Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.

http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.

http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Generic Low-Level Sensor Fusion Framework for Cyber-Physical Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Technische Informatik

eingereicht von

Denise Ratasich
Matrikelnummer 0826389

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Mitwirkung: Univ.Ass. Dipl.-Ing. Oliver Höftberger

Wien, 14.03.2014

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Generic Low-Level Sensor Fusion Framework for Cyber-Physical Systems

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Computer Engineering

by

Denise Ratasich
Registration Number 0826389

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Assistance: Univ.Ass. Dipl.-Ing. Oliver Höftberger

Vienna, 14.03.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Denise Ratasich
Hauptstraße 92, 7304 Nebersdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasserin)

Danksagung

Zuallererst möchte ich meinem Betreuer Prof. Dr. Radu Grosu danken, der mir die Möglichkeit eröffnet hat, eine Diplomarbeit an den mobilen Robotern des Instituts durchzuführen. An Univ.Ass. Dipl.-Ing. Oliver Höftberger ein großes Dankeschön für die produktiven Besprechungen und Diskussionen, die vielen Tipps und das gewissenhafte Korrekturlesen meiner Arbeit.

Meinem Freund Thomas möchte ich auch ganz herzlich danken, für sein Verständnis, das er im Laufe meines Studiums (z.B. für versäumte Wochenenden) entgegengebracht hat, und das finale Korrekturlesen meiner Arbeit.

Ein besonderes Dankeschön möchte ich meinen Eltern aussprechen, die mir meinen Bildungsweg erst ermöglicht haben, und mich all die Jahre unterstützt haben.

Abstract

Sensors usually suffer from imprecision and uncertainty, e.g. measurements are corrupted by noise. Additionally the observations of a sensor may be incomplete, i.e. a single sensor possibly does not cover the entire range of a measured variable. To overcome these problems, sensor fusion techniques are often applied. Sensor fusion combines measurements to improve the description of a property in a system, i.e. to provide data of higher quality.

Although various sensor fusion algorithms exist, there is only few literature comparing the different methods, and still less frameworks collecting sensor fusion algorithms. Several implementations of commonly used algorithms exist but are difficult to configure and less applicable.

The objective of this thesis is the design and implementation of a generic framework that integrates various sensor fusion algorithms with a common interface. The implementation is build on top of the Robot Operating System, which enables a wide usage of the framework. The implemented methods can be configured to combine an arbitrary number of sensors and can be easily integrated into an existing application. The focus within this thesis lies on low-level sensor fusion, i.e. fusion that is processing raw sensory data. In particular so-called state estimation algorithms are discussed. Such fusion methods use a system model to combine the measurements to minimize the error between actual and estimated state of the system.

The accuracy of the algorithms are experimentally evaluated and compared, using a dead-reckoning navigation application on a robot (i.e. the position is estimated considering the translation and rotation of the robot). Several sensors and different configurations are investigated. The parameters of the methods are tuned to reach the best possible accuracy and guidelines for configuring sensor fusion algorithms are stated.

Kurzfassung

Sensoren sind meist unpräzise und unsicher, da die Messungen durch Rauschen verfälscht werden. Überdies liefern Sensoren oft nur ausreichend genaue Werte in einem Teilbereich der zu messenden Größe. Diese Probleme können durch Sensor Fusion bewältigt werden. Sensor Fusion kombiniert Messungen von Sensoren um eine bessere Darstellung einer gemessenen Größe zu erreichen, d.h. Sensor Fusion erhöht die Qualität der Daten. Algorithmen, die Sensor Fusion oder im Allgemeinen Data Fusion implementieren, verbessern zum Beispiel Genauigkeit, Zuverlässigkeit und Vollständigkeit einer beobachteten Größe.

Obwohl viele Sensor Fusion Algorithmen existieren, gibt es nur wenig Literatur, die die verschiedenen Methoden vergleicht und umso weniger Programmbibliotheken, die mehrere Methoden implementieren. Bestehende Implementierungen von üblichen Algorithmen sind meistens schwer zu konfigurieren und für wenige Applikationen anwendbar.

Diese Arbeit befasst sich mit dem Design und der Implementierung eines generischen Frameworks, das verschiedene Sensor Fusion Algorithmen mit identischem Interface integriert. Die Implementierung basiert auf dem Robot Operating System, damit wird eine breite Anwendung des Frameworks ermöglicht. Die implementierten Methoden können für eine beliebige Anzahl an Sensoren konfiguriert werden und in bestehende Applikationen einfach integriert werden. Es wird speziell auf Low-Level Sensor Fusion eingegangen, die rohe Sensordaten verarbeitet und somit die Basis für ein zuverlässiges System ist. Im Detail werden sogenannte State Estimation Algorithmen behandelt. Diese verwenden ein Modell des Systems um die Sensordaten so zu kombinieren, dass der Fehler zwischen geschätztem und tatsächlichem Zustand des Systems minimiert wird.

Die Genauigkeit der Algorithmen wird in einer Test-Applikation experimentell ermittelt und verglichen. Im Zuge dieser Test-Applikation schätzt ein Roboter seine aktuelle Position mittels verschiedener Sensoren und unterschiedlicher Konfigurationen. Den Abschluss bilden Richtlinien und Tipps zur Konfiguration von State Estimation Algorithmen.

Contents

CHAPTER 1

Introduction

More and more embedded systems emerge in various application fields. In comparison to e.g. a personal computer, embedded systems are much less visible computers [17] e.g. controlling a motor, monitoring a chemical process, displaying images on a navigation system or processing radio signals. Embedded systems interact with physical processes consisting of several components. One important group of components are sensors which bridge the physical world with the embedded computers.

Sensors measure properties or attributes of an entity of the system, i.e. variables of interest. These system variables are often physical properties of the system, e.g. the current number of revolutions of a motor or the temperature of a fluid. Such measured variables are often continuous with an infinite range of values which are transformed to discrete values of limited resolution by sensors.

However as soon as a system interacts with its environment several problems arise [8, 21]:

imprecision Sensors cannot provide an exact description of a (physical) property in a system.

Sensors have their own system dynamics, distortions and are corrupted by noise [20]. So sensors always have a specific amount of imprecision, i.e. the value measured by a sensor may vary around the actual value of the entity. Further the amount of imprecision might also depend on the actual value.

