more inaccurate features as long as the algorithm keeps failing. The chosen distance is a trade off between accuracy (smaller steps reduce loss in accuracy) and the number of required iterations (larger steps decrease the number of required iterations).

9.7 Feature matching

The features that are found in the current frame are matched with features that where found in a reference frame. This reference frame is the frame in the feature map that is

closest to the current position.

Two different methods where compared for the feature extracting process. Both of the feature matching methods use the Hamming distance to determine when a feature is the closest match. The Hamming distance is based on how many bits are different between two bit streams.

The first matching method is the openCV brute force matcher. This matcher has two input arguments: the distance measure that should be used (the Hamming distance), and whether or not to use crossCheck. The principle of crosscheck has however not been mentioned before.

Lets assume that two sets of features (Black and White) are matched to each other and this is done by checking each feature in Black to each feature in White. When this is the case, then feature A is closest to feature C and feature B is also the closest feature to feature C (fig. 16). When crosscheck is not enabled then both A and B will have C as their match. However, when crosscheck is enabled then only A and C will be matched because C is closest to A. The matches that have a distance higher than the minimal threshold distance are filtered out.

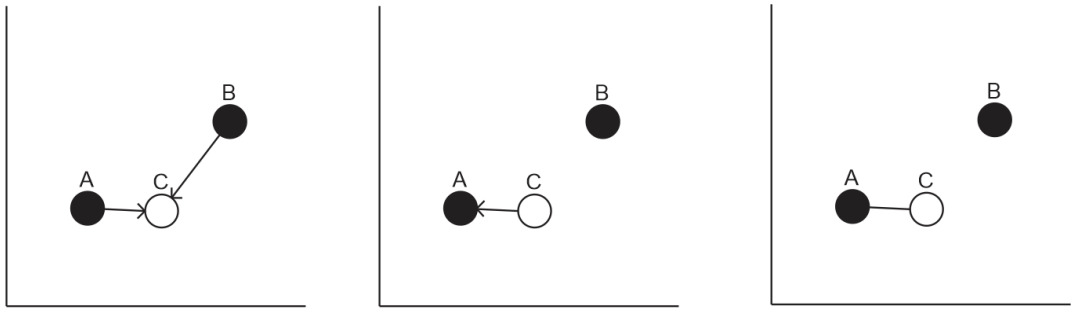


Figure 16: Crosscheck illustration

The second matching method is the hash table matcher. Because this matcher is not implemented in openCV an own implementation was made. This implementation uses an array of linked lists to store the hashtable. Here the first X entries represent the first hash table where X represents the number of entries in a hash table, the next X entries represent the second hash table and so on. Due to practical reasons every entry in a hashtable consist out of 8 bytes, which results in 32 hash tables total ($\frac{256}{8}$). When a minimum distance threshold is used that is less than or equal to 32 then there is only one entry per hash table that needs to be checked.

The two matching methods are compared by looking at the time it takes to match X features where X ranges from 1 to 400. By looking at the results (fig. 17) it is clear that the hashtable matcher is faster than the brute force matcher. For this reason the brute force matcher was used in the final implementation.

