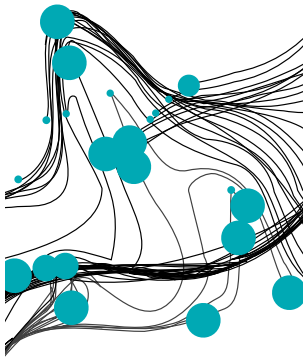
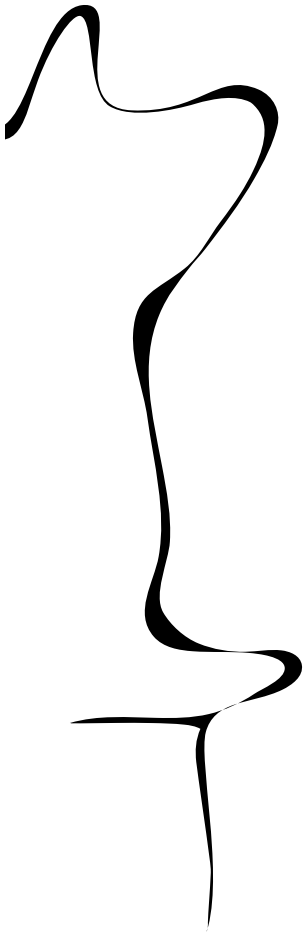


RAM ROBOTICS AND MECHATRONICS



Towards component-based, sensor-independent and back-end independent SLAM

J. (Jeroen) Minnema

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
dr. ir. D. Dresscher
dr. ir. G.A. Folkertsma
dr. F.C. Nex

May, 2020

015RaM2020
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE

UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

CONTENTS

Towards component-based, sensor independent and back-end independent SLAM

Jeroen Minnema, Douwe Dresscher, Gerrit A. Folkertsma and Jan F. Broenink

***Abstract*—Simultaneous Localization and Mapping (SLAM) is the process of simultaneous pose estimation and map creation for mobile robots. Over the years, a lot of research has gone into improving computational efficiency and robustness of SLAM algorithms. This has lead to SLAM implementations tailored to specific algorithms and sensor setups with limited reusability and modularity. The effort required to implement new or modified SLAM implementations—for instance with a different sensor set-up, algorithm or operating environment—is therefore often large and implementation specific optimizations often lead to sub-optimal realizations for different set-ups. This work aims at contributing towards widespread practical implementation of SLAM by investigating the interfaces between sensors and SLAM algorithms in order to come up with a generalized sensor—back-end interface. This is done by analyzing different types of idiothetic and allothetic sensors relevant for SLAM as well as how these are used by both filter-based and graph-based SLAM algorithms. The presented interface provides insight in the influence of sensors and SLAM algorithms on the modularity of SLAM and is used to develop a proof of concept level framework that demonstrates modularity in sensor inclusion and algorithm use.Simulations are used to demonstrate the sensor and algorithm modularity.**

I. Introduction

SIMULTANEOUS Localization and Mapping (SLAM) is the process of estimating a robot's location while concurrently creating a map of the environment. SLAM is a well-studied problem in the field of robotics and is widely regarded as one of the fundamental steps on the road towards fully autonomous mobile robots [1]. Large scale, robust and real-time mapping and localisation is a complex task and therefore traditionally, SLAM related research has been focused on algorithmic improvements with the goal of improving robustness and reducing computational costs [2]. Because of this, many SLAM implementations have been developed in a research context with the goal of showcasing novel algorithms or improvements on existing ones. These algorithms are often optimized for a single use case; i.e. a specific combination of sensors, robot and operating environment. Consequently, different configurations of these SLAM implementations need to be developed with limited reuse of (parts of) previous efforts. In addition, optimizations have led to algorithmic dependencies between parts of the SLAM system that limit reusability and modularity of the SLAM framework.

On the other hand, advances in fields such as self-driving cars [3], unmanned aerial vehicles [4] as well as medical robots [5] resulted in a diverse range of potential robotic platforms—with an equally wide range of sensor and operational requirements—that could potentially benefit from SLAM technology. Moreover some SLAM algorithms are inherently more suited for certain situations than others due to advantages in the area of robust data association (e.g. particle filters [6]), the ability to do offline SLAM (graph SLAM [7]) or low computational costs (Kalman-based SLAM [8]). Effective and efficient implementation of SLAM in these emerging application areas requires reusable and flexible SLAM solutions that reduce the effort required to modify the sensor configuration and SLAM algorithms.

A widespread practical implementation of modular SLAM requires an analysis of the interface and interaction between various sensors and SLAM algorithms. This includes analyzing the different information sources (idiothetic and allothetic sensors), the way this information is handled by the various SLAM back-ends and—depending on the sensor configuration—data association and landmark extraction. The result is a generic "Sensor–back-end interface" integrated in a'modular SLAM framework that is both sensor and back-end independent. The contribution of this work is therefore twofold: A) increased understanding of the information being exchanged by the different sensors and SLAM-algorithms and the corresponding interfaces. And B): a proof of concept of a sensor and back-end independent modular SLAM framework. In order to limit the scope of this work and to facilitate multiple well-known algorithms, it was chosen to limit this work to 2D feature-based SLAM. Although the framework is designed such that it allows generalization to 3D.

The outline of the paper is as follows: in Section II related work is discussed, followed by an analysis of SLAM interfaces (Section III). Based on this analysis a SLAM framework design is presented IV as well as the implementation in Section V. Finally, Sections VI and VII evaluate the framework and compare it with related work.

II. Related work

Several contributions aimed at improving the modularity of SLAM implementations have been made. Often these focus on either sensor modularity–the ability to add or replace additional sensor to a SLAM system– or on developing a SLAM implementation that consists of individual components that can be easily changed or replaced, i.e. a component-based architecture. Firstly, sensors can be added to a SLAM system by doing external (outside the SLAM-algorithm) sensor fusion as shown in [9]. Examples include the fusion of optical and thermal cameras [10] or fusion of different odometry sensors [11]. However, these approaches are limited to specific cases of sensor fusion and do not focus on a generic interface.

Sensor-independent SLAM has been implemented by [12]. Here a Rao-Blackwellised Particle Filter (RB-PF) has been used with the goal of developing a SLAM algorithm that is not restricted to specific sensors or landmarks. This gives sensor flexibility, albeit with a single algorithm and without the ability to additional sensors. Similar results were obtained by [8]: limited sensor modularity with a Kalman filter.

PRO SLAM [13] treats Graph SLAM from a programmer's perspective and presents a modular feature-based implementation of visual Graph SLAM. Sensor flexibility is achieved to a certain extent by RTAB MAP [14] for it allows both LIDAR and visual SLAM, whereas for the latter several camera types are supported. However, other types of sensors such as compasses and IMU sensors are not supported at the moment. RTAB map is also solely based on graph SLAM. The authors of [15] actually attempted to support two different SLAM algorithms: the EKF and the RB-PF, but unfortunately did not succeed in the implementation due to limitations of the software tools used. Finally, [16] addresses the need for reusable component-based SLAM and uses Software Product Lines (SPLs) among other programming paradigms to create a framework that supports two different algorithms, namely the EKF and PF algorithms. Regarding sensor modularity, this work supported replacing of sensors but did not facilitate adding additional sensors to the SLAM-algorithm.

Both on the front of sensor modularity as well as regarding algorithm modularity large steps have been made. However, a SLAM framework that combines complete sensor modularity with the ability to switch between SLAM algorithms, does not exist yet. According to the authors a proper analysis of the interfaces on a conceptual level is currently lacking. This would clarify the possibilities and limitations of both existing and future work in the field of modular SLAM.

III. Anaylsis of Modularity in SLAM

A SLAM system essentially consists of various sensors that collect data and an algorithm that combines this data and uses it to compute an estimated position and map. This section analyses the type of information gathered by the various sensors followed by an analysis of how the back-end algorithms use this data. A functional flow diagram of the system is presented in Figure 1. The categorization of the various sensors and possible intermediate steps such as Landmark extraction, data association and the idiothetic filter are considered later.

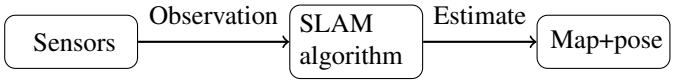


Fig. 1: Schematic view of the essence of a SLAM-system: multiple sensors feed observations (z) into the SLAM-algorithm which combines this data to come up with an estimate of the robot pose and map.

A. Sensor types

Modern robots can be equipped with a vast range of sensors that all provide different information about either the robot's state or its environment. At its core, a SLAM algorithm combines internal and external sensor data to estimate a pose and a map. In [16] the concepts of idiothetic and allothetic sensors were introduced. Here, these definitions are reused and extended. By analyzing the information provided by idiothetic and allothetic sensors, a sensor categorization is developed.

1) Idiothetic sensors: are interoceptive sensors that estimate the robot's pose incrementally ($\mathbf{x} = [x, y, \theta]^\intercal$ in case of a 2D planar robot). This is done with a motion model $u = g(z)$, where u is defined as the pose-change $u = [\Delta x, \Delta y, \Delta\theta]^\intercal$ and z is the sensor's observation. Examples of idiothetic sensors include odometry information obtained by wheel encoders, gyroscopes, IMUs etc. These sensors all have different motion models and observations (z). The output, u, however, has the same format: $[\Delta x, \Delta y, \Delta\theta]^\intercal$. Pose estimation solely based on idiothetic sensors is commonly known as dead-reckoning and is known to suffer from drift due to the incremental nature of the sensors [17].

2) Allothetic sensors: are exteroceptive sensors that use features, landmarks or other properties of the outside world to obtain an observation z. Two types of allothetic sensors exist, relative and absolute allothetic sensors. Absolute allothetic sensors measure absolute poses using the outside world as a reference. Examples include a GPS and a compass. The former uses satellites to obtain an absolute position estimate and the latter measures an absolute orientation based on the earth magnetic field. The practical difference with relative sensors such as a laser-range finder is that the data is presented in an absolute way: $z = [x, y, \theta]^\intercal$ although technically it is still a relative measurement with respect to a single fixed world frame. Because of the difference in data format, a distinction between absolute and relative allothetic sensors is made. All allothetic sensors are accompanied by a measurement model: $z = h(\mathbf{x})$ and its Jacobian.

Relative allothetic sensors form a second type of allothetic sensors. A typical relative allothetic sensor is a laser-range finder, which measures the relative distance of the features around the robot and returns an array of relative distances and directions. Examples of similar relative allothetic sensors include sonars but also all sorts of RGB(-d) cameras that can be used for feature detection. A problem with these relative sensors is that they sometimes receive ambiguous readings—environments may look similar. Depending on the sensor these sensors there often rely on robust landmark extraction and data association. Landmark extraction is the process of obtaining detectable features from raw sensor data. The output of the extractor is a list of landmark poses and—depending on the type of sensor—possibly landmark descriptors (for example BRIEF, SURF and SIFT). This is followed by data association, where the detected features are compared with previously seen features and receive a landmark ID. Landmark extraction and data association are only required for relative allothetic sensors. Once these two steps have been done, the observation is a vector of the relative positions (in polar coordinates, range r and bearing θ) of landmarks lm and their corresponding ID: $z = [lm_{1_r}, lm_{1_\theta}, lm_{1_{ID}}, ..., lm_{m_r}, lm_{m_\theta}, lm_{m_{ID}}]$.

The power of SLAM is that allothetic sensors help correcting for the drift of idiothetic sensors while the pose estimate obtained with idiothetic sensors is used to solve the ambiguity in allothetic readings. This results in a more accurate map and pose estimate than the sensors could achieve individually. Figure 2 shows a summary of the above described sensor classification.

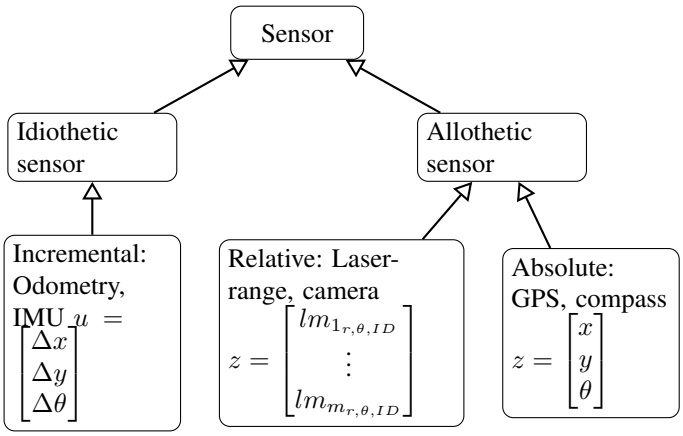


Fig. 2: Categorization of SLAM related sensors along with several examples.

B. SLAM Algorithms

Over the years, a wide range of SLAM algorithms have been developed, ranging from filter based to graph-based approaches. Naturally, many of these treat the data they receive from sensors quite differently. This section analyzes the way idiothetic and allothetic data are processed by SLAM implementations from three different categories of SLAM algorithms namely: Kalman Filter-based SLAM, Graph-SLAM [18] and a Rao-Blackwellised Particle Filter. For the two filter based approaches, FAST-SLAM 1.0 [6] and the Extended Kalman Filter [19] where analyzed specifically as these have formed the basis of many further optimizations. For graph-based SLAM, the process of graph construction in general has been analyzed. This analysis is aimed at the way the sensor data is used by the algorithms with the goal of defining a generic interface. For a detailed explanation of the algorithms refer to [17], [19],[18],[6].

1) EKF SLAM: formulates the SLAM problem in a single EKF and stores the robot pose and landmark positions in a state vector $\mathbf{X}$ and corresponding covariance matrix $\mathbf{P}$. In the prediction step, idiothetic data is processed. This is done with a motion model g, its Jacobian G and the motion model covariance C_x:

$$\mathbf{X_{t+1}} = \mathbf{X_t} + \mathbf{g(z)} \tag{1}$$

$$\mathbf{P} = \mathbf{G^\intercal P G} + \mathbf{C_x} \tag{2}$$

Allothetic data is used for the correction step. For this correction step, the algorithm requires the sensor model $\mathbf{h(X)}$, its Jacobian $\mathbf{H(X)}$ and the measurement covariance $\mathbf{Q}$. The fact that the measurement model **h(X)** is a function of the state, means that either the sensor has to be aware of the current state, $\mathbf{X}$, or the algorithm requires access to the measurement function. The correction step can mathematically be described as [17]:

$$\mathbf{y} = \mathbf{z} - \mathbf{h(X)} \tag{3}$$

$$\mathbf{K} = \mathbf{P H^\intercal (H P H^\intercal + Q_t)} \tag{4}$$

$$\mathbf{X} = \mathbf{X} + \mathbf{(K y)} \tag{5}$$

$$\mathbf{P} = \mathbf{(I - KH)P} \tag{6}$$

Where $\mathbf{y}$ is the innovation and $\mathbf{K}$ the Kalman gain. These two steps show how the EKF algorithm uses idiothetic and allothetic data separately and independently.