E.g. an incremental encoder measuring the speed of a wheel has limited resolution. An encoder provides pulses which can be counted to evaluate the wheel velocity. The number of pulses provided by the encoder per 360° is finite. So the slower the velocity the more inaccurate the measured value gets (there are less pulses to count in a period).

uncertainty Sensors may not provide all attributes of an entity or measurements might be ambiguous.

So in contrast to imprecision, uncertainty depends on the entity observed and not on the sensor. E.g. a simple motion detector connected to an alarm system cannot distinguish between a human or an animal.

non-coverage One sensor may not cover the required range of a property completely.

Limited coverage is often the case with range sensors. An infrared distance sensor may only provide useful values in a specific range due to its structure and power of its components, e.g. from 2 to 30 cm or 10 to 80 cm. This is known as limited *spatial* coverage. On the other side e.g. a sonic range sensor has limited *temporal* coverage. A sonic sensor measures the propagation time of a sonic wave to an obstacle and back to calculate the distance. So when measuring the depth of a lake the update of the distance cannot be made e.g. every ms, more likely every couple of seconds (depending on the actual depth of the lake).

complexity Safety critical systems will use more sensors to build up a more dependable and robust system. This leads to a system with higher complexity which is harder to design and test. So safety cannot be guaranteed that easy.

Imagine the *Active Brake Assist* (a collision avoidance system) of a car. A sensor measures the distance to an obstacle, e.g. another car or a person. An *outlier*, i.e. a high deviation of the measured value w.r.t. the actual value, of this sensor could cause the car to perform an emergency brake without any obstacle far and wide. On the other side the correct functionality can save lives, hence improving the reliability and dependability in safety critical systems is an important topic.

The above example shows that malicious or even inaccurate sensors can cause the system to operate in a wrong way. One cannot trust a sensor without knowing the *certainty* of the measured value. Additional arrangements must be accomplished to overcome the inaccuracy, unreliability and the possibilities of failures.

1.1 Motivation

Sensor fusion combines sensory data to provide a representation of an entity of higher quality [30]. Such algorithms fuse sensor measurements to increase accuracy, reliability, certainty and/or completeness.

Sensor fusion can address the problems stated in the previous section. A system implementing sensor fusion gains performance in several ways [21]:

representation A sensor fusion method with several inputs operates as an abstraction layer. Hence it relieves the complexity of the control application. The output of this so-called *fusion block* provides a richer semantic than its inputs. E.g. the raw measurements of radar and image sensors given by a set of pixels are first combined to get the position in x,y coordinates of a tracked object and then passed to the control application of the system. So the measurements are first fused to a more "useful" value (e.g. the position of the tracked object) for the controller.

Additionally some sensor fusion methods provide a value representing the confidence, e.g. the standard deviation of the output.

certainty The belief in sensor data increases when propagating through a fusion method, i.e. the probability of a specific sensor value representing a property after fusion is higher than that before fusion.

Fusion methods are often used to combine different sensors using different technologies but measuring the same system variable. Such sensor configurations can be used to overcome systematic errors, e.g. offset errors. The certainty of the entity's value increases too, because the sensor values match the actual value better.

Certainty in contrast to accuracy is a more generic term and includes to consider many types of errors (e.g. measurement errors caused by noise and errors affected by environmental factors). An accurate sensor may not provide certain data about a specific entity, e.g. a high speed traffic camera providing noise reduced images (the accurate sensor data) might not be enough to distinguish a truck from a camper seen from above (the type of car is the entity to observe).

completeness Every method brings additional knowledge and therefore a more complete view on the overall system.

Several sensors with different ranges can be combined to cover the possible values of a property. So fusion methods address the problem of non-coverage.

Further a sensor fusion may address the problem of failed sensors. When a sensor measuring property A fails, a fusion method can provide the value of A as long there are sensors measuring the same property or properties where A can be derived from. So failed sensors can be compensated by a fusion method.

The range of application fields is broad. Sensor fusion may be used in medical, robotic, industrial or transportation systems. Because sensor fusion increases reliability and certainty it is well suitable for dependable reactive systems [8].

1.2 Objectives

The aim of this thesis is to work out a generic framework of several sensor fusion methods which can be easily configured and used in various applications.

A first goal of this thesis is to develop a C++ library collecting several sensor fusion methods. So the user of the library has various possibilities to combine available measurements. Therefore various sensor fusion methods will be investigated and sorted out to fit best for low-level sensor fusion, i.e. fusing raw sensor data to get data of higher quality. The methods will be studied and implemented as C++ classes. All methods will be implemented to work in a generic way, i.e.

the methods are designed to be exchangeable among themselves. A common interface will be defined and used throughout the framework.

A second big goal is the development of a configuration method. The framework is tested and used on a robotic platform running the *Robot Operating System* (ROS) [25]. Hence the configuration includes generating a ROS node, instantiating a sensor fusion method and initializing it regarding to the user's needs. The aim of the configuration part is to increase usability and minimize configuration time.

As a final goal the sensor fusion methods will be analyzed. Advantages, disadvantages, computational efficiency and application fields will be figured out. Finally the methods are applied to a specific use case (dead-reckoning navigation) and compared w.r.t. accuracy considering specific sensors and configurations.

1.3 Outline

The next chapter introduces sensor fusion and discusses different types of sensor fusion. Furthermore, some definitions and basics about systems, states and statistics are presented.

Chapter 3 reveals the theory and algorithm of the most popular low-level fusion methods. The conclusion of Chapter 3 will cover a comparison of the proposed fusion methods regarding application fields and complexity among other properties.

Chapter 4 describes the overall design and architecture of the sensor fusion library. Implementation details of the fusion algorithms are stated and the integration into the robot operating system [25] is shown.

In Chapter 5 the algorithms are analyzed and benchmarked with a navigation application. Accuracy, speed and memory usage are compared. This section also includes general guidelines and tips for configuring the parameters for a sensor fusion method.

A summary and outlook on future work is given in Chapter 6.