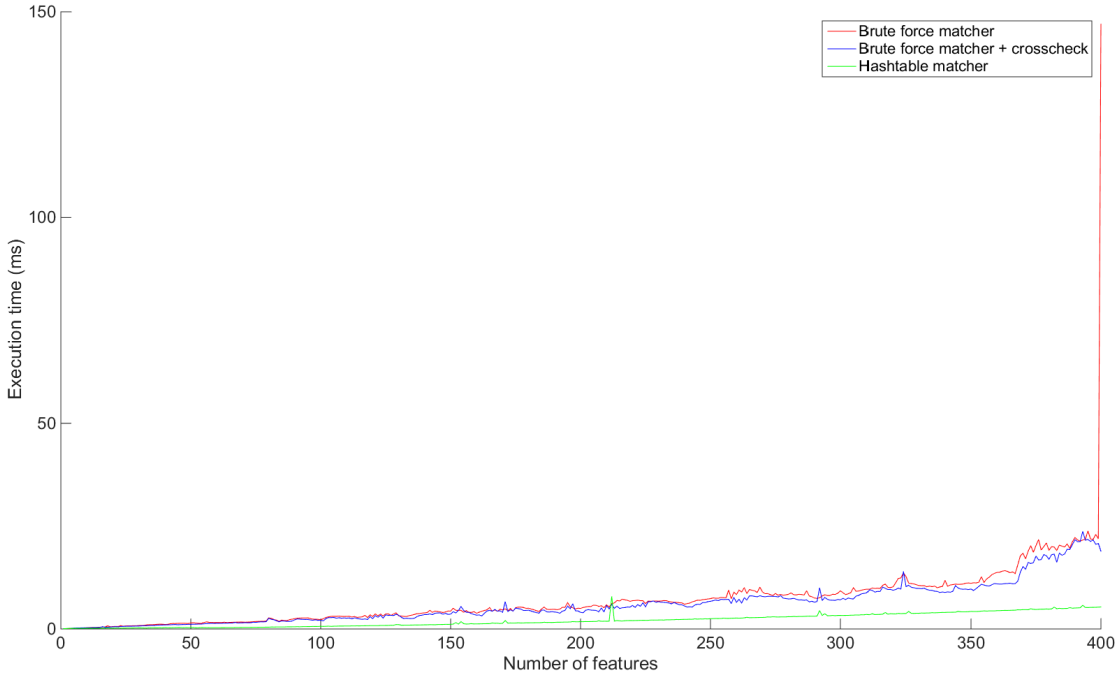


Figure 17: Execution time as a function of a number of input features for the the brute force matcher and the hashtable matcher

9.8 Transformation calculation

After the matching is complete, the matched features are transformed to 3D points using an OpenNI function. This function uses the calibration values to convert a depth value and an X and Y coordinate to a 3D position. Before the transformation is determined, first the features need to be checked if they are not all on a line or on a plane. When the features are on a line, then there are infinite transformations that minimize the distance. This is because there are an infinite amount of rotations around the line. This could result in a wrong transformation. The problem with the features oriented in a plane is that a mirror image of the features is also a solution, which would also results a wrong result.

To be able to deal with this the principle components of the 3D feature points are calculated. This results in calculating the covariance matrix and the eigenvalues of the 3D points of the matches. If the largest eigenvalue is greater than λ times the smallest, then the matched points are in a plane or on a line. When this is the case it is not possible to find an accurate transformation and the maximum distance is increased. If the distance between the eigenvalues is small enough, then the transformation can be found by using RANSAC and the SVD implementation of OpenCV.

As is described in section 4.4, RANSAC works by picking 3 random points from the matches, calculating a transformation that best minimizes the distance between these three points and decides if it is a valid solution by looking at the number of inliers

when the transformation is applied to all the points. Two matched features are seen as inliers when the Euclidean distance between them after applying the transformation is below a certain threshold. This threshold will have to be dynamically calculated for each feature because the resolution of the depth values changes as the depth value gets higher. Because there are no accuracy measurements available for the Xtion the depth resolutions of the Microsoft Kinect are used.

After a valid transformation is found the principle components of the inliers is determined again. This is to prevent that all the matched feature points are not in a plane or on a line but the inliers are. If the feature points pass the principle components test then the transformation is success full.

A problem with adding a frame after traveling a certain distance is that the feature descriptions might change too much to be able to match them successfully. This problem is solved by updating the descriptions of the reference frame by adding the descriptions of the found inliers to the reference frame.

9.9 Scan matching

The main downside of using a method based on features is that it wont work very well when there are not enough good features. In order to try and deal with this a scan matching algorithm was implemented. The method that was used later turned out to be the point to projection ICP method (section 8.1.1).

9.9.1 Scan matching method

The main problem with the ICP methods is that finding the matches between two point clouds takes up a large amount of processing time. By finding a way to reduce the processing time of the matching process it should be possible to decrease the required processing time of the scan matching. The main idea is to use the transformation that was found by the feature matching method described above to directly find correspondences between the two depth images.