2) FAST SLAM 1.0: is an implementation of a Rao-Blackwellised particle filter, which exploits the conditional independence of individual landmarks by estimating the pose and landmarks separately, thereby avoiding the high dimensionality of the statevector which would otherwise lead to extremely high numbers of particles being required. FAST SLAM 1.0 uses particle-based Monte-Carlo localisation to estimate the pose and an EKF for each landmark to estimate the landmark's position [6]. Each particle is a possible representation of the pose vector and has M EKFs. Where M corresponds to the number of landmarks. Similarly to the Kalman filter, idiothetic data leads to a prediction: the paths of the particles are extended by sampling a new pose for each particle using the motion model and motion noise that together form the proposal distribution $p(x_t|x_{t-1}^{[k]}, u_t)$

$$x_t^{[k]} \sim p(x_t|x_{t-1}^{[k]}, u_t) \tag{7}$$

Like for the EKF, besides the measurement, the motion model, its Jacobian and the covariance are required. New allothetic data leads to an update of the weight of the particles, based on the observation z_t. Here the EKFs are also updated. For a derivation of the weight update refer to [6]. Finally, after the weights and EKFs have been updated, the updated weights are used to create a new set of particles.

3) Graph-based SLAM: formulates the problem as a graph where the nodes (vertices) correspond to robot poses in time and landmark positions. The edges between the nodes correspond to measurements. These measurements are written as a set of constraint equations that can be minimized by various non linear solver algorithms. An illustration of a graph-visualization is shown in figure 3. When an idiothetic measurement is received, a new pose vertex is added to the graph as well as a measurement with the relative position and information matrix (the inverse of the covariance matrix). Therefore the only information being required from the sensor are the relative position and information matrix. For allothetic measurements, edges are added in a similar way, the difference is however that in the case of relative allothetic sensors landmark vertices are only added in the case a landmark has not been seen before.

C. Key principles that facilitate Modular SLAM

From the previous analysis the key insight follows that each of the three categories of SLAM algorithms requires three

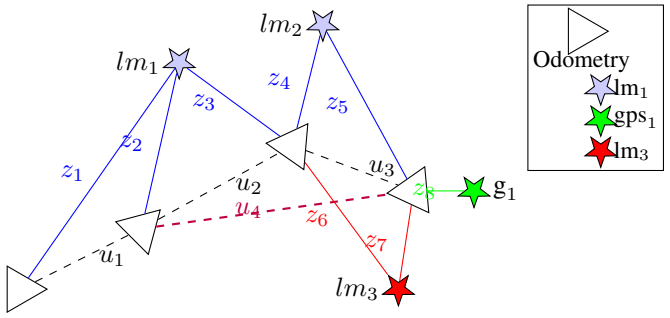


Fig. 3: Schematic view of the structure of a graph-based SLAM problem. Pose nodes are created by odometry measurements, after graph optimization these poses may change slightly. Measurement u_4 indicates the edge corresponding to an additional idiothetic sensor. Measurement z_6, z_7 show an additional allothetic sensor and an absolute allothetic measurement is represented by z_8.

Algorithm	Measurement	Model	Uncertainty
EKF idio.	Pose increment u	Motion model $g(u)$, Jacobian $G()$	Motion covariance C_x
EKF allo.	Observation z	Sensor model $h(x)$, Jacobian $H()$	Sensor noise Q
Graph idio.	Pose increment u	Motion model $g(u)$	Information matrix Ω
Graph allo.	Observation z	Sensor model $h(x)$	Information matrix Ω
PF idio.	Pose increment u	Motion model $g(u)$, Jacobian $G()$	Motion model distribution
PF allo.	Observation z	Sensor model $h(x)$, Jacobian $H()$	Sensor noise Q

TABLE I: Data being exchanged between the sensor and back-ends for the three SLAM algorithms ('idio.' and 'allo.' stand for respectively idiothetic and allothetic information).

aspects from the sensor: the measurement, a measurement model and the uncertainty of the measurement. Depending on the sensor and SLAM algorithm the format of this information may vary slightly, as indicated in Table I. The second important observation that can be made is the fact that each of the three discussed algorithms treats idiothetic and allothetic data differently and independently. Processing of idiothetic data does not require allothetic data and vice versa. These two insights will form the basis of the modular framework design (Section IV).

Due to the separation of idiothetic and allothetic data, it is possible to add additinal sensors to the SLAM algorithm. Additional allothetic sensors are fairly straightforward to add to each of the three back-ends. For a Kalman filter and Fast SLAM (1.0) this involves an additional update step for each sensor. New landmarks being observed by additional sensors lead to augmentation of the covariance matrix and statevector. Observations of landmarks do not affect each other directly. The same holds for graph SLAM, where allothetic expansion leads to additional nodes and vertices being added to the graph.

On the other hand, because of the incremental nature of idiothetic sensors, idiothetic expansion is more complicated. Kalman filter based approaches require a single prediction of the robot pose. Therefore idiothetic sensors would have to be merged outside the SLAM system for the EKF. A similar problem arises for particle filters, although it may be possible to keep a set of particles for each idiothetic sensor and to merge after algorithm execution, based on the weighted sum of both particle sets. For graph SLAM, idiothetic expansion is possible, as this would consist of adding additional constraint equations between previously created pose nodes. However, this introduces an error because of time-synchronization and would either require sensor data synchronization before the idiothetic data is fed into the algorithm or accepting the error this introduces.

IV. Modular framework design

A modular SLAM framework can consist of a back-end and various sensors. The back-end contains the SLAM algorithm and is responsible for the processing of incoming sensor data and publication of the estimated pose and map. The three analyzed algorithms process the idiothetic and allothetic information (table I) differently as described in table II. This can be done by implementing a back-end specific `processIdiothetic()` and `processAllothetic()` method that handle respectively idiothetic and allothetic data according to table II. For example, EKF SLAM deals with idiothetic data by executing the prediction step of the Kalman filter. In order to do this, the back-end requires the pose increment u, motion model $g(u)$, Jacobian $G()$ and motion covariance C_x from the sensor. This is illustrated with the UML sequence diagram in figure 4. The three back-ends are realizations of an abstract base class, as shown by the class diagram in figure 5. Besides this implementation of the back-end, the information from table I has to be specified in a sensor–back-end interface. Furthermore, the sensor analysis pointed out that three different sensor types with different observation formats exist (figure 2), that—depending on the sensor type—may require additional processing steps (such as landmark extraction or filtering). This ultimately results in the software architecture depicted in figure 6.

Back-end	process_idiothetic()	process_allothetic()
EKF	EKF prediction step:	Relative: EKF-SLAM update step of full state. Absolute: update step of just the pose.
Graph SLAM	Add new pose node and measurement of pose increment	Relative: add node and edge if landmark is new, otherwise add an edge. Absolute sensor: always add a node and edge based.
Particle Filter	Update particle paths based on motion model.	Update weights based on sensor observation, update step of landmark EKF's, resample.

TABLE II: Overview of the actions being performed based on new idiothetic and allothetic sensor data.

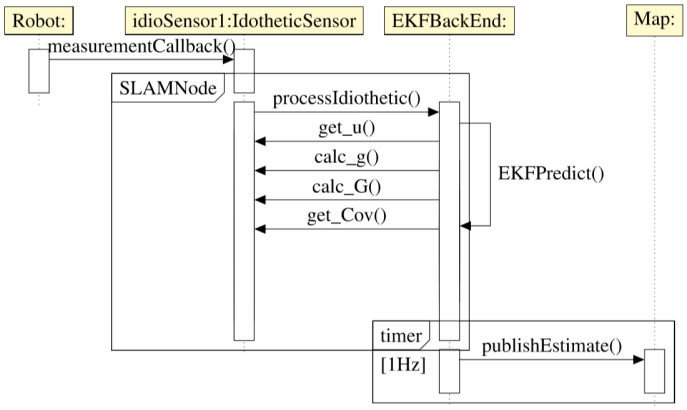


Fig. 4: UML sequence diagram to illustrate the interface between an idiothetic sensor and an EKF. `processIdiothetic()` triggers the prediction step (table II), for which the measurement u, the Jacobean and the covariance are required. This information is requested from the sensor. A timer is used to publish the estimate at a constant frequency. Allothetic data is handled in a similar way.

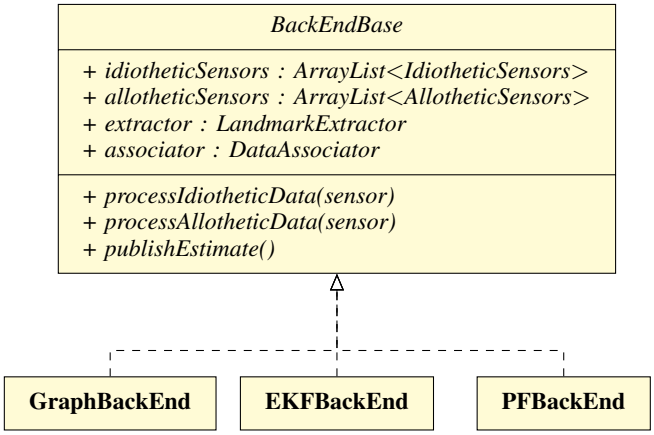


Fig. 5: Class diagram of the back-end classes. The `BackEndBase` class is an abstract class, the three implemented back-ends form the implementation.

A. Sensor–back-end interface

We chose to realize the actual sensor–back-end interface by a range of accessor methods (i.e. getter functions) that make the required information (i.e. the measurement, the uncertainty information and the relevant models according to table I) available to the back-end in the form of arrays. In the earlier mentioned example of handling idiothetic data for an EKF, this boils down to the methods `get_u()`, `calc_g()` `calc_G()` and `get_cov()`. The dimensions of these arrays are dictated by the fact that planar SLAM (SE2) is considered here. The measurement (u or z) format depends on the type of sensor and is either a relative pose increment, an absolute pose or relative observation (see figure 2). The uncertainty information is either presented as a covariance matrix in the sensor domain, or in the case of graph SLAM, mapped to Cartesian coordinates and presented in the Canonical information form. Finally the models are the mathematical mappings between robot-state, world-state and the predicted measurement.

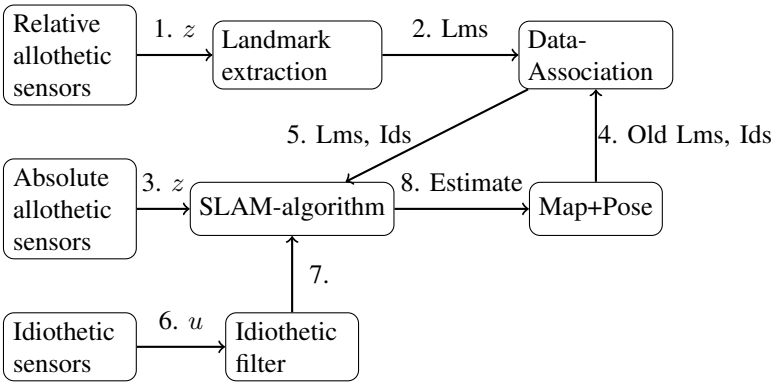


Fig. 6: Architecture of the SLAM framework. Arrows 1,3 and 6 indicate the data obtained via the three sensor types (section: III-A). Arrows 2 and 5 indicate landmark extraction and data association for relative allothetic sensors. Data association may require information on previously landmarks (arrow 4). The idiothetic filter may be required in case multiple idiothetic sensors are being used (explained in 3.C, arrow 7.) The final SLAM estimate is illustrated by arrow 8.

B. Landmark extraction and association

Depending on the sensor, relative allothetic sensors–for instance an RGB(-d) camera)–may require landmark extraction and data association before an associated list of landmarks is obtained. Because sensors can share the same feature extractor and association algorithm, it was chosen to make these separate components instead of adding them to the sensor class directly. Instead composition is used by the back-end class to facilitate sharing of the landmark extractor and data associator. Whether or not extraction and association are required is known at the time of implementation of a sensor and hence considered as a sensor property. The input of the landmark extractor is the raw observation z, for example a camera image. The output is a list of relative positions of the landmarks present in the observation along with a descriptor of the features. The data associator matches the features in the current observation with the previously seen features, either based on likelihood association or using the descriptive features. The result that is used by the SLAM algorithm is a list of relative landmark positions including an ID.

V. Implementation & Demonstration of Modularity

In order to demonstrate the modularity of the designed framework, it has been implemented and several experiments with different configurations have been simulated to demonstrate the ability of the framework to add and replace sensors as well as the usage of three different SLAM algorithms.

The framework has been written in order to work with ROS2 [20] and the sensor and back-end implementations together form a single ROS-node. Simulations have been done with CoppeliaSim [21] which also works with ROS and allows testing of the SLAM framework with several built-in simulated sensors. Several different sensors have been implemented to demonstrate the desired sensor flexibility, these are shown in

Sensor type	Implemented
Idiothetic	Odometry (wheel encoder) IMU
Allothetic, relative	Simulated range-bearing sensor fiducial markers (ArUco)
Allothetic, abosulte	GPS (absolute x,y position) Compass (absolute heading)

TABLE III: Overview of the sensors that have been implemented

table III. Figure 7 shows the results of several simulations using the proposed SLAM framework. These demonstrate proper functioning SLAM using three different back-ends and various different sensor configurations.

VI. Discussion

The presented analysis provides a clear structure of the information that is exchanged by SLAM sensors and algorithms: a measurement, a model and uncertainty information. The results show that, by adhering to this standardized structure, an interface is realized that allows a modular SLAM framework capable of handling addition and replacement of sensors and that can deal with three types of SLAM algorithms.
The question is to which extend this structure contributes towards a modular framework that allows sensor and algorithm flexibility. Given that the interface of a sensor is implemented via the required accessor methods that provide access to the information shown in Table I, adding additional sensors to the back-end involves nothing more than the creation of a new sensor object and appending this sensor to the list of either idiothetic or allothetic sensors that the back-end contains. This locality of change required for additional sensors indicates the modularity of the framework. New algorithms can be implemented by creating a realization of the abstract `BackEndBase` class.

The framework implements three SLAM algorithms, in practice often only one is required, hence implementing the interfaces for all of the sensors and algorithms is often unnecessary. However, by still adhering to the provided structure and implementing the required subset, a flexible SLAM implementation is obtained. This modular implementation allows sensors to be added and replaced with minimal changes to the software.

Another aspect worth addressing is the extend to which current modular SLAM implementations adhere to the proposed interface. RT-SLAM [8] uses a slightly similar interface for the EKF, where the sensor classes contain information about the covariance and Jacobian of the sensors. Regarding Graph-SLAM, ProSLAM [13] is also based on G2O [18] and uses a similar method of graph construction, but on a higher level. Instead of adding individual measurements to the pose graph, ProSlam constructs a graph that consists of smaller local maps where constraints between those local maps are added upon loop-closure. However, this global pose graph used by Pro-SLAM is only used to improve algorithmic efficiency by merging local maps instead of adding every single measurement to the pose graph and not with the goal of facilitating additional sensors in mind. So on the level of individual measurements a different interface is used.

