Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät

Bachelor's Thesis Cognitive Science

3D Navigation using Microsnapshots

Tristan Baumann

30.09.2016

Reviewer

Prof. Dr. Hanspeter A. Mallot
Institut Neurobiologie – Kognitive Neurowissenschaften
Eberhard Karls Universität Tübingen

Supervisor

Gerrit Ecke
Institut Neurobiologie – Kognitive Neurowissenschaften
Eberhard Karls Universität Tübingen

Baumann, Tristan:
3D Navigation using Microsnapshots
Bachelor's Thesis Cognitive Science
Eberhard Karls Universität Tübingen
Period: 01.06.2016 – 30.09.2016

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form keinem andern Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift

Abstract

This bachelor's thesis presents an algorithm that can be used by an agent such as a mobile robot to explore and navigate 3D environments. A topological representation of the environment is obtained by integrating SURF-features, which are salient points of interest detected by a monocular camera, with weak metric odometry data to create a complex graph encoding locations and their relative positions. The agent can then use the graph to plan routes through the environment by calculating the shortest path between two points on the graph. The algorithm is implemented and tested on a hand-held tablet computer supplying both image and odometry data. The algorithm's code is available on the CD provided with the physical copy of this work, as well as downloadable at:
https://www.dropbox.com/sh/b5co89gwawdzwhx/AAByoHXZKDpU-KmmZTCMM6ema?dl=0

Contents

List of Figures

1. Introduction

For more than 40 years now, autonomous navigation of mobile robots has been a central research subject in robotics and artificial intelligence. A popular method to "teach" a robot where it can and cannot move, is designating an area by placing easily recognizable markers: lawnmower robots for example usually navigate the lawn by following metal rails or respecting borders set by metal bars embedded in the ground. These robots are of course highly location-bound; and to create universally applicable robots, placing metal bars in every possible location is neither a feasible (if even possible) solution, nor is it needed: humans for example, are able to navigate most environments perfectly fine, simply by relying on visual and haptic information. Similarly, methods that can be used by robots to explore, represent and navigate environments based on sensor feedback, can be created. Robot navigation strategies generally consist of the following steps [13]:

1. Sensing the environment.
2. Building an environmental representation.
3. Locating itself with respect to the environment.
4. Planning and executing efficient routes in the environment.

Steps one and two are usually referred to as *mapping*, step three as *localization* and step four as *pathfinding.* The steps are of course not independent: for an accurate localization step, an accurate representation of the environment is needed; But to create an accurate representation of the environment, exact localization of the robot is extremely advantageous. Being able to run these two processes at the same time is the core of the problem known as *Simultaneous Localization And Mapping* (SLAM). The problem was allegedly established at the 1986 IEEE Robotics and Automation Conference in San Francisco, California; the term itself was coined in 1995. Since its first proposal, SLAM has been a major driving force behind robot navigation. If a robot is to successfully learn an environment, solving the issues presented by the SLAM problem is a necessity, irrespective of the higher-level processes, plans and goals otherwise used in the navigation algorithm [4] [14].
In the last three decades, SLAM has seen many different theoretical and practical solutions, which have been successfully implemented for and tested in ground-, underwater-, and air-based navigation tasks [4]. With the development of ever faster computation devices even once thought insurmountable problems such as high-resolution, high-speed and high frequency real-time SLAM have since been achieved, for example in the recent high-level solution ORB-SLAM by Mur-Artal et al. [11].
But there is no single "best" solution to SLAM. Rather, many different solutions, each with their own advantages and drawbacks, have been proposed throughout the years. The maybe most significant way, in which SLAM solutions usually differ from one another, is how the representation of the robot's environment, the

map, is handled. Many different ways for mapping have been proposed, one of which is for instance the creation of maps from the information gained by measuring the distance to landmarks or markings around the robot. Some landmark-based maps are able to model distances extremely accurately based on a global coordinate system, others simply sort detected landmarks and obstacles into large-spaced grids. To create landmark- or grid-based maps, 3D information of the environment is required, which can for example be obtained by relying on multiple cameras in a stereo setup: knowledge about distance to different points in space is gained by comparing the difference in viewpoint of partially overlapping images, or by using specialized distance sensors such as sonar or laser range finding. Basing algorithms around visual information is especially attractive, because cameras are ubiquitously available and much cheaper than other distance measuring methods.
In 1983, even before SLAM was solidified as an underlying problem, Moravec [19] presented a robot that was able to navigate around obstacles using depth information gained by a stereo-vision setup. The setup consisted of a single monocular camera mounted on a slider on a mobile, remote-controlled vehicle. By moving the camera laterally along the slider while taking snapshots of the scene in front of the robot, a disparity field containing depth information about the scene could be created from the multiple slight differences in view angle. Using the depth information, an obstacle avoidance route for the robot could then be detected and planned. While the system proved to be quite successful, it was held back by a lack of computing power: one single step of the avoidance path, which consisted of a long pause and then a short movement of around one meter, took 10 to 15 minutes to calculate and execute.
Moravec's system was later picked up and improved upon in the vision system FIDO. The stereo slider system and disparity field were used to extract features from the environment. By correlating the features, obstacles could be assigned 3D coordinates, which were successively used to place the obstacles in a two by two meter occupancy grid map. A robot was then able to navigate the mapped environment by planning a route through the unoccupied cells of the map [22]. A similar approach by Jennings and Murray [20] instead used a trinocular stereo vision system to create the occupancy grid map.
However, while occupancy grid maps provide a simple representation of obstacles in an environment, their accuracy directly depends on the chosen grid size; by reducing the size, the map is able to accommodate for obstacles of different orientations and sizes more accurately, but the computing power required for mapping and navigation also greatly increases. Additionally, the validity of the grid is increasingly dependent on the accuracy of the information provided by the robot's sensors. Little et al. [21] combined an otccupancy map with sparse corner features extracted from trinocular stereo vision as stable orientation points. In this approach, the feature information was only used to match successive frames to supplement the occupancy grid map. In a later approach, Little et al. [14] expanded upon the idea to use prominent features to facilitate mapping and localization: distinctive and robust interest points were detected in and extracted from each scene by relying on the image patch detector SIFT (Scale Invariant Feature Transform) by Lowe [7][32]. A reliable 3D grid map was then created by accurately localizing

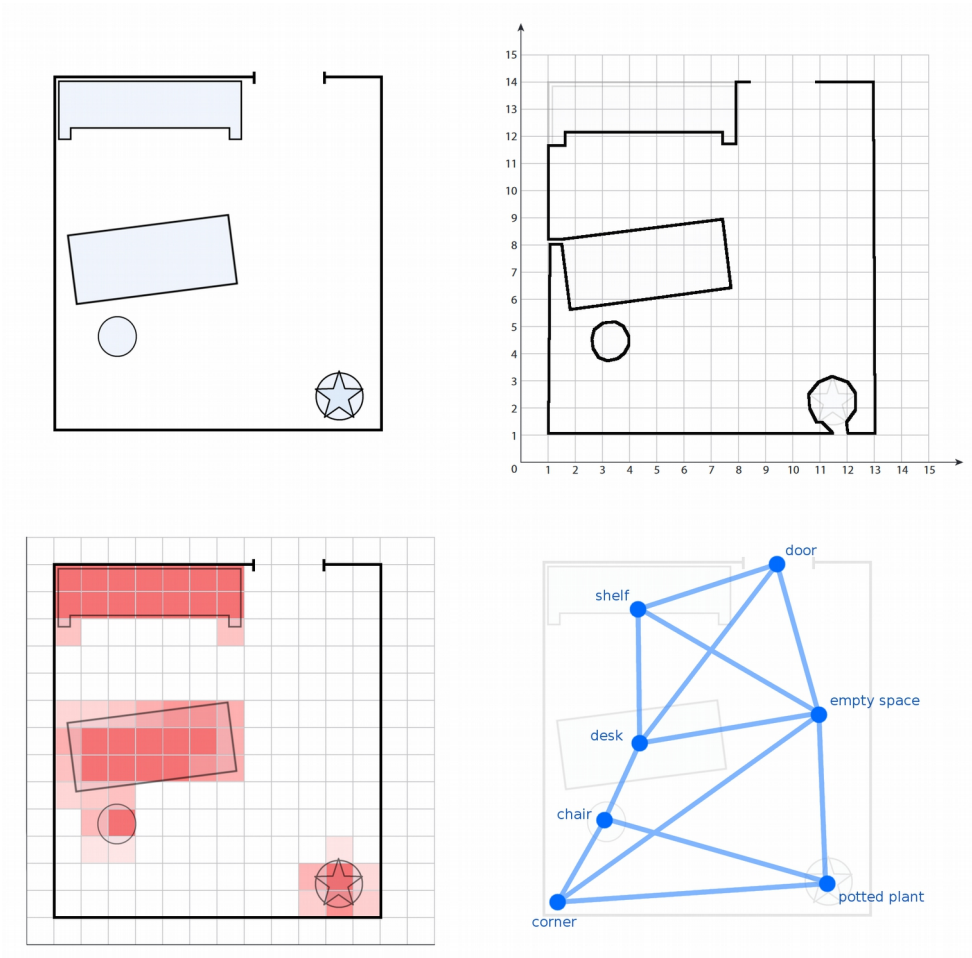

Figure 1. (Heavily simplified) renditions of different map types, all showing the same room, which is depicted in the upper left image. The upper right image shows a Cartesian grid map, with accurately mapped obstacle distances relative to the global coordinate system. The lower left image shows an occupancy grid map, which partitions the room into a coarse global grid. The spaces of the grid are then occupied to various degrees (red patches) by obstacles, and can or can not be traversed. The lower right image shows a topological map: the room is represented as a graph, with nodes being certain locations. The edges could possibly hold distance information such as "3m from door to desk", or movement instructions such as "move right for two seconds to reach the potted plant from the chair." Because of their loose structure, self localization can be quite difficult in topological maps.

each feature in space. Features as a means of localization and mapping enjoy a wide range of application, such as robust camera viewpoint localization and environment estimation for augmented reality [10], or extracting motion from image flow [23].