CHAPTER 2

Introducing Sensor Fusion

Data fusion is the combination of data acquired from multiple sources and related information from associated databases such that the result is in some sense better than it would be without such synergy exploitation [12, 21, 30]. *Sensor fusion* is a specialization of data fusion where the source of data are sensor measurements.

Different types of sensor fusion are distinguished. The purpose or function of the fusion may vary and so the techniques do. Traditional disciplines like digital signal processing, statistical estimation and control theory are used but also artificial intelligence and numerical methods are applied in several fusion methods [12].

There is no fusion method which fulfills all needs in one algorithm, in general the technique to use highly depends on the purposes of the fusion method and the application. The methods are often combined and hierarchically structured, e.g. in navigation the position may be first estimated using statistical methods and then passed to a neural network deciding the next position to drive.

The next section introduces the various types and classifications of sensor and data fusion in general. In Section 2.2 the focus lies on fusion of raw sensor data and the needed theory for subsequent chapters is summarized.

2.1 Types and Classifications

Sensor fusion or data fusion in general can be divided into several levels regarding the purpose and expected outcome of a fusion method [8, 12]:

low-level fusion or *raw data fusion* combines the raw sensor data to be of better quality and more informative.

A low-level fusion method transforms data to consistent units and frames (known as *alignment*), assigns attributes to entities (*association*), estimates or predicts attributes of entities (*estimation*) and refines the estimates of an object's identity or classification (*refinement*) [12].

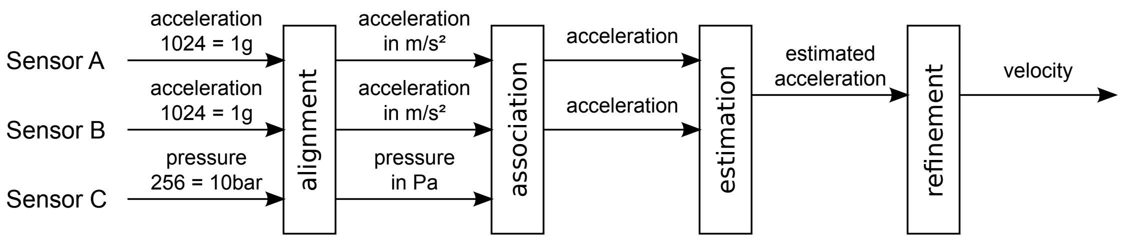

Figure 2.1: Example of low-level fusion.

In the example of Figure 2.1 three sensors measure acceleration and pressure of an entity, e.g. a car's tyre. The aim of the fusion is to estimate the velocity of the wheel. So first the values from the sensors (e.g. results from an analog to digital converter) are transformed to standard units (*alignment*). Then the measurements which can be used for calculating the velocities are selected (*association*). The velocity cannot be derived from the measurements of a pressure sensor, so only the observations from the accelerometers are used. The estimation combines all suitable sensor data (the accelerations) to form an estimate of higher accuracy (*estimation*). Finally the velocity is calculated from the estimated acceleration to provide a more useful value for e.g. a control application (*refinement*).

intermediate-level fusion or *feature-level fusion* extracts representative features from sensor data, e.g. for estimating or classifying an identity.

Feature-level fusion methods extract contextual descriptions out of the (estimated) data, i.e. it determines the meaning of the data. Considering the above low-level fusion example, a feature-level method picks the information e.g. "car is moving or not" from the velocity or "tyre should be pumped up" from the pressure.

high-level fusion or *decision fusion* interprets the collected data, e.g. features or properties of an entity, and involves this data into the overall control application.

Considering the tyre example ones more, fusing the features "tyre should be pumped up" and "car is moving" may trigger an alarm message on the bord computer's display (decision).

For low-level sensor fusion often statistical methods are used, e.g. sequential estimation, batch estimation and maximum likelihood. Feature extraction is often done using knowledge-based systems, e.g. fuzzy logic, logical templates and neural networks. Neural networks are also used in high-level fusion [12, 14].

Another classification is given by [6] and is also discussed in [8, 21]. The sensor fusion methods can be distinguished by the way the sensor data is combined (see Figure 2.2, where entities A, B and C are measured by sensors S_1 to S_5).

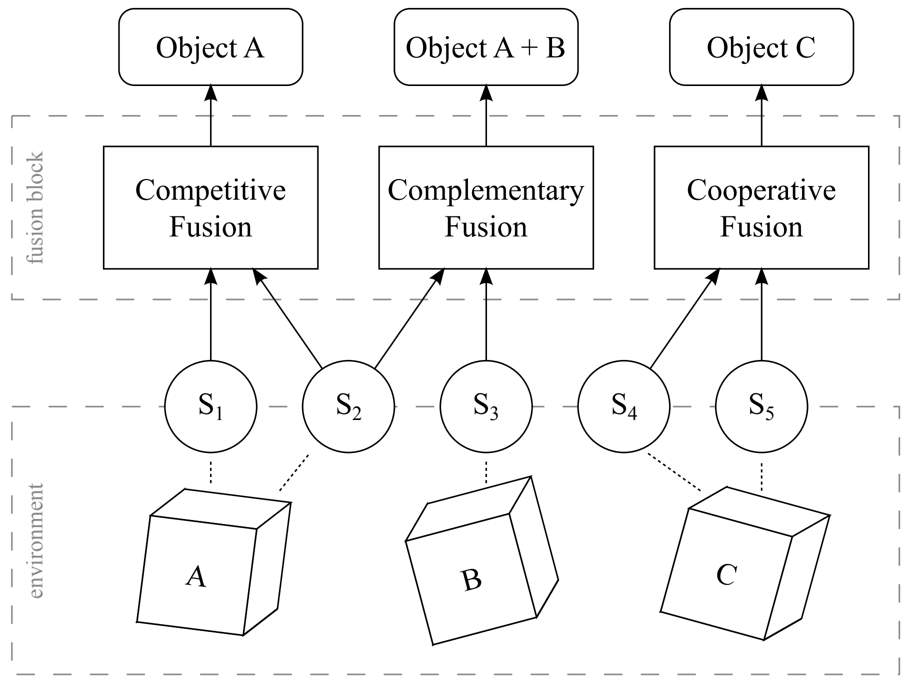

Figure 2.2: Types of sensor configuration [8].

competitive When sensors measure the same property the configuration is called *competitive*. The sensors are then called *redundant*.