This is done by transforming the query depth image to how it would look from the viewpoint of the reference depth image. If this is successful, then finding matches between the transformed query depth image and reference depth image is done by looking at the same X and Y position in both images and transforming both positions to a 3D point. The transformation that is used to transform the viewpoint is the transformation that was found by the feature matching process described above. This is because this transformation results in making the query features match the reference features. Converting a pixel at X,Y in the query depth image to the viewpoint of the reference image is done as follows:

- Get the depth value (D_q) at X,Y of the query frame
- Calculate the 3D position using X,Y and D_q

- Apply the initial transformation found by the feature matching process to the 3D point
- Transform the 3D coordinate back to a 2D depth image coordinate (A,B)

Because of the change of viewpoint it can happen that certain points don't exist in both depth images. An example of this is when moving past an open door. When the camera is far away from the door then it will only see a small sliver of the door, but when the camera moves closer to the door then more of the door will be visible. This causes a problem when a part of the door that is only visible in the query image is moved to the viewpoint if the reference image, then the wall blocking the view of the door is chosen as a match, which is wrong. This problem is dealt with by thresholding the distance between two matches. When the distance is too large then those points are filtered out.

The process of the implementation of the scan matching is illustrated in figure 12. First matching points are found between the reference and the query frame. At the moment the scan matching algorithm is optimized for matching corridors. Corridors have the advantage of consisting of 2 walls, a floor and a ceiling, which are all mostly flat. Because of this, the number of points that are matched can be reduced. The applied reduction works by taking 2 slices from the images, a horizontal and a vertical slice both at the center of the hallway. These slices will ensure that each of the 4 surfaces (left wall, right wall, floor and ceiling) is matched. When dealing with a room with desks,chairs etc. this method will not work as well because of the large variety in depth values (less straight surfaces).

Just like the feature based scan matcher, SVD is used to find the transformation that minimizes the distance between 3D points. A problem occurs when in a slice (horizontal or vertical) the two segments (two walls or ceiling and floor) don't have the same amount of points. When this is the case the SVD has the tendency to rotate towards the segment with the most amount of points, which generates an error. This is probably because of the larger amount of points has a larger influence on the resulting transformation. The way this is dealt with is by clustering the segments and using the start and end of a cluster as points for the transformation calculation. This doesn't however solve the problem of missing the entire left wall for example. When this happens then the transformation will still rotate towards the right wall. An advantage of using cluster endpoints is that it also extracts features like door openings. When a door opening is visible in the depth image, then the cluster will split that wall into two parts. When the endpoints are calculated, then two of them will indicate the start and endpoint of the door.

9.9.2 Feature Map

Every time the new frame is matched to a reference frame it is passed to the feature map where it is stored if a large enough distance is traveled. This distance threshold is implemented in an effort to reduce the accumulation of the error. When a frame is matched to a reference frame then the found transformation contains a small error that occurs due to sensor noise or rounding errors. If every frame where to be stored then

this error would increase faster than when the distance threshold is used. A maximum distance threshold of 25 cm was found to give accurate 3D models.

Besides storing new frames the feature map is also used to supply the query frame with a reference frame, which is the frame that is closest to the predicted location of the current frame.

9.10 Visualization

In order to be able to visually check the results, a visualization tool was created that runs offline on a base station. The goal of the tool is to be able to generate a 3D model based on the recorded depth images, RGB images and the found transformations. Besides showing the model it also enables the user to freely move through the model. The 3D model will consist out of voxels, where each voxel represents a pixel of the RGB image that has a corresponding depth value. Because of the large amount of voxels per image (320*240= 76800), openGl is used to be able to use the GPU for processing. Besides the OpenGL library also openGLFW is used. OpenGLFW is used to be able to create a graphical user interface (GUI) on which the resulting 3D model can be shown and interacted with.

The camera movement is implemented by a translation and a rotation of the world. So instead of moving the camera the entire model is moved. This is implemented by keeping track of the position vector (3 floats) and the view direction vector (3 floats).

The first step in the process is defining a single voxel (cube). This can be done in multiple different ways: by defining 6 square faces, by defining 12 triangular faces or by using indexing. The downside of defining faces is that each 3D position (vertex) will be defined multiple times (a vertex is on the corner of a face). In order to increase the efficiency indexing is used. Indexing uses two separate lists: one list contains the vertex locations and the other contains the indices of the vertices that make up the faces, which results in only having to store each vertex (3 floats) once. Because the processing of triangles is more efficient than the processing of squares, triangular faces are used.

