Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Cybernetics

Bachelor Thesis

Probabilistic Approach to Landmark Management in Visual Odometry

Pavel Potoček

Thesis Advisor:
doc. Ing. Tomáš Svoboda., PhD

Praha, 2014

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne …………………   ………………………
Podpis autora práce

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student:	Pavel P o t o č e k
Study programme:	Cybernetics and Robotics
Specialisation:	Robotics
Title of Bachelor Project:	Probabilistic Approach to Landmark Management in Visual Odometry

Guidelines:

Propose a robust point-based algorithm for visual odometry on a mobile outdoor robot. Focus on the problem of tracking landmarks. The new algoritm should mitigate the scale drift in long sequences. Implement the solution as a node in ROS (Robot Operating System) and integrate it into the system runing on the mobile robot. Validate on real data from several scenes.

Bibliography/Sources:

[1] John Mullane, Ba-Ngu Vo, M. D. A. & Vo, B.-T. (2011), 'A Random-Finite-Set Approach to Bayesian SLAM', IEEE Transactions on Robotics 27.

[2] Kundu, A.; Krishna, K. M. & Jawahar, C. V. (2011), Realtime Multibody Visual SLAM with a Smoothly Moving Monocular Camera, in 'Computer Vision, 2011. ICCV 2011. IEEE International Conference on'.

[3] Montemerlo, M.; Thrun, S.; Koller, D. & Wegbreit, B. (2002), FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem, in 'Proceedings of the AAAI National Conference on Artificial Intelligence'.

[4] Divis, Jiri. (2013), 'Visual Odometry from Omnidirectional Camera', Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics.

Bachelor Project Supervisor: doc.Ing. Tomáš Svoboda, Ph.D.

Valid until: the end of the summer semester of academic year 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Pavel P o t o č e k

Studijní program: Kybernetika a robotika (bakalářský)

Obor: Robotika

Název tématu: Pravděpodobnostní přístup pro správu klíčových bodů ve vizuální odometrii

Pokyny pro vypracování:

Navrhněte robustní algoritmus pro výpočet pozice robota z visuálních korespondencí. Soustřeďte se na problém udržení klíčových bodů. Nový algoritmus by měl především zlepšit přesnost ve výpočtu měřítka a snížit drift u dlouhých sekvencí. Algoritmus implementujte jako uzel (node) v prostředí Robot Operating System (ROS) a začleňte ho do celkového systému běžícího na mobilním robotu. Otestujte na reálných datech z různých scén.

Seznam odborné literatury:

[1] John Mullane, Ba-Ngu Vo, M. D. A. & Vo, B.-T. (2011), 'A Random-Finite-Set Approach to Bayesian SLAM', IEEE Transactions on Robotics 27.
[2] Kundu, A.; Krishna, K. M. & Jawahar, C. V. (2011), Realtime Multibody Visual SLAM with a Smoothly Moving Monocular Camera, in 'Computer Vision, 2011. ICCV 2011. IEEE International Conference on'.
[3] Montemerlo, M.; Thrun, S.; Koller, D. & Wegbreit, B. (2002), FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem, in 'Proceedings of the AAAI National Conference on Artificial Intelligence'.
[4] Divis, Jiri. (2013), 'Visual Odometry from Omnidirectional Camera', Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics.

Vedoucí bakalářské práce: doc. Ing. Tomáš Svoboda, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2014

Many thanks to Tomáš Svoboda, the advisor of this thesis, for frequent consultations, great advice and remarkable helpfulness. I would also like to thank Hana Potočková for text correction and Jan Vakula for his assistance with the setup of the development environment.

Děkuji vedoucímu své práce doc. Ing. Tomáši Svobodovi, PhD. za časté konzultace, dobré rady a vstřícnost. Dále děkuji Bc. Haně Potočkové za jazykovou korekturu a Bc. Janu Vakulovi za pomoc s nastavením vývojového prostředí.

The work was supported by EC project FP7-ICT-609763 TRADR and by the CTU project SGS13/142/OHK3/2T/13. Any opinions expressed in this paper do not necessarily reflect the views of the European Community. The Community is not liable for any use that may be made of the information contained herein

Abstract

We implemented a FastSLAM 2.0-based algorithm for visual trajectory estimation and applied it to the NIFTi robot. We performed multiple experiments to validate the algorithm and measured its performance in various settings. We showed that our algorithm outperforms the existing solution in many of them. We proposed future changes to the algorithm that have a potential to further increase the performance.

Anotace

V této práci jsme vyvinuli nový algoritmus pro vizuální odhad trajektorie NIFTi robota, který vychází z FastSLAM 2.0 algoritmu. Experimentálně jsme algoritmus ověřili a změřili jeho přesnost v různých podmínkách. Ukázali jsme, že ve většině z nich dokáže trajektorii odhadnout lépe než stávající algoritmus. Identifikovali jsme příčiny problémů a navrhli jsme změny, které by vedly k jeho dalšímu zlepšení.

Contents

Chapter 1

Introduction

NIFTi is a tracked robot focusing on urban search and rescue (see figure 1.1). Human-robot teams work together to explore a disaster area, to assess the situation or to locate victims[1]. As such, it is essential for the robot to know its own position. Visual odometry is one of the ways to acquire it and the aim of this paper is to improve upon an existing visual odometry algorithm.

Current implementation[8] has got some major shortcomings, such as insufficient landmark lifetimes and a high drift in scale. Especially the drift in scale forbids the usage of the translational part of the estimate; only rotation is used. This work aims to provide a better-performing algorithm, while paying a special attention to the drift elimination and scale stability.

First, we need to define several terms that will be used throughout this paper:

Visual odometry (VO) is the process of estimating the egomotion of an agent (e.g. vehicle, human, and robot) using only the input of a single or multiple cameras attached to it[21].

Visual SLAM or vSLAM (Simultaneous Localization and Mapping) is similar to the visual odometry, but VO is only concerned in the local consistency of the trajectory, whereas SLAM with the global consistency[21].

landmark is a distinct feature of surroundings that can be captured by a camera. Its appearance and 3D position can be determined; the positions of landmarks are computed as a part of the vSLAM routine.

feature is a projection of a landmark into an image plane. It is thus determined by its 2D position and appearance. Information about landmarks is obtained via measurements of corresponding features.

feature association is a process of associating a single landmark with features in different images. Features, corresponding to a single landmark, are detected across multiple images, forming a sequence (vector) of 2D positions. Members of the sequence are called **matching features.**

There are two main approaches to visual odometry. Appearance-based methods use intensity information of all the pixels in input images. Feature-based methods

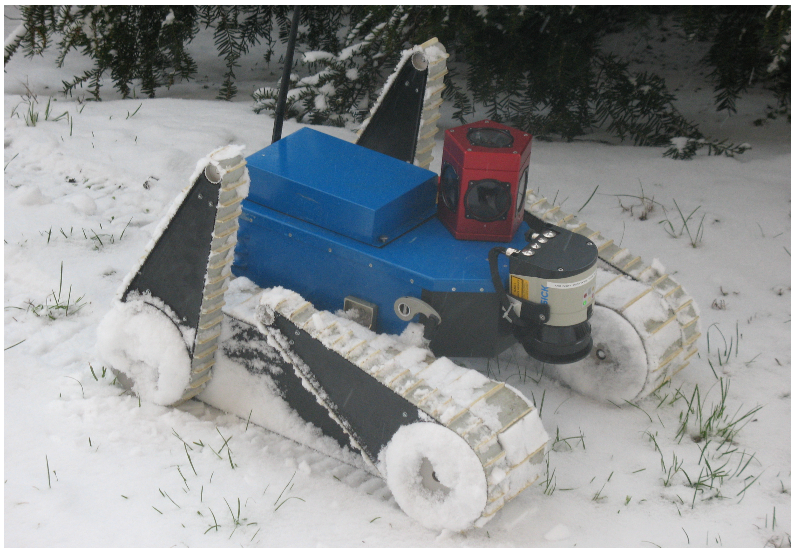


Figure 1.1: NIFTi robot. The omnidirectional camera used in this paper is mounted on top of it (painted in red).

only use repeatable features extracted from the images. Feature-based methods are generally more accurate and less computationally expensive than appearance-based methods[21]. Therefore, most VO implementations are feature-based and this paper details only feature-based approaches.

In following sections, we will first describe a pairwise approach to visual odometry and then work our way up to the best currently known vSLAM algorithms.

1.1 Pairwise approach to visual odometry

Imagine we have two images of a scene, taken from different positions. We can detect features in them and associate them with landmarks, forming a set of associated feature-pairs. Relative poses of the two calibrated cameras can be inferred from five such feature-pairs by a 5-point algorithm (Figure 1.2)[18]. This ability to compute a pair of cameras can be extended to a full VO solution by using the 5-point algorithm repeatedly and fusing the camera pair transformations to compute a full camera trajectory estimate. There are several techniques for such an extension. One of them is visualized in Figure 1.3.

The 5-point algorithm does not deal with outliers. However, outliers are present due to the feature association and feature extraction errors. In this environment containing outliers, we have to use a robust technique; the most widely used one is RANSAC (or its modifications), described in [10].

To make the estimate more accurate, an optimization technique known as *sliding bundle adjustment* can be employed. It is a non-linear optimization technique that aims to improve positions of cameras and landmarks to give a more consistent estimate. It can be described as a following function:

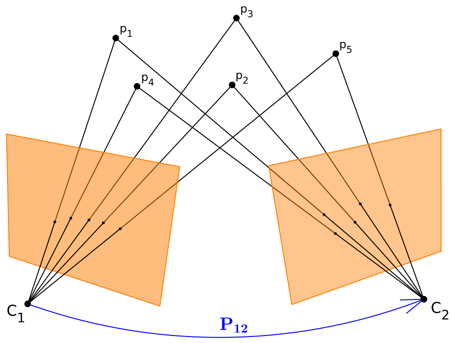


Figure 1.2: Visualization of the 5-point algorithm. This algorithm infers $\mathbf{P}_{12}$ (up to a translation scaling coefficient) from projections of five landmarks.

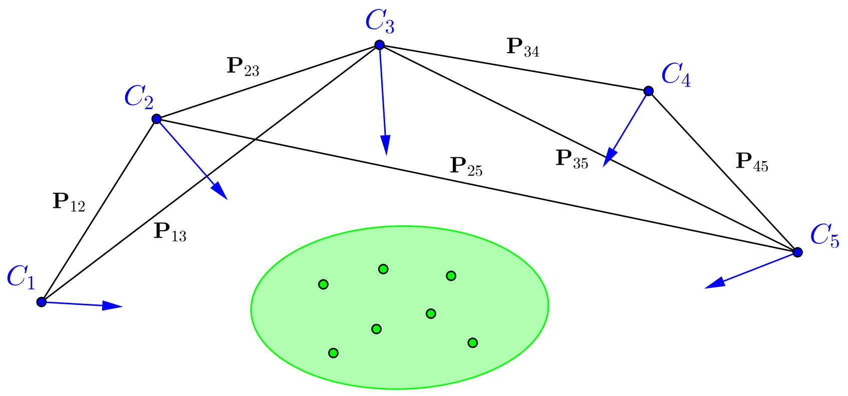


Figure 1.3: Full visual odometry solution based on the 5-point algorithm. With enough constraints, we can reconstruct the poses of all nodes, up to a single global scale. Scene, consisting of landmarks, is shown in green.

$$\begin{matrix} \textit{sliding} \\ \textit{bundle} \\ \textit{adjustment} \end{matrix} :: \begin{pmatrix} \text{camera poses} \\ \text{landmark positions} \\ \text{landmark projections} \end{pmatrix} \to \begin{pmatrix} \text{improved camera poses} \\ \text{improved landmark positions} \end{pmatrix}.$$

The term *sliding* indicates that the optimization is only over a fixed number of most recent camera poses. Older camera poses do not influence further computation in any way and can be safely thrown away.

This approach to visual odometry has been successfully used in a range of applications[16, 24, 22]. But there are some inherent weaknesses:

1. It has no notion of landmark uncertainty. Some landmarks could be estimated with better precision, than others (depending on observation parallax and the number of observations). Those landmarks should have a bigger weight in the process of camera pose estimation.

2. It is very common that landmarks are not well-constrained in their distance from the observer but are well-constrained in the direction. The 5-point algorithm is unable to express this property.

3. It does not fail gracefully. When the RANSAC scheme does not find an all-inlier sample, the process usually returns nonsense. Those cases can be detected and possibly the frame could be thrown away, but then their contribution to estimation is zero, even if there *is* some information that could help with the estimate.

4. The way of correcting existing landmarks *via* bundle adjustment is not sufficient. The algorithm works only over the last n camera poses. This seems like quite an arbitrary choice.

Better solution to visual odometry would have to incorporate landmark and camera uncertainties in some way or another. In the following chapters, we would be discussing approaches that do just that. And as the extra gains in accuracy (especially in a monocular setup) are significant, those approaches are more likely to be used to get a globally-consistent trajectory estimate. We will, therefore, refer to them as SLAM techniques.

1.2 EKF-SLAM

EKF-SLAM was introduced in 1990[23] and remained a *de-facto* standard in the 90s. In EKF-SLAM, both camera and landmark poses are jointly propagated in time via Extended Kalman Filter (EKF). This gives us a better approximation to an underlying process. In VO, we computed the mean values of camera and landmark poses. Now, we assume that the landmark and camera poses are normally distributed and we compute also their uncertainties.

Assume there is a fixed map $\mathcal{M}$ consisting of N landmarks l_i.

$$\mathcal{M} = \{l_1, l_2, \ldots, l_N\}.$$

Those landmarks can be observed by a sensor; the result of the observation is a feature z_j. The features are uniquely associated with the landmarks, and the association is a known function a, mapping feature indices to landmark indices: $a(j) = i$. Moreover, we have some initial estimate of all landmarks in the map. The robot pose, denoted as x_t (in time t), is dependent on the previous pose x_{t-1} and the control input u_t:

$$x_t = f_t(x_{t-1}, u_t),$$

where f_t is the robot's kinematics. All landmarks l_i, together with the robot's pose x_t, will be estimated by an EKF. We introduce a state vector $S_t = (x_t, l_1, l_2, \ldots, l_N)$ and propagate its estimate, approximated by a mean value and a covariance matrix, in time. Each observation z_j is a constraint between the robot's pose x_t and a landmark $l_{a(j)}$, improving the estimates of both of them. With each time step, the robot's pose is updated *via* the function f_t, and landmark estimates remain constant (the map is static).