Instead of relying on grids, a completely different approach to mapping can be found in the use of topological maps: a topological map is usually a graph consisting of nodes corresponding to points or landmarks in space. The edges and nodes of the graph can then for example encode metric information between different

points in space[1], or contain directional or behavioral information needed to reach each point. There are numerous ways on how to approach the creation of a topological map, such as what constitutes a node or how to handle possible sensor uncertainties [22]. As topological maps are usually much less defined and less metrically accurate than their grid-based counterparts, they allow for much faster map operations such as pathfinding, but determining the robot's position relative to the map is much more difficult. For a quick comparison of map types, see Fig.1.

There have of course been approaches that combine the accuracy and complexity of fine grid-based mapping with the simplicity and efficiency of topological mapping: Thrun [18][34] described an algorithm that first created a grid-based map by integrating data from different sources such as vision- or sonar-based sensors. The grid-based map was then partitioned intelligently into critical regions, from which a topological map was created. Finally, both the topological and grid-based map were used to quickly calculate routes for a robot to enable it to navigate a multi-room environment: route planning was done on the topological map, described as several orders of magnitude faster than similar calculations on a grid-based map. The grid-based map was then used for accurate control and motion fine-tuning. Their technology found usage in the University of Bonn's entry RHINO for the 1994 AAAI mobile robot competition, where it managed to win the second price in the “office-cleanup”-category. Later, during a six-day testing period in the museum Deutsches Museum Bonn, RHINO was also used to successfully interact with and guide a large number of visitors while being able to safely navigating dense crowds [35]. The robot's sensor capabilities were additionally enhanced by sensors, and it was able to move at speeds of around 35 cm/s in dense crowds, and up to 80 cm/s otherwise, which resulted in an overall traveled distance of over 18,5 km during the six days. RHINO was however supplied with a hand-constructed map of the museum, as it would have been unable to navigate as accurately with a self-created map (if even able to map such a large area) [18][35]. Self-mapping was later realized in the museum tour-guide robot MINERVA, which was able to map and navigate an environment by taking pictures of the ceiling with a directed camera, and then creating a mosaic-like map from the different images [36].

A major problem of all map types, but especially topological maps, is detecting if a node or location have been visited before. This problem is known as the *loop closing* problem: after a robot has completed a loop motion, it should be able to recognize the place as already visited, instead of creating an ever larger map. But caution needs to be taken: if a loop is mistakenly detected and closed, for example if two distant but similar locations are falsely detected as the same, the entire map, or at least navigation, can be compromised: a route planned over the falsely closed loop can obviously not be navigated. Different solutions on how to detect and close loops exist of course, such as comparing the entire map with itself every few steps, and closing a loop if two matching places are found [15].

1 Topological maps containing metric information are often referred to as topo-metric maps [1].

There are yet other approaches to map building, such as the one detailed in the work of Harris and Pike [14]: the proposed algorithm worked by moving a camera along a path, during which corner features were detected in the image sequence. From the change in the feature's position on the camera image, localization of both camera and features in space could be derived in near real-time. The System is described as accurate for short- and medium-length distances, but because the camera's motion and perceived 3D environment were consistent in error. At long distances, the actual position could possibly deviate substantially from the one derived from the camera images. This problem is usually referred to as *drift.* Also, the system did not try to close loops in any way.
Smith et al. [25] detailed how to update an uncertain map once more accurate information becomes available; all correlations of the localization and mapping problems are united in a single state vector and covariance matrix, which is then updated sequentially by an extended Kalman filter (EKF). A Kalman filter is an optimal estimate for linear system models that contain some amounts of uncertainty in both the measurement and transition systems. To also estimate optimally for nonlinear systems, techniques from calculus such as Taylor series expansions are used to linearize a model respective to a certain working point. The nonlinear equivalent to the Kalman filter is the EKF.
In the late 1990s computing power reached the point where it became feasible to practically test solutions based on the work of Smith et al.: the EKF approach to the SLAM problem was implemented and subsequently proven in multiple different robot systems, and demonstrated the importance of maintaining estimate correlations. This solidified EKF as the core estimation technique in many SLAM solutions, generalizing the problem as a Bayesian probability problem [9].
A major downside of EKF based solutions is the management of large-scale maps, as with growing map size the EKF's computational complexity rises and accuracy decreases due to the linearization operations employed by the filter. To solve this issue, various strategies, such as splitting the map into multiple sub-maps, have been implemented. Especially when coupled with highly accurate distance sensing methods like laser range finding or sonar, maps of impressive sizes can be created, as for example shown by Gutmann and Konolige [27] or Bosse et al. [26] where globally consistent maps of large indoor environment were created, some with a path length of over 2 km. Loops in these environments were closed by comparing parts of the generated maps, and connecting similar-looking ends. This process can be sped up by only comparing small sub-maps at a time, a common method to simplify loop closing [15]. Using faster alternatives to the EKF, successful mapping and localizing in large outdoor areas has also become possible [16].
Navigation approaches do not necessarily need to rely on landmarks, features, or other reference points and their distance measurements: instead, model-based approaches, where the environment is compared to an internal 3D model, let a robot navigate the environment. In the work by Kosaka and Kak [2] for example, a geometrical model of the environment is assumed, and landmarks in the model are matched to landmarks extracted from a monocular camera image. Using the uncertainties created by matching the real environment with the estimated model,

correlation is established. Then, a Kalman filter is used to calculate the position and pose of the robot. In a similar approach, Kak et al. [28] implemented and tested a system that used models of the environment constructed in a CAD program which were then compared to visual data obtained by a robot's camera. Additionally, the robot used odometry data to calculate its position. When the difference between model and camera images passed a certain threshold, the robot position was updated and the odometry data corrected.
Approaches that use a-priori-models have very low computation costs compared to map-creating approaches, but because the environment needs to be known in advance, they can obviously not be used to let robots navigate unknown environments.

The approaches presented all rely on distance measurements to map their environment, either by utilizing stereo discrepancies, odometry, or depth information gained by laser range finders or sonar. Most algorithms rely on 3D measurements of the environment for navigation and scene modeling. However, there have also been approaches that obtain depth information by making assumptions regarding the 3D environment, based only on monocular camera setups. In a work of Lebègue and Aggarwal [24], for example, a system that was able to navigate indoor areas and create a CAD model of them by using continually updated orientation data of objects, was presented. Orientation data of different objects was obtained by extracting line segments from a monocular wide-angle image, and then adding the segments to a CAD model: the line segments were considered edges of surface patches, which were then used to estimate objects in the environment. Crucially, their algorithm assumed that each object in an indoor scene can be represented by linear segments oriented in a relatively limited set of directions. The algorithm then tried to recover these line segments over subsequent frames to solidify their position in the CAD model and increase its accuracy. One of the key advantages of the algorithm was its ability to easily detect doors or similar passable openings, as they are usually large empty spaces framed by orthogonal lines.
However, because the view angle of the robot was still quite limited, even with the use of a wide-angle lens, it needed to keep a certain distance to its environment to detect the line segments and was thus for example unable to navigate and map tight passages [13][24].

Yet another way to map and navigate the environment, using only monocular imaging is not found in robotics and SLAM approaches, but biology: in their 1987 paper, Cartwright and Collett [6] describe how bees can navigate from their hive to a food source using snapshots of the environment taken at the hive and food sources. By comparing its current view to a snapshot, the bee can then home in on the food source or hive, which means traveling in the direction that reduces the difference between current view and snapshot. For this to work consistently, the snapshot and current view filter close objects, because they hold few to no environmental information (a blade of grass, for example, passes the bee much faster than a distant tree). Once sufficiently homed in on the target location, the bee then switches to a snapshot containing close landmark information to accurately locate

the target food source or hive. Contrary to robot navigation tasks, for its navigation, the bee does not need any metrical distance information of its environment at all. Similarly, ants recognize familiar routes, and are able to home in on the routes if they are displaced or get lost. Baddeley et al. [8] present a robot navigation method which uses holistic route representations as its map; a robot was able to find, recognize and follow familiar routes and home in on them if displaced.
Another example from biology is the ability of humans to learn a route or environment simply by being shown pictures of the route or environment, without relying on any metrical data. This has led to the idea of a so-called *view graph* as an underlying topological navigation and mapping structure. A view graph is a graph consisting of different views of an environment as the graph's nodes, for example a bee's snapshots or the route pictures presented to a human; and non-metric movement or directional information on how to reach one view from another as the graph's edges [29].
This serves as a basis for work by Franz et al. [5] where a robot was able to navigate an environment without any metric data at all: by using a single monocular camera facing a conical mirror, 360° panorama snapshots were obtained at different locations. The panorama snapshots were then inserted into a graph, and the robot was able to navigate between neighboring viewpoints simply by homing in to a nearby snapshot. To decide when a new snapshot should be inserted into the graph, and when a nearby snapshot was reached, simple difference threshold functions were used. While the algorithm worked well in their experimental conditions, Franz et al. also described some practical limitations of their work: because their method did not use any metrical information, only locations and views resulting in non-ambiguous snapshots could be used for the graph, as the algorithm had no way to differentiate between two different matching locations, which could for example result in false loop closing. One way to reduce or eliminate this problem, other than including metric information, would for example be to increase resolution and contrast of the panorama snapshots to accommodate for more possible views.
In a later work by Hübner and Mallot [3], the view graph based algorithm was expanded by adding weak metrical data, the robot's odometry, to the graph's edges to create globally consistent maps, which could then be used for route planning and following. Errors in the odometry were compensated by additionally relying on a homing algorithm. Similar to the work before, a new snapshot was added to the graph when it was sufficiently different from all other snapshots. However, using a threshold strategy for deciding when to add a new snapshot caused the following problem: while a new snapshot was rarely taken in the middle of wide open spaces where differences between close viewpoints were low, the algorithm added a lot of snapshots when in close proximity to an obstacle, because the view would greatly change between each movement step.