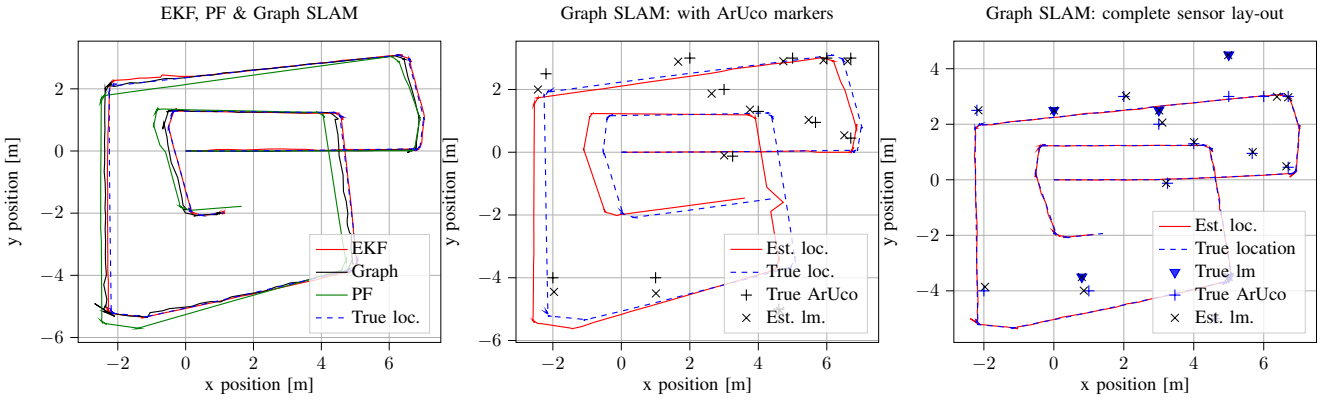
Furthermore, [15] addresses the sensor–back-end interface briefly as well and made the design choice to store sensor specific information as the measurement model in the back-end instead of within the sensor. The limiting effect of this on the modularity is addressed and suggested as a possible improvement. The different sensors that RTAB Map [14] is capable of handling communicate with the back-end through standardized ROS messages. From this it can be deduced that the sensors contain the required models and uncertainty information, similarly to the interface proposed here. However, all sensor types that are supported by RTAB Map supports are integrated in the framework differently and not through a systematic approach.

The allothetic interface proposed by [16] consists of a vector of associated range-bearing measurements–very similar to the interface proposed here. However, that interface does not provide any information on the measurement covariance and sensor model, which means that this information has to be implemented in the back-end, something which is not desired from modularity perspective as it makes the back-end sensor dependent.

VII. Conclusion

This work presents an analysis of the main components of a SLAM system and their impact on the modularity of SLAM with the goal of developing a modular SLAM framework that enables sensor and back-end flexibility and by doing so, decreasing the effort of developing SLAM for new applications. The sensor analysis has identified three different categories of sensors: idiothetic, relative allothetic and absolute allothetic sensors. Each of these sensor categories measures different information types and this information therefore has to be handled differently. It has become clear that SLAM approaches based on Kalman filters, Particle filters and graph optimization all require three aspects from the sensor: a measurement, a model and uncertainty information. This property together with the fact that each of these algorithms treat idiothetic and allothetic sensor data independently was used to develop a modular framework. The structure allows straightforward adding and replacing of various sensors and works with these types of SLAM algorithms. By adhering to the proposed structure and implementing the desired parts, a flexible SLAM implementation can be obtained.The proposed structure has been implemented and simulations demonstrate the framework's modularity.

Some combinations of algorithms and sensors can not be implemented by the current implementation of the framework. For instance, not all SLAM algorithms treat idiothetic and allothetic data completely separately. An example of this is the fact that FAST SLAM 2.0 uses both



(a) (b) (c)

Fig. 7: Simulation results demonstrating the framework's modularity by showing proper functioning with various configurations. Figure (a): three EKF, PF and graph SLAM with odometry and an (artificial) range-bearing sensor. Figure (b) and (c) demonstrate sensor flexibility. Figure (b): Graph SLAM, odometry and visual SLAM (ArUco markers). Figure (c): Graph SLAM, all implemented sensors (table III).

idiothetic and allothetic information for the prediction step. The effect of these improvements and possible solutions require further investigation. On an implementation level, the support for additional idiothetic sensors should be improved. For filter based approaches this requires an additional form of filtering or sensor fusion before idiothetic data is fed into the back-end. For graph SLAM, time-synchronization of idiothetic data has to be realized to allow constraints from different sensors to be linked to two pose nodes. Finally, it would be interesting to study the effect of facilitating scan-matching in the framework, for this would allow improved integration of sensors such as LIDAR.

REFERENCES

[1] H. Durrant-whyte and T. Bailey, "Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms," pp. 1–9, 2006.

[2] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, I. Reid, and J. J. Leonard, "Past , Present , and Future of Simultaneous Localization And Mapping : Towards the Robust-Perception Age," 2016.

[3] F. Kunz, D. Nuss, J. Wiest, H. Deusch, S. Reuter, F. Gritschneder, A. Scheel, M. Stubler, M. Bach, P. Hatzelmann, C. Wild, and K. Dietmayer, "Autonomous driving at Ulm University: A modular, robust, and sensor-independent fusion approach," *IEEE Intelligent Vehicles Symposium, Proceedings*, vol. 2015-Augus, no. Iv, pp. 666–673, 2015.

[4] A. Ravankar, A. Ravankar, Y. Kobayashi, and T. Emaru, "Autonomous Mapping and Exploration with Unmanned Aerial Vehicles Using Low Cost Sensors," *Proceedings*, 2018.

[5] D. Scaradozzi, S. Zingaretti, and A. Ferrari, "Simultaneous localization and mapping (SLAM) robotics techniques: a possible application in surgery," *Shanghai Chest*, 2018.

[6] M. Montemerlo, S. Thrun, D. Roller, and B. Wegbreit, "FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges," in *IJCAI International Joint Conference on Artificial Intelligence*, 2003, pp. 1151–1156.

[7] G. Grisetti, R. Kummerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based SLAM," *IEEE Intelligent Transportation Systems Magazine*, 2010.

[8] 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," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6962 LNCS, pp. 31–40, 2011.

[9] I. Toroslu, "Effective Sensor Fusion of a Mobile Robot for SLAM Implementation," pp. 76–81, 2018.

[10] M. Magnabosco and T. P. Breckon, "Cross-spectral visual simultaneous localization and mapping (SLAM) with sensor handover," *Robotics and Autonomous Systems*, vol. 61, no. 2, pp. 195–208, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.robot.2012.09.023

[11] Z. Zhang, S. Liu, G. Tsai, H. Hu, C.-c. Chu, and F. Zheng, "PIRVS: An Advanced Visual-Inertial SLAM System with Flexible Sensor Fusion and Hardware Co-Design," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1–7, 2018.

[12] C. Schroeter and H. M. Gross, "A sensor-independent approach to RBPF SLAM - Map match SLAM applied to visual mapping," *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, pp. 2078–2083, 2008.

[13] D. Schlegel, M. Colosi, and G. Grisetti, "ProSLAM: Graph SLAM from a programmer's perspective," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3833–3840, 2018.

[14] M. Labbé and F. Michaud, "RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation," *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446, 3 2019.

[15] M. Ristroph, "A Component-Oriented Approach to Simultaneous Localization and Mapping," pp. 1–14, 2008.

[16] M. A. Abdelhady, D. Dresscher, and J. F. Broenink, "Reuse-oriented SLAM Framework using Software Product Lines."

[17] S. Thrun, "Probabilistic robotics," *Communications of the ACM*, vol. 45, no. 3, pp. 52–57, 2002.

[18] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "G2o: A general framework for graph optimization," *Proceedings - IEEE International Conference on Robotics and Automation*, no. June, pp. 3607–3613, 2011.

[19] H. F. Durrant-Whyte, "Uncertain Geometry in Robotics," *IEEE Journal on Robotics and Automation*, 1988.

[20] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, 2009.

[21] E. Rohmer, S. P. N. Singh, and M. Freese, "CoppeliaSim (formerly V-REP): a Versatile and Scalable Robot Simulation Framework," in *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.

APPENDIX A
BACKGROUND ON KALMAN-BASED SLAM

Kalman-based SLAM

Jeroen Minnema

April 2020

1 Introduction

Kalman based SLAM techniques formed the first major branch of successful SLAM [1]. During the years that followed, extensions and modifications were added to the basic Kalman estimator to improve robustness and usability for SLAM systems with large amounts of landmarks. The goal of this appendix is to briefly explain the version of the Kalman filter used in this project, namely SLAM based on the extended Kalman Filter (EKF). Because this is something widely available in textbooks on robot navigation (such as [3]), we keep this section brief and focus on advantages and drawbacks as well as on how improved versions of the EKF would fit in the framework that has been developed. The reason the EKF was chosen is that it is the most simple version of a SLAM algorithm that actually works. A short assessment to which extend other Kalman-based SLAM techniques could be used by the developed SLAM framework is done in Section 4

The basic Kalman filter is a recursive Bayes filter that consists of a prediction and a correction step and that assumes linear motion and sensor models as well as Gaussian noise. The Kalman filter has a wide range of applications, specifically in the context of online feature based 2D SLAM the problem boils down to solving:

$$p(x_t, m | z_{1:t}, u_{1:t}) \tag{1}$$

Where x_t is the robot pose $x = [x, y, \theta]^T$, m is the map consisting of a vector of landmarks lm_i that represent the x and y coordinates of the observed landmarks: $m = [lm_{1,x}, lm_{1,y}, ..., lm_{m,x}, lm_{m,y}]^T$. Furthermore u is the control input (idiothetic information) and z the allothetic observation. In practice, the statevector is defined as the pose and landmarks combined: $x_t = (x, y, \theta, lm_{1,x}, lm_{1,y}, ..., lm_{m,x}, lm_{m,y})^T$. This statevector has a length equal to $3 + 2n$ where n is the number of visible landmarks. In addition, the covariance matrix of the full statevector P is defined. This is wirtten as:

$$\mathbf{X} = \begin{bmatrix} x \\ m \end{bmatrix} \tag{2}$$

$$\mathbf{P} = \begin{bmatrix} \Sigma_{xx} & \Sigma_{xm} \\ \Sigma_{mx} & \Sigma_{mm} \end{bmatrix} \tag{3}$$

2 EKF SLAM

The extended Kalman filter consists of an additional linearization of the current estimate of the statevector and covariance matrix. The prediction step predicts the motion of the robot using the motion model $g()$, the Jacobian of the motion model $G()$ and the covariance of the motion model C_x are used to propagate the covariance matrix.

$$\mathbf{X_{t+1}} = \mathbf{X_t} + \mathbf{g(u)} \tag{4}$$

$$\mathbf{P} = \mathbf{G^T P G} + \mathbf{C_x} \tag{5}$$

Allothetic data is used for the update step. The measurement model $h(x)$, its Jacobian $H()$ and the measurement covariance Q are used. The innovation y which consists of the difference between the calculated and measured observation is calculated. Toggether with the Kalman Gain K, this is used to propagate the statevector and the covariance matrix:

$$\mathbf{y} = \mathbf{z_t} - \mathbf{h(X)} \tag{6}$$

$$\mathbf{K} = \mathbf{P H^T (H_t P_t H_t^T + Q_t)} \tag{7}$$

$$\mathbf{X} + \mathbf{X} + (\mathbf{K} \cdot \mathbf{y}) \tag{8}$$

$$\mathbf{P_t} = (\mathbf{I} - \mathbf{KH})\mathbf{P} \tag{9}$$

[2].

3 EKF SLAM advantages and drawbacks

The advantage of the EKF SLAM algorithm lies in its algorithmic simplicity. Furthermore, in the case of linear models with Gaussian noise it can be mathematically proven that it is actually an optimal estimator. However, there are severy disadvantages as well:

- The EKF can easily diverge if the non-linearities in the models are sufficiently large.
- The complexity of the algorithm and memory consumption are $O(n^2)$ where n is the number of landmarks. Which makes it computiationally intractable for maps with large amounts of features.
- The EKF is limited to feature-based SLAM only.
- The EKF is very sensitive to faulty data associations, i.e. loop-closures and can't recover from this properly.

4 Other Kalman based SLAM filters

4.1 Unscented Kalman Filter (UKF)

The EKF linearizes via a Taylor expension, an improved way of doing this is the Unscented Transform [3], which computes a set of so-called sigma points and transforms each of the points through the non-linear function. Then a new Gaussian distribution is computed based on these transformed points. UKF SLAM has two advantages compared to the EKF:It forms a better approximation for non-linear models. Furthermore, no Jacobians are needed for the uncsented transformation. Although the UKF scales similarly as the EKF ($O(n^2)$, it is slightly slower. Furthermore, the UKF is also restricted to Gaussian distributions.

4.2 The (Sparse) Extended Information Filter (S)EIF SLAM

Alternative to formulating the SLAM problem with moments (the statevector is often written as the mean μ), it is possible to formulate the problem in the information form, also known as the canonical representation: $\Omega = \Sigma^{-1}$, the information matrix is defined as the inverse of the covariance amtrix. Similarly the information vector ζ is defined as $\zeta = \Sigma^{-1}\mu$. Writing the problem in this alternative formulation allows so called sparsification of the innovation matrix. Instead of updating the matrix for all landmarks that have been seen, only 'active' landmarks that have been seen in the recent history are tracked. As a consequence, the SEIF SLAM algorithm has a linear complexity: $O(n)$. SEIF SLAM also calculates the Jacobian of the measurement and motion model, but does not require any additional information.

5 Conclusion

A brief overview of Kalman filter based SLAM approaches has been provided. An important conclusion is that each of the three analyzed algorithms requires no other information other that the measurement itself, its uncertainty, the measurement and motion models and their Jacobians. Table 1 gives an overview of the analyzed Kalman-based SLAM approaches and compares these with the Rao-Backwelized particle filter and GraphSLAM as well.

References

[1] Hugh F. Durrant-Whyte. Uncertain Geometry in Robotics. *IEEE Journal on Robotics and Automation*, 1988.

[2] Cyril Roussillon, Aurélien Gonzalez, Joan Solà, Jean Marie Codol, Nicolas Mansard, Simon Lacroix, and Michel Devy. RT-SLAM: A generic and real-time visual SLAM implementation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6962 LNCS:31–40, 2011.

[3] Sebastian Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.