1.2.1 Landmark initialization

We assumed that all landmarks have an initial estimate. However, this is generally not the case. We typically do not have any prior information about the landmarks in the map. The process of acquiring initial estimate from first measurements of a landmark is called *landmark initialization.* Various approaches to landmark initialization are described in this chapter.

A monocular camera is a projective sensor which measures the bearing of landmarks. To infer the depth of a landmark, the camera must observe it repeatedly as it translates through the scene, each time capturing a ray from a landmark to its optic center. The angle between the captured rays is a *parallax.* It allows landmark's depth to be estimated. It is clear that the landmark can not be represented by a normal distribution in Euclidean coordinates after a single observation: the depth is not constrained. figure 1.4 shows that even after two or more observations, the landmark uncertainty still might not be normally distributed. As this is is contrary to the requirements of EKF-SLAM, several approaches have been developed to tackle this problem[3].

Delayed initialization is an approach that divides observed landmarks into two categories: partially and fully observed. Each of these categories is propagated in time independently, but only fully observed landmarks contribute to the robot's pose estimate. Fully observed landmarks are the landmarks that can be represented by a normal distribution in Euclidean space (they have acquired a sufficient parallax), and are incorporated into a normal EKF routine. Partially observed landmarks can be represented, e.g. by multiple gaussian hypotheses; the unlikely hypotheses are pruned and when only one hypothesis remains, the landmark becomes fully initialized[3].

Inverse-depth parametrization is an approach that uses non-euclidean representation of landmarks throughout the EKF filter. It is based on an observation that landmarks in a monocular setup can be linearly represented by a following 6-dimensional vector P:

$$P = (x, y, z, \theta, \phi, \rho),$$

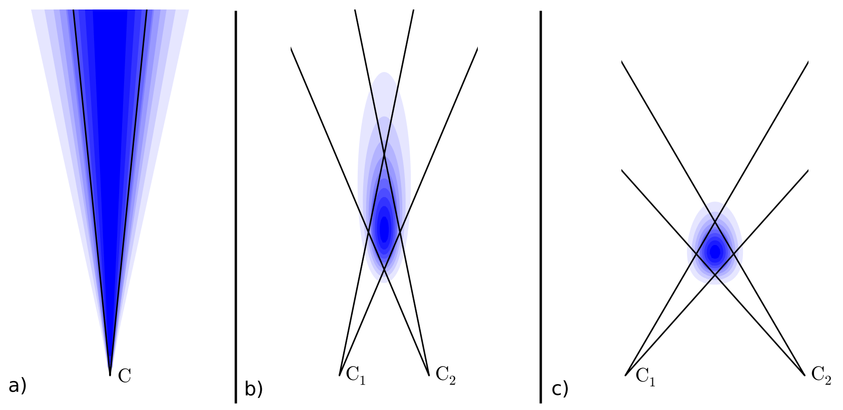


Figure 1.4: Landmark position density after observation by one (a) or two (b,c) cameras. The cameras observe the bearing of a landmark with gaussian uncertainty (black lines are at 1-sigma). The resulting pdf is shown in white-blue gradient. Figure a) shows that a single-camera observation can not be modeled by a normal distribution in Euclidean space. For multiple cameras at a low parallax (b), the distribution is still not gaussian (it has got a heavy tail). With increasing parallax, the approximation gets more accurate (c).

where x, y, z are the coordinates of the camera the landmark was first observed with; θ, ϕ are azimuth and elevation angles defining landmark direction from that camera, and ρ is inversely proportional to the landmark distance from the camera. With this parametrization, it is possible to represent landmarks at infinity, landmarks at a finite distance, and even landmarks, initialized from only one observation (Figure 1.4a)[5]. Even the landmarks that do not exhibit any parallax (yet) can be used to help constrain the bearing of the robot. This is in contrast with delayed parametrization, where the landmark estimate must collapse to a finite representation before being used to actually help the estimate.

1.3 FastSLAM

The key limitation of EKF-SLAM is the computational complexity. EKF propagates the mean value and the covariance matrix of the whole world-state $S_t = (x_t, l_1, l_2, \ldots, l_N)$. In a 3D setup, the dimension of the state vector grows linearly with the number of landmarks: $S_t \in \mathbb{R}^{6+3N}$. The number of elements of the covariance matrix grows quadratically. All of them must be updated, even if a single landmark is observed[14]. The quadratic complexity limits the number of landmarks only to a few hundred, whereas a natural environment contains millions of features. FastSLAM is a technique introduced by Montemerlo *et al.* in 2002[14] that was designed to lower the computational complexity of EKF-SLAM while maintaining its accuracy.

1.3.1 FastSLAM in detail

The robot pose is a probabilistic function of the robot controls u_t and the previous pose x_{t-1}:

$$p\left(x_t|u_t, x_{t-1}\right),$$

where $p\left(\cdot\right)$is a probability density function (pdf). Sensor measurement probability can be written as

$$p\left(z_t|x_t, \mathcal{M}\right).$$

Landmark position estimates θ_k are dependent on the measurements $z_{a^{-1}(k)}$ associated with them and a robot trajectory $x_{1:t}$:

$$p\left(\theta_k \middle| x_{1:t}, z_{a^{-1}(k)}\right)$$

It can be observed that if the robot trajectory $x_{1:t} = x_1, x_2, \ldots, x_t$ were known, individual landmark estimates θ_k are independent. If an oracle provided us with a complete robot trajectory, the problem of determining landmark locations could be decoupled into N independent estimation problems[14].

Based on this observation, FastSLAM decomposes the SLAM problem into a robot localization problem and a collection of EKF-based landmark estimation problems, conditioned on a robot trajectory. The robot trajectory is represented by a *particle filter*. Each particle is a guess of robot's trajectory, and for each particle, an independent map estimate is computed. A representation of an accuracy of the map is computed for each particle. If the map associated with a particle is very accurate, the particle may multiply; if the map is not accurate, the particle may be terminated. This is known as a *resampling* process.

FastSLAM has been successful in eliminating the computational burden of EKF-SLAM. While EKF-SLAM's time complexity is $O\left(N^2\right)$, FastSLAM takes $O\left(M \log N\right)$ time to update state, where M is the number of particles. It is possible to achieve real-time performance with a large number (e. g. 100 000) of landmarks[2].

1.3.2 Robot pose estimation

Crucial for the performance and accuracy of FastSLAM algorithm is the quality of the robot pose estimation/guess. The guess is drawn from a distribution that can be obtained in a number of ways.

Constant velocity model forms a distribution that is dependent only on previous poses of the robot. It is suitable in setups, where there is no odometry information: for example a hand-held camera. This model is sensitive to sharp acceleration; the robot's path has to be smooth.

Robot kinematics can be used for a more accurate estimation, when available. The resulting distribution is dependent on the control input u_t and on the previous robot pose x_{t-1}.

FastSLAM 2.0 is an approach that improves the proposal distribution by an additional dependency on the most recent measurement z_t. This provides a more accurate estimate of the robot pose, particularly when the measurements are very accurate relative to robot kinematics (as is frequently the case in visual SLAM). FastSLAM 2.0 can, in some setups, outperform both FastSLAM and EKF-SLAM by a large margin. It has even been shown that it converges for a single particle in linear SLAM problems[15].

5-Point RANSAC algorithm is specific to a visual SLAM setup. It estimates vehicle pose distribution according to the most recent observations using a 5-point algorithm. This has an advantage of accuracy, and it is independent of robot kinematics. It is applicable even in setups with large camera resolution and low frame-rate. An implementation of this system, along with experimental results, is described in [13].

1.4 RFS-SLAM

In the previous approaches, we assumed for simplicity perfect feature associations. In practice, this is certainly not the case. The methods described can cope with some uncertainty. However, accuracy of feature association is still crucial to a successful SLAM solution. Random Finite Set SLAM (RFS-SLAM) does not require any prior data association step, and thus proves superior particularly in situations of high association ambiguity[17]. This approach propagates in time positions of landmarks, pose of the robot **and** the number of landmarks in a Bayes-optimal fashion. The number of landmarks and data association are both propagated in a unified update routine.

Another useful property of RFS-SLAM is that it does not require a separate routine for loop closure detection; it closes the loops by default, if the drift is sufficiently small.

The computational cost of this approach is higher, than that of FastSLAM; it is equal to $O(NZM)$, where N is the number of landmarks, Z is the number of observed features at a single frame, and M is the number of trajectory particles. The computational burden thus increases linearly with the map size.

Up to the author's best knowledge, RFS-SLAM has not been implemented in a visual setup yet. This may be caused by the fact that in visual SLAM, there is typically a fairly good data association estimate available from the feature appearance similarity. Thus, the classical FastSLAM algorithm performs well. The gain in increased robustness does not justify higher runtime complexity.

Chapter 2

Algorithm

We developed a SLAM solution based on the FastSLAM 2.0 algorithm introduced in [15] and briefly described in section 1.3. This chapter details the algorithm that we implemented, paying special attention to various details specific to this work. We sketched the algorithm in figure 2.1 and we will refer to this figure throughout this chapter.

Our algorithm has got the following inputs:

- Image sequence acquired by an omnidirectional camera, that was transformed into a single panoramic image (figure 2.1 a).
- Optionally, a prior estimate of the camera pose using robot kinematics (figure 2.1 b) including:
  - the rotation estimate from a gyroscopic sensor,
  - the travel distance estimate from the wheel odometry.

The algorithm outputs an estimate of the camera pose (figure 2.1 c).

Robot kinematics are an optional input. They can be used to improve a camera pose proposal distribution. The experiments showed that the algorithm performs well even with a simple constant pose proposal. However, the travel distance estimate will constrain the scale of the simulation, and the rotation estimate can help with capturing quick rotational movements of the camera.

2.1 Camera description

The NIFTi robot has a Ladybug 3 camera onboard. It is an omnidirectional camera system consisting of six 2-megapixel cameras. The images of the six cameras can be stitched by the Virtual Camera module into a single panorama. Virtual Camera offers many panoramic image representations: spherical, cylindrical, pinhole and others. The spherical model has been chosen because of one property: if the camera rotates around its vertical axis, the landmarks do not change their appearance. This is convenient, because the rotation of a robot around the vertical axis is a very common maneuver. The features are upon detection converted from pixel coordinates into a

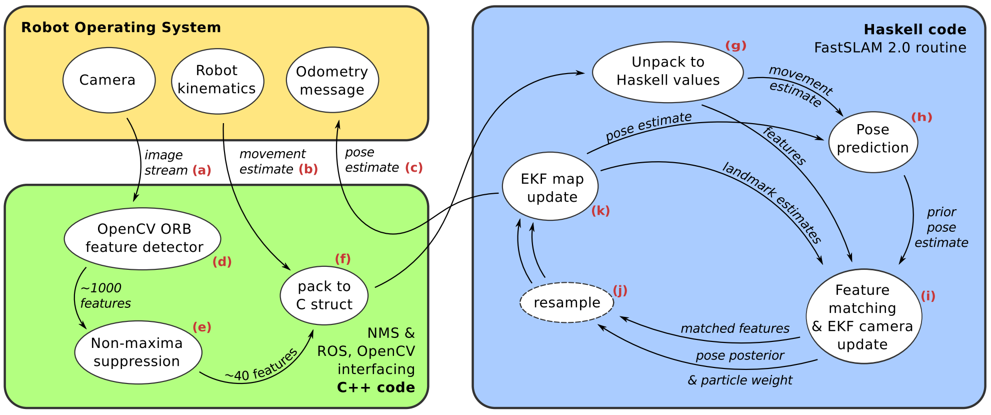


Figure 2.1: SLAM solution overview. The solution consists of two parts (green and blue), the orange part is not a part of the algorithm. Ellipses represent functions operating over data (represented by arrows). They do not exactly match the actual C++ or Haskell functions used. The brown letters are used in the text for reference.

(θ, φ) pair described in section 2.4 that is used throughout the computation. This way, the internal representation of landmarks is independent of the camera model chosen.

2.2 Feature detection and non-maxima suppression

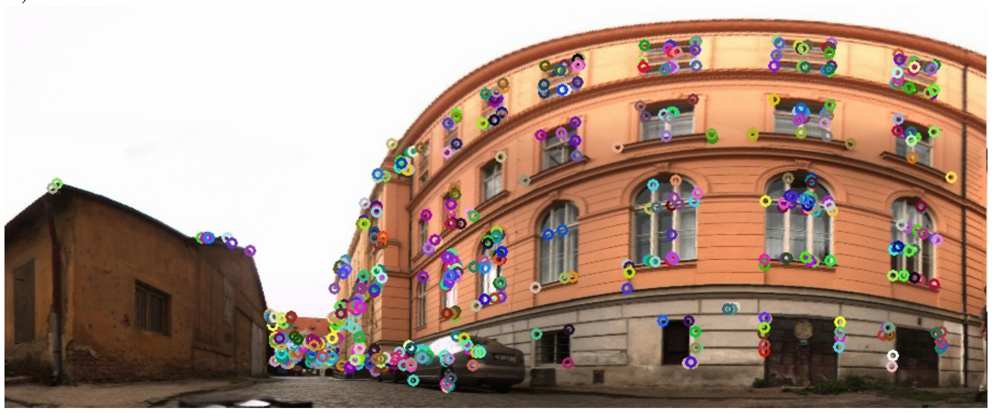
Features are detected from the panorama using the OpenCV ORB detector/descriptor[20] (figure 2.1 d). Pyramid layers are not used, because the non-maxima suppression routine would return different scales of features rather randomly, decreasing matching success rates dramatically.