There are multiple ways of creating a data structure that results in being able to draw multiple voxels on the screen. One way is by defining an indexed cube for each voxel. The downside of this is that when dealing with millions of voxels a large amount of memory is required. In order to optimize this a method called instancing is used. Instancing works by having one cube and only storing the translational data for each voxel. This significantly reduces the required amount of memory because now only 3 floats (instead of 18 (6 vertices * 3 floats)) are necessary per voxel.

9.10.1 initialization

When the visualizer is started, first the initialization step is performed. In this step the openGLFW window is created and all the required data is written to openGL buffers. This data consists of the translation and color of each voxel that is shown in the final model.

Besides filling the buffers also the vertex shader and fragment shader code is loaded. The vertex shader code, is code that is run to update the position of each vertex based on input arguments. This process only effects the position of the vertices not the color of the pixels on the screen. The pixel colors is processed by the fragment shader code. The advantage of using shaders is that they can be run in parallel on the GPU, which enables the application to process multiple voxels at the same time.

9.10.2 Main loop

In the main loop there are two things that happen: drawing the voxels and checking for events.

Drawing the voxels is done by passing 3 matrices to the vertex shader. The first matrix is the projection matrix. This matrix ensures that vertices that are far away are put closer to each other, while vertices that are close to the camera are spread far apart, which is done to give the user the illusion of depth (3D). The second matrix is the scaling matrix. In this implementation the matrix just makes sure that all the voxels have the same size.

The final matrix that is passed to the vertex shader is the world to view matrix. This matrix applies the movement and the orientation of the camera to every voxel. Based on these 3 matrices and the translations that where entered into the buffer at the initialization the vertex shader is able to calculate the correct position and orientation for each vertex in each voxel.

The second part of the main loop is checking for events. These events are the keyboard presses and the mouse movement that enable the user to move around the model. If an event was triggered then the camera is updated according to the user input. Shown below are the calculations that are performed for every movement option. Here MOVEMENTSPEED is a constant that indicates the movement speed, position is a vector containing the x,y and z position, viewDirection is a vector representing the viewing direction in x,y and z and UP is the vector that defines which way is up in x,y and z.

- Move forward: position += MOVEMENTSPEED*viewDirection
- Move backward: position -= MOVEMENTSPEED*viewDirection
- Move up: position += MOVEMENTSPEED*UP
- Move down: position -= MOVEMENTSPEED*UP;
- Strafe left: position += MOVEMENTSPEED * cross(viewDirection,UP)
- Strafe right: position -= MOVEMENTSPEED * cross(viewDirection,UP)
- Horizontal rotation: viewDirection = rotate(x rotation, UP)*viewDirection
- Vertical rotation: viewDirection = rotate(y rotation, cross(viewDirection,UP))*viewDirection

The forward, backward, up and down movements are all straight forward. Here the position is updated by adding the direction in which the movement occurs to the current position. Strafing works in a similar way but here the direction of movement must first be calculated. Because the direction of movement is perpendicular to the viewing direction and the UP vector, the cross product is calculated to find the movement vector.

Updating the viewing direction is done by using rotation matrices. These matrices are generated by using the glm rotate function. This function uses the angle of rotation and the vector to rotate around as input arguments and generates the corresponding rotation matrix. In the case of horizontal rotation, the vector to rotate around is the UP vector but for vertical rotation this is not the case. Here the rotation vector is the vector perpendicular to UP and viewDirection just like in the strafing calculations. The generated rotation matrices are then used to rotate the viewing direction.