Back-end	KF	EKF	SEIF	RB-PF	GraphSLAM
Complexity	n^2	n^2	Constant	$\propto m \log n$	$\propto Edges$
Assumed distribution	Gaussian	Gaussian	Gaussian	Pose:Any Landmarks: Gause | pose	Gaussian (but deals with outliers well)
Linearization	All linear	Once (downside: poor scaling)	Once	Not needed	Relinearization every iteration
Flexibility	+ Front-end and Back-end - feature based	like KF	Like KF	++ multiple hypotheses parallel feature & scan based	++ multi modal option to add constrains easily. - grows with trajectory
Large scale	-	-	++ constant with n	+ log n	- to ++ depends on implementation

Table 1: Overview of various SLAM algorithms.

Particle filter based SLAM

Jeroen Minnema

May 2020

1 Introduction

This document is the result of an analysis of particle filters with the goal of determining how particle filters could be implemented in the framework developed during my master thesis. Initially the goal was to implement the three major back-ends in the framework (EKF, Graph-based and particle filter based). Because of time constraints the choice was made to limit the particle filter implementation to a theoretical analysis. It was expected that, because the particle filter is still a probabilistic, filter based approach, the interface would probably be somewhat similar to that of the EKF. Hence, graph-slam was considered to be more interesting to investigate. During the exploration phase a brief analysis of the functioning was made, presented in the project plan. Since then some changes were made in the framework and its interfaces. This document was written to reflect on the impact of these design changes on the implementation of the PF and will most likely be added as an appendix to the report/paper.

2 Basic principles of the particle filter for SLAM purposes

One of the major drawbacks of Kalman filter based SLAM techniques originates from the fact that they are only capable of dealing with Gaussian distributions for the measurement and motion models. An approach for dealing with arbitrary distributions has been the core motivation for particle filter localisation (and later on mapping techniques). Arbitrary distributions can be represented by using multiple weighted samples that represent the posterior. Mathematically the set χ consisting of J particles each representing a possible robot state x associated with weight w and represented by the posterior p can be defined as follows:

$$\chi = \{\langle x^{j]}, w^{j]}\rangle\}_{j=1,...J} \tag{1}$$

$$p(x) = \sum_{j=1}^{J} w^{[j]} \delta_{x^{[j]}}(x) \tag{2}$$

The left plot in figure 1 shows how samples can be used to represent a Gaussian distribution. In practice we want to represent arbitrary distributions, for instance the one $f(x)$ shown in the middle plot, this is the target distribution. The Gaussian proposal distribution $g(x)$ can be used to draw samples from the arbitrary target distribution $f(x)$ by using the principle of importance sampling. Weights for the particles are calculated based on how likely they are (i.e. how well they represent the distribution): $w = f/g$. This principle is called importance sampling and forms the basis of the particle filter. The particle filter finds the pose distribution by repeating the following three steps.

1. Sample the particles with a proposal distribution: $x_t^{[j]} \sim \pi(x_t|..)$
2. Update the importance weight of the particles: $w_t[j] = \frac{target(x_t^{[j]})}{proposal(x_t^{[j]})}$
3. Resample using the updated weights to create J new samples where each sample i is based on the probability $w_t^{[i]}$.

2.1 Monte Carlo localization

Before addressing the full SLAM problem we first explain the core concept by introducing Monte Carlo localization. Here allothetic information is used to improve the robot's state estimate but the landmark's positions are not estimated (i.e. no map is being made). Each particle represents a possible robot state (in our 2D case x, y, θ). The proposal distribution is the motion model, for instance the robot's odometry

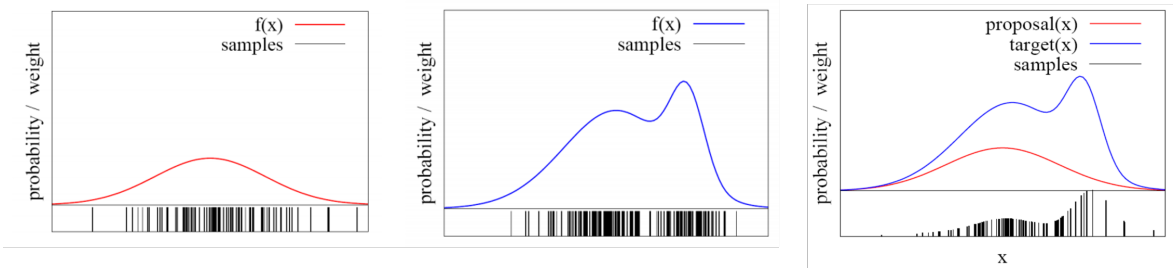


Figure 1: Caption

(u_t). This is often called the prediction step, similarly to EKF-SLAM. Each particle is propagated during this step according to the motion model. The sensor model is used to update the weights based on the observation z_t given the robot's pose and the map m. Mathematically:

$$x_t^{[j]} \sim p(x_t|x_{t-1}, u_t) \tag{3}$$

$$w_t[j] = \frac{target(x_t^{[j]})}{proposal(x_t^{[j]})} \propto p(z_t|x_t, m) \tag{4}$$

Where x_t and x_{t-1} represent the robot's pose at time t given control input u_t. The concept of resampling remains and boils down to replacing unlinkely samples by more likely ones, this effectively removes most of the unlikely samples from the filter and is required in practice because a finite (and often limited) number of samples can be used to limit computation costs.

2.2 SLAM using particle filters

Next, the localisation problem is extended to full SLAM, so this includes landmark estimation. Each particle represents a single hypothesis of the state. So for feature-based (offline) SLAM this state vector is defined as:

$$x = (x_{1:t}, m_{1,x}, m_{1,y}, ...m_{M,x}, m_{M,y})^T \tag{5}$$

In the case of many landmarks, this results in a highly dimensional state vector. The problem with this is that defining the state-vector like this would require enormous amounts of particles to represent the likely regions of possible state space. This is the reason that the step from Monte Carlo localisation –where the statevector is of low dimensionality– towards full SLAM took a while [2]. Rao-Blackwellization is a practical solution that has been widely used for SLAM [1]. The idea is to use the fact that once the robot's pose is known, mapping is straightforward because –given the poses– landmarks are conditionally independent, i.e. landmark m_1 does not depend on m_2 . This dependency is used by factorizing the posteriors such that the particle set is only used to model the robot's path and not the landmarks as well. For each sample (i.e. pose hypothesis) the corresponding map is computed in a later state of the algorithm. The core of this is factorization of the SLAM posterior using Bayes's theory:

$$p(x_{0:t}, m_{1:M}|z_{1:t}, u_{1:t}) = p(x_{0:t}|z_{1:t}, u_{1:t})p(m_{1:M}|x_{0:t}, z_{1:t}) \tag{6}$$

Rao-Blackwellized particle filters exploit the independency of thandmarks by estimating the robot's pose and landmark posteriors seperately. For example FAST-SLAM [3] estimates the pose using Monte Carlo localisation and the landmarks using M 2-dimensional EKF's. One for each landmark. The key steps of the original FASTSLAM (1.0) algorithm are:

1. Extend the paths by sampling a new pose for each particle using the motion model based proposal distribution: $x_t^{[k]} \sim p(x_t|x_{t-1}^{[k]}, u_t)$
2. Update the particle weight based on the observation z_t: $w^{[k]} = |2\pi Q|^{-\frac{1}{2}} exp\{-\frac{1}{2}(z_t - \hat{z}_t^{[k]})\}$
3. Update the landmark belief, similarly to the update step of the EKF
4. Resample, similarly to Monte Carlo localization.

Here $\hat{z}$ is the prediction of the measurement and Q is the measurement covariance matrix, which is updated like for the EKF. The mathematical derivation of the importance sampling (weight update) is beyond the scope of this appendix and can be found in [2] or [3]. Later algorithmic improvements have been made which resulted in FastSLAM 2.0 which also uses the observation during the initial sampling phase. This will turn out to be important for implementation in our framework. Implications of the FASTSLAM algorithm are per-particle data association leading to increased robustness for faulty data associations. Compared to the original EKF, FastSLAM is more capable of dealing with large amounts of landmarks. Furthermore, the probabilistic approach is known for being capable of dealing with occupancy grids and sensors that are based on scan-matching. Something the EKF cannot do.

3 Feasibility of implementing FAST SLAM within the framework

Now that the basic operating principles of Rao Backwelised particle filters have been explained, it is possible to look at the key design choice made for the modular framework and how these would work in case of implementing FAST SLAM. The following three have been considered to be important in this respect:

- Acting on an 'on-new-data' basis.
- Handling Idiothetic and Allothetic sensor data separately.
- The ability to handle multiple sources of idiothetic and allothetic data.

3.1 Separating idiothetic and allothetic data

One of the observations that has been made for EKF and graph SLAM is that these algorithms treat idiothetic and allothetic data very differently. A Kalman filter does the prediction step with new idiothetic data and the update step on new allothetic data. Similarly, graph SLAM adds pose nodes with idiothetic data and (depending on the sensor) landmark nodes when new allothetic data is received. In principle the same holds for Rao-Backwellized particle filters. Considering Fast-SLAM 1.0, this separation is visible as well. Idiothetic data is only used for generating new samples based on the proposal function. Allothetic data is used for updating the weights. The covariance matrix that is used for this is updated in the same way as an EKF, hence it also doesn't require any idiothetic data. Resampling does not require any data, hence one could argue to put this function in a timed loop, similarly to optimizing the graph. This is probably a suboptimal solution, as the allothetic information is only actually used by resampling. Hence it most likely makes more sense to resample the data every time the weights are updated.

The only potential issue that's visible is the fact that an improved version of Fast-SLAM (Fast-SLAM 2.0) actually uses the allothetic observations for the proposal distribution. This would require some form of idiothetic and allothetic synchronization and is most likely not a dealbreaker for our framework but would require some additional effort. Perhaps simply using the last allothetic data available would work out fine. This has to be tested before any further conclusions can be drawn.

3.2 Acting on an 'on new data' basis

In practice it is very likely that the frequencies of idiothetic and allothetic data do not match. Hence the algorithm should be capable of dealing with repeated messages of the same information type. Studying FastSLAM 1.0 implementations shows that this most likely will not be an issue. Animations show that odometry propagation decreases the certainty of the robot's belief. New allothetic information and resampling decreases the pose uncertainty. This appears to be working very similarly to graph and EKF SLAM.

3.3 Multiple idiothetic and allothetic sources

Based on the previous sections, it is expected that this should not be a problem as well. This follows more or less implicitly from being capable of acting on arrival of new data and allowing multiple consecutive idiothetic or allothetic messages to be processed.

In addition to this, the per-particle data association is a big advantage for implementation of different sensor types such as bearing/range only data association. Typically a range only observation results in a multimodal probability distribution. Something which the Particle filter is very good at (CITATION).

4 Conclusion

This document explained the key concepts of the particle filter and analyzed how it could be implemented in the existing framework. Although one can only be sure after succesful implementation, based on a brief study it should be possible to fit a Rao Backwellized particle filter based on FastSLAM 1.0 in the existing framework. This would probably boil down to:

- On new idiothetic data: Propagate samples using the (motion model based) proposal distribution.
- On new allothetic data: Update the weight based on the observations and the updated landmark belief. Followed by resampling using the weights just found.
- Every X seconds (in a timed-loop). Publish the current estimate.

No major potential issues were identified, although upgrading to FastSLAM 2.0 would require some synchronization of idiothetic and allothetic data in order to be able to perform the prediction step. Advantages of the Particle filter include: increased robustness for faulty data association, improved results compared to the EKF-based approach, the ability to produce occupancy grid maps and a first step towards supporting scan-based SLAM solutions. Therefore adding FastSLAM 1.0 is considered to be a very useful step that should be executed if the time budget allows.

References

[1] Hugh Durrant-whyte and Tim Bailey. Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms. pages 1–9, 2006.

[2] Sebastian Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.

[3] Sebastian Thrun, Michael Montemerlo, Daphne Koller, Ben Wegbreit, Juan Nieto, and Eduardo Nebot. Fastslam: An efficient solution to the simultaneous localization and mapping problem with unknown data association. *Journal of Machine Learning Research*, 4(3):380–407, 2004.

Graph-based SLAM

Jeroen Minnema

April 2020

1 Introduction

Whereas EKF and PF SLAM are filter based approaches, graph-based methods approach the SLAM problem differently. A pose graph is used to represent the problem and every node in the grap corresponds to either a robot pose or a landmark position. The edges between these nodes correspond to spatial constraints between them. The idea of graph SLAM is first to build this graph and then to find a node configuration that minimizes the error introduced by the constraints. For this a nonlinear solver is used, for example algorithms like Gauss-Newton or Marquardt Levenberg. As the focus of this assignment is on analyzing and improving on the modularity of the back-end and not on the algorithm and mathematics themselves, the reader is referred to either [1] or [2] for the complete mathematical formulation of the graph SLAM problem.

2 Building the graph

Idiothetic measurements correspond to new pose nodes. Using the motion model $g()$, the pose increment $u = [\Delta x, \Delta y, \Delta\theta]^T$ is calculated. This is added to the previous pose to create a new node at the location $x + u$. The measured pose increment u is added as a constraint, i.e. edge. Furthermore, the canonical representation of the uncertainty (i.e. the inverse of the covariance matrix) of the motion is added to the edge. This concludes an idiothetic measurement for graph SLAM.

If a new landmark, for instance lm_1, is observed, this landmark is added as a node to the graph as well. The initial location of the node is usually the current belief of the robot pose plus the relative observation z. The edge is initialized with the measurement z and the covariance Q. If the same landmark is seen from another position, simply another edge is added to this landmark. This set of motion and measurement constraints can then be optimized results in slightly altered locations (beliefs) of the pose and landmark nodes. Figure 1 illustrates this process of building the graph. Next, adding additional allothetic sensors to the graph is considered.

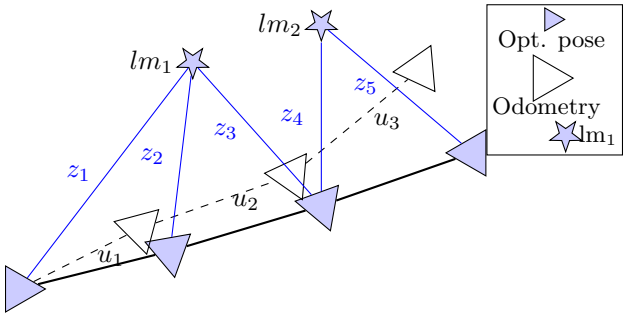


Figure 1: Schematic view of the structure of a graph-based SLAM problem. Pose nodes are created by odometry measurements, after graph optimization these poses may change slightly.