Based on these concepts, in the following, an algorithm is described, that is able to create a view graph of an environment, and to constantly update the graph with new location- and threshold-independent snapshots, only using a regular monocular camera and weak odometry data. This is achieved by selecting *microsnapshots*,

multiple small points of interest or features, in each scene, rather than using entire (panoramic) images. The features are then connected to a graph by their relative distances, as obtained by odometry data. Finally, the graph can be used to calculate routes through the mapped environment by utilizing a path searching algorithm. The algorithm is not implemented or tested on a robot. Instead, the exploration is simulated by manually moving around a tablet computer, which is able to supply both the camera image and odometry through motion sensors. In section 2, a more detailed explanation of the algorithm, as well as the setup used, is given. section 3 describes the feature detection by giving an in-depth explanation of the used SURF feature detector. View graph creation and pathfinding using Dijkstra's shortest path algorithm is detailed further in section 4. Various short experiments concluded to test the algorithm are described in section 5. Lastly, in section 6, the algorithm's issues and shortcomings as well as possible solutions are discussed, and the work is concluded.

2. Setup

To guide an agent through space using visual information, the visual information constantly needs to be mapped to the space the agent moves in. Every time the agent moves, both the visual information and the position of the agent in space need to be updated. Instead of simply mapping an entire scene to a certain point in space, multiple points of interest are detected by the SURF feature detector (see section 3) in each scene, and connected to each other by their relative distances, creating the nodes and edges of a constantly growing view graph. The environment is explored using a tablet computer held and moved around by a human, rather than a mobile robot equipped with a camera. Similar to a robot's odometry feedback, weak distance data is gained by the tablet computer's motion sensors. Finally, the algorithm is able to calculate a route from the current location to a specified target location by finding the shortest path between two features in the view graph using the Dijkstra shortest path search algorithm (see section 4). Experimentally, only rotational movements could be tested and verified, as the tablet computer's accelerometer failed to provide sufficiently accurate translational movement data.

The algorithm was developed and tested on a desktop computer running the Microsoft Windows 10 operating system, equipped with a quad-core Intel i5-6500 CPU (4 x 3.2 GHz), a Nvidia GeForce GTX 750 Ti GPU, and 8 GB of RAM. Images were taken by Dell Venue 11 Pro 5130 tablet computer, running the Microsoft Windows 8.1 operating system, equipped with an Intel Atom Z3770 CPU (1.46 GHz), 2 GB of RAM, and a 8 megapixels camera filming at 24 frames per second. To reduce bandwidth usage, the images were resized to 640x480 pixels, set to grayscale, and then converted into a lossless JPEG format, which resulted in a size of about 110.000 bytes per image. The image data was then sent to the desktop computer over WLAN using the UDP data transfer protocol, which the

desktop computer then received on a separate thread. Due to the nature of WLAN and the UDP, image data was sometimes corrupted or not received at all. These cases were detected and the algorithm simply skipped the corrupted or missing images.
The field of view of the tablet computer's camera covered 60° horizontally and 40° vertically, which equals to around 0.1° per pixel at the chosen resolution. Additionally, the tablet computer was equipped with both accelerometer and gyrometer sensors. However, due to insufficient accuracy, only the gyrometer data was used to localize points in space. Although the images were taken at 24 frames per second, due to computational restraints, the actual mapping and pathfinding algorithms only ran at frequencies of around 12 Hz to 14 Hz.
The algorithm was programmed in C++; image manipulations and feature extraction were implemented using the OpenCV computer vision library and its implementation of the SURF feature detector [37]. The view graph and graph operations such as Dijkstra's shortest path algorithm use the LEMON Graph Library [38]. Multithreaded WLAN data transfer of camera images and sensor readings is achieved by relying on the Boost.Asio library [39].

3. Feature detection

To create a view graph, rather than taking a snapshot of the full scene at certain points in time or when a differentiation threshold is met (e.g. in [3][5]), multiple different microsnapshots are extracted each time the scene updates. The microsnapshots are not arbitrarily chosen parts of the image, but distinctive points of interest, or *features*. The most important property of these features is their repeatability: the algorithm needs to be able to find and extract the same features over multiple images of the same scene, even under (slightly) different viewing conditions such as variations in lighting or perspective, small changes to rotation of the viewing angle, or distance to the same scene.
There are multiple different approaches to finding and extracting features: a popular approach for 3D reconstruction of a scene is for example the use of corner and edge detectors [31]. However, while a corner detector can match a distinctive set of edges and corners to each scene, corner features alone are often not very distinctive from each other, and particularly lack robustness to viewing angle. A more useful approach is found in image patch feature detectors. Feature detectors such as SIFT by Lowe [7][32], or SURF (Speeded-Up Robust Features) by Bay et al. [30] find interest points based on mathematical transformations of the image, and then use a patch of a certain size around each interest point as a distinctive descriptor. More recently, even faster and more distinctive and robust feature detectors such as the ORB feature detector, which is described as an order of magnitude faster than both SIFT and SURF, have been developed [11].

In this bachelor's thesis, the SURF method by Bay et al. [30] is used for feature

detection, as a C++ implementation for both SIFT and SURF is readily available in the OpenCV computer vision library [37], and it is both computationally faster and more robust against white noise than SIFT. In the following segment, a brief description of the SURF algorithm is given[2].

The full feature detection process contains three steps (of which the first two are described in 3.1 and 3.2, respectively): First, repeatable interest points need to be detected. Then, each interest point is described by a description vector containing information about its close neighborhood. Finally, the features are matched with each other to find out if a detected feature is new and needs to be added to the view graph, or is already part of it. The matching step works by comparing the description vectors of two features, and calculating their difference, for instance by calculating their Euclidean distance. As the dimensionality of the description vector directly impacts calculation time, a shorter description vector is preferable, albeit less descriptive.

3.1 SURF interest point localization

To gain complex features that are still robust against commonly occurring deformations, both detectors and descriptors are chosen to be scale and in-plane rotation invariant. Image skewing, anisotropic scaling and other perspective effects are assumed to be second-order effects that are (at least partially) covered by the overall robustness of the descriptor. The SURF method ignores color information completely, which allows for faster computation and slightly increases robustness against illumination changes.
Interest points are detected by using a very basic Hessian-matrix approximation over the image. For the approximations, the image is transformed into its integral image, as integral images allow for fast time-constant calculation of image intensity over any upright, rectangular area, such as convolution filters. The entry of an integral image $I_\Sigma(\mathrm{x})$ at a location $\mathrm{x} = (x, y)^\mathrm{T}$ represents the sum of all pixels in the input image I within a rectangular region formed by the origin and x.

$$I_\Sigma(\mathbf{x}) = \sum_{i=0}^{i \le x} \sum_{j=0}^{j \le y} I(i, j)$$

The interest points are precisely detected as small blobs at locations where the determinant of the Hessian matrix is maximal. Given a point $\mathrm{x} = (x, y)$ the Hessian matrix $H(\mathrm{x}, \sigma)$ in x at scale σ is defined as follows:

2 The description of the SURF feature detector given in this section is a brief summary of the description given in the original SURF paper by Bay et al. [30]. All images and formulas used in this section are also reproduced from the same paper.

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

$L_{xx}(x, \sigma)$ is the convolution of the Gaussian second order derivative $(\delta^2 / \delta x^2)g(\sigma)$ with the image I in point x, and similarly for $L_{xy}(x, \sigma)$ and $L_{yy}(x, \sigma)$. The second order Gaussian derivatives are not actually calculated, but rather approximated using box filters (Fig. 2) and can be evaluated at very low computational cost using integral images.

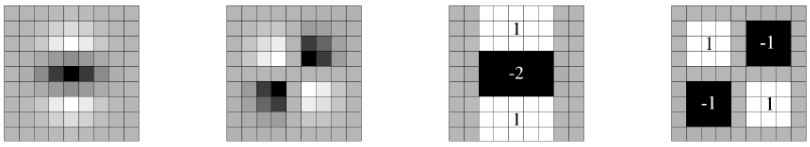

Figure 2. Left to right: The Gaussian second order partial derivative in *y*- (L_{yy}) and *xy*-direction (L_{xy}), and their approximations in *y*- (D_{yy}) and *xy*-direction. (D_{xy}). The grey regions are equal to zero.