ORB frequently produces features that are closely packed to groups around corners. This is not desirable, since it increases the computational burden while introducing little new information. Hence, a non-maxima suppression algorithm (NMS, figure 2.1 e) is implemented to curb the number of features and make them evenly distributed across the image. The effect of this routine is visualized in figure 2.2.

If any two features are too close together (euclidean distance in image coordinates is smaller than a threshold), the one with a weaker response is removed. The threshold distance is dependent on the feature response rather than being constant. It increases with a decrease in response of the stronger feature in a given pair. This way, in the areas of the image with worse features (measured by their response), fewer features are retained. The average response is thus increased, hopefully returning more persistent landmarks in general.

The routine is implemented naively and has got a complexity of $O\left(n^2\right)$ for n detected features. The parameters have been tuned to give an approximate average of 50 features per image out of the ~ 1000 features detected by the ORB detector.

a)



b)

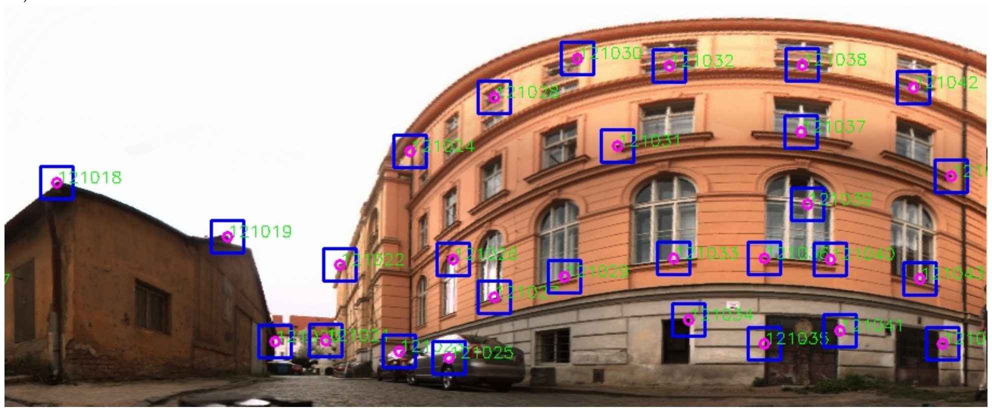


Figure 2.2: Non-maxima Suppression visualization: before (a), and after (b). In the image (a), features are visualized using small circles in the size of the detection area of the FAST detector used by ORB. In the image (b), purple circles visualize FAST detector areas. Blue squares show the BRIEF descriptor areas and green numbers are unique feature identifiers. Since pyramid layers are not used, all the descriptors are of the same size.

2.3 Recursive state estimation

In this section, the core state estimation routine is detailed. It is very similar to another FastSLAM 2.0-based routine described in [9].

At each time step, three stages of computation take place: prediction, observation and update. At the beginning of each frame, the pose of the camera is represented by a particle cloud. In the prediction step (figure 2.1 h), a linear probabilistic transition function is applied to each particle, yielding a gaussian mixture representation of the camera pose. Then, in the observation step, this predicted pose estimate is used in the feature matching routine(figure 2.1 i). Using matched features, the camera pose estimate is EKF-updated (figure 2.1 i). The updated pose (still a gaussian mixture) is sampled (figure 2.1 j) in order to get a new particle cloud. For every particle in the cloud, the map is EKF-updated (figure 2.1 k), again using the matched features. The output of this stage is used as an input to the next time step. The updates of the landmarks and the camera are separated from each other, so that the conditional independence of landmarks from each other is maintained[15].

2.3.1 Feature matching & EKF camera update

Features are associated to landmarks by an active search for the best matches to previously observed landmarks. Landmarks are projected into the (θ, φ) plane, using a predicted camera pose. As both the landmark estimate and pose estimate are gaussian, the result is also gaussian. Every feature within a 3σ region is checked for the descriptor similarity with the landmark. If just one feature descriptor matches the landmark descriptor better than a threshold, the match is successful. The landmark's descriptor is then updated to the most recent observation.

With each successful match, the particle weight is multiplied by the observation likelihood and the camera pose is updated *via* a standard EKF routine. This usually constrains the camera pose, giving an opportunity to constrain other landmark reprojection estimates. Unsuccessful matches lower the particle weight times a set amount.

This process introduces a dependency of the results on the ordering of landmarks[1]. To take an advantage of this, landmarks are sorted in a random order, independently for each particle. Because the search regions for the features get smaller with each matched feature, outlying matches are likely to happen only in the first few landmarks, when the search region is not constrained enough. Due to the random sort, this happens only in some of the particles that are be subsequently eliminated due to their lower weight, thus increasing robustness.

2.3.2 Landmark representation

For landmark representation, we chose Inverse Depth Parametrization[5]. Landmark estimates form a 6D vector and a 6×6 covariance matrix. The vector can be written as

$$(x, y, z, \theta, \phi, \sigma)\,,$$

[1]This is in contrast with [9]: they kept the update routine independent of the order in which landmarks are processed.

where (x, y, z) is a position of the camera that the landmark was first observed in, (θ, ϕ) is the azimuth-elevation bearing of the landmark relative to the (x, y, z) position, and σ is an inverse of the landmark distance.

As the landmarks are updated *after* sampling the camera pose, the (x, y, z) coordinates of the landmarks are known exactly. It follows that the covariance matrix is zero in the rows and columns corresponding to the first three coordinates:

$$cov \sim \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & \cdot \end{bmatrix}$$

Thus, only the relevant 3×3 submatrix could be stored and used in computations. This optimization is not used in order to keep the code as clear as possible; it does not form a performance bottleneck of the algorithm.

2.3.3 Landmark pruning

There are two reasons why the landmark pruning needs to be implemented:

1. As landmarks are used to actively search for corresponding features, the update takes time proportional to their number.
2. Some landmarks get occluded due to the robot movement and are no longer observable.

Were the landmarks not pruned, both the computational time and the number of outliers would increase. Only relevant landmarks must be used for the computation. Every landmark has a health value associated. If the landmark is observed, the health is increased:

$$\text{health} += 1 - \frac{1}{10}\,\text{health}.$$

If the landmark is not observed, the health is decreased by one. When the health gets below zero, the landmark is removed.

2.3.4 Stationary camera

Similarly to [9], we observed that the state estimation degrades if the camera is stationary for a prolonged period of time. This is solved by detecting that the camera is static and dropping the observation before the main estimation routine. The detection is simple: all features after the non-maxima suppression at the time t are searched for a nearest neighboring feature at the time $t - 1$. If the nearest neighbor is in exactly the same position, the feature is considered static. If more than $1/5$ of all features are static, the frame is dropped.

This routine drops only the frames where the camera pose (including rotation) is static. It should not be sensitive to rotation, but it turns out that this inaccuracy does not matter in reality.

2.4 Coordinate system conventions

Positions in 3D space are described using a right-handed euclidean coordinate system. The positive Z coordinate is pointing towards the vehicle's front side, the positive X coordinate is pointing to the right, and the positive Y coordinate is pointing downwards. This coordinate system choice is different from the one used in ROS. The conversions are implemented in the `tf.cpp` file.

Feature bearings are represented by an azimuth-elevation pair (θ, φ), where positive φ angles point upwards, positive θ angles point clockwise, and $(0, 0)$ is in the direction of positive Z axis.

Camera pose is represented in two ways as described in section 3.5.3. `ExactCamera` is a straightforward (x, y, z) representation and a rotation matrix. `GaussianCamera` uses a 6-vector to represent the camera pose, $(x, y, z, \alpha, \beta, \gamma)$. (x, y, z) is a camera position vector, and (α, β, γ) are Tait-Bryan angles, describing camera orientation in a minimal and linearizable way. Intrinsic $y - x' - z''$ axes are used; they do not exhibit singularity during any common robot movement.

Chapter 3

Implementation

This section details the implementation from programmer's point of view. The most important design specifics, data structures, functions and control flow characteristics are described.

3.1 Language choice

To enable a rapid pace of development and testing, a programming language for implementation was sought that matches following criteria:

1. It should be garbage collected, to avoid trivial bugs and memory leaks.
2. It should be terse, fast to type and fast to read.
3. It should provide a considerable level of type safety and compiler checks, to catch as many bugs as possible at compile time.
4. It should perform reasonably well, so that the code does not unconditionally need to be rewritten into C++ before deployment.
5. It should have mature libraries and ROS and OpenCV interfaces.

C++ fails 1. and 2., Java utterly fails in 2., Python and Matlab fail 3. and 4.

Haskell fails in 5. It does have mature libraries and does have ROS and OpenCV interfaces[7][25]. But the ROS interface in Haskell is nowhere as well documented as the C++ counterpart, and the OpenCV interface does not fully support OpenCV 2.4 yet.

Since no single language matches the criteria, we decided to use two different languages: C++ and Haskell. We use the C++ to interface with ROS and OpenCV, and for the core of the algorithm, we use Haskell. The extra burden of implementing C++/Haskell interface is justified, because of the advantages Haskell provides:

- It is a very high-level language. The programmer can focus on the algorithm instead of low-level details.

- It is similar to Python in code terseness, having a large edge over C++.
- Since all parameters to functions must be explicitly specified, the data flow is explicit and easily visible. It also encourages writing decoupled modules with small and well-defined interfaces.
- In contrast to Python, it has got sufficient performance.
- It has got a very strong type system, which is able to catch most bugs at compile-time. This is crucial to successfully developing a solution to such a large problem as visual SLAM. I observed that most bugs that pass compilation are just a misplaced minus signs.
- It has got a great program property testing library[6].
- An extremely simple-to-use 3D graphics library is available for visualization[11].

3.2 Implementation overview

The algorithm communicates with the sensoric equipment and other code on the NIFTi robot through the Robot Operating System (ROS). This system manages a set of loosely coupled processes, called nodes, and their interactions. The result of this paper is a single node, named `fastSLAM_2`. The algorithm was tested with ROS Fuerte Turtle on Ubuntu 12.04 (32 bit). For building instructions, see the file `INSTALL` in the root directory of the node.

Haskell is the main language for implementation and contains the entry point of the program; parts written in C++ are linked as a shared library. The C++ part of the code does all the interfacing with ROS and OpenCV, and communicates the data to the second part, written in Haskell. The Haskell code does the bulk of the mathematics involved, but to keep the inter-language interface as small as possible, some mathematics have been implemented in C++ instead (namely, non-maxima suppression and some coordinate system conversions).

It is possible to serialize the data on the inter-language interface, so the two parts can be executed separately, thus improving the speed of testing and development of both of them.

3.3 Execution and threads

The program is started by executing the `main` function defined in the `RosMain.hs` file. It immediately calls a C++ function `main_c`, which spawns a thread that initializes ROS and enters the main ROS message loop. This loop is active throughout the whole program runtime, collecting ROS messages and saving the results into global variables that are used by other parts of the program.

After the initialization is finished, control is returned to the `main` function, which proceeds to start the main program loop. The loop consists of three steps:

1. Call to `getFrame` to get the most recent observations from ROS. This in turn calls a C function `extract_keypoints()`.

2. Call to `filterUpdate` to update the FastSLAM 2.0 filter with new observations. This is written entirely in Haskell, and does not use the FFI.
3. Call to `publishTf` to publish the updated pose estimate as a ROS message. The C function `publish_tf` is called to perform the ROS interfacing.

All three calls in the main loop are synchronous.

The `getFrame` function returns the most recent observation available. The observations (images from the camera) are produced by ROS constantly at a rate of $1-6\,\text{Hz}$. This rate is not tied to the update rate of our algorithm, so a producer-consumer problem arises. As the update rate of our algorithm is rather constant and low delay is desired, a single-item queue is used. `getFrame` returns four values:

- the frame ID used for debugging,
- time delta between two most recent frames,
- extracted features after non-maxima suppression,
- pose transition estimate from robot's odometry. In the case this is not used, it can be set to identity.

The time delta is used to estimate the transition covariance, which together with transition estimate forms a probabilistic transition function.

The `filterUpdate` routine is a Markov process: its input is its output from the previous frame, together with the probabilistic transition function and the set of extracted features. It returns an updated map estimate and an updated camera pose estimate. Both updated map and camera estimates are used as an input to `filterUpdate` during the next time step. It is described in more detail in section 2.3.

After the update, the camera pose estimate is published as a ROS message by the `publishTf` function.

3.4 Haskell foreign function interface

Haskell foreign function interface (FFI) is used to call Haskell functions from C and C functions from Haskell. Only the latter is used in our program. That is, C functions are called by Haskell functions. Since there is no C++ interface, all the data and functions that are used from Haskell are defined in the C++ code in a C-compatible format. The interface is visualized in figure 3.1.

Haskell implementation of the FFI requires manual structure unpacking: determining the field offsets and sizes by hand. This is very error-prone, so the c2hs library[4] is used to generate the interfacing code automatically. This provides us with a nice property: when the data structures are changed, the alignment is adjusted by c2hs automatically.

Note that the entry point of the program is in the Haskell part of the code. As the command-line arguments are essential for the ROS initialization routine, they must be passed to the C++ code. They are joined into a single string (`char *args`) to keep the argument marshaling simple.