10 Results

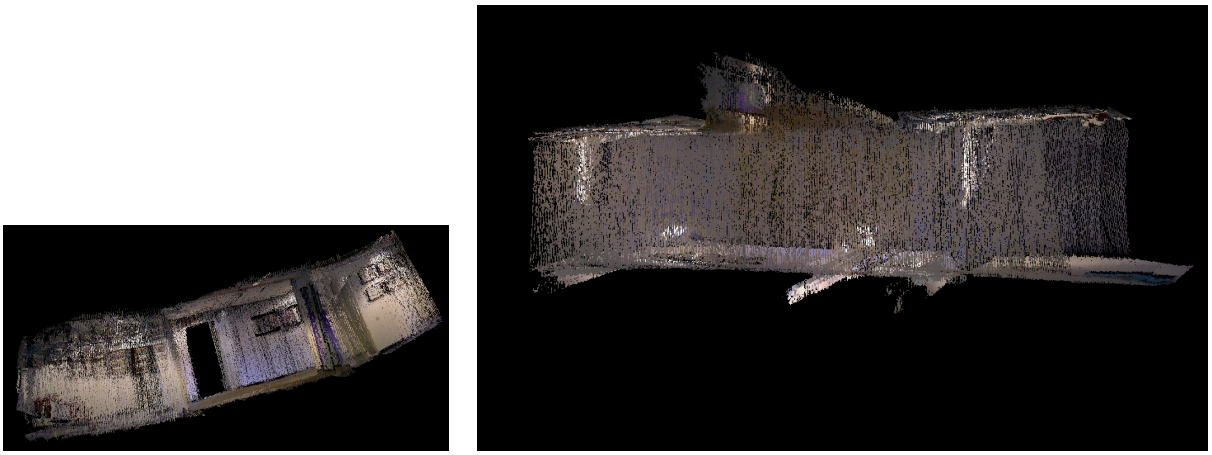
The main goal of this project is to develop a light weight 3D SLAM algorithm that can be run onboard of a hexacopter. This results in creating an implementation that can run at a high frame rate without using a GPU or a multi-core CPU.

10.1 Execution speed

In order to be able to asses the execution speed of the implementation a timing measurement has been performed. This measurement shows that the average total frame processing time is 56ms with a standard deviation of 33ms. The majority of the processing time ($\sim$ 48ms) is taken up by the feature based scan matching method. When two frames are easy to match, then only a few RANSAC iterations and no distance threshold increases are required. When, however, two frames are difficult to match, 100 (maximum iteration amount) RANSAC iterations will be performed per distance threshold, which results in a long processing time (100 ms+). By decreasing the maximum number of allowed iterations and increasing the distance threshold step size it is possible to reduce the maximum processing time. The downside of reducing the maximum number of iterations is however that it is possible that the optimal solution exists but is not found.

10.2 Model generation

Two 3D models are generated based on two different datasets. The first dataset is a relatively small dataset where the Xtion is put onto a cart and driven through a hallway for about 10 meters (S in figure 20), which results in the 3D model shown in figure 18. In order to be able to compare the result with the reality, the generated model and an RGB image from the dataset are shown side by side (fig. 19). Because no ground truth data is available it is not possible to determine the translation and rotation errors that are made during the model generation. It can however be seen that the model is horizontally aligned but there is a vertical error accumulation.



(a) Side view of the model (b) Top view of the model

Figure 18: 3D model of the small dataset

(a) Internal view of the model (b) RGB image

Figure 19: Result comparison

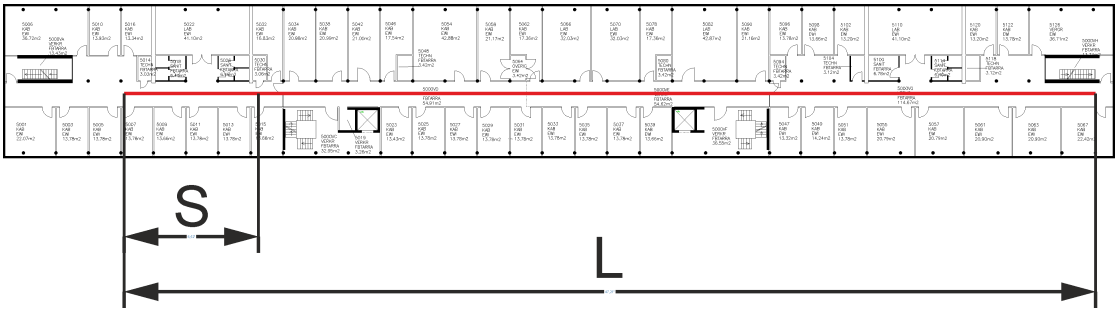


Figure 20: Floorplan showing the dataset path

The second dataset is of the same hallway but instead of traveling 10 meters on a cart, 100 meters was traveled (L in fig. 20) . This results in a model where the accumulation of the error is clearly visible (fig. 21). Due to memory problems it is not possible to show all the frames in the feature map. Because of this, only 160 frames are used for the model generation, which causes the holes at the start.