The approximation results in only minor accuracy loss at substantial computation time increases. The 9x9 box filters in Fig. 2 are approximations of a Gaussian with $\sigma = 1.2$ and represent the highest spatial resolution for interest point blob detection. The box filters are denoted by D_{xx}, D_{xy}, and D_{yy}, respectively. They are intentionally kept as simple as possible to further increase computation speed. The determinant is calculated as follows:

$$\det(\mathcal{H}_{\text{approx}}) = D_{xx}D_{yy} - (wD_{xy})^2$$

ω is theoretically a relative weight calculated off of the filter responses to balance the expression for the Hessian matrix' determinant. This is needed for energy conservation between the actual Gaussian kernels and their approximations. The actual ω used in the SURF algorithm is kept constant at 0.9.
The determinant gives the interest point blob response strength for location x in the image *I*. The response strength is then stored in a blob response map, and the process is repeated over different scales. Usually, different scales are implemented

using an image pyramid, where an image is repeatedly smoothed with a Gaussian and then sub-sampled to achieve higher scales; the integral images used in SURF however allow for Gauss smoothing of any size on the original image: instead of down-sampling the image size, the different scales are analyzed by up-scaling the filter size, which brings various advantages such as fast computation speed and no aliasing due to down-sampling. The smallest scale is the 9x9 box filter (σ = 1.2) presented in Figure 2, and the next scale pyramid layers are obtained by filtering the image with progressively larger filters. The different response maps that are gained by this filtering method are then united in a so-called *octave*. An octave represents a total scaling factor of two, with each octave being subdivided into a constant number of scale levels.

The sizes of the different filters that make up an octave's scale level are dependent on the box filter shape: because the relative sizes are kept intact, the next possible scaling level after a 9x9 filter is a 6 pixels increase to a 15x15 filter, for example (Fig. 3). The next two filter size levels of this smallest octave are 21x21 and 27x27 pixel. While the octave's scaling factor of two is already reached (and passed) at 21x21 pixel, the largest filter size is used to create some overlap between neighboring scale levels. For each successively larger octave, the filter size increase is doubled (from 6 to 12 to 24 to 48 etc.). The intervals, at witch the interest points are sampled are also doubled with each octave to reduce computation time. Because the sampling of scales is quite crude even at the smallest filter size (9x9 to 15x15 is an increase of 1.7), a second finer scale sampling is repeated on the same image after its size has been doubled via linear interpolation.

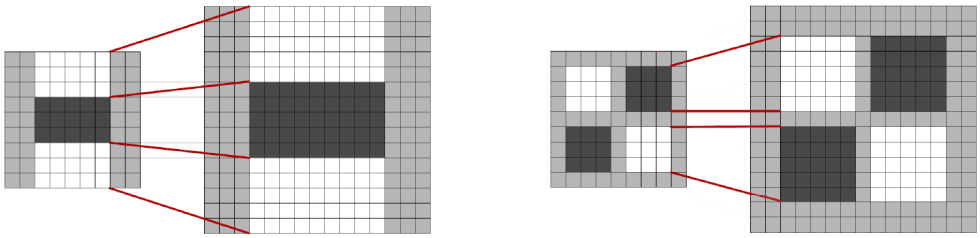

Figure 3. Filters D_{yy} (left) and D_{xy} (right) for the scale levels 9x9 and 15x15, respectively. The length of the dark lobe in the D_{yy} filter can only be increased by an even amount of pixels in order to guarantee the presence of a central pixel.

3.2 SURF descriptor extraction

The interest points found in 3.1 are now each assigned a descriptor vector. The descriptor specifies the distribution of image intensity within the interest point's neighborhood. The intensity information is gained via the use of so-called Haar wavelet filters (Fig. 4) in x and y directions. The entire description vector is constituted of 64 dimensions.

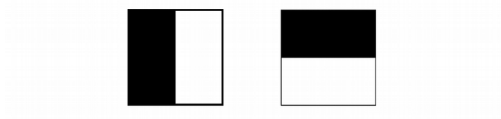

Figure 4. Haar wavelet filters to compute the responses in x (left) and y (right), respectively. The dark part has a weight of -1, the light part +1.

Because the SURF algorithm is supposed to be rotation invariant, the first step of descriptor extraction consists of assigning each feature a reproducible orientation: in a radius of 6 s around each interest point, with s being the scale level at which the interest point was detected, Haar wavelets with size 4 s are used to calculate in x and y direction. Afterwards, the responses are weighted with a Gaussian (σ = 2 s) centered at the interest point. The different responses are then represented as points in space. By summing up the response strengths with a sliding window of size $\pi/3$, a local orientation vector can be calculated (Fig. 5).

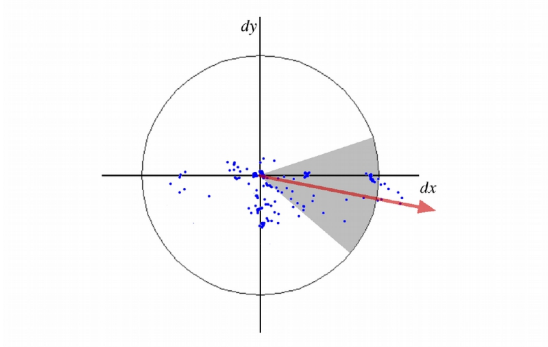

Figure 5. Orientation assignment: a sliding orientation window of size $\pi/3$ detects the dominant orientation of the Gaussian weighted Haar wavelet responses at every sample point within a circular neighborhood around the interest point.

The second step of descriptor extraction consists of constructing a 20 s large square region around the interest point, which is aligned to the just selected orientation vector. This region is then split up into smaller 4x4 square sub-regions, in each of which Haar wavelet responses are calculated at 5x5 regularly spaced sample points. The responses are calculated in horizontal (dx) and vertical (dy) direction respective to the interest point's orientation (Fig. 5). Wavelet responses are also weighted with a centered Gaussian (σ = 3.3 s) to further increase robustness against deformation and localization errors. Additionally, to gain information about the polarity of the intensity changes, the sum of absolute response values, $|dx|$ and $|dy|$, is also extracted. The sums over dx, dy, |dx| and |dy| are each a single entry in the description vector, which, for 4x4 sub-regions, results in a 4x4x4 = 64 dimensional description vector.
An interest point and its descriptor make up a feature. Features detected in a scene can now be matched by comparing their description vectors, for example by calculating the Euclidean distance between them. If the distance is shorter than a certain threshold, the feature can be considered the same.

4. Graph creation

4.1 Feature insertion

In the algorithm presented in this bachelor's thesis, the SURF feature detector is used to extract a number of features in each scene. As the algorithm is supposed to run in real-time or near real-time, feature detection (and graph calculation) is done at around 12 Hz - 14 Hz. Even on a resolution of 640x480 pixels (the image size used), SURF feature detection is able to find over 300 features in a single frame. This would mean that over 3600 features would need to be calculated, matched to each other, and matched to old features every second – thankfully, SURF and its C++ implementation of the OpenCV library [37] allow for multiple different strategies to increase computation speed.
The first, and probably most significant, speed increase is achieved by skipping SURF's interest point orientation calculation. For horizontal exploration of an environment, feature rotation invariance is actually not needed (as the environment is not explored at strong rolling angles, sideways or upside-down). Thus, each feature can be assumed to be upright, which allows for skipping of a major part of the descriptor extraction calculation to greatly increase computation speed. Skipping the orientation step is a feature already implemented in the original SURF algorithm, called SURF-U (upright) [30]. Note that upright SURF features are still robust against rotations up to +/- 15°.
Furthermore, because less features mean faster follow-up calculations, the number of detected features is reduced by two methods: for one, it is possible to increase the threshold an interest point needs to pass in the Hessian matrix to be considered a feature. Depending on the threshold chosen, the amount of features detected in each scene is tremendously reduced.
The amount of features is then further reduced by filtering out “bad” features, such as features that “flicker” because the algorithm fails to find them in consecutive frames: the features are filtered by comparing the currently detected features to the features detected in the two preceding frames to one another. Only features that can be detected in all three frames are considered further by the algorithm. This method however somewhat limits the maximum speed the camera can be moved, as no features are detected at all if the image changes too quickly, for example during fast movements. The motion blurring occurring during fast movements might also prevent many features from being detected[3]. Additionally, to further reduce feature number, the SURF octaves are set to the low number of three; this causes the detector to find features at less different scales, and also limits scale invariance. This is an acceptable trade-off, because scale invariance is not particularly important (or even wanted, see section 6.3). While without these corrections around 200 to 300 features are found in each image, the number drops to a more manageable 20 to 40 features per frame after filtering (Fig. 6). Finally, a

3 As SURF features are supposed to offer some robustness against blurring, this never seemed to cause major problems during testing.

hard cap of 50 is set as the maximum amount of features considered in each frame, but this number was rarely reached during testing.

Figure 6. Features detected in a scene are marked by a black circle. The black line in the circle shows the feature's orientation, which is the same for every feature, as the orientation calculation step is skipped.