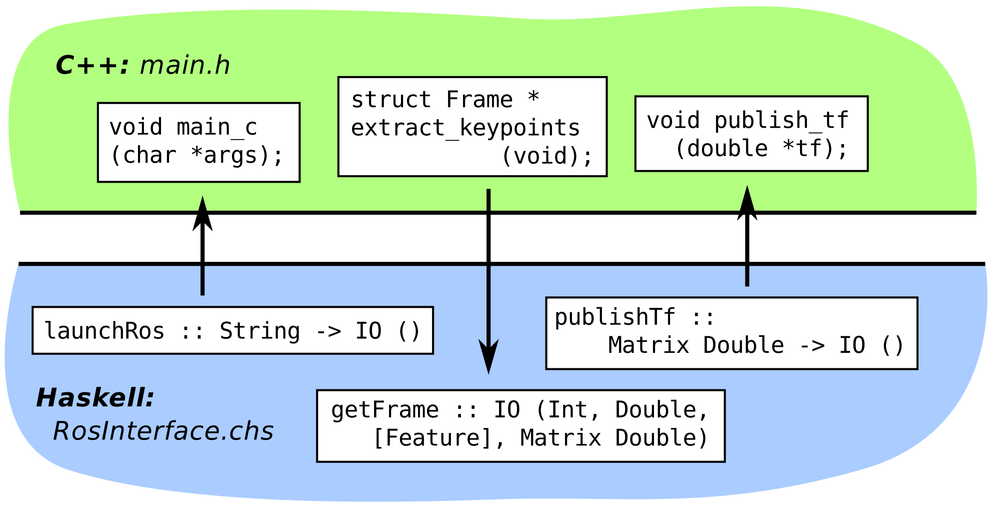

C++: main.h
void main_c (char *args);
struct Frame * extract_keypoints (void);
void publish_tf (double *tf);
launchRos :: String -> IO ()
publishTf :: Matrix Double -> IO ()
Haskell: RosInterface.chs
getFrame :: IO (Int, Double, [Feature], Matrix Double)

Figure 3.1: Haskell–C++ communication. All three C functions called from Haskell are visualized (top three squares), along with the data flow (arrows). On the Haskell side, the functions are converted into Haskell functions. Note, the analogous type signatures of functions connected by arrows. The function `getFrame` returns unpacked contents of the `struct Frame`, corresponding to the C fields `(id, dt, kps, tf)`.

3.5 Data structures

In this section, various data structures used throughout the program are described.

3.5.1 Features

Because features must be processed in both C++ and Haskell parts, they are stored in three distinct formats.

Features are produced in the `detect_keypoints` functions. This function returns features in a format native to OpenCV. The descriptors are stored in a `cv::Mat` matrix, one descriptor per row. All the other valuable information, such as the position in the image plane and response strength are stored in a `std::vector<cv::KeyPoint>` array.

Before the features are saved into a global variable and then sent over FFI to Haskell, they are converted by a `keypoints_to_structs` function into a C-compatible array `struct Keypoint*`. By this point, the image-space feature positions are converted to an angular azimuth-elevation pair described in section 2.4. `struct Keypoint` is defined in `main.h` as follows:

```
typedef struct Keypoint {
    int id;                  // A globally-unique id
    double px, py;           // azimuth-elevation pair
    int octave;              // image pyramid level
    float response;          // response strength
    int descriptor_size;
    char *descriptor;        // a BRIEF descriptor
} keypoint_t;
```

In this representation, the features are passed to the Haskell code. Upon receiving, they are immediately converted by a `getKeypoint` function into a Haskell-native format defined in `Landmark.hs`:

```
data Feature = Feature
    { fid :: FID                  -- unique ID
    , flm :: Maybe Landmark       -- optional association
    , fpos :: (Double, Double)    -- azimuth-elevation pair
    , response :: Double
    , descriptor :: Descriptor
    }
```

Code and comments have been shortened for clarity. This structure is interlinked with other data structures used throughout the Haskell code. Note that the previous representation has some fields that are not visible in the Haskell structure: `descriptor_size` is incorporated in the `Descriptor` type, and `octave` is not used.

3.5.2 Landmarks

Landmarks are stored in a structure `Landmark`, defined in `Landmark.hs`:

```
data Landmark = Landmark
    { lid :: LID                    -- unique ID
    , lmu :: Vector Double          -- 6D mean vector
    , lcov :: Matrix Double         -- 6D covariance matrix
    , ldescriptor :: Descriptor     -- descriptor of the last feature
    , lhealth :: Double             -- landmark health
    }
```

Comments have been added for clarity. Landmarks are organized in maps, where a map is a set of landmarks:

```
type Map = S.Set Landmark
```

There is one map instance per particle. The set is implemented as a size balanced binary tree[26] and supports a $O(\log n)$ insert operation. Since the maps are not copied during the map update (only the references are), and the Glasgow Haskell Compiler uses copy-on-write memory model, we get runtime complexity equivalent to the tree-based optimization of the FastSLAM algorithm[14] with no extra work.

3.5.3 Camera pose

Camera pose is passed between Haskell and C++, hence it must use multiple representations. The first estimate of the camera pose is acquired from ROS as a `tf::StampedTransform`, but it is immediately converted into an OpenGL-compatible 4×4 homogenous matrix of type `double*` (a column-major array). This array is passed to the Haskell code and converted into Haskell-native representation.

In the Haskell code, two camera representations are used (defined in `Camera.hs`), and they are frequently switched. The first one, `ExactCamera`, represents an exactly-known camera:

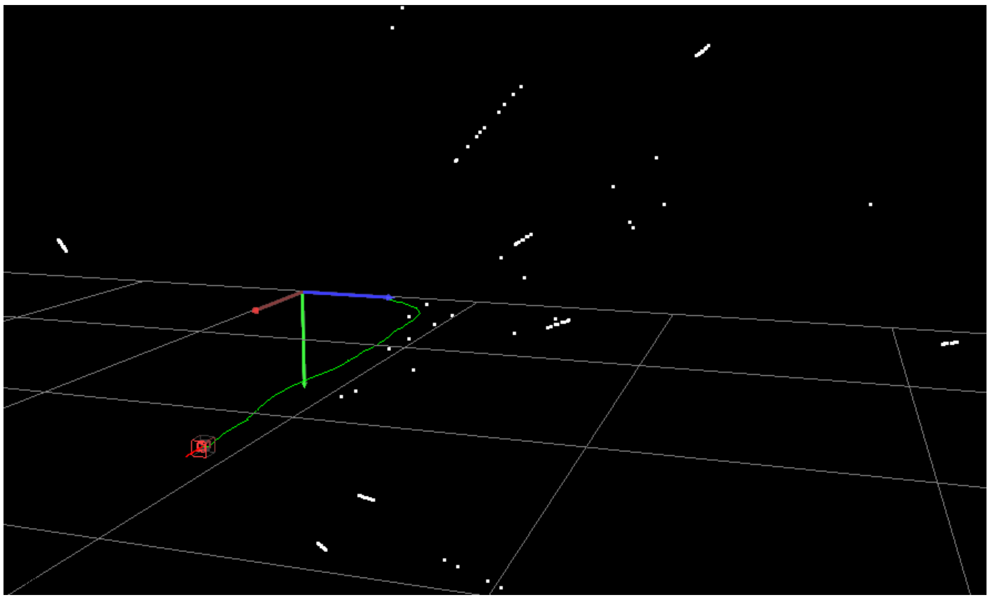


Figure 3.2: Haskell SLAM visualization. In this screeenshot, several entities are displayed: robot trajectory (green), robot pose hypotheses (red cubes at the end of trajectory), landmark estimates (white particles generated from landmark probability density functions; 10 per landmark) and axes with wireframe ground plane for visual reference.

```
data ExactCamera = ExactCamera
    { cpos :: Vector Double     -- 3D position
    , crot :: Matrix Double     -- 3x3 rotation matrix
    }
```

Its rotation is represented by a matrix rather than rotation angles because rotation composition is often desired. The second representation, `GaussianCamera`, represents a gaussian camera pose estimate:

```
data GaussianCamera = GaussianCamera
    { cmu :: Vector Double      -- 6D pose mean
    , ccov :: Matrix Double     -- 6x6 pose covariance
    }
```

Both snippets have been commented for clarity. For mathematical details of the camera representation, see section 2.4.

Finally, after the algorithm returns a pose estimate, it is passed back to the C++ code via the same OpenGL-compatible representation as before. It is subsequently converted into a ROS odometry message and published.

3.6 Data visualization

Both C++ and Haskell parts of the program have separate visualization functionality focusing on different data structures. The C++ part visualizes acquired images using OpenCV drawing functions, and overlays them with detected features or features after non-maxima suppression. It can also save the images into files for future reference. The visualization code is located in `visualize.cpp`. figure 2.2 is an example of an image that was produced using this code.

The Haskell part visualizes the SLAM update routine in 3D using the not-gloss library[11]. It is defined in `Display.hs` and has a separate entry point `main`. It does not run online; instead, it uses serialized data that can be produced by the `saveData` routine in `RosMain.hs`. In this way, it runs separately from ROS, providing a faster and more flexible testing environment. The Haskell visualization shows robot pose hypotheses, landmark estimates and the resulting robot trajectory. It is possible to observe the scene from various angles by a virtual camera and to advance step-by-step in simulation. A screenshot is shown in figure 3.2.

Chapter 4

Experiments

We analyzed the algorithm performance in various environments and compared it with other means of acquiring robot pose. Namely, with the 2013 visual odometry algorithm by Jiří Diviš[8], and with a combined tracks/gyroscopic odometry (INSO)[12]. The influence of incorporating INSO into the FastSLAM routine as a proposal transition function was also measured.

4.1 Experimental setup

ROS supports recording the whole robot state into a file (called a *bag file)* during its operation. The bag files can be played back, publishing all topics as recorded. It very much simulates the real experiment. The main purpose of the bag playing and recording functionality is to enable repeating experiments and algorithm debugging. The NIFTi team has recorded an extensive set of experiments that we used to validate our algorithm. All experiments were conducted offline, but as detailed in section 4.7, we believe that our algorithm will work in real-time on the actual robot.

Several trajectory estimation techniques were compared:

INSO (Inertial Navigation System Odometry): an estimate of the angular acceleration is acquired from a gyroscopic sensor and combined with an estimate of the movement speed from the track odometry. These values are integrated over time to form a pose estimate.

Visual Odometry: the 2013 visual odometry algorithm by Jiří Diviš[8].

FastSLAM: the FastSLAM 2.0 implementation described in this thesis; constant-pose transition model is assumed.

FastSLAM + INSO: the FastSLAM 2.0 implementation described in this thesis; transition model is acquired from INSO.

We will refer to those algorithms in the following text by the names written above in bold.

Five different scenes were included in this thesis. They cover both indoor and outdoor environments and show the performance of our algorithm in each of them. The locations of the bag files used for the scenes are in table 4.1.

Scene Name	Bag file location
KN yard	`data/20140409_prague_rail_data_for_vodom_KN_yard /ugv_2014-04-09-15-14-27.bag`
Street	`datasets/zurich_dataset/leica_reference /20130412_hallway_street/street1 /ugv_2013-04-12-15-57-38.bag`
Krč forest traversability	`data/20130624_Krc_rainforest /traversability_1/ugv_2013-06-24-09-55-09.bag`
Lab to yard	`data/20130124_from_lab_to_the_yard /ugv_2013-01-24-16-42-10.bag`
Big squares	`datasets/zurich_dataset/vicon_reference /(0_1)Big_Squares/ugv_2013-03-25-16-17-37.bag`

Table 4.1: Locations of the bag files, representing the scenes. The paths are relative to `ptak.felk.cvut.cz:/datagrid/nifti`.

FastSLAM parameters

All experiments were conducted with the same set of parameters unless otherwise stated. Fine-tuning the parameters for concrete scenes would provide an unfair advantage over the Visual Odometry algorithm, and it is not a realistic usage pattern in practice. The parameters used are:

Parameter	Value
Number of particles	20
Measurement SD	1 px
Transitional SD	x, y: 0.10 m/s; z: 0.17 m/s
Rotational SD (FastSLAM)	y: 0.22 rad/s; x', z'': 0.10 rad/s
Rotational SD (FastSLAM + INSO)	all axes: 0.01 rad/s

In the table above, SD is a standard deviation. The transitional and rotational uncertainties increase with the time between two frames, so the units are per-second.

The rotational SD is lowered for the INSO variant of the algorithm in order to take advantage of the precise rotational estimate INSO offers. Other than that, the parameters for the two variants are identical, including the transitional SD. This ensures that when the tracks skid, the algorithm is able to correct the translational error. In this way, we are not sacrificing robustness by using the prior odometry information.

In the feature matching routine, we check a 3-sigma surrounding of a landmark reprojection for a corresponding feature. Hence, for the algorithm to fail, the robot would have to move at three times greater velocities than the velocities in the table above.

Descriptions of all the tested scenes follow.

4.2 KN Yard

This is a very easy dataset with a large, non-occluded view on well-lit surroundings. It was shot in the campus on Karlovo Náměstí (KN) in Prague on April 9, 2014. An illustrative image is provided in figure 4.1. Several bag files were recorded, but since

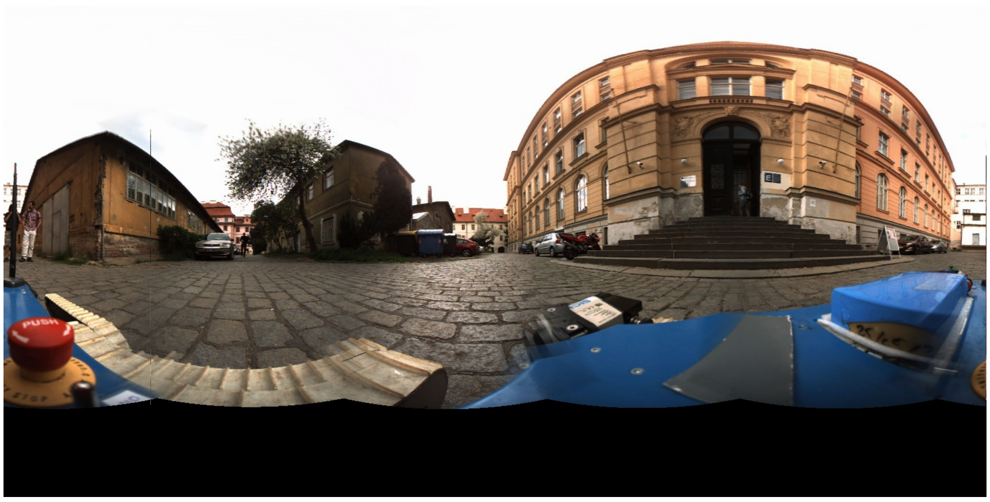


Figure 4.1: KN Yard 2014, an image from the omnidirectional camera. Note that the flippers were lowered specifically to make the view as clear as possible.

the algorithms performed similarly in all of them, only one of them is described here. It is 369 seconds long and the robot covers approximately 120 meters.