This is illustrated with figure 2. Adding an additional relative allothetic sensor that observes different landmark types (such as lm_3 is trivial, as long as it leads to similarly formulated constraint equations. Absolute allothetic sensors, such as an absolute position measurement made by a GPS sensor, are slightly more interesting: a new node at the measured position is created for every measurement. As this location should (in theory) be identical

to the previous pose node that has been estimated with the odometry sensor, the corresponding edge (i.e. the error) is set to zero. For a GPS sensor the uncertainty could be set according to the HDOP of the sensor (its internally reported uncertainty estiamte). An additional idiothetic sensor simply results in additional constraint

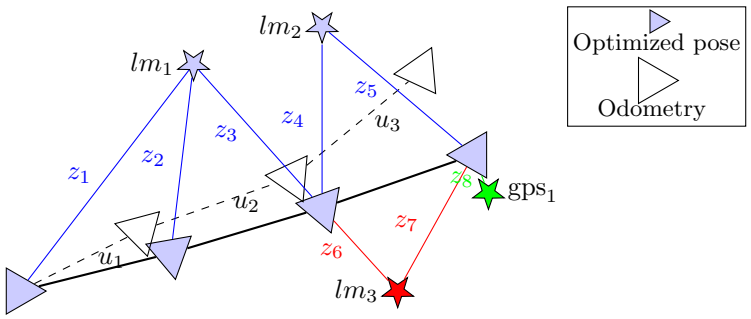


Figure 2: Showing how an additional idiothetic sensor can be modeled: by adding an additional constraint equation that is mapped to the relevant nodes.

equations being added to the graph, as u_4 in figure 3 shows. However, this assumes that the measurements of the idiothetic sensor have been time-synchronized first. Finally, additional idiothetic and allothetic sensors are

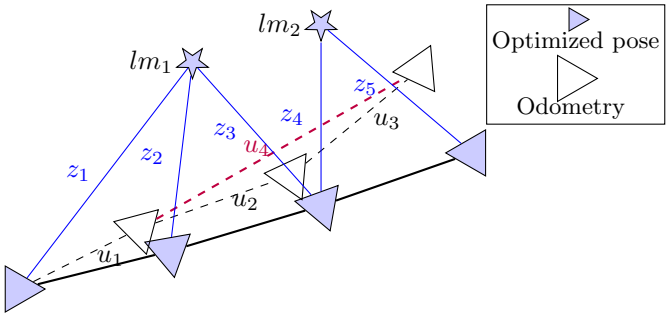


Figure 3: Showing how an additional idiothetic sensor can be modeled: by adding an additional constraint equation that is mapped to the relevant nodes.

combined in figure 4.

3 Remarks

Some interesting differences between graph SLAM and filter based approaches have been found:

- Compared to filter based approaches, graph SLAM is capable of handling multiple idiothetic sensors, as long as time-synchronization takes place.
- No pose uncertainty information is obtained by optimizing the graph. Something which is achieved with the Kalman-based approaches.
- Graph SLAM is a form of offline SLAM, i.e. the entire pose graph is being optimized at once, compared to EKF and PF SLAM that do online SLAM where only the most recent pose is being optimized.

4 Conclusion

Also for graph SLAM, idiothetic and allothetic data is handled separately and independently. The three different sensor types are added to the graph differently although the data that's exchanged is the same, i.e. pose and uncertainty information.

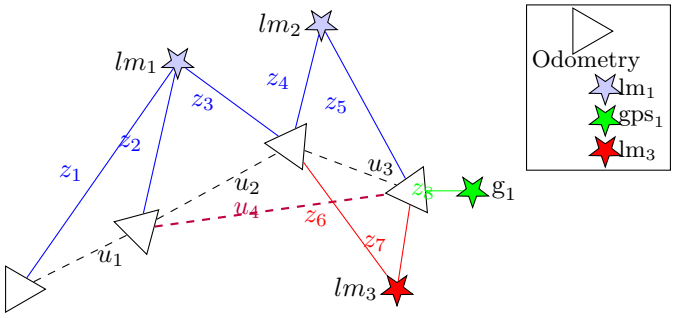


Figure 4: Schematic view of the structure of a graph-based SLAM problem. Pose nodes are created by odometry measurements, after graph optimization these poses may change slightly. Measurement u_4 indicates the edge corresponding to an additional idiothetic sensor. Measurement z_6, z_7 show an additional allothetic sensor and an absolute allothetic measurement is represented by z_8.

References

[1] Giorgio Grisetti, Rainer Kummerle, Cyrill Stachniss, and Wolfram Burgard. A tutorial on graph-based SLAM. *IEEE Intelligent Transportation Systems Magazine*, 2010.

[2] Sebastian Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.

Framework Design

Jeroen Minnema

April 2020

1 Introduction

This appendix explains and motivates the key design choices and interfaces of the framework developed during this assignment and serves as additional material next to the paper and documented code. The structure of this appendix is as follows: first the structure of a SLAM system is derived. The result of this is a functional flow diagram indicating the components present in the SLAM system. Next, the interface between the various sensors and the back-end is explained in more detail. Section 3 describes each of the individual components. Finally, in order to provide a modular interface with either simulation software or robotic platforms, ROS [1] was chosen to provide a flexible interface. Section 4 explains the integration of the framework in ROS.

1.1 Structure

At its core a SLAM system combines observations from various sensors (z) and uses this to estimate the robot pose and evnironment around the robot. This is schematically illustrated with Figure 1.

Figure 1: Schematic view of the essence of a SLAM-system: multiple sensors feed observations z into the SLAM-algorithm which combines this data to come up with an estimate of the robot pose and map.

As explained in the paper, sensors can be divided into three categories: idiothetic sensors that measure the robots pose incrementally, relative allothetic sensors that use the outside world relative to the robot to provide information and absolute allothetic sensors that use external references to obtain an absolute position estimate. This categorization is illustrated by Figure 2. Based on this sensor categorization, we can derive the structure of the SLAM framework. Figure 2 shows that three different sensors produce observations in different data formats. In addition to this, relative allothetic sensors may require further processing steps that consists of landmark extraction and data association. Furthermore, idiothetic expansion requires a single estimate in order to work with filter based back-ends. Therefore some kind of sensor fusion or filtering is required outside the back-end algorithm in these cases. These additional

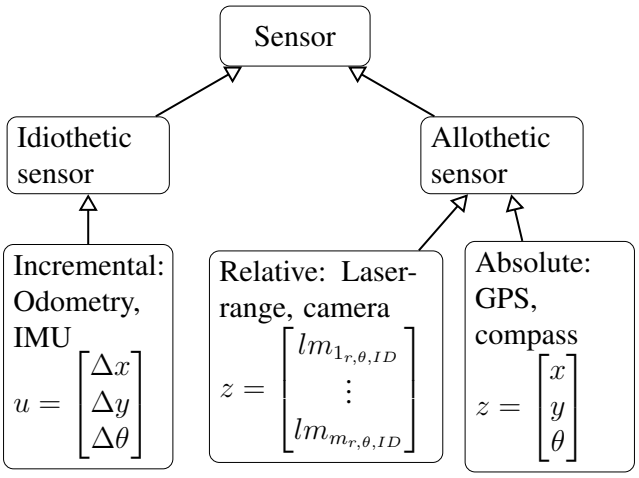


Figure 2: Categorization of SLAM related sensors along with several examples.

components result in the functional flow diagram shown in Figure 3. Note that the idiothetic filter has not been implemented in this work, for sensor fusion or filtering is considered to be a known technology that is not necessarily relevant to assess the feasibility of a modular SLAM framework. The numbers in the figure correspond to the following information being send:

1. Relative allothetic observation, can either be a raw observation (an image, laser scan etc.) or a list of relative landmark observations that may or may not contain an ID: $[lm_{1\theta}, lm_{1r}, lm_{1ID}, ...lm_{m\theta}, lm_{mr}, lm_{mID}]^T$
2. List of landmarks and feature descriptors
3. Absolute allothetic observation z: measurement of the robot's state vector $[x, y, \theta]^T$
4. List of associated landmarks (with id)
5. Previously seen landmarks, Ids
6. Idiothetic observation(s) $u = [\Delta x, \Delta y, \Delta\theta]^T$
7. Single idiothetic, incremental pose update u: $[\Delta x, \Delta y, \Delta\theta]^T$
8. Slam estimate: state vector: $[x, y, \theta, lm_{1x}, lm_{1y}, lm_{1id}, ...lm_{nx}, lm_{ny}, lm_{nid}]^T$

2 Sensor–back-end interface

A SLAM back-end that consists of sensors and the SLAM algorithm has been developed. The core of this contribution is formed by designing a sensor–back-end interface that allows adding additional sensors and using different SLAM algorithms. In the analysis in the paper on the way idiothetic and allothetic information are being used by the various SLAM algorithms it was concluded that Idiothetic and allothetic data are used separately and independently. This means that an idiothetic measurement can be handled repeatedly without an intermediate

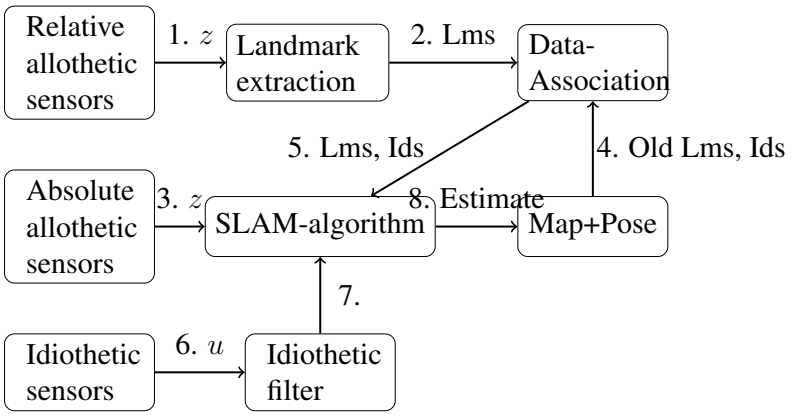


Figure 3: Architecture of the SLAM framework. Arrows 1,3 and 6 indicate the data obtained via the three sensor types. Arrows 2 and 5 indicate landmark extraction and data association for relative allothetic sensors. Data association may require information on previously landmarks (arrow 4). The idiothetic filter may be required in case multiple idiothetic sensors are being used (explained in 3.C, arrow 7.) The final SLAM estimate is illustrated by arrow 8.

allothetic measurement. The result of this analysis can be summarized in two tables: Table 1 shows the information that is required for the three algorithms to handle respectively idiothetic and allothetic data. Table 2 shows the respective actions that are involved in these steps.

Algorithm	Measurement	Model	Uncertainty
EKF idio.	Pose increment u	Motion model $g(u)$, Jacobian $G()$	Motion covariance C_x
EKF allo.	Observation z	Sensor model $h(x)$, Jacobian $H()$	Sensor noise Q
Graph idio.	Pose increment u	Motion model observation	Information matrix Ω
Graph allo.	Observation z	Sensor model observation	Information matrix Ω
PF idio.	Pose increment u	Motion model $g(u)$, Jacobian $G()$	Motion model distribution
PF allo.	Observation z	Sensor model $h(x)$, Jacobian $H()$	Sensor noise Q

Table 1: Data being exchanged between the sensor and back-ends for the three SLAM algorithms ('idio.' and 'allo.' stand for respectively idiothetic and allothetic information).

The observation of the independence between idiothetic and allothetic data is also very useful considering the goal of adding multiple sensors. Something which inevitably leads to repeated measurements of a specific sensor. Because of this, it was chosen to make the back-end act on an "on-new-data" basis. In practice this means that when a sensor receives data, it triggers respectively a processIdiotheticData () or processAllotheticData () function in the back-end. These functions are back-end specific and consist of the actions described

Back-end	process_idiothetic()	process_allothetic()
EKF	EKF prediction step:	Relative: EKF-SLAM update step of full state. Absolute: update step of just the pose.
Graph SLAM	Add new pose node and measurement of pose increment	Relative: add node and edge if landmark is new, otherwise add an edge. Absolute sensor: always add a node and edge based.
Particle Filter	Update particle paths based on motion model.	Update weights based on sensor observation, update step of landmark EKF's, resample.

Table 2: Overview of the actions being done based on new idiothetic and allothetic data.

in table 2. In order perform these actions, sensor specific information is required, such as the measurement, the uncertainty and possibly the Jacobian. The information that is being exchanged is shown in Table 1.

For the relation between the back-end and the sensors, it was chosen to use composition as this makes adding new sensors as simple as creating a new object of the type of the sensor to be added and to add this to a list of either idiothetic or allothetic sensors that is stored in the back-end. Similarly, Figure 5 shows the sequence diagram of processing an allothetic

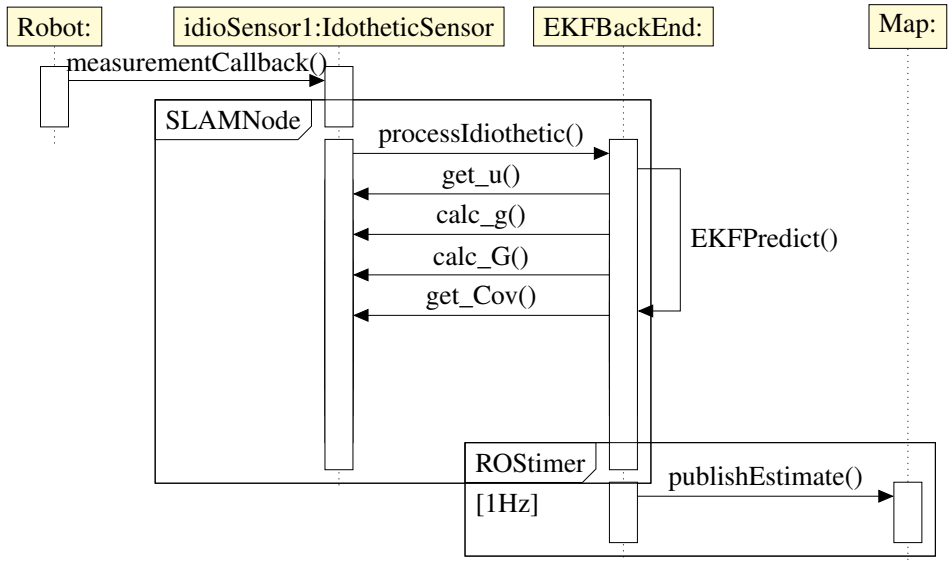


Figure 4: Sequence diagram to show the interaction between idiothetic sensors and the back-end, in this case an EKF back-end. `processIdiothetic()` triggers the prediction step of the Kalman filter, for which the measurement u, the Jacobian and the covariance are required from the sensor.

measurement for an EKF back-end. Here the allothetic measurement triggers the update step of the EKF.