These methods increase *accuracy*, *reliability* and *robustness* [21].

complementary In a so-called *complementary* configuration the sensors are combined to give a more complete representation of an entity.

E.g. an array of distance sensors at the front and at the back of a mobile robot gives a more complete view of possible obstacles around the robot.

Fusion methods of this type resolve *incompleteness*.

cooperative In a *cooperative* configuration two independent sensors are fused to derive a specific information which would not be available with a single sensor.

E.g. the position of an object can only be detected by at least two distance sensors through triangulation. One sensor would not be sufficient to specify the location.

Considering above example, one would agree that the accuracy is defined by the sensor with the poorest accuracy. Hence these methods are very sensitive to the individual information sources and *decrease accuracy* and *reliability* [8].

W.r.t. a fusion system's input/output characteristic another classification by Dasarathy can be made, see Table 2.1.

Name	**Description**
Data Input/Data Output	Input data is smoothed or filtered.
Data Input/Feature Output	Features are generated out of the input data.
Feature Input/Feature Output	Features are reduced or new features are generated.
Feature Input/Decision Output	Features are fused to decisions.
Decision Input/Decision Output	Input decisions are fused to output decisions.

Table 2.1: Dasarathy's Fusion Model [21].

This classification is similar to the one of [12]. The car tyre example can be applied here too. The acceleration and pressure corresponds to the data input whereas the velocity corresponds to the data output. So low-level fusion is kind of data input/data output. Data input/feature output ("car is moving or not" is extracted out of the velocity) and feature input/feature output maps to feature-level fusion. Decision fusion includes feature input/decision output ("car is moving" and "pressure low" leads to an alarm) and decision input/decision output.

2.2 Low-Level Sensor Fusion

Regarding the types and classifications given in the previous chapter the library implemented throughout this thesis contains *low-level fusion* or *Data Input/Data Output* algorithms, i.e. the algorithms are intended to process raw sensor data and output filtered results.

The algorithms can be used with various sensor configurations. The proposed configurations may be combined too. The chosen configuration depends on the application and the available sensors.

The methods explained in Chapter 3 have a common underlying theory. As mentioned in the previous section low-level fusion algorithms are often based on statistical methods. So basic terms and concepts of control theory and statistics are summarized in the next sections. For further introductions to probability theory w.r.t. robotics see [26, 27]. Various sensor fusion methods are explained in [21].

2.2.1 Systems and States

The *world* or *environment* is assumed to be a dynamic system which is characterized by its *state* x [27].

$$x = \begin{pmatrix} x_1 \\ x_2 \\ : \\ x_n \end{pmatrix}$$

The *state variables* $x_1, .., x_n$ represent properties of interest of the system where n is called the *state size*. E.g. in a tracking application the position of an object is of interest, therefore the state vector may look like $x = (x_o \, y_o \, z_o)^T$ where x_o,y_o and z_o define the position of the tracked object in a 3-dimensional space. Further variables of interest may be speed and heading of the object.

A system can be described by its *state-space model* [18], e.g. the differential equations describing the dynamics of the system. The state-space model is represented by the *state transition model* and *measurement model*.

state transition model This is the next state function f of a system (sometimes also called *process model*). It maps the previous state x_{k-1} and additional inputs u_{k-1} to the current state x_k considering the process noise w.

$$x_k = f(x_{k-1}, u_{k-1}) + w_{k-1} \tag{2.1}$$

x_k represents the actual state at time step k. u is the *control input* and provides information about the *change* of the state. The process noise w models the uncertainty of the state transition model. Note, the model is only an abstraction of the real world. E.g. take a robot driving in a room. The pose of the robot is a typical state variable. Also the location of objects or people in the room could be contained in the state vector. The robots desired velocity may be the control input.

Applying the state transition model is also called *time update*. Considering the previous state and additional control inputs the *expected* state is "predicted" with the state transition model. E.g. the expected position of a tracked object (current state x_k) can be updated with its velocity and heading when taking the elapsed time into account.

measurement model This is the output function h. It maps the current state x_k and additional inputs to the measurements z_k, i.e. the response of the system.

$$z_k = h(x_k) + v_k \tag{2.2}$$

v is called the measurement noise and it models the noise characteristics or accuracy of the sensors. E.g. a robot stops 20 cm in front of a wall (the state), then the range sensor at the front should measure a distance of 20 cm to the wall (expected measurement $h(x_k)$). But as sensors are noisy the actual value may deviate by v_k, e.g. the sensor gives us 21 cm instead of 20 cm.

Note that the noise is assumed to be purely additive, i.e. the noise is not contained in the functions f or h and is simply added to the expected state or measurement respectively. This is in general the case. However when using these models the process and measurement noise are always assumed to be independent from each other.

The above functions form the physical model for the estimation algorithms in the framework developed in this thesis. To find the state transition and measurement model, the system's differential equations have to be determined. Differential equations of a dynamic system are generally of the following vector form:

$$\frac{d}{dt}x(t) = f(x(t), u(t))$$

But for the estimation, the *solution* of the above form for the discrete case is needed, where the next state is a function of the current state.

$$x_k = f(x_{k-1}, u_{k-1})$$

This form can be achieved by solving the differential equations numerically e.g. with Matlab and replacing dt with T.

If a model is linear, the functions f and h simplify to the matrices A, B and H respectively. So the state transition model may be written as

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \tag{2.3}$$

and the observation model as

$$z_k = Hx_k + v_k \tag{2.4}$$

When the state x is of size n and the number of measurements equals m, then A is of size $n \times n$ and H of size $m \times n$. If the size of u equals l, B is of size $n \times l$. So the state transition model expands to

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}_k = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}_{k-1} + \begin{pmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,l} \\ b_{2,1} & b_{2,2} & \dots & b_{2,l} \\ \dots & \dots & \dots & \dots \\ b_{n,1} & b_{n,2} & \dots & b_{n,l} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_l \end{pmatrix}_{k-1} + \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}_{k-1}$$

and the linear observation model to

$$\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{pmatrix}_k = \begin{pmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,n} \\ h_{2,1} & h_{2,2} & \dots & h_{2,n} \\ \dots & \dots & \dots & \dots \\ h_{m,1} & h_{m,2} & \dots & h_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}_k + \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{pmatrix}_k$$

Hence x_1 is a linear combination of x_1, x_2 to x_N first (x_1 depends on state variables where the appropriate coefficient $a_{1,i} \neq 0$). Furthermore the state change of x_1 is added with a linear combination of the control inputs u_1 to u_L. So these two terms give the expected value of x_1. Finally random noise w_1 is added to model the uncertainty of x_1.