Trajectory estimates are plotted in figure 4.2. The trajectory forms a closed loop for an easy reference of accuracy. All the algorithms estimated the trajectory well. The final position matched the starting position best in *FastSLAM + INSO,* falling short of completing the circle only by approximately 2% of the trajectory length. Visual Odometry was the worst, drifting noticeably in scale towards the end.

4.3 Street

This dataset was recorded in Zurich on April 12, 2013. For an image, see figure 4.3. The robot drives on a pavement along a straight road, then crosses the street and drives onto the opposite pavement. It returns back along the street, and when it is opposite to the starting position, it crosses the street back and returns to the starting spot. This forms a closed rectangular trajectory of a total length of approximately 140 m. This scene contains some moving cars and pedestrians, so it is more challenging than the Yard dataset. The robot also uses flippers during the drive, which are not masked out by any of the algorithms; some features are detected on them, decreasing the performance.

The estimated trajectories are shown in figure 4.4. The Visual Odometry algorithm accumulated a significant scale drift along the route. The drift of FastSLAM in both variants was lower. In this scene, FastSLAM + INSO used only the rotational part of the INSO estimate to demonstrate that the scale drift is lowered even without using the translational part. There are two reasons for it:

- The search regions for the feature matching routine are considerably smaller due to the smaller rotational covariance. This increases the number of successful matches and decreases an outlier amount. This effect is amplified by the fact

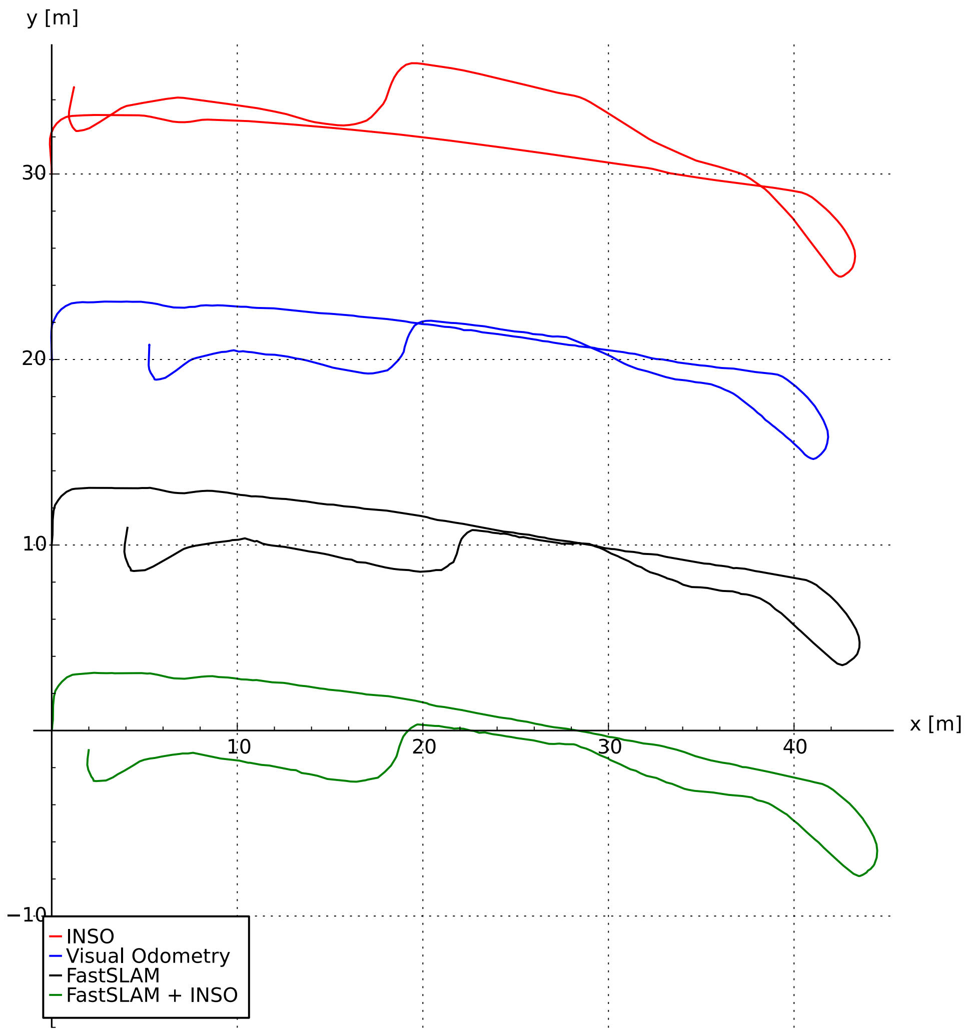


Figure 4.2: KN Yard 2014, trajectory comparison. The trajectories are aligned in scale to INSO and laid out vertically. The robot started at the end of the trajectory that touches the vertical axis. The INSO trajectory drifted clockwise in rotation. Other trajectories drifted in scale. The scale drift was the largest in the Visual Odometry algorithm, and smallest in the FastSLAM + INSO algorithm.

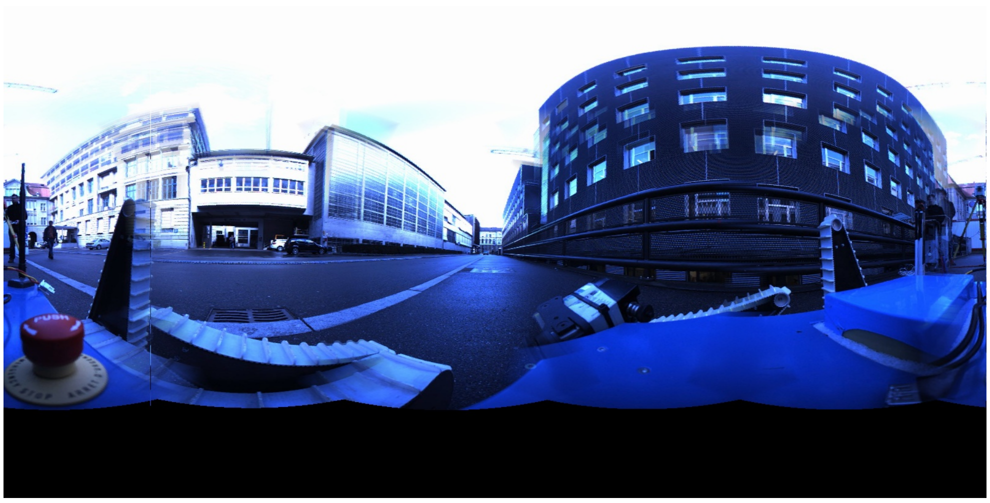


Figure 4.3: Street dataset, an image from the omnidirectional camera.

that the buildings around the robot consist of repetitive patterns (windows etc.). If multiple repetitions fall into a single search region, it is sometimes not possible to chose the right match.

- Fewer particles are wasted on the rotational estimate, and they can help with the translational part of the estimate instead.

The FastSLAM algorithm sometimes does not have enough persistent features to constrain the scale perfectly and it drifts.

4.4 Lab to yard

The Lab to yard dataset was recorded in the campus on KN on January 24, 2013. It is 646 seconds long and the robot traverses approximately 90 m. This data set was chosen to show the systematic scale drift described in section 4.8. The robot starts indoors in a small room (figure 4.5 a). Then, it drives outdoors to the yard (figure 4.5 b) and traverses a loop. It returns back inside through the same door and drives back to its starting position. The trajectory estimates are plotted in 4.6.

It is apparent from the trajectories that the FastSLAM estimates drift considerably in scale. To compare the scales better, speed versus time estimates are plotted in figure 4.7. Since the speed estimate is proportional to the scale estimate, by directly comparing different speed estimates we also compare the scale estimates. The INSO trajectory has got the most reliable scale estimate, the other trajectories are compared against it.

The scale drift of both FastSLAM variants is caused by the fact that the scene scale estimate converges to a single value as described in section 4.8. The scale drift occurs quite rapidly, because there are little persistent landmarks near the doors. The scale is thus not well constrained by the observations and it drifts towards the optimum given by the inverse depth parametrization.

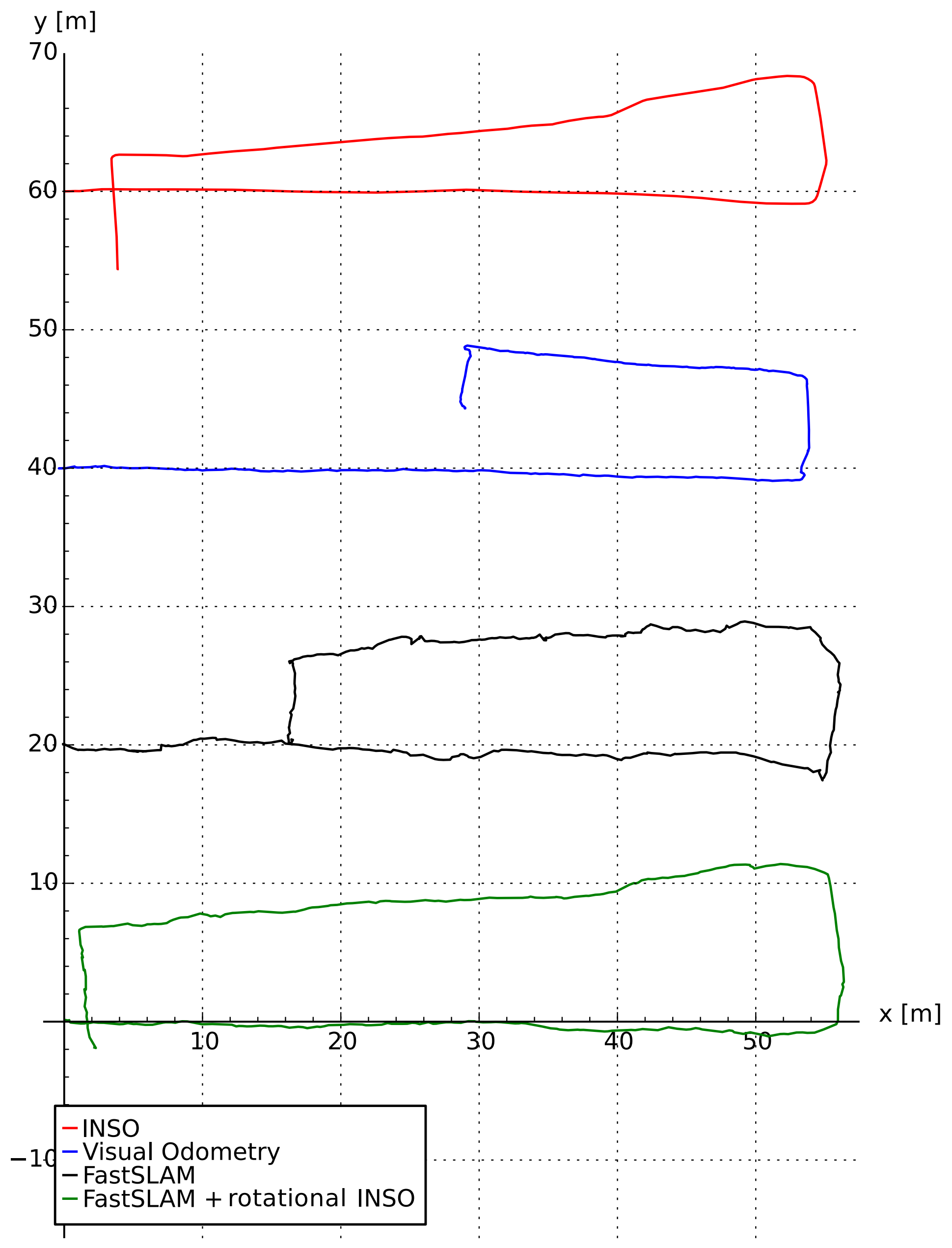
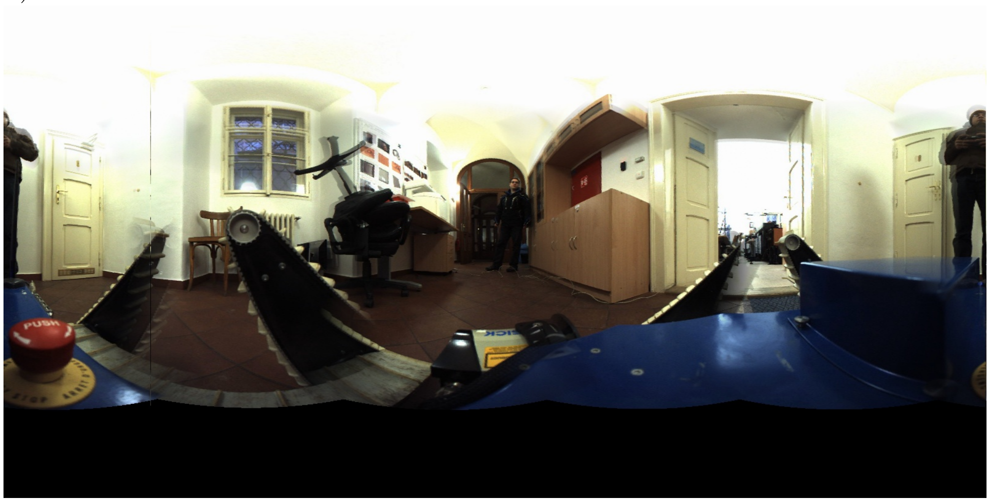


Figure 4.4: Street dataset trajectory comparison. The trajectories have been aligned in scale to INSO and laid out vertically. The robot started its motion at the end of the trajectory that originates on the y axis. Only the rotational part of the INSO estimate was used in the green trajectory. The parameters were identical to the ones used in FastSLAM + INSO algorithm.

a)



b)

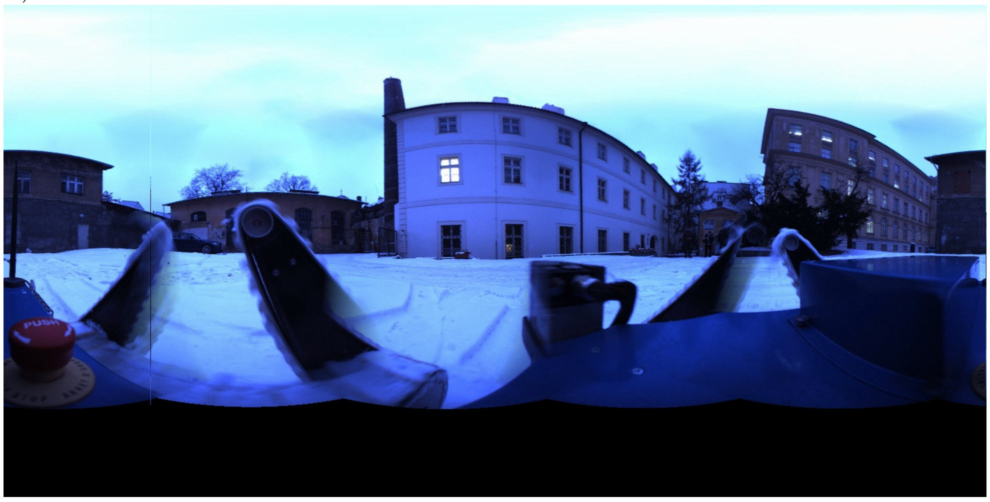


Figure 4.5: Lab to yard dataset. Indoor (a) and outdoor (b) parts of the scene.