4.2 Graph connection

The 20 to 40 features detected in each frame are first matched with the features already in the view graph. If no match is found, the feature is considered a *new* feature, and is added to the view graph by connecting it to seven randomly chosen *old* features, which were detected in the preceding frame and are currently not visible. As the graph is directed, each connection contains two edges, one from the old to the new feature, and one from the new to the old one. The number of connections is limited in order to reduce the number of edges added to the view graph in each step, to increase computation speed. The number seven was chosen as it proved high enough to not arbitrarily disconnect the graph[4], while usually still being substantially lower than the maximum number of connections that would occur if each feature was connected to all features in the frame before.
If two features are found matching, the newly detected feature is usually considered to already be part of the graph, and no new feature is added. The old feature is then also connected to up to seven features in the preceding frame, which results in automatic loop closing. A fatal problem that could possibly occur during feature detection and matching is that two features, that are actually in different places, are considered the same: if the matching new feature was falsely considered old, a false loop would be closed. This could then compromise the entire graph, and especially the pathfinding using Dijkstra's shortest path algorithm (section 4), as this falsely closed loop would represent a shortest connection between two possibly extremely distant locations. To prevent this from happening, each time a matching feature pair is found, the features close to them (the other currently detected features for the new feature, and the in the graph directly neighboring

4 This is extremely unlikely to happen. On average, over 200 connections are made between two different frames. During testing, the graph only ever disconnected if multiple consecutive frames were skipped due to data transfer corruption, or when the camera was moved extremely fast or abrupt. Even then, connection can be re-established by scanning the same area again.

features for the old feature) are also compared. Only if a percentage of features higher than a certain threshold is also found matching, the features are ultimately considered the same. The algorithm is biased towards falsely detecting an old feature as new, rather than falsely detecting a new feature as old, as the latter can possibly have catastrophic consequences. Loops are still highly likely to be closed: because a number of features are detected each scene, even if the majority are detected wrongly as new in a situation where a loop should be closed, a single feature can be enough to close the loop for graph searching and pathfinding purposes. While testing, a similarity threshold of 50 % (more than half of the features close to two feature must match for it to be considered the same) seemed to produce good results.

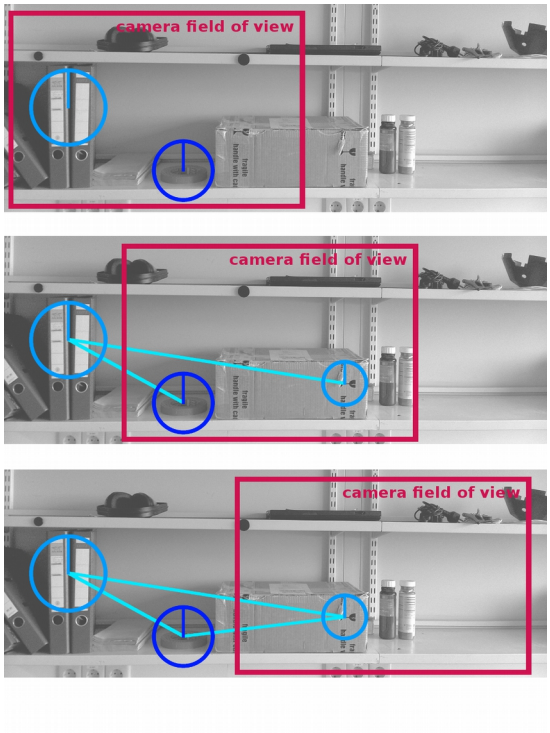

Figure 7. Schematic graph connection, from top to bottom: first picture: in the first frame, two features (blue) are detected, but not connected, as they are both concurrently visible. Second picture: as soon as the camera moves to the right, the dark blue feature and a newly detected one are connected to the old feature which is not in the camera's field of view anymore. Again, no connection is made between the two concurrently visible features. Third picture: finally, the new feature can be connected to the dark blue one, as it left the camera's field of view.

The view graph's edges are created by connecting all currently visible features to the preceding frame's features, except the ones that are visible in both frames. Features visible in both frames do not cause addition of new edges to the graph in order to limit graph growth while the camera is not moving[5] (Fig. 7). The (rotational) distance between the two features, meaning the rotation needed to move the camera from one feature to the other, is then calculated by their relative position in the image and the difference in viewpoint between the two frames. The difference in viewpoint is inferred from the rotational velocity (in degrees per second) received from the tablet computer's gyrometer: both yaw distance d_z (rotation in horizontal direction) and pitch distance d_y (rotation in vertical direction) are obtained by multiplying the respective rotational velocity v_z and v_y with the time t

5 Even while not moving, new edges are still occasionally added to the graph because of "flickering" features that are for example only detected every other frame.

passed since the last frame in seconds. Thus yaw distance in degree is obtained by $d_z = v_z \times t$, and pitch distance by $d_y = v_y \times t$ respectively.
Rolling distance (rotation along the view axis) is neglected by the algorithm, as it is not needed for exploration, and also allows for use of the much faster upright SURF features, as they do not need to be rotation invariant.
The distance obtained by the calculations describes the rotation from the old feature to the new one. Because path finding needs to be able to work in both directions, and the rotations are not absolute, a second edge describing the inverse rotation from the new to the old node is created. If an edge between two nodes already exists, it is not created a second time, but the value of the edge is instead updated with the average between the old and new value.
In theory, adding linear translational distances to the view graph would work analogous. In fact, edges are still added if the tablet computer is only moved in a line without rotating it. However, as the sensor data obtained by the tablet computer's accelerometer is not even sufficiently accurate to calculate only the translation direction testing was difficult during experimentation.

4.3 Dijkstra's algorithm

The navigation algorithm presented in this work is not only able to map the environment by using features extracted from multiple different viewpoints, but also able to calculate routes for navigation of the mapped environment by finding the shortest path from the current scene to a user-selected target feature in the view graph. As both the view graph and perspective change, in each frame, the route is fully recalculated; at the graph sizes tested (2000 - 4000 nodes), the calculation had no major impact on the overall computation time[6]. The shortest path is found by using the algorithm known as Dijkstra Shortest Path Search (SPS), or simply Dijkstra's algorithm, which was first described by Dijkstra in 1959 [33]. Dijkstra's algorithm was specifically chosen because of its readily available implementation in the used LEMON code library [38].
Dijkstra's algorithm[7] is an algorithm able to find the shortest path between two nodes *P* and *Q* in a connected, directed graph with non-negative edge weights. Rather than attempting to “explore” the graph towards the destination, it does this by iteratively calculating the shortest distance between the *source node P* and all other nodes in the graph, expanding outwards from the source until the target node has been reached. The strategy is based on the fact that if *R* is a node on the shortest path from *P* to *Q*, knowledge of the latter implies knowledge of the shortest distance from *P* to *R*. The algorithm stops if the shortest distance to *Q* has been

6 There still are different ways to reduce the computation time, if so desired: For example only calculating a route every other (or more) frames, or only re-calculating parts of a route. If the graph becomes large enough that route calculation slows down noticeably, different algorithms or graph managing strategies, such as splitting the graph into sub-graphs, could be employed (see also section 6).

7 The explanation of Dijkstra's algorithm given in the following is an adaption of Dijkstra's original description in [33].

found, or if the shortest distance to all connected nodes has been calculated (in which case no path from *P* to *Q* exists) (Fig. 8).

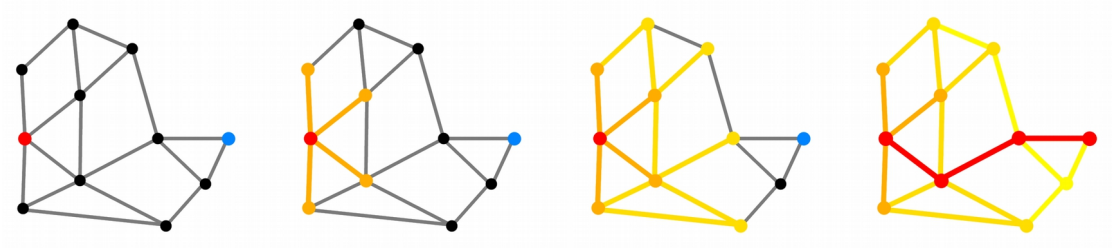

Figure 8. Dijkstra's algorithm makes no attempt to directly search a path from the starting node (red) towards the target node (blue). Instead, the only factor considered in determining which node to choose next, is a node's proximity to the source. The algorithm slowly expands outwards (the more yellow a node, the further away it is from the source) until the destination is reached. Depending on the graph's topology, the process can be quite time consuming: in the example above for instance, the entire graph has to be searched before the shortest path is found.

Initially, a starting node *P,* called *source*, needs to be selected. The distance of a node *R* is the distance from the source node *P* to *R*. Each node is then assigned[8] their currently known shortest distance d_a, which is set to zero for *P* and to infinity for all other nodes in the graph (to represent that the shortest distance is not yet known); the algorithm will update these distances with every step. Then, the source node is selected as *current* node, the node which is considered for distance calculation in each step of the algorithm. All nodes (including the source node) are marked as unvisited, and put into the *set of unvisited nodes*.