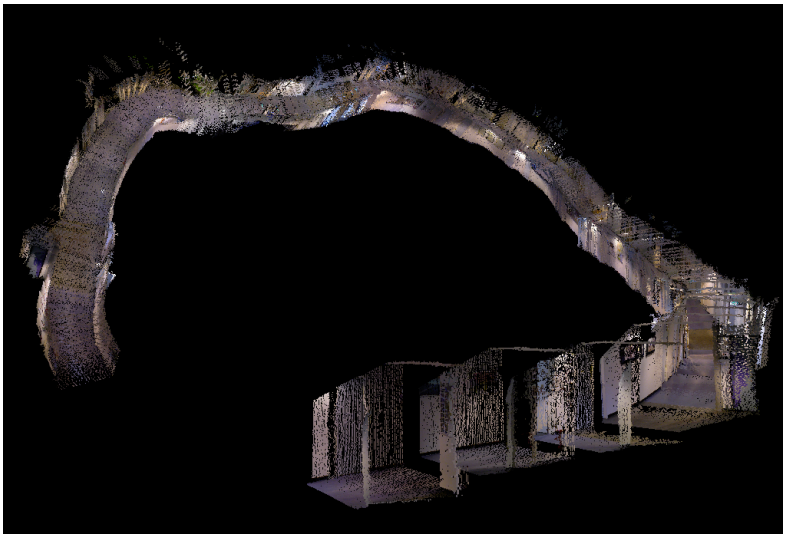


Figure 21: 3D model of the large dataset

11 Conclusion and future work

In this work a basic framework was created for a SLAM system using the Xtion pro live from Asus. The system is able to determine the location of the sensor and merge new frames with previous frames with an average frame rate of 17 frames per second, which was measured on a HP EliteBook 8570W.

To be able to assess the performance of the algorithm when it is run onboard a drone, two tests can be performed. The first test is to generate a dataset that is recorded from a drone moving in 3 dimensions. This test will determine the influence that drone has on the images. An example of this is that the vibrations caused by the motors decrease the quality of the images.

The second test is to determine the average frame rate on a low power pc like a Raspberry Pi. A rough estimate of the frame rate results in a frame rate of 3-6 fps, assuming that the Raspberry Pi is 3-5 times slower than the HP 8570w. This frame rate would be enough to generate a 3D map. If the test shows however that the frame rate is too low, then it might be possible to reduce the image resolution. This reduction will free up processing time, which could be used by a method that can deal with low resolution images. If the current methods would be applied to lower resolution images, then the accuracy of the 3D map will be reduced.

During the implementation it became clear that the openNI library is unstable at times. A possible reason for this could be that the library does not work on a 64bit windows 8.1 machine. It is recommended to look into this problem in order to reduce the memory usage that is caused by storing the images on the hard-disk.

The generated map has an accumulation of localization errors, which is a common problem with SLAM systems. Multiple different methods have been proposed in the literature to try and reduce or eliminate the error. Examples of this are:

- Use an IMU (inertial measurement unit) combined with a motion model to try and predict the location and orientation of the robot. This prediction can then be combined with the calculated transformation (observation) using an EKF (Extended Kalman Filter) for example.
- Improve the graph that is generated in the feature map by using graph optimization methods like TORO [23] or g^{2}o [26].
- Use weights for the transformation calculation. At the moment all the matched features that are used in the transformation calculation are weight equally. It might be possible to improve the accuracy by giving features that are matched over multiple frames a higher weight than features that are matched once or twice.
- Filter out the depth image noise.
- Match planes around features instead of single coordinates.

The current implementation is build to only add an image to the feature map when a certain distance is traveled. This results in the drone not being able to go around

tight corners. A solution to this would be to also add a frame when a certain amount of degrees is rotated.

Generating a 3D map of the environment is only one part of the final implementation. The drone also needs to be able to plan a path through the building on its own. There are multiple different approaches for this:

- Wall following methods.
- Move towards the closest unexplored area.
- Move towards locations that result in the highest map accuracy increase (most information gain).