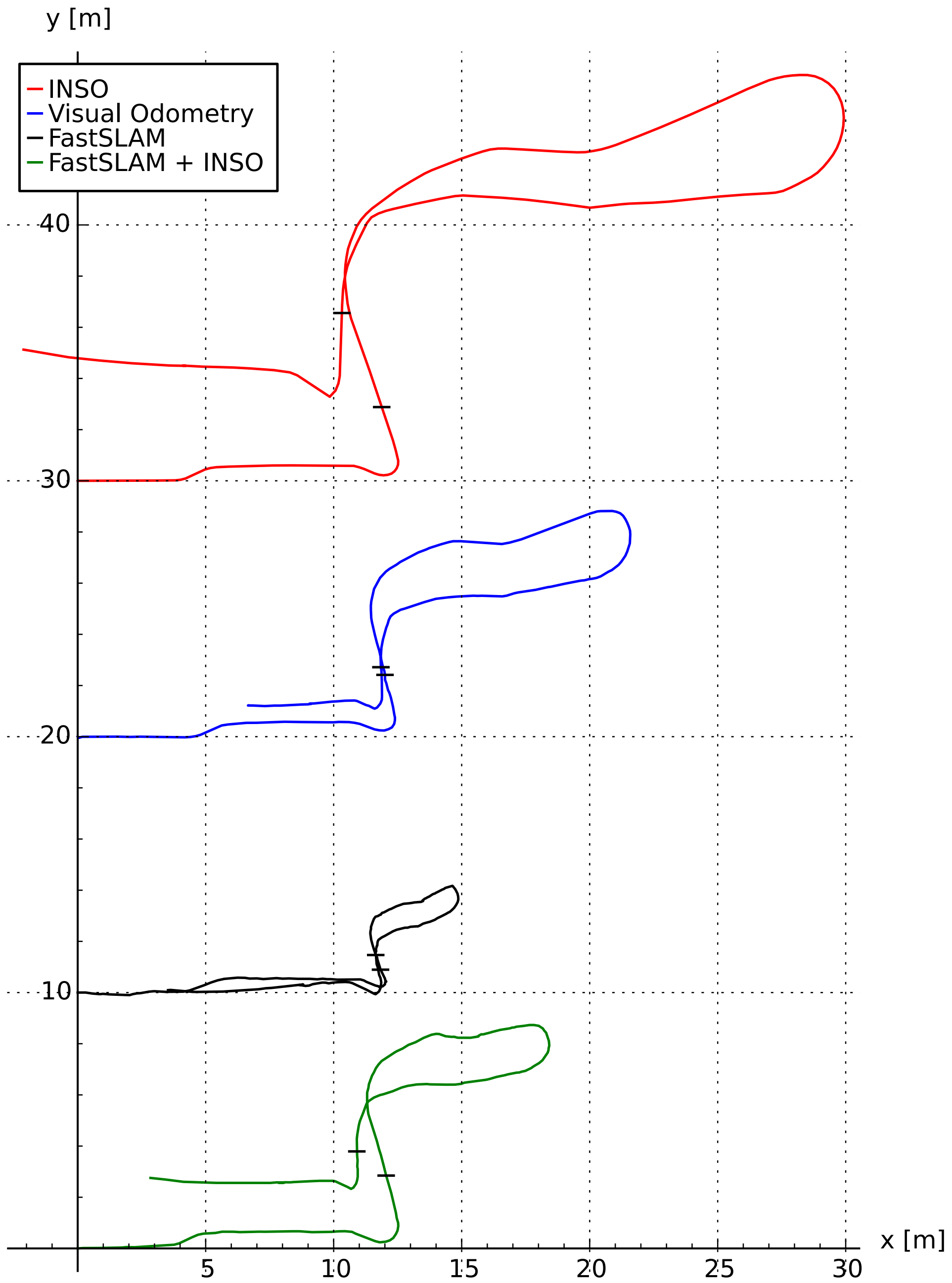


Figure 4.6: Lab to yard dataset trajectory comparison. The trajectories have been aligned in scale to INSO and laid out vertically. The robot started indoor with an x coordinate equal to zero, then rode outside and back inside. The short horizontal lines mark the doorway the robot used to drive outside and inside. With a perfect trajectory estimate, the two horizontal lines would coincide. Visually estimated trajectories drifted significantly in scale, INSO drifted in rotation.

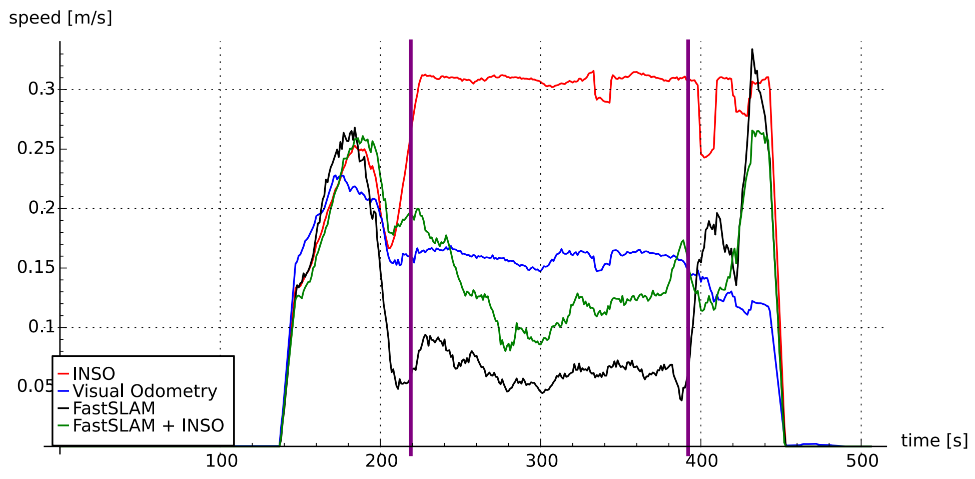


Figure 4.7: Lab to yard dataset. The robot speed estimate is plotted to show the drift in scale of different estimation methods. Vertical violet lines show the times of indoor-outdoor and outdoor-indoor transitions. The speeds have been aligned so that the beginning of the graph is identical for all trajectories, trajectories are filtered by a 10 sec wide sliding average. Both FastSLAM variants underestimated the scale after leaving the building, and they returned to the correct scale after entering the building again. The *FastSLAM + INSO* algorithm did not perform the scale changes as quickly as the *FastSLAM* variant, showing that it is slightly more resilient to the scale change.

This scale drift could be corrected by making the landmark initialization (much) less frequent. The improvement described in section 4.9.1 might help this.

4.5 Krč forest traversability

This data set was captured in Prague on June 24, 2013. The recording session took place in the Krč forest and several bag files were recorded. We chose to test a bag file that records the robot traversing a single log three times. This scene is very challenging for visual odometry for several reasons:

- Most detected features are in the leaves. Their appearance frequently changes due to the robot movement and wind.
- The robot tilts around a horizontal axis while traversing the log. This deforms the features due to the spherical projection.
- The robot makes sharp movements while traversing the log.

The results of the estimation are plotted in figure 4.10. The only visual algorithm that was able to provide a reasonable trajectory was FastSLAM + INSO. FastSLAM diverged and the Visual Odometry algorithm drifted ten-times in scale.

One of the causes of the poor performance of the FastSLAM algorithm is the non-maxima suppression routine. The leaves form a pattern that contains a huge number

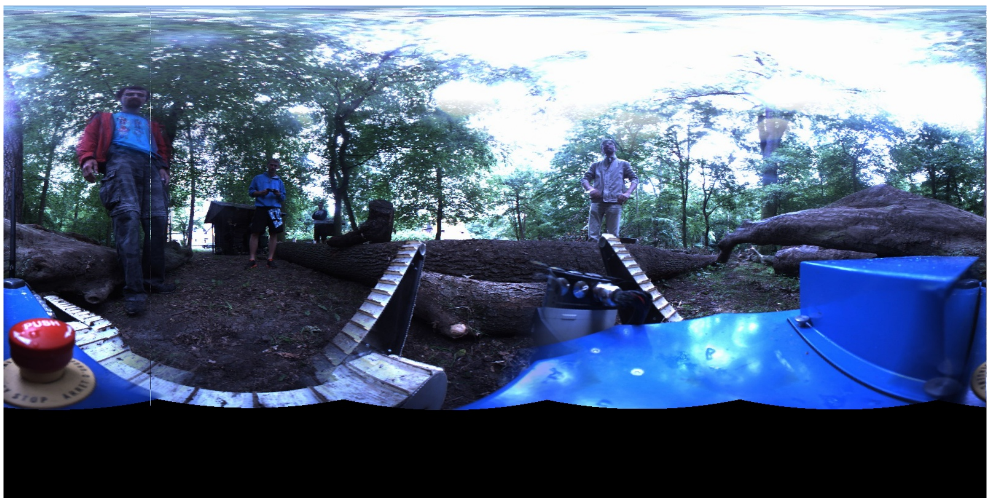


Figure 4.8: Krč Forest scene captured by the omnidirectional camera. The robot traverses three times the log that is visible in the center of the image.

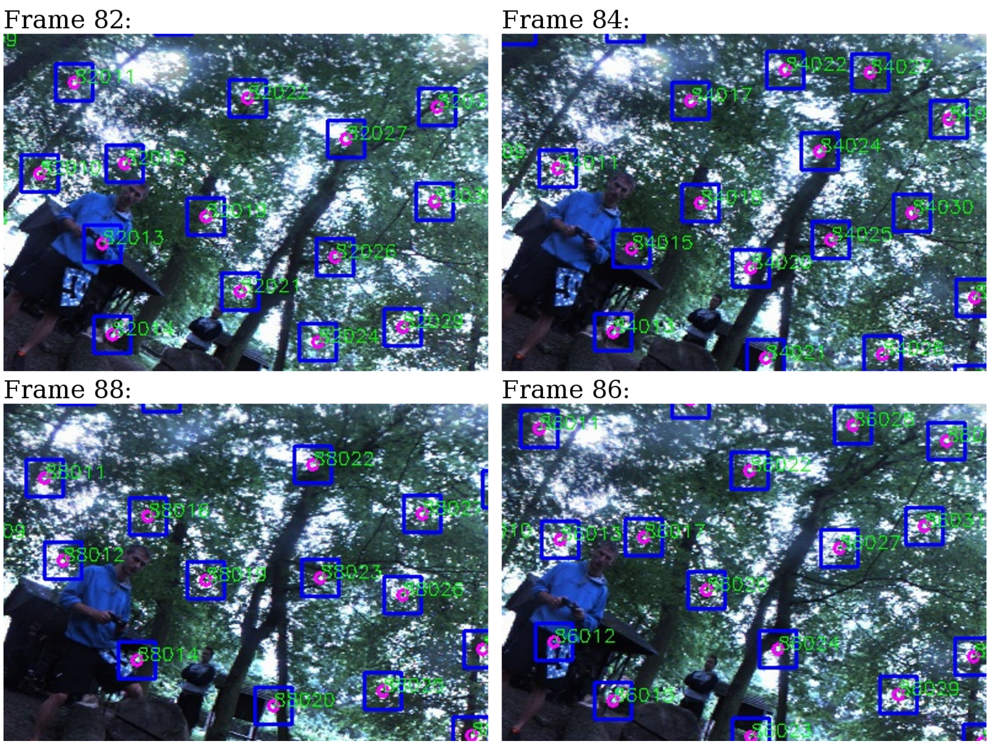


Figure 4.9: Krč Forest scene: details of four consecutive frames showing the landmarks after NMS in the leaves. Note that features that could be associated over multiple frames are scarce.