1. Calculate a tentative distance d_t between the source node and each of the current node's unvisited neighbors by summing up the current node's assigned distance d_a and the distance between the current node and its neighbors. Compare the tentative distance d_t calculated for each node to its currently assigned distance d_a. If d_t is smaller than d_a, set d_a to d_t. Otherwise, the old d_a is kept. After all neighboring nodes of the current node have been checked, the node is marked *visited* and removed from the set of unvisited nodes. The shortest path to this node is known, and it will not be considered by the algorithm any further.
2. If the target node *Q* has been marked visited and removed from the set of unvisited nodes, the shortest distance to *Q* (its assigned distance d_a) has been found and the algorithm can stop. Similarly, if the set of unvisited nodes is empty, the algorithm also stops, as the shortest distance to all nodes in the graph has been calculated.
3. Select the node with the smallest d_a from the set of unvisited nodes, and set it to the new current node. Then return to step 1.

8 In Dijkstra's original description, rather than marking nodes or assigning values to them, instead a set of shortest known distances exists, which starts out empty and is then slowly filled by the algorithm.

Once distance calculation is finished, if a shortest distance to Q has been found, the shortest path can then be determined by moving backwards through the graph from Q to P by selecting the neighboring node with the smallest d_a in each step.
An agent could now use this path to navigate the environment by following the movement directions encoded by each edge. If, for example, the path describes a rotation of a certain angle to reach a target view, it can be reached by following the rotations specified by the path one after another (Fig. 9). The total distance to the target can also quickly be inferred by treating the graph's edges like multi-dimensional vectors and summing them up; the length of the resulting vector is the total distance to the target. Additionally, the vector also represents the shortest possible distance between source and target, a straight line. While not implemented in this algorithm, vector addition could possibly allow for finding of faster routes and shortcuts – although the algorithm would then need a method to detect if factors such as obstacles prevent the shorter route from being taken.

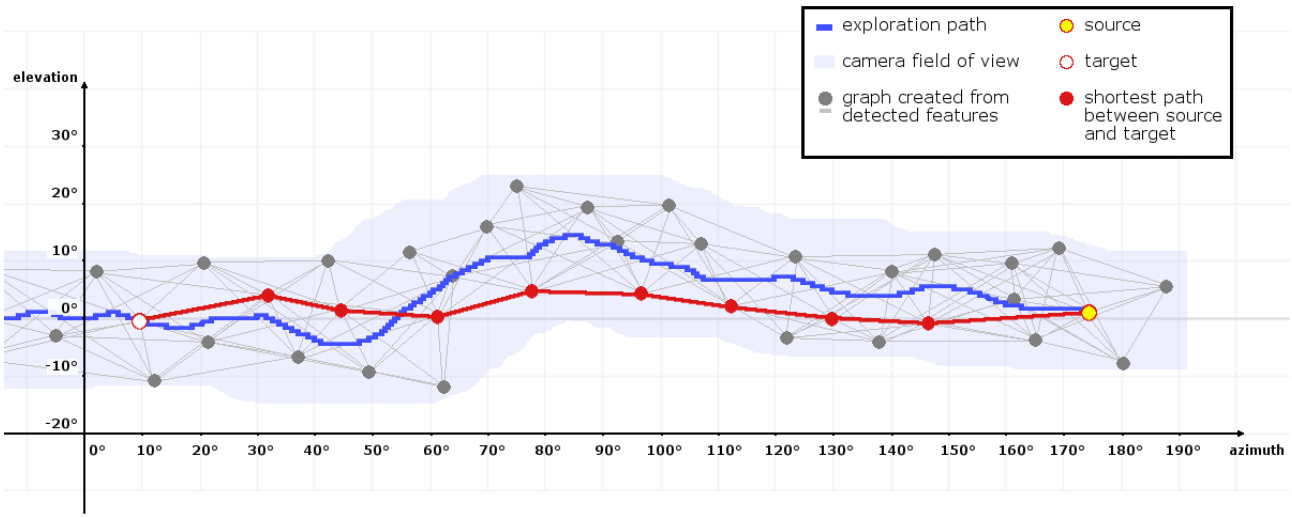

Figure 9. Mock-up: an environment is explored by rotating a camera around 180° to the right (dark blue path). The environment is mapped by adding features detected in the camera's field of view (bright blue area) to a topological graph (gray). Using Dijkstra's algorithm, the shortest path (red) is then calculated from the current position (yellow) to a target node (white). While the path found is much shorter than for example backtracking the exploration path, it is not the shortest possible movement (which would be a straight rotation along 0° elevation). Note that the field of view of an actual camera may cover a much larger area than the here depicted 30° x 20°.

5. Testing and Experimentation

The presented algorithm was tested under a multitude of different conditions, which can be roughly summarized in three different experiments:

In the first experiment, the algorithm was tested by placing the tablet computer on a pivot joint allowing for horizontal rotation and featuring a degree display. The tablet computer was manually rotated while calculating a path to a target feature; the rotational distance between the current view and the target feature returned by the algorithm was then compared to the rotational distance on the pivot joint's degree display. Additionally, the algorithm's ability to close loops was tested by rotating the tablet computer over more than 360°. The algorithm was both able to return accurate distance measurements, and close the loop after a full rotation. The loop closing was verified by the shortest path returned: while a target feature situated at a location close to the exploration's start would result in a longer path the further the tablet computer was rotated, the algorithm would immediately switch to a path describing the shorter rotation into the opposite direction as soon as a full rotation was finished (Fig. 10).

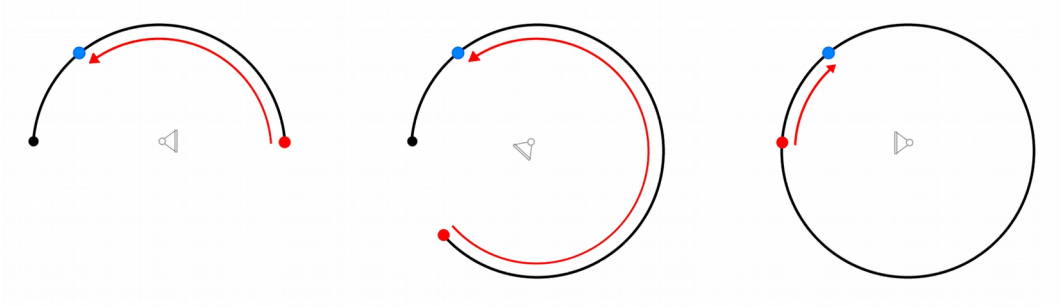

Figure 10. Left: the explored and mapped environment (black), target feature (blue), current position (red dot) and shortest path towards the target (red line). Middle: as exploration is continued in a clockwise circle, the length of the shortest possible path towards the target increases. Right: as soon as a full rotation has been completed, a loop is closed, and a much shorter path towards the target can be taken.

In a second series of experiments, the tablet computer was held by a person and rotated both in horizontal and vertical direction. While this method did not allow for accurate verification of the returned rotational distance, the information returned by the algorithm still seemed accurate at easily verifiable angles such as 90° or 180° [9] for horizontal angles. While vertical rotation by itself also produced good results, some (expected) issues occurred when mixing both horizontal and vertical rotations. For one, the algorithm was not able to close a loop by rotating

9 Because the distance between features is dependent on their relative position in the camera image, and the rotation angle covered by the field of view changes if the camera is rotated in a larger circle, drift occurs in the calculated distance data. As this only minorly impacted the distances at tested rotation ranges, no correction was done.

the tablet computer 180° in horizontal, and then 180° in vertical direction, as feature rotation is deliberately ignored by the algorithm, and the upside-down known features were simply detected as new. Because roll rotations around the view axis were also ignored, it was also possible to deliberately "cheat" the system, for example by moving the tablet computer 90° upward, then rotating it 90° around the view angle, and finally moving it 90° downward again to simulate a 90° rotation in horizontal direction. The algorithm would however assume that the tablet computer was not rotated horizontally at all, but simply moved up and down. The algorithm was not expected to solve these issues – it is supposed to mainly model exploration of a horizontal environment with some variation in elevation.

The third set of experiments focused on the algorithm's ability to map translations: while no translation distances are detected when the tablet computer is moved linearly, new edges are still added as the scene changes. The distances represented by the edges are then described by their rotational distance, which is simply zero or close to zero for translations. As such, the algorithm is able to calculate routes using translations as normal, but their correctness is difficult to ascertain.
In one series of experiments, the camera was rotated a certain angle (e.g. 180°) away from the target feature, and then linearly moved to a different location, creating a translation edge. If rotated at the new location, the distance angle towards the target feature would never drop below the distance at which the tablet computer was moved away from the first location, because the only possible path towards the target feature included the translation edge (Fig. 11). As soon as the tablet computer was moved back to the original location, loops were closed, and a shorter path could be found again.

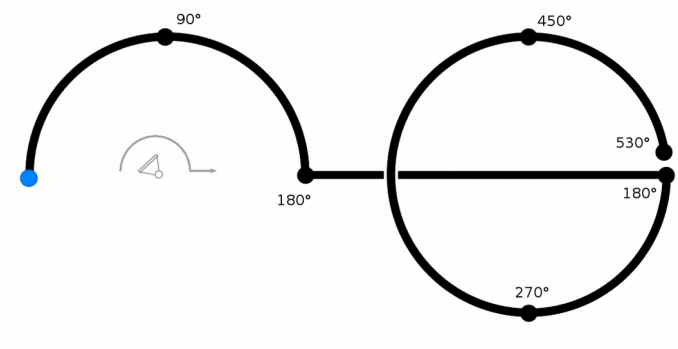

Figure 11. The environment is explored by first rotating the camera in a semicircle, then moving it to another location, where rotational exploration is continued (black line). The rotational distance towards the target feature (blue) increases beyond 360° on the second circle, as the only possible path back to the target must include the translational movement.