Example. Considering a moving train where its velocity should be estimated and two sensors measuring the acceleration and velocity of the train are available for fusion (there are no control inputs which may be used). The sensors are sampled every 100ms. So the velocity and acceleration (the state variables) may be updated every $T = 100ms$ with following state transition model:

$$\begin{aligned} v &= v + a \cdot T \\ a &= a \end{aligned}$$

The velocity is integrated from the acceleration which is assumed to be constant. The acceleration cannot be predicted or described by other state variables, hence the above formula $a = a$ is the best option possible. Note the model is held simple and is not perfect, e.g. the acceleration is assumed to be "sluggish". But this uncertainty about the model can be featured by the process noise (more in Chapter 5).

However the sensors directly reflect the velocity and acceleration respectively so the measurement model evaluates to

$$\begin{aligned} m_v &= v \\ m_a &= a \end{aligned}$$

Estimating the state $x = (v\, a)^T$ and the measurement vector $z = (m_v\, m_a)^T$ with a Kalman filter the state-space model is described by (without noise)

$$\begin{aligned} x_k &= \begin{pmatrix} 1 & T \\ 0 & 1 \end{pmatrix} x_{k-1} \\ z_k &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x_k \end{aligned}$$

2.2.2 Probability Fundamentals

Problems of sensors, unpredictable environments, inaccurate models and limited computation rise the uncertainty of the system's state. So rather relying on the "best guess" of the state, the uncertainty is described by probability distributions [27].

So instead of describing a variable as a single value (e.g. the expected state) one can use the *probability density function* (PDF) to specify the possibility of an *event*, i.e. a variable takes on a specific value. The PDF gives the probability of each (possible) value of the variable. E.g. a temperature sensor in a room at 25 degrees above zero will sense 25 degree with very high probability. But the sensor may be inaccurate so a value of 23 or 27 degree is possible too. Sensing the value of 10 degree is very unlikely.

Such probabilities are described by nonnegative numbers. The sum of the probabilities for all possible events equals one. E.g. the probabilities to dice a '1','2',...,'6' is $1/6$, the probability to roll one of these numbers (regardless which one) is 1.

A variable specified by a PDF is called a *random variable* (RV). So the probability of a RV X to take on the value x (event $X = x$) is the PDF evaluated at x or short $p(x)$. The PDF can be characterized by the mean and variance of the random variable.

mean The most likely value is called the *mean* or the *expected* value $\mu = \hat{x} = \mathbb{E}\{X\}$ of RV X. The mean is the average of samples taken from a random variable.

variance The spread of the PDF can be described by the *variance* $\sigma^2 = \mathbb{E}\{(X - \mu)^2\}$. Sensors tend to float around its mean caused by noise. The higher the variance the more measurements deviate from the mean. So a low variance features an accurate sensor.

E.g. the amount of water in a tank (the random variable X) is sensed 1000 times and assumed to have a Gaussian distribution [27]. The average of the measurements (concrete values x_1 to x_{1000}) evaluates to 200 liters. So the expected volume of water is 200 liters (mean of X). The average of the squared deviations of x_1 to x_{1000}, i.e. the variance, results in 2 liters. Then the volume might be described as the function in Figure 2.3.

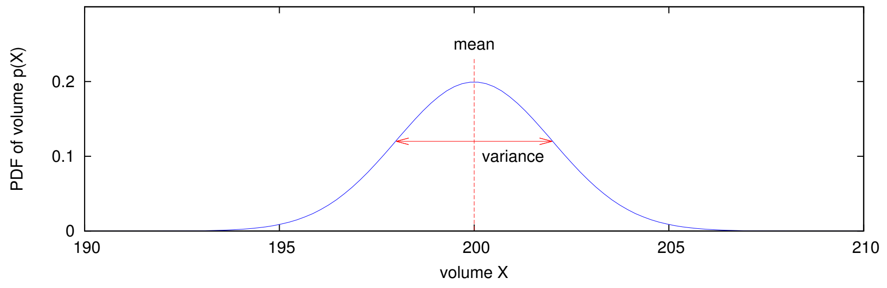

Figure 2.3: The probability distribution of the volume in a tank.

Often RVs carry information about other RVs, i.e. the random variables are *not independent* from each other. A probability denoted by

$$p(x|y) = p(X = x|Y = y)$$

is called *conditional probability*. It specifies the probability of $X = x$ when the value of a RV Y is already known ($Y = y$). E.g. the measurements offset, or bias respectively, of a MEMS accelerometer increases with the temperature due to the expansion of its mechanical structure. In other words, when the temperature raises e.g. by 30 K than the mean of the acceleration increases, i.e. the acceleration's PDF changes. Hence the probability of the acceleration is conditioned on the temperature.

An important rule with conditional probabilities is *Bayes rule* [27].

$$p(x|y) = \frac{p(y|x)\, p(x)}{p(y)} \tag{2.5}$$

Probabilities are not bounded to the 1-dimensional space, so one can extend a random variable to a *random vector*. A random vector is simply a collection of random variables.

covariance The covariance describes the spread of the PDF of a random vector x, i.e. $Cov(X) = \mathbb{E}\{(X - \mu)^2\}$. The term covariance is similar to the variance, but describes the squared expected deviation for the higher dimensional case. The covariance is a symmetric matrix where the coefficients $c_{i,j}$ and $c_{j,i}$ describe the dependency between two random variables X_i and X_j.

$$C = \begin{pmatrix} \sigma_1^2 & c_{1,2} & \dots & c_{1,n} \\ c_{2,1} & \sigma_2^2 & \dots & c_{2,n} \\ \dots & \dots & \dots & \dots \\ c_{n,1} & c_{n,2} & \dots & \sigma_n^2 \end{pmatrix}$$

E.g. the dependency between random variable X_1 and X_2 are given by $c_{1,2}$ and $c_{2,1}$. A coefficient of the diagonal of the matrix represents the variance of the appropriate random variable. When all variables are independent from each other the covariance evaluates to

$$C = \begin{pmatrix} \sigma_1^2 & 0 & ... & 0 \\ 0 & \sigma_2^2 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & \sigma_n^2 \end{pmatrix}$$

2.2.3 State Estimation

The purpose of the sensor fusion methods developed in this framework is to estimate the state of the system out of sensor data. Each variable of the models proposed in Section 2.2.1 are no longer exact nor specific values, rather they are random variables or more generally random vectors. These random vectors (state x, measurements z, control inputs u) can be described by their distribution [27].