Currently the 3D model is made by drawing a voxel for each pixel in the RGB image. A more efficient way would be to draw a square that always faces the camera, which will result in having to draw 1 square instead of 6. Another optimization that is required is to reduce the time it takes to generate the 3D model. A possible solution for this would be to use the GPU instead of the CPU to convert each pixel in the images to 3D coordinates.

References

[1] K Somani Arun, Thomas S Huang, and Steven D Blostein. "Least-squares fitting of two 3-D point sets". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 5 (1987), pp. 698–700.

[2] Chris Harris and Mike Stephens. "A combined corner and edge detector." In: *Alvey vision conference*. Vol. 15. Manchester, UK. 1988, p. 50.

[3] Brian Yamauchi. "A frontier-based approach for autonomous exploration". In: *Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on*. IEEE. 1997, pp. 146–151.

[4] Stan Birchfield and Carlo Tomasi. "Depth discontinuities by pixel-to-pixel stereo". In: *International Journal of Computer Vision* 35.3 (1999), pp. 269–293.

[5] J-S Gutmann and Kurt Konolige. "Incremental mapping of large cyclic environments". In: *Computational Intelligence in Robotics and Automation, 1999. CIRA'99. Proceedings. 1999 IEEE International Symposium on*. IEEE. 1999, pp. 318–325.

[6] Andrew J Davison. "Real-time simultaneous localisation and mapping with a single camera". In: *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*. IEEE. 2003, pp. 1403–1410.

[7] Soon-Yong Park and Murali Subbarao. "An accurate and fast point-to-plane registration technique". In: *Pattern Recognition Letters* 24.16 (2003), pp. 2967–2976.

[8] Hartmut Surmann, Andreas Nüchter, and Joachim Hertzberg. "An autonomous mobile robot with a 3D laser range finder for 3D exploration and digitalization of indoor environments". In: *Robotics and Autonomous Systems* 45.3 (2003), pp. 181–198.

[9] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[10] Sebastian Thrun et al. *Simultaneous mapping and localization with sparse extended information filters: Theory and initial results*. Springer, 2004.

[11] Søren Riisgaard and Morten Rufus Blas. "SLAM for Dummies". In: *A Tutorial Approach to Simultaneous Localization and Mapping* 22 (2005), pp. 1–127.

[12] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "Surf: Speeded up robust features". In: *Computer Vision–ECCV 2006*. Springer, 2006, pp. 404–417.

[13] L. Freda, F. Loiudice, and G. Oriolo. "A Randomized Method for Integrated Exploration". In: *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*. Oct. 2006, pp. 2457–2464. DOI: 10.1109/IROS.2006.281689.

[14] Edward Rosten and Tom Drummond. "Machine learning for high-speed corner detection". In: *Computer Vision–ECCV 2006*. Springer, 2006, pp. 430–443.

[15] Sunghwan Ahn and Wan Kyun Chung. "Efficient SLAM algorithm with hybrid visual map in an indoor environment". In: *Control, Automation and Systems, 2007. ICCAS '07. International Conference on*. Oct. 2007, pp. 663–667. DOI: 10.1109/ICCAS.2007.4406982.

[16] Zhenhe Chen, Jagath Samarabandu, and Ranga Rodrigo. "Recent advances in simultaneous localization and map-building using computer vision". In: *Advanced Robotics* 21.3-4 (2007), pp. 233–265.

[17] Saeid Fazli and Lindsay Kleeman. "Simultaneous landmark classification, localization and map building for an advanced sonar ring". In: *Robotica* 25.3 (2007), pp. 283–296.

[18] Michael Montemerlo and Sebastian Thrun. "FastSLAM 2.0". In: *FastSLAM: A Scalable Method for the Simultaneous Localization and Mapping Problem in Robotics* (2007), pp. 63–90.

[19] Andreas Nuchter, Kai Lingemann, and Joachim Hertzberg. "Cached kd tree search for ICP algorithms". In: *3-D Digital Imaging and Modeling, 2007. 3DIM'07. Sixth International Conference on*. IEEE. 2007, pp. 419–426.

[20] Masahiro Tomono. "Robust 3D SLAM with a stereo camera based on an edge-point ICP algorithm". In: *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE. 2009, pp. 4306–4311.