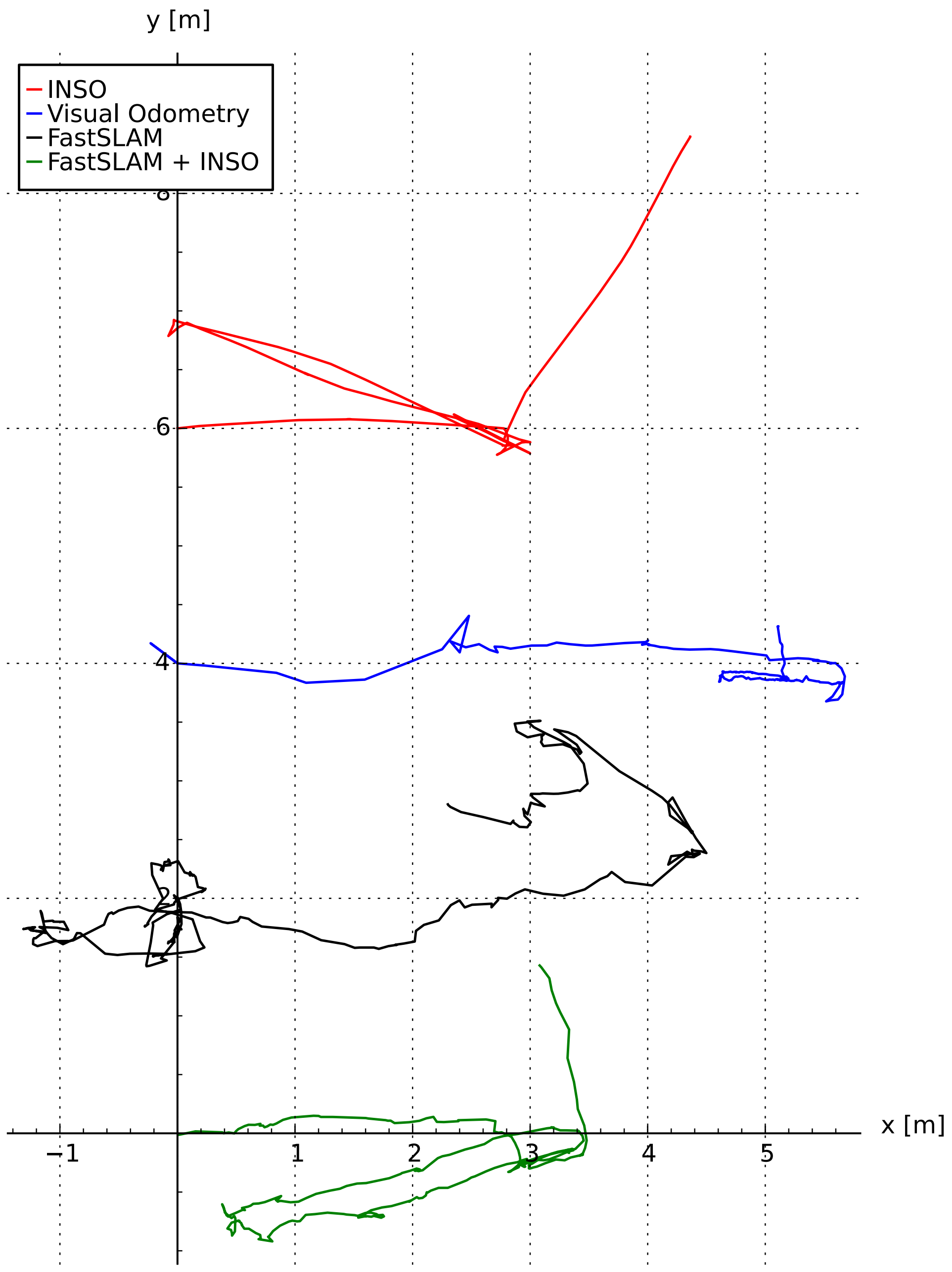


Figure 4.10: Krč forest traversability trajectory estimates. This is a very challenging scene for both INSO and visual odometry. INSO drifts in rotation, Visual Odometry drifts 10 times in scale. FastSLAM diverges. FastSLAM + INSO drifts in rotation, but the scale is fixed well.

of features detectable by the FAST detector. Many of them are detected with large response strengths. The NMS routine chooses only a small subset of them and due to a response strength noise, the subset is fairly random in practice (see figure 4.9). This increases the number of outliers and decreases the number of inlying matches, causing the FastSLAM algorithm to diverge.

The FastSLAM + INSO algorithm estimates a reasonable trajectory because the matching regions are considerably smaller, decreasing the number of outliers dramatically. The resulting trajectory could be used with an advantage over the INSO. In this particular traversability scene, the tracks did not skid. But if they did, the INSO estimate could be less accurate than FastSLAM + INSO.

The Visual Odometry algorithm uses a much less aggressive NMS routine. This enables the algorithm to successfully estimate at least the rotational part of the trajectory correctly.

To improve the estimate, the NMS routine should be made less aggressive. However, the computational complexity increases quadratically with the number of detected features, so it is not possible to make this change straight away. A more elaborate change that eliminates the NMS routine completely is proposed in 4.9.1.

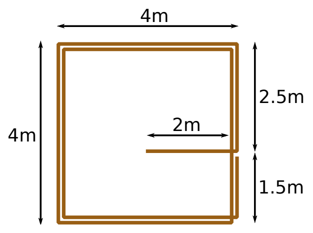


Figure 4.11: Big squares trajectory size and shape.

4.6 Big squares

This is an indoor scene, where the robot circles around a predefined trajectory (figure 4.11). It was recorded in Zurich on March 25, 2013. The scene is pictured in 4.12 and the resulting trajectory estimates are shown in figure 4.14.

This scene proved quite challenging; several reasons for decreased estimation performance were identified.

1. The robot does very sharp turns in the corners. Because the scene is indoor, the edges typically contain one or more blurred images. This decreases matching scores.

2. The room is small. The panorama is not stitched perfectly due to the fact that the centers of cameras' projections do not coincide.

3. The features deform due to the spherical projection as the robot moves. In fact, no landmark survived over more than two turns in either of the FastSLAM variants.

The estimation accuracy could be improved by removing the spherical projection and using the raw camera images instead. This would eliminate stitching errors and spherical deformation. It will also enable detection of the features on the ceiling, which provide an excellent reference of the robot position.

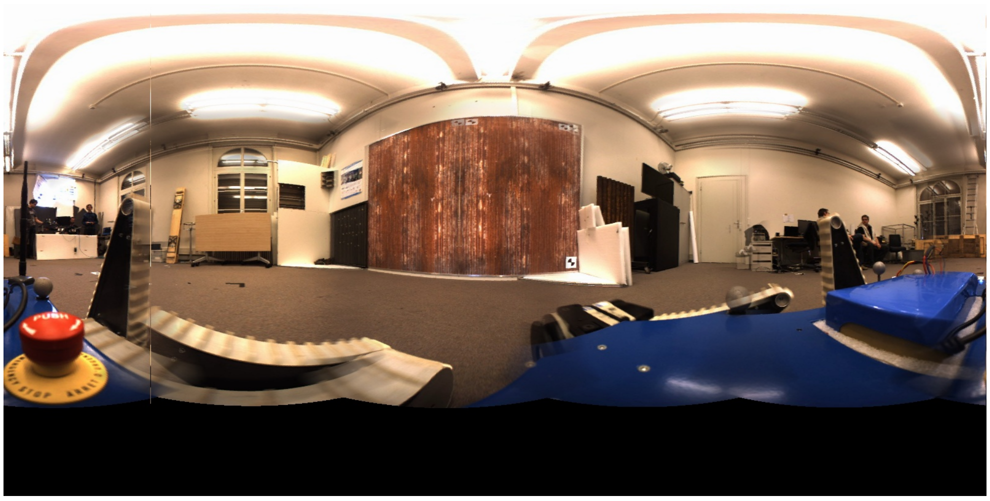


Figure 4.12: Big squares scene.

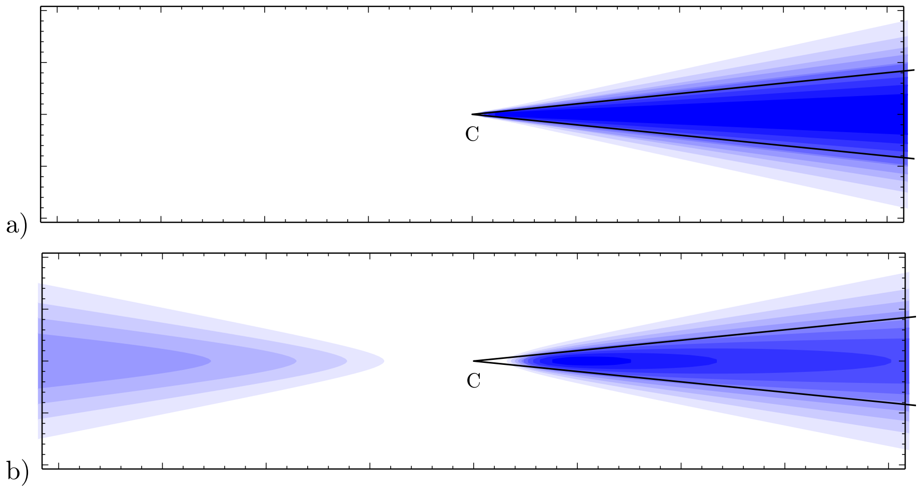


Figure 4.13: Landmark likelihood comparison. This image shows the landmark likelihood after a single measurement: the true likelihood (a), and the likelihood represented by inverse-depth parametrization (b). The difference has been exaggerated by decreasing the covariance in inverse depth ten times. The likelihood in (b) is not uniform along the observation ray. The blue area on the left of (b) is not relevant to this discussion and does not decrease the performance.

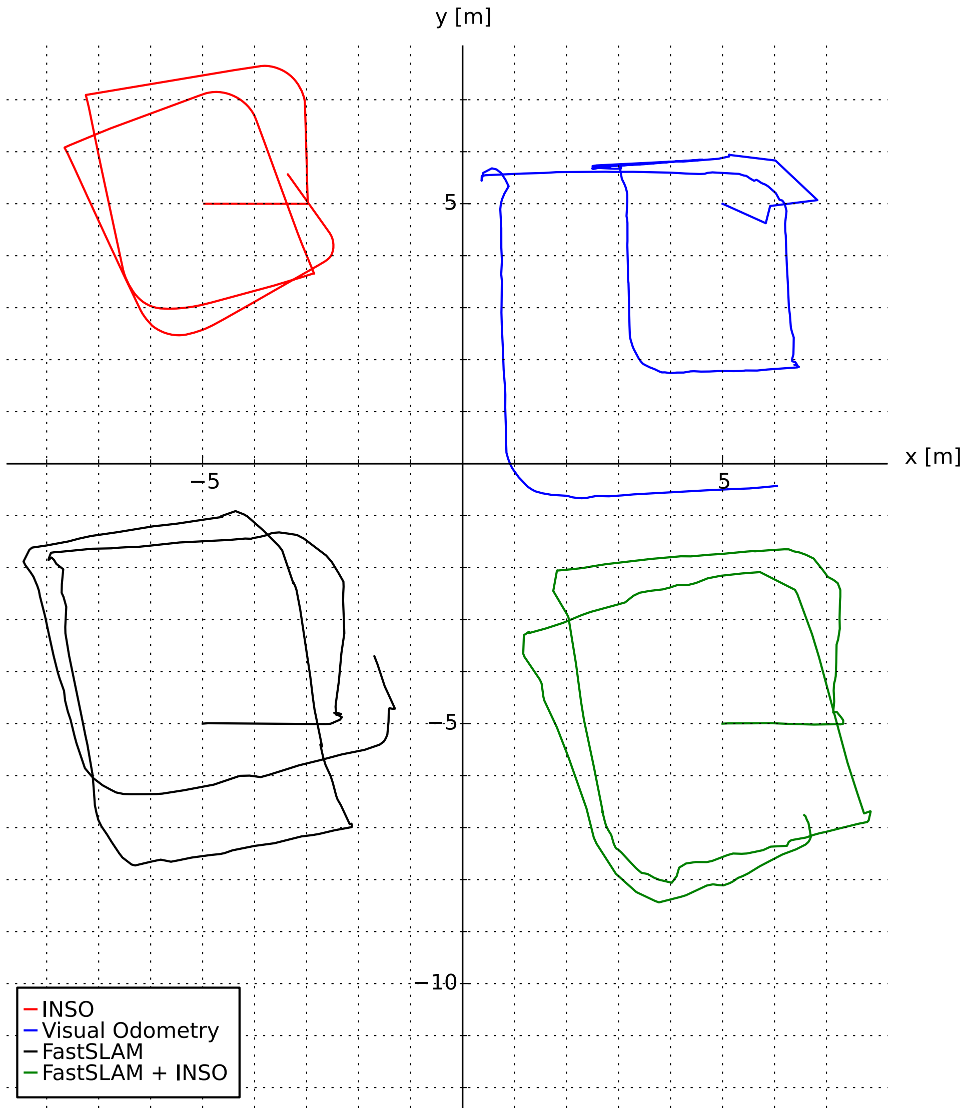


Figure 4.14: Big Squares trajectory estimate comparison. The trajectories were aligned to the INSO trajectory in scale. The Visual Odometry trajectory failed to capture the first two turns correctly, but remained correct in rotation. Both FastSLAM variants drifted slightly in both scale and rotation. The INSO trajectory drifted significantly only in rotation.

Routine	**Time per frame**
C++ thread	48 ms
INSO transformation acquisition	4 ms
Feature detection	41 ms
Non-maxima suppression	2 ms
Haskell thread	225 ms

Table 4.2: Runtime performance in the Yard dataset. Because the threads are executed in parallel, the overall time needed to process a single frame was 225 ms.

4.7 Runtime performance

In this section, the runtime performance of the algorithm is briefly discussed. The program runs in two threads:. One of them processes the images acquired from ROS: it detects the features using ORB and performs the non-maxima suppression. We will refer to this one as a *C++* thread. The other one contains the main update routine (*Haskell* thread). Performance characteristics of both of them were recorded in the Yard dataset and are shown in table 4.2.

On the testing machine[1], it was possible to process four frames per second on average. This is slightly lower than the average frame-rate of the camera (approx. 5 Hz). Some frames were dropped without any significant impact on estimation performance. It is thus very well possible that the algorithm would work on the robot in real-time without any changes.