In the system model proposed in Section 2.2.1 the current *belief*, i.e. the internal knowledge about the system's state, depends on the available data namely measurements and controls.

$$\begin{aligned} bel(x_k) &= p(x_k | z_1, z_2, .., z_k, u_1, u_2, .., u_{k-1}) \\ &= p(x_k | z_{1:k}, u_{1:k-1}) \end{aligned}$$

Note that the use of u_k or u_{k-1} varies in the literature. Here it is assumed that the control input is set right *after* measuring z and evaluating x. Hence for estimating x_k controls up to u_{k-1} are available.

The sensor fusion methods described in Chapter 3 are so-called recursive state estimation algorithms. *Recursive* because these algorithms use the previous state to estimate the current one and need not to log the entire history of states and inputs (*Markov assumption* [27]). The underlying theory for recursive state estimation is the Bayes filter [26,27]. (A *filter* uses previous and actual measurements to estimate the state. So the state estimators are henceforth also called filters.)

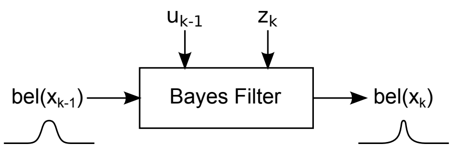

Figure 2.4: Bayes Filter.

The aim of the Bayes filter is to *update* the belief incorporating latest inputs (measurements, controls). In general each measurement decreases the uncertainty of the belief, i.e. the error covariance of the state's PDF shrinks with a new measurement (see Figure 2.4). But after some iterations the covariance of the state converges to a specific value (bounded by process and measurement noise covariance) as long the model and noise covariances do not change.

The evaluation of the current belief is divided into two steps depicted in Figure 2.5. First the belief of the next state is *a priori* calculated out of the belief of the previous state x_{k-1} and the control input u_{k-1}, the so-called *time update* or *prediction*. Next the belief is updated by injecting the new measurement z_k, which is called the *measurement update* or *correction*.

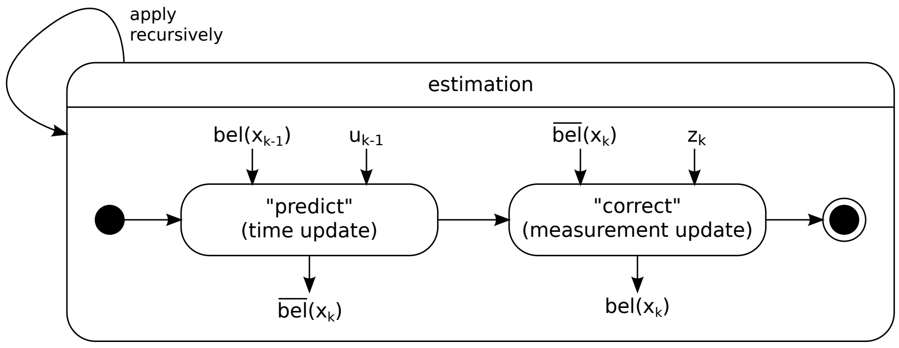

Figure 2.5: Breakdown of a recursive state estimator.

To perform the prediction and correction the state-space model introduced in Section 2.2.1 is used. The belief is passed through the state transition model f to form the a priori state's distribution $\overline{bel}(x_k)$. E.g. a time update in robot navigation considers the time elapsed and the last position of the robot to estimate the current position. Next the a priori belief is multiplied by the probability that the actual measurement may have been observed using the measurement model (considering the a priori estimated state).

Note that using the state-space model for the update of the belief is not the only option for Bayes filters [3]. However the filters in the next chapter use the state transition and measurement model to predict and correct the belief.

CHAPTER 3

Sensor Fusion Methods

This chapter will introduce several recursive state estimation methods derived from the Bayes filter described in the previous section. For each method theory and algorithm is stated and pros and cons are listed.

Finally an evaluation regarding computational efficiency, accuracy and implementation is stated in Section 3.5. The last section covers other fusion methods and explains why they are not included to the list of estimation algorithms to implement within this thesis.

3.1 Kalman Filter

The Kalman filters are the most popular state estimation algorithms. The method was published in 1960 by R.E.Kalman and is studied in lots of papers. The basic idea is to represent the state as a Gaussian random variable (GRV), where its distribution or density respectively can be fully described by the mean and covariance. So instead of propagating the complete distribution of the state for the next state's estimation only the mean and covariance have to be recalculated [20,32].

3.1.1 Theory

The Kalman filter uses the physical model of a system (sometimes called the *plant*) to estimate a statistical variable. For this problem the system is described by difference equations, i.e. the *state-space model* [32].

The original form of the Kalman filter only works with linear systems. In the previous chapter the state-space model is introduced, represented by two functions f and h. Because the state-space model here is linear, one can simplify the model to matrix multiplications [18].

$$\begin{aligned} x_k &= Ax_{k-1} + Bu_{k-1} + w_{k-1} \\ z_k &= Hx_k + v_k \end{aligned}$$

Algorithm 3.1: Kalman Filter.

```
input  : state transition model $A$, $B$,
         measurement model $H$,
         process and measurement noise covariance $Q$, $R$,
         previous state $\hat{x}_{k-1}$ and covariance $P_{k-1}$,
         control input $u_{k-1}$, measurements $z_k$
output: state $\hat{x}_k$, covariance $P_k$ after sensing measurements $z_k$

  // time update
1 $\hat{x}_k(-) = A\hat{x}_{k-1} + Bu_{k-1}$;
2 $P_k(-) = AP_{k-1}A^T + Q$;
  // measurement update
3 $K_k = (P_k(-)H^T)(HP_k(-)H^T + R)^{-1}$;
4 $\hat{x}_k = \hat{x}_k(-) + K_k(z_k - H\hat{x}_k(-))$;
5 $P_k = (I - K_kH)P_k(-)$;
```

A is the *system* or *transition matrix* which relates the previous state x_{k-1} to the current state x_k. The *input matrix* B relates the control input u to the state x. H is called the *output* or *observation matrix*. It relates state x to the measurements z.