[21] Michael Calonder et al. "Brief: Binary robust independent elementary features". In: *Computer Vision–ECCV 2010*. Springer, 2010, pp. 778–792.

[22] Peter Henry et al. "RGB-D mapping: Using depth cameras for dense 3D modeling of indoor environments". In: *In the 12th International Symposium on Experimental Robotics (ISER*. Citeseer. 2010.

[23] G. Hu, Shoudong Huang, and G. Dissanayake. "Evaluation of Pose Only SLAM". In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. Oct. 2010, pp. 3732–3737. DOI: 10.1109/IROS.2010.5649825.

[24] Fengchi Sun et al. "Research on active SLAM with fusion of monocular vision and laser range data". In: *Intelligent Control and Automation (WCICA), 2010 8th World Congress on*. IEEE. 2010, pp. 6550–6554.

[25] Chen Friedman et al. "Towards model-free SLAM using a single laser range scanner for helicopter MAV". In: *AIAA Guidance, Navigation, and Control Conference*. 2011.

[26] Rainer Kummerle et al. "g 2 o: A general framework for graph optimization". In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE. 2011, pp. 3607–3613.

[27] Heon-Cheol Lee et al. "Robust scan matching with curvature-based matching region selection". In: *System Integration (SII), 2011 IEEE/SICE International Symposium on*. Dec. 2011, pp. 1257–1262. DOI: 10.1109/SII.2011.6147629.

[28] Stefan Leutenegger, Margarita Chli, and Roland Yves Siegwart. "BRISK: Binary robust invariant scalable keypoints". In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2548–2555.

[29] Ethan Rublee et al. "ORB: an efficient alternative to SIFT or SURF". In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2564–2571.

[30] Alfredo Toriz et al. "A simultaneous planning localization and mapping approach for robust navigation". In: (2011).

[31] David Leonardo Acevedo Cruz. "Calibration of a multi-kinect system". In: (2012).

[32] Alexandre Alahi, Raphael Ortiz, and Pierre Vandergheynst. "Freak: Fast retina keypoint". In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. Ieee. 2012, pp. 510–517.

[33] Michael Riis Andersen et al. *Kinect depth sensor evaluation for computer vision applications*. Tech. rep. Århus Universitet, 2012.

[34] Felix Endres et al. "An evaluation of the RGB-D SLAM system". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE. 2012, pp. 1691–1696.

[35] Mohammad Norouzi, Ali Punjani, and David J Fleet. "Fast search in hamming space with multi-index hashing". In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE. 2012, pp. 3108–3115.

[36] Joscha Fossel et al. "OctoSLAM: A 3D mapping approach to situational awareness of unmanned aerial vehicles". In: *Unmanned Aircraft Systems (ICUAS), 2013 International Conference on*. IEEE. 2013, pp. 179–188.

[37] H Gonzalez-Jorge et al. "Metrological evaluation of microsoft kinect and asus xtion sensors". In: *Measurement* 46.6 (2013), pp. 1800–1806.

[38] H. Haggag et al. "Measuring depth accuracy in RGBD cameras". In: *Signal Processing and Communication Systems (ICSPCS), 2013 7th International Conference on*. Dec. 2013, pp. 1–7. DOI: 10.1109/ICSPCS.2013.6723971.

[39] Zhe Ji et al. "Probabilistic 3D ICP algorithm based on ORB feature". In: *Information Science and Technology (ICIST), 2013 International Conference on*. Mar. 2013, pp. 300–304. DOI: 10.1109/ICIST.2013.6747555.

[40] Yuichi Taguchi et al. "Point-plane SLAM for hand-held 3D sensors". In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, pp. 5182–5189.

[41] Yuichi Taguchi et al. "Point-plane SLAM for hand-held 3D sensors". In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, pp. 5182–5189.

[42] Daobin Wang et al. "Lidar Scan matching EKF-SLAM using the differential model of vehicle motion". In: *Intelligent Vehicles Symposium (IV), 2013 IEEE*. IEEE. 2013, pp. 908–912.

[43] Chen Friedman et al. “Towards Model-Free SLAM Using a Single Laser Range Scanner for Helicopter MAV”. In:

[44] *SICK LMS 100 documentation*. http://www.hizook.com/files/publications/SICK_LMS100.pdf.