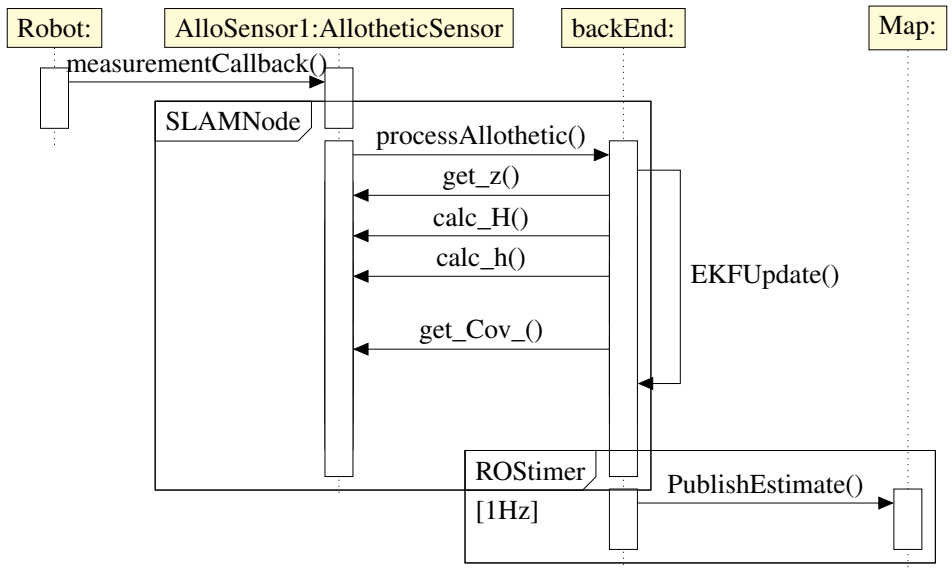


Figure 5: Sequence diagram to show the interaction between Allothetic sensors and the back-end, in this case the EKF back-end.

2.1 Resulting interface

The complete resulting sensor–back-end interface that is required to implement each of the three back-ends follows from Table 1 and 2 and consists of the following parts:

1. *The sensor observation:* allothetic sensors: z, idiothetic sensors: u, according to the data format as specified in Figure 2
2. *Sensor models:* allothetic sensor: $h()$, idiothetic sensor $g()$
3. *Measurement uncertainty:* Represented as covariance C_x or information (inverse of covariance) Ω
4. *Jacobians* of the measurement or motion models, respectively $H()$ and $G()$

This data is made available through respective getter functions, such as `get_u()`, `get_z()`, `calc_g()` etc. These return the measurements, models or uncertainties relevant to the SLAM algorithm in the form of arrays. The dimensions of these arrays are dictated by the sensor type (see Figure 2) and the fact that planar SLAM is considered. Furthermore, if the sensor has n outputs (for example two in the case of an odometry sensor based on two wheels), consequently the covariance matrix is an nXn matrix and the Jacobian $G()$ has n rows. The UML-sequence diagram, Figure 4 provides an example for handling idiothetic data for the EKF. The interface is standardized using abstract parent classes for the sensors as well as the back-ends. Child that form realizations of these abstract classes have been created.

2.1.1 Interface class diagram

The concept of abstract parent classes and realizations in child classes has been used to ensure reusable and interchangeable components in the framework. For the back-end

this boils down to a class with a list of idiothetic and allothetic sensors and (optionally, depending on the allothetic sensors) an extractor and data associator. The implementation of the `processIdiotheticData()` and `processAllotheticData()` depend on the SLAM algorithm that is implemented. Figure 11 illustrates this structure. A similar structure of an abstract base class and realization in the child class has been used for allothetic and idiothetic sensors. Section 3 provides a more detailed explanation of both the back-end and sensor components.

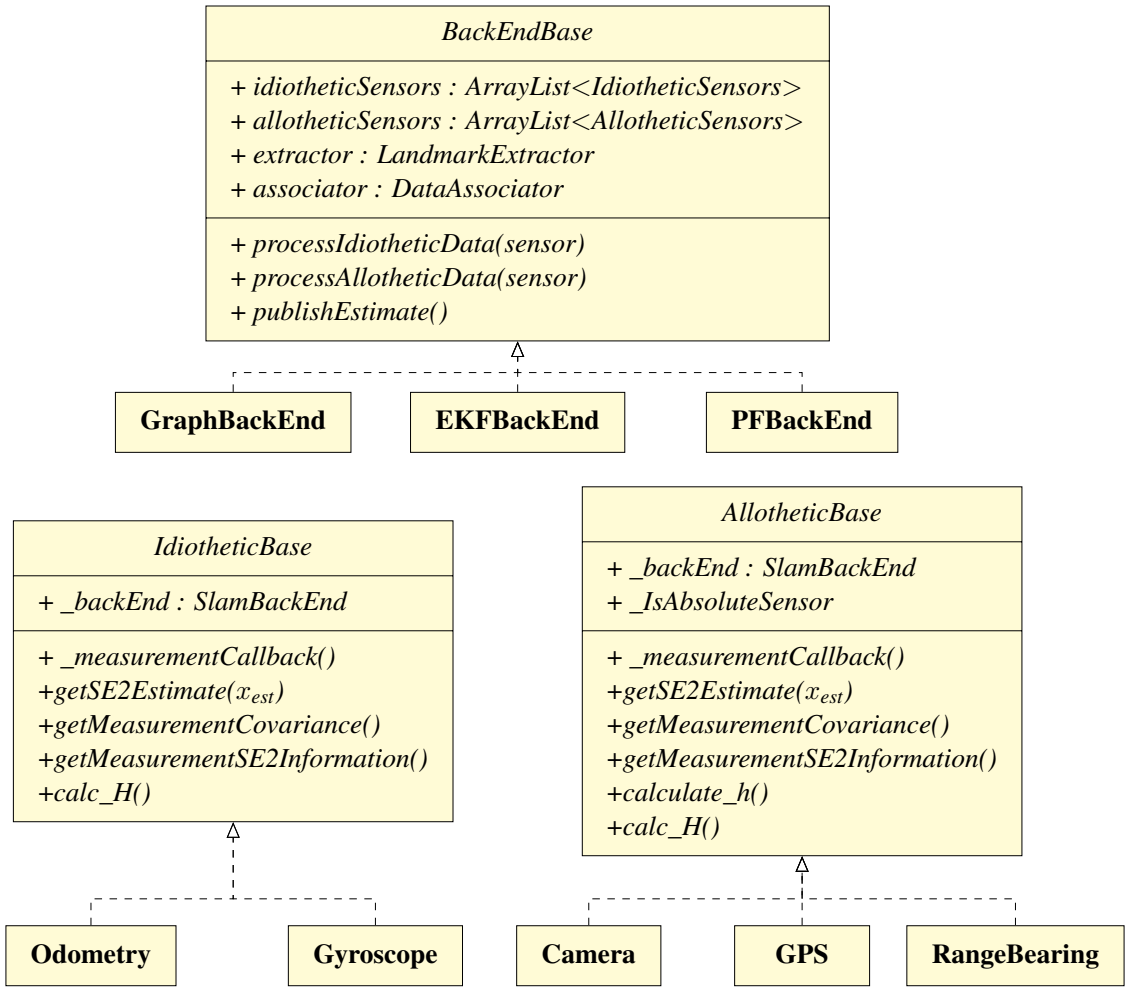


Figure 6: Class diagram of the back-end classes. The `BackEndBase` class is an abstract class, the three implemented back-ends form the implementation.

3 Detailed component description

3.1 Idiothetic sensors

Because every idiothetic sensor has to provide the same information to the back-end, it was chosen to use an abstract idiothetic base class, methods such as the motion model are sensor specific and hence implemented in the child classes. The resulting class-diagram is shown in Figure 7. As can be read in the paper, adding additional idiothetic sensors is more complicated than adding multiple allothetic sensors, for filter based approaches fundamentally require a single pose estimate and graph-SLAM requires a form of time synchronization. This is currently solved by mapping additional idiothetic sources to the pose nodes that have been generated by a sensor that is defined as the "primary" idiothetic sensor. Ideally some form of time synchronization would take place, but the proposed solution works in order to show the proof-of-concept of adding additional idiothetic sensors to graph based SLAM.

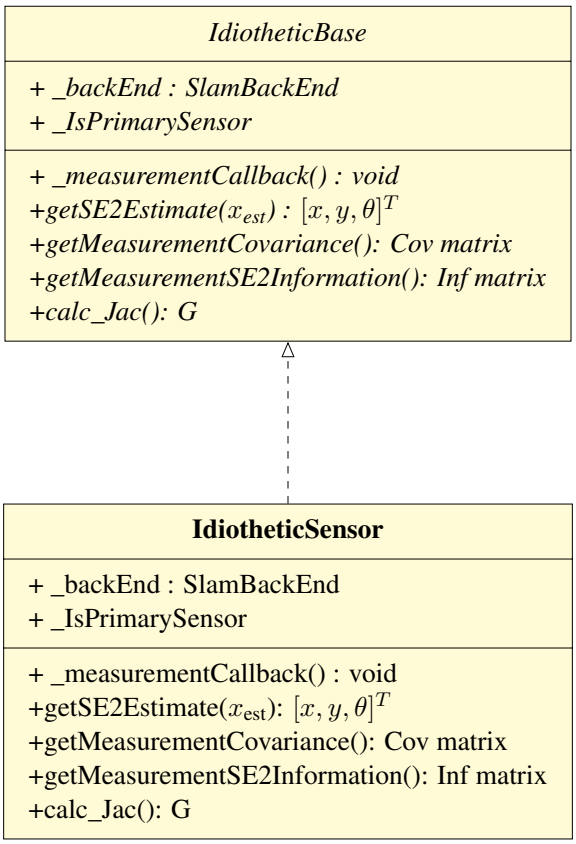


Figure 7: Class diagram of the idiothetic sensor classes. The base class contains abstract methods that and every idiothetic sensor (odometry, gyroscope) implements these and thus forms a realization of the `IdiotheticBase` class.

3.2 Allothetic sensors

For allothetic sensors, a similar construction with an abstract parent-class has been used. This results in the following class diagram, Figure 8.

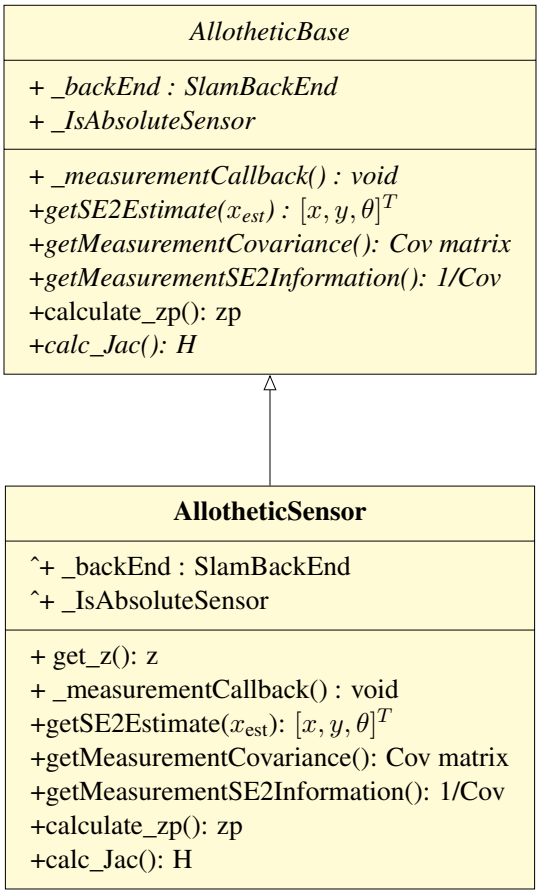


Figure 8: Schematic view of the Allothetic sensor classes to show how an allothetic, other sensors share the same methods but have their own implementations of for instance the Jacobian sensor model.

3.3 SLAM back-ends

As has been explained before, composition is used for the idiothetic and allothetic sensors, the landmark extractor and the data association class: various sensors can be added to the list of idiothetic and allothetic sensors as well as a feature extractor and association. This leads to the class diagram shown in Figure 9, an example of a sensor layout is presented in Figure 10.

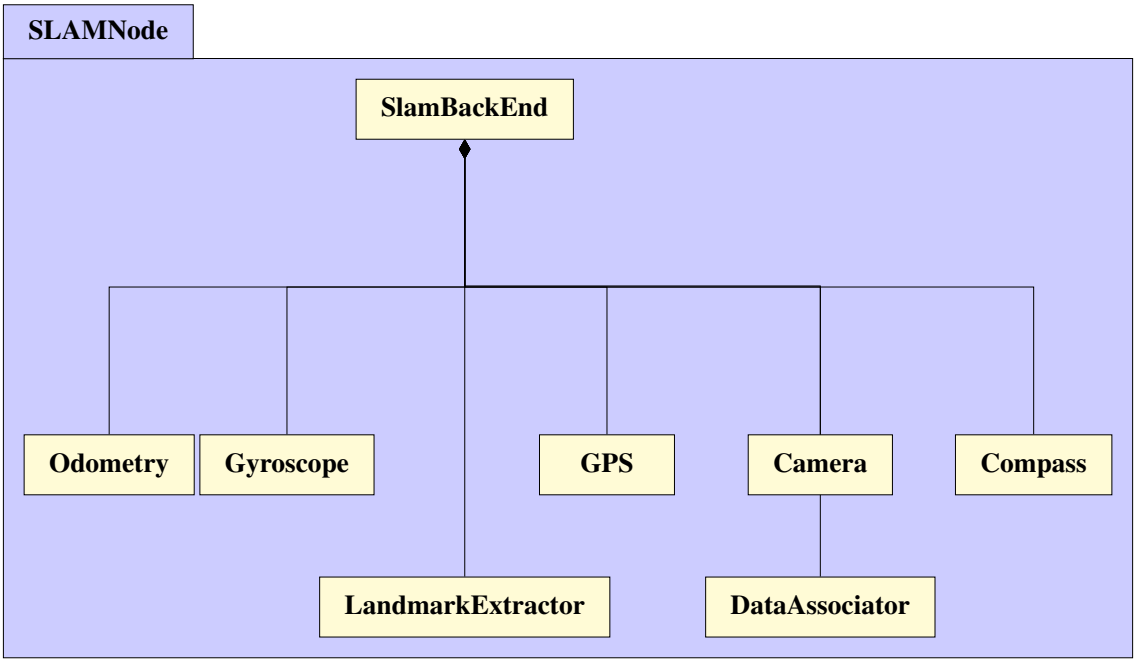


Figure 9: Class diagram of the SLAMNode with an example of a possible sensor configuration.

Also for the back-ends an abstract base class has been used as the majority of the methods are similar across the three back-ends and therefore overridden in the child-classes. This leads to the class diagram shown in Figure 11. Each of the three back-ends has its own implementation of the methods processIdiotheticData () and processAllotheticData (), the implementation of these function corresponds with the parts of the SLAM algorithm corresponding to Table 1.