Considering the following assumptions, the Kalman filter is optimal, i.e. it minimizes an error of actual and estimated state, namely the error covariance [20]:

- The state is a Gaussian random variable.
- Process and measurement noise are independent from each other.
- The process and measurement noise are white and Gaussian with process noise covariance Q and measurement noise covariance R.

$$p(w) \sim \mathcal{N}(0, Q)$$
$$p(v) \sim \mathcal{N}(0, R)$$

The matrices A, B, H, Q and R might vary over time. Here they are assumed to be constant to avoid nonessential time step indices k.

Assuming an initial mean $\hat{x}_0$ and error covariance P_0 (the initial belief of the state), the belief can be updated using the Kalman filter algorithm stated in Algorithm 3.1. The estimation can be divided into two parts, the *time update* where the next state is "predicted" and the *measurement update* which "corrects" the predicted state by incorporating the actual measurements.

time update The first step in a Kalman filter is to predict the next state by applying the state transition model (line 1). The resulting state is called the *a priori* state $\hat{x}_k(-)$. It is the expected state of the system when considering the previous state $\hat{x}_{k-1}$ and the control input

u_{k-1}. Line 2 of the algorithm updates the state's covariance. The state transition matrix is applied to calculate the covariance of the a priori state w.r.t. the linear transformation. Adding the process noise covariance features the uncertainty of the a priori calculated state.

measurement update Line 4 evaluates the *a posteriori* state, i.e. the a priori state corrected by the measurements. The weighted difference of actual z_k and expected measurement $H\hat{x}_k(-)$ is the term which corrects $\hat{x}_k(-)$. So the higher K_k the more will be trusted the actual measurement. On the other hand when the state's covariance $P_k(-)$ tends to zero, i.e. the current state has a very low variance, the measurements will be almost ignored ($K_k \to 0$). K_k is called the *Kalman gain* and minimizes the a posteriori error covariance [32]. Finally the state's covariance is corrected (line 5). The additional information gained by the measurements should decrease the covariance [27].

Note that the calculation order is important, the updated K_k is used for the a posteriori state estimate x_k. The next step should be measuring (to get an actual z_k) and then updating the a posteriori estimates.

Figure 3.1 shows the belief of the state in the two stages of the Kalman filter. The belief models the position of an entity moving to the right (along the x-axis).

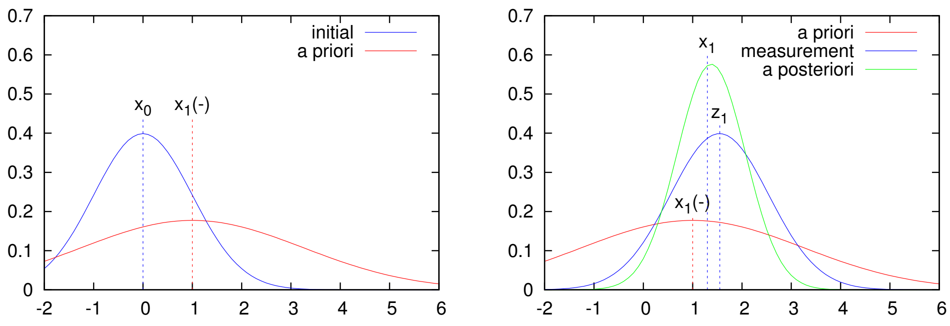

Figure 3.1: Update of the beliefs in prediction (left) and correction (right).

The initial position x_0 is 0 with variance 1. The time update calculates the expected position $x_1(-)$ to be 1. In general the time update increases the uncertainty or error covariance respectively, i.e. $P_1(-) > P_0$, because the process noise covariance is added. An increased covariance is featured by a flattened Gaussian probability distribution of the a priori state. Measurements generally decrease the state's covariance during the first filter iterations. After few iterations (depending on the initial belief) the state covariance and Kalman gain converge to values bounded by the process and measurement noise covariance.

Tutorials, more detailed explanations and the derivation of the Kalman filter can be found in [20, 27, 32].

3.1.2 Pros and Cons

Advantages.

- Optimal, i.e. the estimated state is the best possible estimate, under certain conditions (e.g. Gaussian PDFs).
- Very simple.
- Little computational complexity in comparison to other state estimation methods.

Disadvantages.

- Only suitable for linear models.
- The state is assumed to be Gaussian distributed. This works well for unimodal state distributions, i.e. the state estimate represents a *single* guess with specific uncertainty. States with multimodal beliefs cannot be estimated with the (original) KF. E.g. a robot might have three guesses where it is located, featured by three "peaks" in the PDF [27].
- Restriction to zero-mean Gaussian noise. This is not a big problem at all because in most cases the distribution of the noise is unknown and can be assumed to be Gaussian.

3.1.3 Applications

The above proposed Kalman filter can only be used when the system is linear. Linear systems are often desirable because they are easier to manipulate and the system theory is more complete and practical [20]. However this assumption is rarely fulfilled [27], e.g. the pose estimation of a robot is nonlinear because the robot can drive on circular paths.

Kalman filters are often used for position, kinematic and attribute estimation [12], e.g. tracking of a target's position and velocity, or simple tracking of a specific state variable, image processing, navigation and statistics in economy. In fact every system characterized by a continuous state and noisy measurements can be handled using a Kalman filter (with the assumptions and restrictions discussed above) [26].

3.2 Extended Kalman Filter

Many applications cannot be restricted to linear models, e.g. estimating the position of a robot. Applying a nonlinear function to a Gaussian belief distorts the belief and the KF approach is not applicable any more (all random variables need to be Gaussian and stay Gaussian). The extended Kalman filter (EKF) overcomes this problem by linearizing the nonlinear functions of the model [27].