Issues occurred however, if many features of the original location were visible from the second location, for example if the locations were close or directly next to each other: because SURF features are scale-invariant and robust against shifts in the viewing angle, and translation data to verify locations was not available, the algorithm would wrongly assume its current position as the same as the original location, possibly falsely closing loops, which would then result in an incorrect route calculation.

6. Discussion

6.1 Summary

In this bachelor's thesis, a method to learn maps of an environment, as commonly used for robot navigation tasks, using only monocular visual information and weak metrical data derived from a rotation sensor, was presented. Maps were created by extracting so-called features, prominent points of interest, from images taken in different locations. The features were then used to create a graph, with the nodes being the features, and the edges the distances between the features, as obtained from the rotation sensor, creating a topological map of the environment. Finally, a route to traverse the map could be calculated by finding the shortest path between two feature nodes in the graph. The algorithm was employed and tested on a two-device-setup consisting of a computationally weaker tablet computer equipped with a camera and motion sensors, and a more powerful desktop computer to handle the calculation steps.

While the algorithm was both able to successfully map and find navigable routes in an environment, it was somewhat limited by various factors, most prominently the inability to map translational movement. In the following, various issues and limits the algorithm encountered, as well as possible solutions, are discussed.

6.2 Translational movement

The original plan for the algorithm included both translation and rotation as possible movements, albeit always separated into either translation or rotation to skip complications caused by curve calculations. However, the tablet computer's accelerometer, the sensor used to detect linear accelerations, provided insufficient data[10]. Movement detection from optical flow was briefly considered, but not implemented due to the limited time frame provided for work on a bachelor's thesis. Linear movement detection would be less of an issue if the algorithm was used to navigate an actual robot, as the robot's odometry could have been used.
Both the gyrometer data used in this bachelor's thesis, and the odometry data that could possibly be gained from a robot, need to be consistent for the algorithm to work: errors in scale, for example consistently measuring 10° as 9°, 20° as 18° and so on, can simply be corrected by multiplication, but if the offset is drifting wildly each time a distance is measured, for example 10° as 9° one time, and 10° as 15° another time, the graph's structure may become very inconsistent. In the algorithm, wrong measurements are somewhat remediated by taking the average of two measured distances if an edge is detected again, but this only helps as long as

10 Distances calculated from linear acceleration are very susceptible to drift in the first place, as the acceleration needs to be integrated twice. Rotational distance on the other hand is gained from integrating rotational velocity only once, and is as such much more reliable.

the differences measured are small, or the same environment is explored many times. Additionally, because sensor data is received over WLAN using UDP, a small chance that the data may be corrupted into completely different numbers exists[11]. If constructing a graph based on bad sensor data is to be avoided at all costs, it would also be possible to create a graph using no metrical data at all – simple directions could suffice for route calculation: the first direction specified by the shortest path is followed until the target location is reached. The direction changes slowly over time, as the route is calculated fully in each step, and the agent moves closer to the target location. Because some movements needed to close in on the target location might be very short and exact, this method is somewhat difficult to precisely verify by humans, but it could easily be implemented to navigate a robot. Unfortunately, a complete lack of metrical data makes exploration (and resulting graph creation) more difficult: locations close to one another are easily confounded, as many matches are found between the features detected in both locations, and they cannot be further differentiated by metrical data (see section 5, third set of experiments).

6.3 Feature detection

Confounding of close locations probably occurs especially because of the properties of SURF features: during forward motions, for example, matching features are quickly detected because of the SURF features' scale invariance. It is possible to reduce scale invariance by reducing the number of octaves used by the SURF filters. However, if both rotation and scale invariance, the distinctive qualities of SURF features, are ignored, and robustness to changes in view angle may also not be wanted, it might prove advantageous for the algorithm to use a different, simpler, and possibly faster method for feature detection. A detector using less Gaussian smoothing for noise reduction, or the method described by Gálvez-López and Tardós [12] which uses much faster to compute, rotation and scale invariant FAST features with BRIEF descriptors, while still being able to detect and close loops as efficiently as methods using SURF features, may be preferable.
For the presented algorithm, SURF features were mainly chosen thanks to their readily available C++ implementation, and no other methods or feature detection were considered due to time constraints.
Furthermore, while the features can be used both to map and navigate an environment, they cannot be used for obstacle avoidance, as the distance *towards* the features is not measured in any way. Thus, if the algorithm was to be implemented for robot navigation, the robot would need to be equipped with a separate obstacle avoidance system, such as short-range infrared sensors.
The fact remains that the presented algorithm is computationally resource intensive, especially during the feature detection and matching steps. While the code was not optimized heavily, and the hardware used (see section 2) was able to run

11 This chance is much smaller than image corruption, as the sensor data consists of a single double (8 Byte). It is much more likely that the entire double is lost during packet loss, in which case the algorithm simply uses the last available sensor data.

the algorithm in real-time, it remains to be seen if the algorithm could be implemented in a small-scale autonomous mobile robot without wireless connection to a more powerful computer – the tablet computer used for testing was for example unable to process the SURF feature extraction in real-time, even if the image resolution was reduced further.

6.4 Graph management and pathfinding

During testing, the graph usually grew to sizes from 2000 to 4000 features after a few minutes, mainly depending on the speed the environment was scanned with (faster speed means less features due to camera blurring and the employed three-frame-comparison filter), and how much of the environment was scanned. After an horizontal 360° circle was scanned, graph growth slowed down, as most features detected were already part of the graph. The computation speed of neither feature matching nor Dijkstra-pathfinding seemed to be negatively impacted by the found graph sizes. However, if an entire environment was to be gradually explored (if the algorithm was used by a robot for example), instead of just relying on single points, the graph would quickly grow to much larger sizes. It is easily imaginable that at some point, the graph would grow too large to still allow for real-time or near real-time feature matching and pathfinding. Fortunately, there are different strategies that can be employed to deal with large graphs, such as the partition of the graph into multiple smaller sub-graphs:
Imagine the environment to be two separate rooms connected by a single door. Instead of a big graph covering both rooms, two smaller sub-graphs, one for each room, could be used. Feature matching and pathfinding could then only use the sub-graph of the current room, which would lead to a substantial increase in computation speed. This approach however poses some problems: during mapping, the algorithm would for instance somehow have to be able to detect that a new room had been entered, and creation of a new sub-graph could be started. Additionally, if the exploring agent moved back from the second into the first room without noticing, the automatic loop closing would fail, as the features would only be compared to the second room's sub-graph[12]. The sub-graphs' loop closing problem could be remediated by comparing the entire sub-graphs to one another. To keep up fast computation speeds, this comparison could be spaced out to regular intervals, say every sixty frames. Similarly, the feature matching step could greatly be sped up by only comparing the newly detected features to close features in the graph rather than the full graph, and again using full-graph comparison to close loops at regularly spaced intervals.

Another possible strategy to reduce graph size is node deletion. Certain nodes in a graph are possibly only detected and added once, and then never found again – a moving obstacle, such as a person moving through the scene would not show up

12 Doors and similar openings could for example be detected by integrating the algorithm with the line-segment-based approach proposed by Lebègue and Aggarwal [24] (described in section 1).

in successive scans of the same location. While the nodes gained from the person are never found again, they still take up space in the graph (and thus have an impact on computation speeds). Similarly, there could be nodes that are simply never rediscovered due to factors such as camera errors, illumination changes, or perspective. But how can these “superfluous” nodes be detected and subsequently deleted? One answer is found in graph- or edge-weighting: if, for example, the same features and the edges connecting them are found multiple times, the edges could be assigned a weight. Each time the edge would be found, its weight would increase, and more importantly, the weight of all other edges connected to the same feature node would decrease. At some point of repeatedly mapping the same location, certain edges (such as those connecting to a feature that was only found once) would have a very low weight. An edge having a weight lower than a certain threshold could then simply be deleted form the graph. Finally, if a node would lose its connection to the graph in that way, it could be removed. This strategy would also allow a graph to “unlearn” faulty or unreliable information about the environment.
Instead of outright feature node deletion, the nodes and edges could also simply be rendered inactive, for example by marking them a certain way, or moving them to a set of inactive edges and nodes (or an inactive sub-graph). The inactive nodes could then, similar to the sub-graph strategies, be compared to the active nodes in spaced intervals, and possibly be reactivated. Inactivation instead of deletion would for example prevent the graph from disconnecting large parts by deleting a crucial connecting node or edge, simply because that exact node has not been visited for a while.

Graph weighting could additionally be used to increase pathfinding speeds: a often chosen and traveled route could be assigned additional weight. A route assigned more weight could preferably be chosen by a pathfinding algorithm, albeit not Dijkstra's algorithm: Dijkstra's algorithm calculates the distance to all nodes in the graph, based on proximity to the source node (Fig. 11), irrespectively of any weights assigned to the graph.
Rather than using Dijkstra's algorithm, different pathfinding algorithms like the A* algorithm [17] (or any of its more recent variations), which utilizes heuristics such as exploration in target node direction, or where to continue pathfinding when an obstacle is detected, could be used to both efficiently use the set weights and find a path (although possibly not the shortest) in much faster time.
Even if graph weighting or a different pathfinding algorithm are not employed, there are still ways to speed up the Dijkstra pathfinding process: by starting the path search from both the source node and backwards from the target node and meeting halfway, calculating distances to large parts of the graph can possibly be skipped.