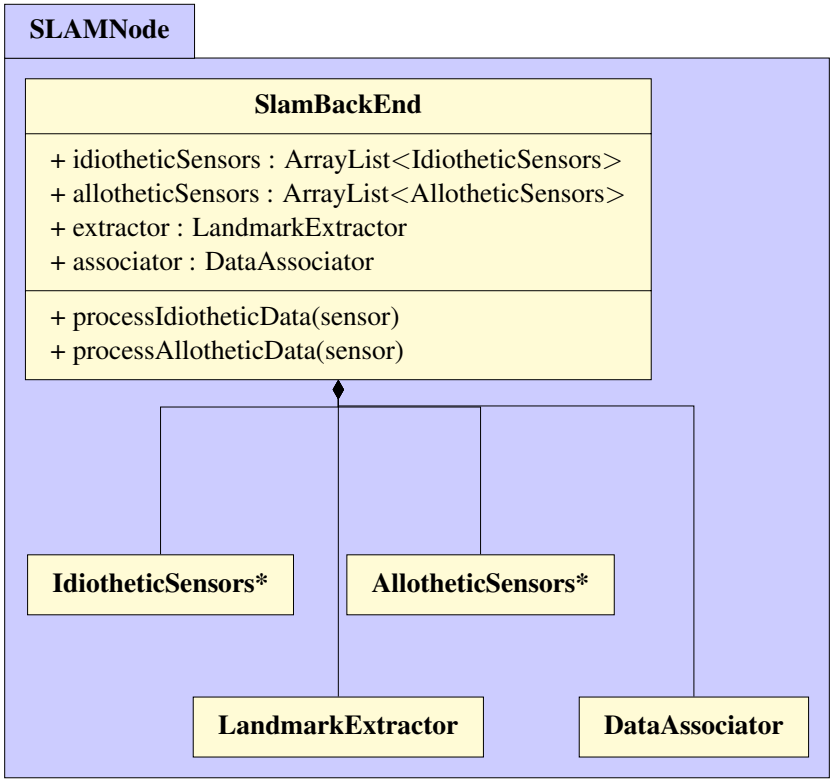


Figure 10: Class diagram of the SLAMNode. Note that in practice any of the three supported back-ends can be chosen for the `SLAMBackEnd`: `EKFSlamBackEnd`, `GraphSlamBackEnd` or `PFSlamBackEnd`
as well as any combination of (multiple) idiothetic and allothetic sensors.

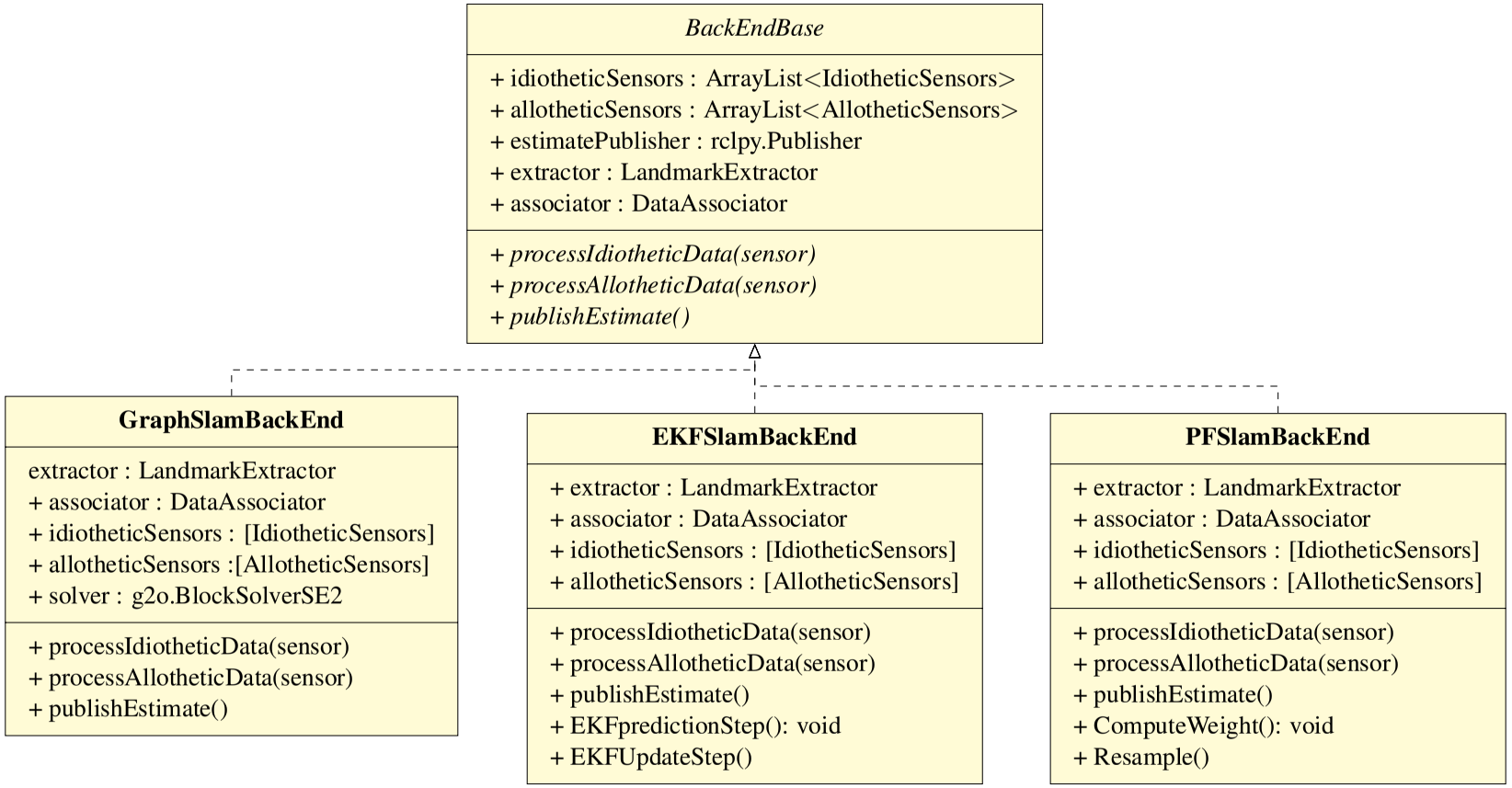


Figure 11: Class diagram of the back-end class, showing how the core methods are being shared among the three different back-ends. Each back-end has its own implementation of these methods, and possibly some additional members or methods.

3.4 Landmark extraction & Data association

Landmark extraction (and data association) are optional steps that may be required for relative allothetic sensors. These steps involve the extraction of suitable landmarks froma raw sensor observation such as a camera image or laser scan followed by data association which is the process of matching the currently observed features with the previously observed once based on either the similarity of their appearance or the likelihood of finding the feature at that specific location. Or a combination of both. Because potentially the same association or extraction algorithm can be used by various sensors, it was chosen to include these as separate components in the back-end instead of integrating these steps in the sensor code.

Whether or not association and extraction are required is known at time of implementation of the sensor in the framework and hence a sensor property. By calling the allothetic sensor's method requiresExtraction () within the processing of allothetic data, it is determined whether or not these steps are required.

3.4.1 Note regarding implementation

The implementation of extraction and association consists of a proof-of-concept using AruCO Markers. These are fiducial markers designed for robust and easy detection of features. The implementation of the OPENCV library has been used. The markers are detected in the extractor using the OPENCV library and compared with the dictionary of possible markers in the data associator. In a realistic environment, the associator continuously adds the detected features to its own dictionary of previously seen landmarks. Here the dictionary is dictated by the library and constant.

3.5 Architecture

The resulting class diagram of the architecture of the SLAM-node is shown in Figure 12. That is, without the additional supporting classes (for instance, a sensor simulator or the map node responsible for plotting the results).

4 ROS integration

As explained in the paper, the software makes use of Coppelia Sim [2] and ROS [1]. The framework that has been explained until now forms a single ROS node, the SLAMNode. This node connects to various other nodes that represent the robot, its sensors and the map. This interface is realized in the form of a publish-subscribe structure with ROS-topics. The communication diagram shown in Figure 13 gives an overview of the different ROS-nodes and how they interact. The core of the algorithm, i.e. the back-end and sensor implementations are contained within the SLAMNode. The sensors either suscribe to the topics created by the robot, or in case of a relative beacon sensor that's simulated, to an allotheticSimulator node. The output of the SLAM node, i.e. the estiamte of the pose and map, is send to the MapNode which facilitates plotting. Each back-end has its own node, so the software has a EKFBackEndNode, a GraphSLAMBackEndNode and a ParticleFilterBackEndNode. The numbers in Figure 13 correspond to the following messages:

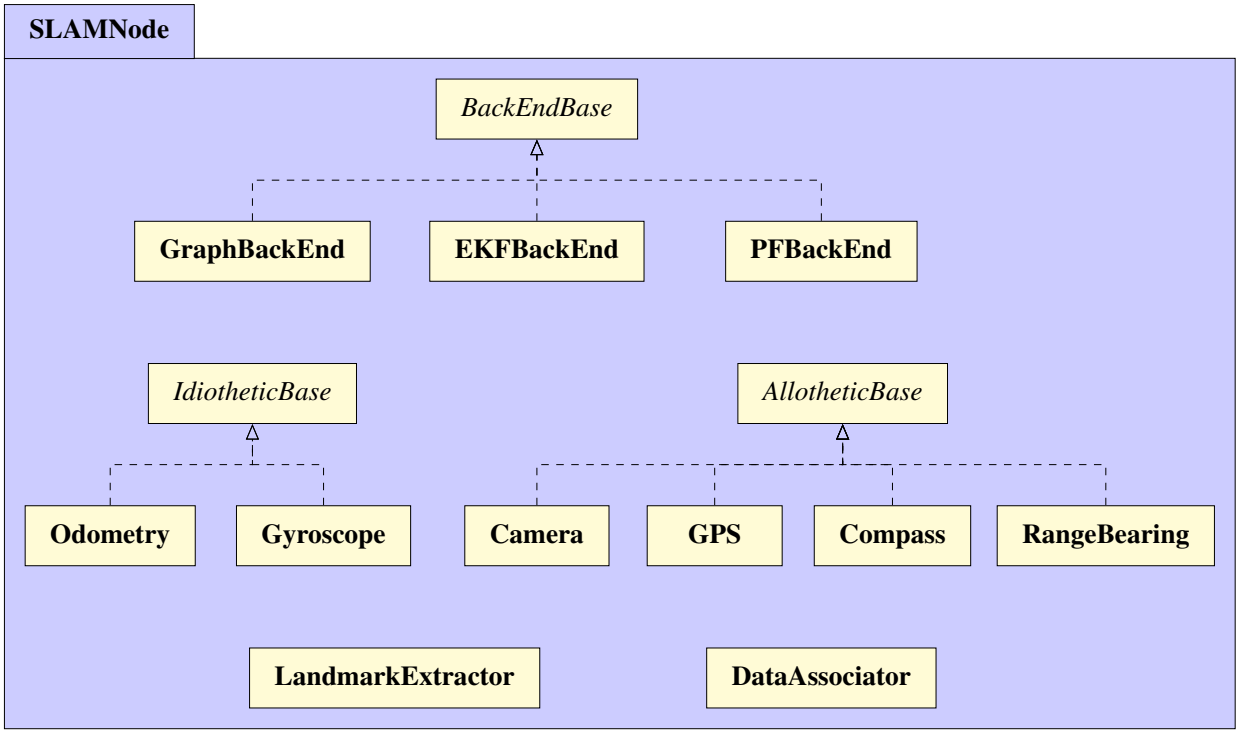


Figure 12: Class diagram of the SLAMNode showing the implemented architecture.

1. StartMsg. Used to start the simulation in CoppeliaSim automatically. Type: std_msgs/Bool
2. TruePosMsg. True robot position, used for comparing estimate and true trajectory. Type geometry_msgs/2DPose
3. SensorMsgs. Messages of the various sensors being used by the robot. Type (depends on sensor, see list below or for a detailed description, refer to appendix X which contains sensor specific documentation).
4. AllotheticMeasurementMsg. Simulated allothetic data, list of relative range-bearing measurements. Type: custom_msgs/AllotheticMeasurement
5. TrueLandmarksMsg. Coordinates of true landmarks. Type: custom_msgs/Landmarks
6. TruePosMsg. True robot pose. Type: geometry_msgs/2DPose
7. EstimateMsg. Estimated robot pose and landmark locations. Type: custom_msgs/Estimate

4.1 ROS nodes

Each of the ROS nodes is the framework is briefly discussed below.

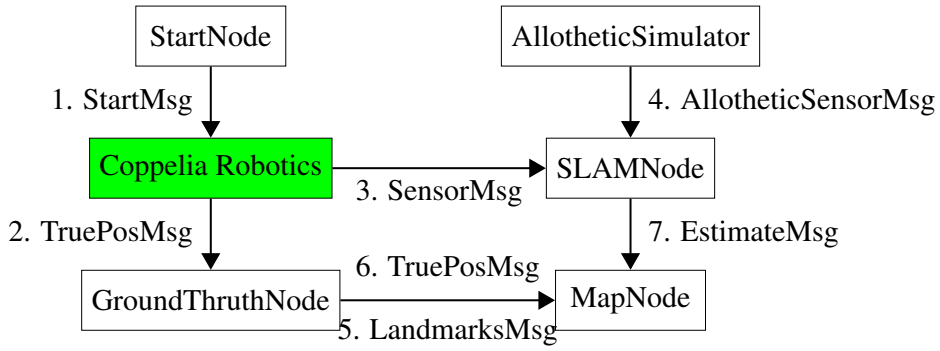


Figure 13: Overview of the different ROS-nodes that have been used for the program. Except from Coppelia Robotics (green), every block in this block-diagram is a separate ROS-node. Note that the SLAMNode consists of the back-end and several sensor objects (the exact number of which depends on the SLAM implementation). All of the data that enters the back-end goes through the sensor classes. In case landmark extraction and data association are present as well, the back-end has instances of these objects as well.

StartSim (ROS-node)
Input: -
Function: Allows starting of the simulator from Python by publishing on the StartSim topic.
Output: *Bool* StartSim ROS-message.

Allothetic Simulator(ROS-node)
Input: True landmark location, true robot pose, sensor uncertainty (Simulation parameter)
Function: Simulates a set of range-bearing measurements by using the true landmark and robot pose to calculate the relative range-bearing. Measurement noise is added to these measurements.
Output: Observation z (in this case this includes the landmark ID as we assume known association), this is implemented in the custom ROS AllotheticSensorMessage.

Ground Thruth (ROS-node)
Input: True Robot position and yaw-angle
Function: Keeps track of the true robot pose (actually maps the orientation to $[0, 2\pi]$) and generates the true landmark locations.
Output: True Robot pose, True Landmarks

Map (ROS-node)
Input: Current (online-) Estimate, True pose.
Function: Stores the full estimated and true trajectories (this might change or become more important for offline SLAM) and plots these as well as the current covariance ellipse.
Output: Plot of the result (possibly in the future the map publishes the estimate?).

Back-End (ROS-nod
Input: Sensor data via sensors (attributes of the back-end class). This includes data coming directly from Coppelia Sim and simulated data, such as the artificial AllotheticSimulator .
Function: Core of the SLAM algorithm that acts on an "on-new-data" basis.
Output: Estimate of the pose and landmarks, this is a custom ROS Estimate message.