4.8 Systematic scale drift

This section discusses the drift that was visible in the “Lab to yard” dataset in section §4.4. It is a result of the inverse-depth landmark representation. After a single observation, the landmark likelihood is uniform along the observation ray (assuming a uniform prior). But in the inverse-depth parametrization, this is not the case; the likelihood has got a maximum in a certain distance from the camera. This causes a drift of the recently-initialized landmarks towards this distance.

In our algorithm, this drift is amplified by the fact that the average life-time of landmarks is typically very short. In the Yard dataset, 47 features were observed in each frame on average. Out of them only 20 were associated with landmarks, so 27 new landmarks were created every frame. As landmark initialization is very frequent, the scale of the map converges rapidly towards the maximal likelihood given by the inverse-depth initialization.

The solution to this cause of the drift is to make the feature initialization less frequent. The changes described in section 4.9.1 would have this effect; the initialization rate could easily drop to 1 per frame.

[1]Tested in a Ubuntu 12.04 (32 bit) virtual machine, Intel Core i3-2310M, 2×2.10 GHz.

4.9 Possible improvements

This section details possible improvements to our FastSLAM algorithm implementation.

4.9.1 Non-maxima suppression elimination

An obvious inefficiency comes from the non-maxima suppression routine (NMS). Currently, the same set of features that comes from NMS is used for both feature initialization and feature matching. This is OK for the feature initialization routine: we want only the most prominent features to be incorporated into the recursion. But the matching routine should use all the candidate features to choose the best one. This would have a further advantage of being able to include many pyramid layers to improve feature scale invariance. We expect this improvement to have a great impact on the estimation capabilities of our algorithm.

At the moment, the matching routine is not fast enough for processing thousands of features in real time; spatial indexing of features could be added to alleviate this problem. Also, multiple closely-packed features with closely-matching descriptors would likely emerge from the search, raising a need to change the criteria for match inclusion.

The NMS routine would then serve only to choose the new features to be incorporated into the recursion, thus requiring no repeatability of observations. This would allow us to implement another routine (used in [19]): a randomly-translated grid would be overlaid over the image space, and the best feature would be searched only in cells, where there is none yet. This would make the features uniformly distributed and their number constant.

4.9.2 Direct use of captured images

By avoiding the spherical projection and panorama stitching, following benefits could be achieved:

- The deformation in the top parts of the image would be smaller.
- The ceiling could be used for estimation in indoor scenes.
- Some processing time could be saved.

On the other hand:

- Features would be deformed by a perspective projection instead of a spherical projection. This could be solved by un-deforming the patches around the features accordingly.
- The implementation would be more complicated, requiring us to do the image-to-ray conversion and a single projection center approximation ourselves.
- If used together with the NMS elimination improvement, the implementation would be trickier because of the missing NMS routine. Duplicated features on image edges would have to be somehow eliminated.
- Higher pyramid layers would not detect any features near the image edges.

Chapter 5

Conclusion

We implemented a new algorithm based on FastSLAM 2.0 for visual trajectory estimation on the NIFTi robot. It was implemented in Haskell for flexibility and ease of development. We validated it offline on real data in the form of pre-recorded ROS bag-files.

The experiments showed that the algorithm performs well in various settings and improves upon the previous algorithm. Generally, it produces less drift in scale and comparable drift in rotation. Other advantage is that the algorithm can use the data available from inertial navigation system (INSO). With those data, it outputs reasonable estimates even in very challenging scenes (Krč forest), where the estimate from the previous algorithm was not usable. This provides the robustness that was needed to use the algorithm output in other parts of the robot software.

Based on the analysis of the experiments, some shortcomings of the algorithm were identified. We discovered that our algorithm sometimes significantly drifts in scale. The issue is specific to the cases where the surroundings change in scale. For example, when the robot transitions between indoor and outdoor environments. We identified the cause of this drift as the feature initialization being too frequent.

We proposed changes to our algorithm that have the potential to fix some of the identified problems. The non-maxima suppression routine could be eliminated to improve the algorithm performance in scenes with many detectable features. As a by-product, feature initialization would be less frequent, lessening the scale drift. To improve the algorithm performance particularly in indoor scenes, raw images could be used instead of the spherical projection. That way, the features on the ceiling could be detected and used.

The improved translational estimate that our algorithm provides would hopefully enable the translational output of our algorithm to be incorporated into the robot self-localization routine. It helps to constrain the location of the robot particularly in very large scenes, where the surroundings are outside of the range of the laser range-finder.

List of Abbreviations

EKF	Extended Kalman Filter
FFI	Foreign Function Interface
INSO	Inertial Navigation System Odometry
KN	Karlovo náměstí
ORB	Oriented FAST and Rotated BRIEF
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
VO	Visual Odometry
vSLAM	Visual SLAM

Bibliography

[1] Nifti: Natural human-robot cooperation in dynamic environments. `http://www.nifti.eu/mission`.

[2] T.D. Barfoot. Online visual motion estimation using fastslam with sift features. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 579–585, 2005.

[3] Kostas E. Bekris, Max Glick, and Lydia E. Kavraki. Evaluation of algorithms for bearing-only slam. In *ICRA*, pages 1937–1943. IEEE, 2006.

[4] Manuel M T Chakravarty. c2hs: C->Haskell FFI tool that gives some cross-language type safety. `http://hackage.haskell.org/package/c2hs-0.17.2`, April 2014.

[5] J. Civera, A.J. Davison, and J. Montiel. Inverse depth parametrization for monocular slam. *Robotics, IEEE Transactions on*, 24(5):932–945, 2008.

[6] Koen Claessen. QuickCheck: Automatic testing of Haskell programs. `http://hackage.haskell.org/package/QuickCheck-2.7.3`, March 2014.

[7] Anthony Cowley. Tools for working with ros in haskell. `https://github.com/acowley/roshask`, 2013.

[8] Jiří Diviš. Visual odometry from omnidirectional camera. Master's thesis, Charles University in Prag, 2013.

[9] Ethan Eade and Tom Drummond. Scalable monocular slam. In *in IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 469–476. IEEE Computer Society, 2006.

[10] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[11] Greg Horn. not-gloss: Painless 3D graphics, no affiliation with gloss. `http://hackage.haskell.org/package/not-gloss-0.6.0.0`, November 2013.

[12] Vladimir Kubelka. inso. `http://cw.felk.cvut.cz/wiki/misc/projects/nifti/sw/inso`, 2013.

[13] Gim Hee Lee, F. Fraundorfer, and M. Pollefeys. Rs-slam: Ransac sampling for visual fastslam. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 1655–1660, 2011.

[14] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, Edmonton, Canada, 2002. AAAI.

[15] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, Acapulco, Mexico, 2003. IJCAI.

[16] E. Mouragnon, M. Lhuillier, M. Dhome, F. Dekeyser, and P. Sayd. Real time localization and 3d reconstruction. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 363–370, 2006.

[17] J. Mullane, Ba-Ngu Vo, M.D. Adams, and Ba-Tuong Vo. A random-finite-set approach to bayesian slam. *Robotics, IEEE Transactions on*, 27(2):268–282, 2011.

[18] D. Nister. An efficient solution to the five-point relative pose problem. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(6):756–770, 2004.

[19] C. Roussillon, A. Gonzalez, J. Solà, J.-M. Codol, N. Mansard, S. Lacroix, and M. Devy. RT-SLAM: A Generic and Real-Time Visual SLAM Implementation. *ArXiv e-prints*, January 2012.

[20] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. *Computer Vision, IEEE International Conference on*, 0:2564–2571, 2011.

[21] D. Scaramuzza and F. Fraundorfer. Visual odometry [tutorial]. *Robotics Automation Magazine, IEEE*, 18(4):80–92, 2011.

[22] D. Scaramuzza, F. Fraundorfer, and R. Siegwart. Real-time monocular visual odometry for on-road vehicles with 1-point ransac. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 4293–4299, 2009.

[23] R. Smith, M. Self, and P. Cheeseman. Autonomous robot vehicles. chapter Estimating Uncertain Spatial Relationships in Robotics, pages 167–193. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[24] J.-P. Tardif, Y. Pavlidis, and K. Daniilidis. Monocular visual odometry in urban environments using an omnidirectional camera. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 2531–2538, 2008.

[25] Ville Tirronen. CV: OpenCV based machine vision library. `http://hackage.haskell.org/package/CV-0.3.7`, January 2013.

[26] unknown. containers: Assorted concrete container types. `http://hackage.haskell.org/package/containers-0.4.2.1`, February 2012.

Appendix A

How to Read Haskell Code

This part provides an introduction to reading the code written in Haskell. The goal of this section is to provide a reader with at least a limited sense of the structure of Haskell code. For more comprehensive tutorials, see

- Haskell Meta Tutorial - http://www.haskell.org/haskellwiki/Meta-tutorial
- The Haskell wikibook - http://en.wikibooks.org/wiki/Haskell
- Real World Haskell - http://book.realworldhaskell.org/

A.1 Introduction

Haskell is a very complex language, so only the basics of both syntax and semantics are covered. It is a purely functional language. This means that

1. Values and functions are immutable (there are no variables). This effectively means there are no loops either; list operations and recursion are used instead.
2. Functions do not have any side effects: they only set their return value.
3. Functions that are called with the same arguments twice return always the same value.

Haskell is also lazily evaluated and strongly statically typed. The types can be inferred by the compiler. When mutability or impurity is desired it can be achieved through the use of monads, which are not covered in this text.
The indentation of lines is significant (similar to Python) and serves to signify logical code structure.

Comments

There are two types of comments, analogous to the C comments:

```
-- this is a single-line comment
{- this comment can span
   multiple lines -}
```

A special syntax is used for documentation autogeneration:

```
--| A comment used to document the following line
--^ A comment used to document the preceding line
```

Identifiers

Types have the first letter capital, while normal functions must have the first letter lowercase. By convention, both types and functions are written in CamelCase. Identifiers start with an upper/lower-case letter followed by numbers, letters, and single quotes. So, for example, `foo'` is a valid function name, distinct from `foo`.

A.2 Values and functions

A simple value can be defined as

```
a = 6                -- a number
l1 = [1,2,3,4,5,6]   -- a list
l3 = (1, "ahoj")     -- a pair
```

Functions are defined much in the same way.

```
power x = x * x
add x y = x + y
```

Functions do not have their arguments enclosed in parentheses when evaluated.

```
-- function evaluation
power 5              -- equals to 25
add 1 2              -- equals to 3
power (add 1 2)      -- equals to 9
```

Function application always has a higher precedence than binary operators. Functions are left-associative.

```
power 5 - 2          -- equals to 23
power (5 - 2)        -- equals to 9
add (2+3) 6          -- equals to 11
```

If a function application should have a lower precedence than the binary operators, it is annotated by the `$` operator. The `$` operator is frequently used instead of parentheses.

```
power $ 1 + 3        -- equals to 16
```

Every infix operator is just a binary function whose name consists of only non-alphanumeric characters. New operators can be defined and they are often imported from various libraries. Infix operators can be used in prefix notation by wrapping them into parentheses:

```
(+) 1 2                -- equals to 3
(*) 4 ((+) 2 3)        -- equivalent to: 4 * (2 + 3)
```

Binary functions can also be used in an infix way, wrapping them into back-ticks. These two are equivalent:

```
5 `add` 6
add 5 6
```

A.3 Types

Any expression can have an optional type annotation. Top-level definitions are often annotated for clarity, other definitions less so. The compiler infers the type itself. Type annotations are always delimited by the quad-dot (`::`) and the syntax is

```
foo :: Argument1 -> Argument2 -> ... -> Result
```

There are some basic types predefined. The most frequently used ones are `Int`, `Float`, `Double`, `Bool` and `Char`, which are analogous to the C types.

Here are some examples of annotated expressions:

```
six :: Int
six = 6

-- A binary function that results in an Int type.
add :: Int -> Int -> Int
add a b = a + b

-- Inline type annotation is possible
addOne i = i + 1 :: Int
```

The lists of values have a type in square parentheses. The following could be used to produce a list of four repeating Ints:

```
repeat :: Int -> [Int]
repeat x = [x,x,x,x]
```

New types (analogous to C structs) are defined as

```
data Pixel = Pixel Int Int

-- this time with named fields
data Pixel2 = Pixel { x :: Int
                    , y :: Int }
```

and enums are defined using the pipe character:

```
data Bool = True | False
```

Custom types are used like this:

```
pix :: Pixel Int Int
pix = Pixel 5 6

truthVal :: Bool
truthVal = False
```

A named field can be directly accessed:

```
x pix                -- equals to 5
x pix + y pix        -- equals to 11
```

A.4 Miscellaneous

A function can be defined multiple times. Some arguments may be (partially or fully) specified and the first matching variant of the function is chosen.

```
-- The classical recursive definition of the factorial function.
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

A list is a homogeneous structure: elements have the same type. It can be expressed in two equivalent ways:

```
[1,2,3,4,5]
1:2:3:4:5:[]
```

The second way is frequently used in pattern matching in function definitions. In the following snippet, the value `i` is the head of the list, `rest` is the rest of it. When the list is finally empty, the first definition of the function `sumList` is used.

```
sumList :: [Int] -> Int
sumList [] = 0
sumList (i:rest) = i + sumList rest
```

Appendix B

Compact Disk Contents

`/thesis.pdf` – an electronic version of this thesis.

`/report` – the thesis project, along with all the files needed to render it into a pdf file. This folder contains the main LYX project and all the images and files used in the thesis. Their respective GIMP, Inkscape and GeoGebra projects are also included.

`/report/bp.lyx` – the LYX thesis project,

`/fastslam_2` – the folder containing a ROS node that was developed as a part of the thesis,

`/fastslam_2/src` – source code of the algorithm,

`/fastslam_2/INSTALL` – instructions on how to build, install and use the algorithm,

`/math` – the Sage[1] worksheets used to draw the graphs and to do various computations,

`/math/csv` – the data recorded in ROS by the `ctu_data_logger` node and used to draw the graphs.

[1]Sage is a free open-source mathematics software system licensed under the GPL license. *http://www.sagemath.org/*