6.5 Conclusion

In this bachelor's thesis, a proof of concept of a view-based navigation and pathfinding algorithm relying on a topological map created from salient points of interest and weak metrical data, was presented. The algorithm has been proven to work, albeit with some severe limitations; due to the methods employed, such as only relying on rotational data, the algorithm had various failings such as being unable to distinct between close locations. Multiple solutions and suggestions for possible improvements have been made; additionally, the code itself can possibly be optimized and improved in all areas. Many improvements could simply not be implemented due to the limited time constraints set for work on a bachelor's thesis. The next step, as well as a possible solution to many problems, would be the implementation and testing of the algorithm on an actual robot – the author remains hopeful that the findings of this work will be useful in future endeavors regarding robot navigation.

Acknowledgments

I sincerely thank Gerrit Ecke, Dr. Hansjürgen Dahmen, Prof. Dr. Hanspeter A. Mallot, the participants of the Bsc-seminar, and the Eberhard Karls Universität Tübingen's department Institut Neurobiologie – Kognitive Neurowissenschaften for making this work possible.

References

[1] Möller, R., Krzykawski, M., Gerstmayr-Hillen, L., Horst, M., Fleer, D., & De Jong, J. (2013). Cleaning robot navigation using panoramic views and particle clouds as landmarks. *Robotics and Autonomous Systems*, *61*(12), 1415-1439.

[2] Kosaka, A., & Kak, A. C. (1992). Fast vision-guided mobile robot navigation using model-based reasoning and prediction of uncertainties. *CVGIP: Image understanding*, *56*(3), 271-329.

[3] Hübner, W., & Mallot, H. A. (2007). Metric embedding of view-graphs. *Autonomous Robots*, *23*(3), 183-196.

[4] Durrant-Whyte, H., & Bailey, T. (2006). Simultaneous localization and mapping: part I. *IEEE robotics & automation magazine*, *13*(2), 99-110.

[5] Franz, M. O., Schölkopf, B., Mallot, H. A., & Bülthoff, H. H. (1998). Learning view graphs for robot navigation. *Autonomous agents*, 111-125. Springer US.

[6] Cartwright, B. A., & Collett, T. S. (1987). Landmark maps for honeybees. *Biological cybernetics*, *57*(1-2), 85-93.

[7] Brown, M., & Lowe, D. G. (2002, September). Invariant Features from Interest Point Groups. *BMVC* (No. s 1).

[8] Baddeley, B., Graham, P., Philippides, A., & Husbands, P. (2011). Holistic visual encoding of ant-like routes: Navigation without waypoints. *Adaptive Behavior*, *19*(1), 3-15.

[9] Davison, A. J., Reid, I. D., Molton, N. D., & Stasse, O. (2007). MonoSLAM: Real-time single camera SLAM. *IEEE transactions on pattern analysis and machine intelligence*, *29*(6), 1052-1067.

[10] Klein, G., & Murray, D. (2007, November). Parallel tracking and mapping for small AR workspaces. *Mixed and Augmented Reality, 2007*, *ISMAR* 2007, 225-234. IEEE.

[11] Mur-Artal, R., Montiel, J. M. M., & Tardós, J. D. (2015). Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, *31*(5), 1147-1163.

[12] Gálvez-López, D., & Tardos, J. D. (2012). Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics* *28*(5), 1188-1197.

[13] Shah, S., & Aggarwal, J. K. (1997). Mobile robot navigation and scene modeling using stereo fish-eye lens system. *Machine Vision and Applications*, *10*(4), 159-173.

[14] Se, S., Lowe, D., & Little, J. (2001). Vision-based mobile robot localization and mapping using scale-invariant features. *Robotics and Automation, 2001. Proceedings 2001 ICRA*, Vol. 2, 2051-2058. IEEE.

[15] Williams, B., Cummins, M., Neira, J., Newman, P., Reid, I., & Tardós, J. (2009). A comparison of loop closing techniques in monocular SLAM. *Robotics and Autonomous Systems*, *57*(12), 1188-1197.

[16] Nieto, J., Guivant, J., Nebot, E., & Thrun, S. (2003, September). Real time data association for FastSLAM. *Robotics and Automation, 2003. Proceedings*, Vol. 1, 412-418. IEEE.

[17] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100-107.

[18] Thrun, S., & Bücken, A. (1996). *Learning Maps for Indoor Mobile Robot Navigation* (No. CMU-CS-96-121). Carnegie-Mellon University Pittsburgh, PA, Dept. of Computer Science.

[19] Moravec, H. P. (1990). The Stanford cart and the CMU rover. *Autonomous Robot Vehicles,* 407-419. Springer New York.

[20] Murray, D., & Jennings, C. (1997, April). Stereo vision based mapping and navigation for mobile robots. *Robotics and Automation, 1997. Proceedings*, Vol. 2, 1694-1699. IEEE.

[21] Little, J. J., Lu, J., & Murray, D. R. (1998, October). Selecting stable image features for robot localization using stereo. *Intelligent Robots and Systems, 1998. Proceedings*, Vol. 2, 1072-1077. IEEE.

[22] Bonin-Font, F., Ortiz, A., & Oliver, G. (2008). Visual navigation for mobile robots: A survey. *Journal of intelligent and robotic systems*, *53*(3), 263-296.

[23] Bouguet, J. Y., & Perona, P. (1995, June). Visual navigation using a single camera. *Computer Vision, 1995. Proceedings*, 645-652. IEEE.

[24] Lebegue, X., & Aggarwal, J. K. (1992, May). Extraction and interpretation of semantically significant line segments for a mobile robot. *Robotics and Automation, 1992. Proceedings*, 1778-1785. IEEE.

[25] Cheeseman, P., Smith, R., & Self, M. (1987). A stochastic map for uncertain spatial relationships. *4th International Symposium on Robotic Research*, 467-474.

[26] Bosse, M., Newman, P., Leonard, J., Soika, M., Feiten, W., & Teller, S. (2003, September). An Atlas framework for scalable mapping. *Robotics and Automation, 2003. Proceedings*, Vol. 2, 1899-1906. IEEE.

[27] Gutmann, J. S., & Konolige, K. (1999). Incremental mapping of large cyclic environments. *Computational Intelligence in Robotics and Automation, 1999. CIRA'99. Proceedings*, 318-325. IEEE.

[28] Kak, A. C., Andress, K. M., Lopez-Abadia, C., Carroll, M. S., & Lewis, J. R. (2013). Hierarchical evidence accumulation in the PSEIKI system and experiments in model-driven mobile robot navigation. *arXiv preprint arXiv:1304.1513*.

[29] Schölkopf, B., & Mallot, H. A. (1995). View-based cognitive mapping and path planning. *Adaptive Behavior*, *3*(3), 311-348.

[30] Bay, H., Tuytelaars, T., & Van Gool, L. (2006, May). Surf: Speeded up robust features. *European conference on computer vision*, 404-417. Springer Berlin Heidelberg.

[31] Harris, C., & Stephens, M. (1988, August). A combined corner and edge detector. *Alvey vision conference*, Vol. 15, 50.

[32] Lowe, D. G. (1999). Object recognition from local scale-invariant features. *Computer vision, 1999*, Vol. 2, 1150-1157. IEEE.

[33] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*(1), 269-271.

[34] Thrun, S. (1998). Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, *99*(1), 21-71.

[35] Burgard, W., Cremers, A. B., Fox, D., Hähnel, D., Lakemeyer, G., Schulz, D., Steiner, W., & Thrun, S. (1998, July). The interactive museum tour-guide robot. *AAAI-98 Proceedings*, 11-18.

[36] Thrun, S., Bennewitz, M., Burgard, W., Cremers, A. B., Dellaert, F., Fox, D., Hähnel, D., Rosenberg, C., Roy, N., Schulte, J., & Schulz, D. (1999). MINERVA: A second-generation museum tour-guide robot. *Robotics and automation, 1999. Proceedings*, Vol. 3. IEEE.

Code library references

[37] http://opencv.org/
Documentation and download of the OpenCV (open source computer vision) library, including an implementation of SURF. The version used in this work is *OpenCV 3.1 for Windows*.

[38] http://lemon.cs.elte.hu/trac/lemon
Documentation and download of the LEMON (Library for Efficient Modeling and Optimization in Networks) graph management library. The version used in this work is *LEMON 1.3.1*.

[39] http://www.boost.org/doc/libs/1_61_0/doc/html/boost_asio.html
Documentation and download of the Boost.Asio (Asynchronous Input and Output) multithreading and network data transfer library. The version used in this work is *Boost.Asio 1.61.0.*

Image References

Figures 2, 3, 4 and 5 originate from:

[30] Bay, H., Tuytelaars, T., & Van Gool, L. (2006, May). Surf: Speeded up robust features. *European conference on computer vision*, 404-417. Springer Berlin Heidelberg.

All other images were specifically created for this bachelor's thesis.