4.2 ROS messages

Besides standard ROS messages (Float32, Point, Pose, Bool), the following custom messages have been defined to facilitate the framework. Note that for compatibility reasons, the back-end should also publish a ROS Pose message –that could be used by for instance path planning software– besides the currently used custom Estimate() message, although this is currently not being used.

RangeBearingLandmark(ROS-message)
Float32 Bearing
Float32 Heading
Int Id

RangeBearingLandmarks(ROS-message)
RangeBearingLandmark[] list

Estimate(ROS-message)
Pose x
RangeBearingLandmarks landmarks
Float32 C_x11
Float32 C_x12
Float32 C_x21
Float32 C_x22
The latter four being the four entries of the x-y covariance matrix of the state vector.

References

[1] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, 2009.

[2] E. Rohmer, S. P. N. Singh, and M. Freese. CoppeliaSim (formerly V-REP): a Versatile and Scalable Robot Simulation Framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.

Mathematical sensor Description

Jeroen Minnema

April 2020

1 Introduction

The purpose of this document is to provide some clear documentation on the sensors that have been implemented in the framework, especially regarding the mathematical models used, i.e. the measurement model and its Jacobian.

2 Odometry

Sensor type: Idiothetic
Measures: changes in angular position since the previous measurement for both the left and right motor, resp: $\Delta\alpha_l$ and $\Delta\alpha_r$.
ROS-topic: coppelia_leftWheelSpeed, coppelia_rightWheelSpeed

Measurement function $g()$ Given the diameter d and distance between the two motors L first the distance traveled by the left and right motor are calculated:

$$\Delta S_l = d\pi \frac{\Delta\alpha_l}{2\pi} \tag{1}$$

$$\Delta S_r = d\pi \frac{\Delta\alpha_r}{2\pi} \tag{2}$$

$$\tag{3}$$

By assuming that the motion is relatively small and hence that the angular arc traveled by the robot has approximately the same length as the straight distance, we obtain after some manipulations[1]:

$$\Delta S = \frac{\Delta S_l + \Delta S_r}{2} \tag{4}$$

$$d\theta = \frac{\Delta S_r - \Delta S_l}{L} \tag{5}$$

$$\Delta x = \Delta S \cos\left(\theta + 0.5 * \Delta\theta\right) \tag{6}$$

$$\Delta y = \Delta S \sin\left(\theta + 0.5 * \Delta\theta\right) \tag{7}$$

$$u = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{bmatrix} \tag{8}$$

[1] https://www.cs.princeton.edu/courses/archive/fall11/cos495/COS495-Lecture5-Odometry.pdf

Jacobian The Jacobian is calculated as follows:

$$G = \begin{bmatrix} \frac{\partial \Delta x}{\partial x} & \frac{\partial \Delta x}{\partial y} & \frac{\partial \Delta x}{\partial \theta} \\ \frac{\partial \Delta y}{\partial x} & \frac{\partial \Delta y}{\partial y} & \frac{\partial \Delta y}{\partial \theta} \\ \frac{\partial \Delta \theta}{\partial x} & \frac{\partial \Delta \theta}{\partial y} & \frac{\partial \Delta \theta}{\partial \theta} \end{bmatrix} = \begin{bmatrix} 0 & 0 & -\Delta S \sin(\theta + 0.5 * \Delta\theta) \\ 0 & 0 & \Delta S \cos(\theta + 0.5\Delta\theta) \\ 0 & 0 & 0 \end{bmatrix} \tag{9}$$

Remarks

3 Gyroscope

Sensor type: Idiothetic
Measures:
ROS-topic: coppelia_gyroData

Measurement function

Remarks Because no additional idiothetic sensors were implemented for EKF and PF SLAM, the Jacobian wasn't implemented.

4 Range-bearing sensor

Sensor type: Relative allothetic
Measures: z, a relative allothetic observation, i.e. a vector of relative range bearings and– depending on the sensor–IDs (r, θ: $z = [lm_{1,\theta}, lm_{1,r}, lm_{1,ID} ... lm_{m,theta}, lm_{m,r}, lm_{m,ID}]^T$), however for the measurement function and its Jacobian, an individual observation of landmark i is considered: $z^i = [r^i, \theta^i]$
ROS-topic: slam_allotheticMeasurement

Measurement function

Remarks

4.1 ArUco camera

Sensor type: Relative allothetic
Measures:
ROS-topic: coppelia_camera

Measurement function The measurement function $h()$ is designed for the observation of an individual landmark and is used for reconstruction, i.e. based on the current believe of the

landmark lm_i and robot statevector x_{robot}.

$$\delta = \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} = \begin{bmatrix} lm_{i,x} - x_{\text{robot}} \\ lm_{i,y} - y_{\text{robot}} \end{bmatrix} \tag{10}$$

$$q = \delta^T \delta \tag{11}$$

$$z^i = \begin{bmatrix} \sqrt{q} \\ \text{atan2}(\delta_y, \delta_x) - \theta_{\text{robot}} \end{bmatrix} \tag{12}$$

Remarks Besides the feature extraction and data association, the AruCo marker is actually just a range-bearing sensor, hence the measurement model is the same.

Jacobian Note that for the EKF and Particle filter, the Jacobian of the full statevector–the robot pose and *all* landmarks that have been observed at that point in time–is calculated. However, for all landmarks except for lm_i this results in zeros, so here we consider the low-dimensional Jacobian ${}^{\text{low}}H^i$ which consists of the partial derivatives over $(x, y, \theta, lm_{i,x}, lm_{i,y})$ by applying the chain rule this results in:

$$H = \begin{bmatrix} \frac{\partial h}{\partial x} & \frac{\partial h}{\partial m} \end{bmatrix} = \frac{1}{q} \begin{bmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & \sqrt{q}\delta_x & \sqrt{q}\delta_y \\ \delta_y & -\delta_x & -q & -\delta_y & \delta_x \end{bmatrix} \tag{13}$$

5 GPS

Sensor type: Absolute allothetic
Measures: Absolute robot position $z = [x, y]$
ROS-topic: coppelia_truePosition

Measurement function

$$z = \begin{bmatrix} x_{\text{robot}} \\ y_{\text{robot}} \end{bmatrix} \tag{14}$$

Jacobian

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{15}$$

Remarks Noise is added to the true position in order to get a realistic measurement. For a real GPS added to a robot, it would make sense to use the GPS's own accuracy estimation, mainly the dilution of precision, HDOP/VDOP[2] information instead of estimating the covariance manually.

6 Compass

Sensor type: Absolute allothetic
Measures: Absolute heading $z = [\theta]$
ROS-topic: slam_trueRobotTheta

[2] `https://en.wikipedia.org/wiki/Dilution_of_precision_(navigation)`

Measurement function

$$z = \begin{bmatrix} \theta \end{bmatrix} \tag{16}$$

Jacobian

$$H = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{17}$$

Remarks Noise is added to the true orientation in order to simulate a realistic measurement. In reality, if one would be aware of the standard deviation of the earth magnetic field at the operating location, this could be included in the measurement model.

Simulation Environment

Jeroen Minnema

April 2020

1 Introduction

The goal of this appendix is to provide some details on the simulation environment. The initial role of the simulation software was mainly as a development and testing tool as the goal was to perform the final tests on a real world robotic platform. However, unfortunately due to the COVID-19 situation during spring 2020, physical experiments were no longer possible, therefore the simulations form the prime results of this paper and showcase the functioning of the modular SLAM framework. First, the simulation environment and the connection with the SLAM framework is explained. Furthermore, a brief tutorial on how to run the demo runs is provided. Hints on compiling and installing the required software are provided on the RaM gitlab page, where the software is currently located `https://git.ram.eemcs.utwente.nl/minnemaj/msc_slam`.

2 Simulation environment

As explained in the paper, ROS is used together with CoppeliaSim. CoppeliaSim contains a simulation environment of a robot that's navigating through a building and contains a basic anti collision algorithm such that upon starting the simulation the robot will explore the environment automatically. This robot is based on the example robot in the CoppeliaSim tutorials and is equipped with various (artificial) sensors, listed in table 1. These sensors publish their data on ROS topics on which the SLAM Node is subscribed.

3 Running the Demo simulation runs

The most convenient way of running the demo examples is by running one of the ROS launch files that directly start all relevant nodes and start the simulation. Assuming one has cloned the Git repository of this project. Open CoppeliaSim by navigating to the installation directory and executing coppeliaSim.sh. Open the scene named "DemoScene.ttt". Open a terminal and navigate to the main directory: \slam_ws. Run the following commands:

Sensor type	Implemented
Idiothetic	Odometry (wheel encoder)
	IMU (Gyroscope)
Allothetic, relative	Simulated range-bearing sensor
	Aruco based range-bearing
Allothetic, abosulte	GPS (absolute x,y position)
	Compass (absolute heading)

Table 1: Overview of the sensors that have been implemented

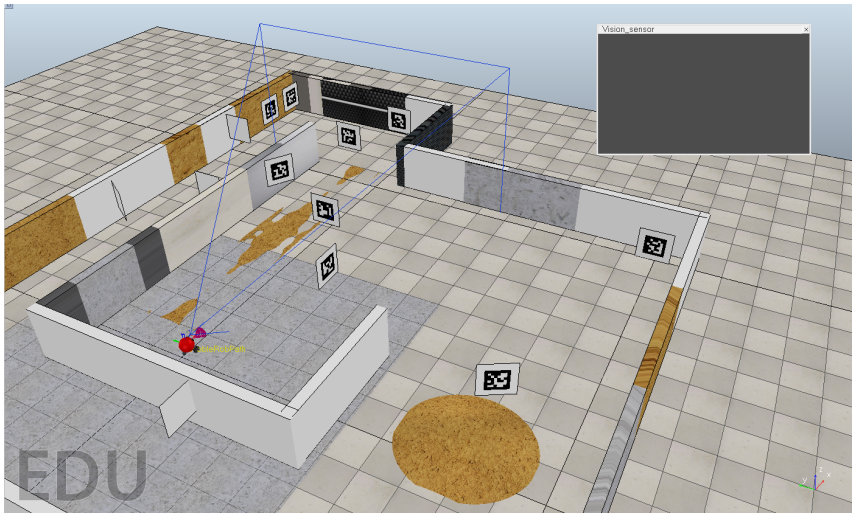


Figure 1: Screenshot of the CoppeliaSim environment

```
colcon build
. install/setup.bash
ros2 launch slam_framework launchGraphSim.launch.py
```

This launches the graph SLAM back-end. Alternatively, for the other two back-ends, launch the other two launch files: launchPFSim.launch.py and launchEKFSim.launch.py.

3.1 Changing the sensor configuration

If one desires to change the sensor configuration, this can be done by creating a sensor of the desired type in the __init__() function of the back-end and adding it to the list of sensors. Figure 2 provides a code snippet on how to do this. The README files on Git provide further details. Please note that any software changes do require the software to be rebuild before launching the simulation. This is done with colcon build while being in the root directory of the repository.

```
64        # Creating idiothetic Sensors
65        newIdiotheticSensor = OdometrySensor("coppelia_leftWheelSpeed","coppelia_rightWheelSpeed",self)
66        self.idiotheticSensors.append(newIdiotheticSensor)
```

Figure 2: Code snippet to illustrate how additional sensors can be added to change the sensor configuration

ArUco marker fiducial detection

Jeroen Minnema

April 2020

1 Introduction

This section elaborates on how AruCo markers have been used for this project and explains how the calibration process has been done and could be redone if different cameras (or a real robot) would be used in the future. Fiducial markers were chosen because they form as they provide a means of using visual SLAM with limited implementation time because a robust and efficient off-the-shelf extraction and association algorithm can be used. Another advantage of fiducial markers is that they were likely to work both in simulation and in practice. ArUco markers are fiducial markers that can be automatically generated and can be reliably detected. Furthermore an easy to use interface is provided by the OpenCV ArUco library [1]. An example of an ArUco marker is provided in figure 1

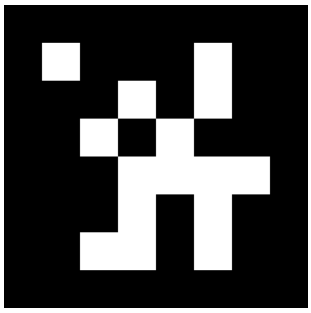


Figure 1: Example of an ArUco marker

2 Calibration with CoppeliaSim

In order to be able to estimate the distance of the detected markers, the camera parameters and the true size of the image have to be known. CoppeliaSim does not clearly provide the exact camera properties, but fortunately OpenCV provides algorithms to facilitate camera calibration with a CharUco Chessboard image. This requires a set of images of the chessboard made by orienting the camera at various angles. This was done by building a calibration scene in CoppeliaSim, figure 2 shows this. The calibration scripts that are located in the /aruco_tests/ directory are documented and explain the details of the calibration. The resulting accuracy was not investigated in great detail but measurements appeared to be within 0.1m of the true distance, which was considered to be acceptable for the required purpose.

[1] `http://www.uco.es/investiga/grupos/ava/node/26`

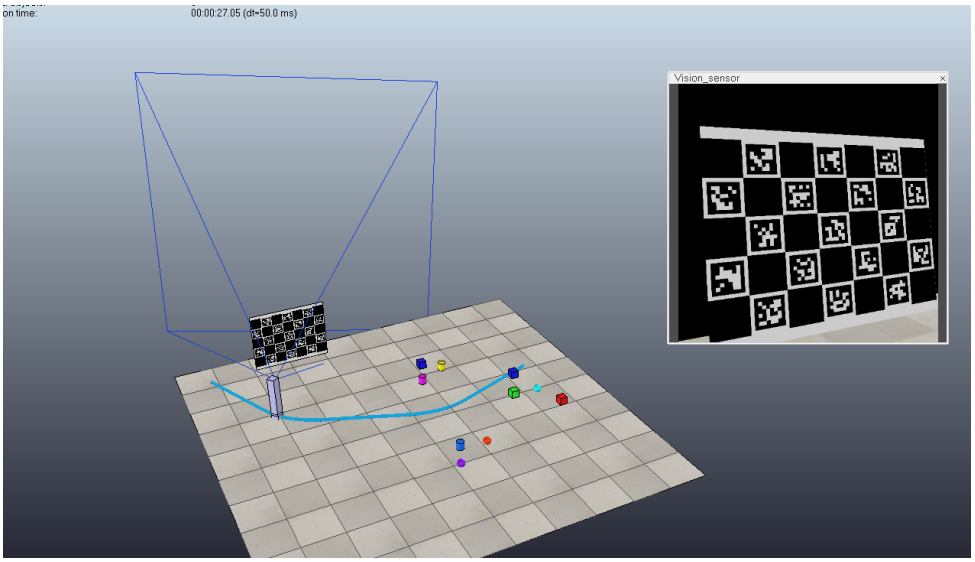


Figure 2: ArUco calibration with CoppeliaSim, the Camera is moving along the purple line to provide images from different viewing angles. The images are saved and read by a python script that performs the calibration and obtains the camera parameters.