3.2.1 Theory

The extended Kalman filter can estimate states of nonlinear systems (introduced in Section 2.2.1).

$$\begin{aligned} x_k &= f(x_{k-1}, u_{k-1}) + w_{k-1} \\ z_k &= h(x_k) + v_k \end{aligned}$$

This is done by modeling the system as *locally* linear, i.e. the nonlinear state-space functions f and h are approximated by a tangent at the mean of the current belief (see Figure 3.2).

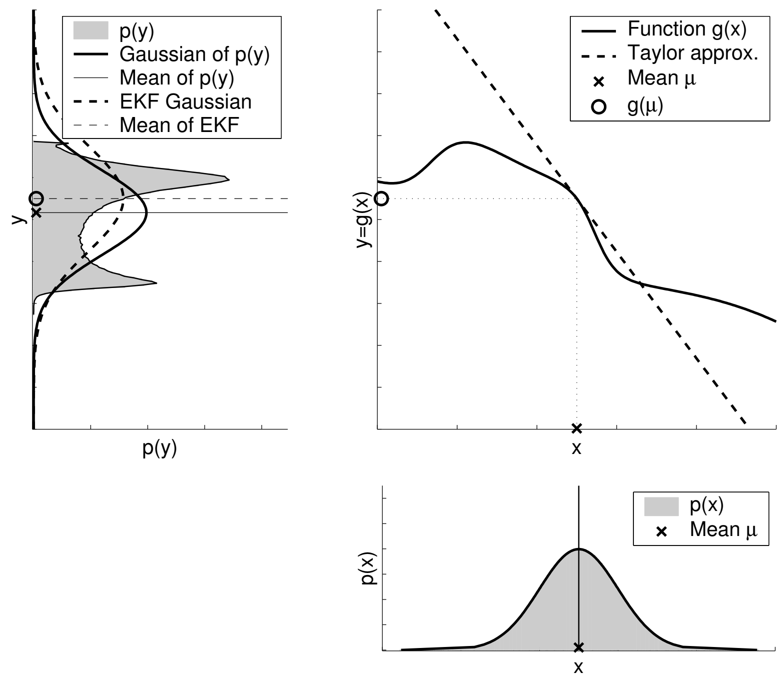

Figure 3.2: Linearization applied by the EKF [27].

The PDF of a Gaussian random variable X (lower right graph) passed through a nonlinear function g (upper right graph) results in a non-Gaussian posterior PDF $p(y)$ (shaded curve in the upper left graph). This non-Gaussian PDF $p(y)$ is approximated by passing the prior PDF $p(x)$ through the *tangent* (dashed line in upper right graph) at the current mean.

Of course the linearization produces an error (cf. the deviation of means in the upper left graph of Figure 3.2). But this error can be kept small as long the uncertainty of the state is low, i.e. a flattened belief is more affected by non-linearities. Furthermore a high degree of non-linearity or multimodal functions f or h may give a poor approximation [27].

The EKF uses the first order Taylor expansion to linearize the state transition model f and and measurement model h. The first order Taylor expansion of $f(x_{k-1}, u_{k-1})$ at the previous posterior estimate $\hat{x}_{k-1}$ evaluates to

$$\begin{aligned} f(x_{k-1}, u_{k-1}) &\approx f(\hat{x}_{k-1}, u_{k-1}) + f'(\hat{x}_{k-1}, u_{k-1}) \cdot (x_{k-1} - \hat{x}_{k-1}) && (3.1a) \\ &= f(\hat{x}_{k-1}, u_{k-1}) + A_k(x_{k-1} - \hat{x}_{k-1}) && (3.1b) \end{aligned}$$

and $h(x_k)$ is approximated at the a priori estimate $\hat{x}_k(-)$ with

$$\begin{aligned} h(x_k) &\approx h(\hat{x}_k(-)) + h'(\hat{x}_k(-)) \cdot (x_k - \hat{x}_k(-)) && (3.2a) \\ &= h(\hat{x}_k(-)) + H_k(x_k - \hat{x}_k(-)) && (3.2b) \end{aligned}$$

The derivatives vary (note the time index k at the derivatives A and H), because they are evaluated considering the current mean of the state. These matrices, once calculated, consist of concrete values describing the tilt of the linearization at the current mean.

So additionally to the state space functions the derivatives of f and h have to be specified. A_k is called the *Jacobian* of f and is defined by the partial derivatives of f with respect to x. H_k represents the first derivative of h, which is the Jacobian matrix of the partial derivatives of h with respect to x.

$$A_{k[i,j]} = \frac{\delta f_{[i]}}{\delta x_{[j]}}(\hat{x}_{k-1}, u_{k-1}) \tag{3.3}$$

$$H_{k[i,j]} = \frac{\delta h_{[i]}}{\delta x_{[j]}}(\hat{x}_k(-)) \tag{3.4}$$

The basic algorithm is stated in Algorithm 3.2 which almost equals the one from the Kalman filter (Algorithm 3.1). But instead of the linear system matrices the appropriate approximations are used. For a derivation of the EKF see [32] or [27].

time update The value of A, the Jacobian of f, varies over time (the state and control input may change), so this matrix has to be calculated first (line 1) to provide a linear approximation of f at the current mean or estimate respectively.

Line 2 predicts the a priori state estimate $\hat{x}_k(-)$ with the nonlinear state transition function. Although f is approximated by Equation 3.1b, the Jacobian A is not needed to calculate $\hat{x}_k(-)$, because the approximation coincides with f exactly at the current mean x_{k-1} used to evaluate $\hat{x}_k(-)$ (A is calculated at the current mean x_{k-1}).

$$\begin{aligned} \hat{x}_k(-) &\approx f(\hat{x}_{k-1}, u_{k-1}) + A_k \overbrace{(\hat{x}_{k-1} - \hat{x}_{k-1})}^{0} \\ &= f(\hat{x}_{k-1}, u_{k-1}) \end{aligned}$$

A Gaussian distribution remains Gaussian only if it is propagated through a linear function. So for calculating the a priori covariance the (current) approximation of f namely A_k is used (line 3). Hence A_k is (only) used to update the spread of the belief.

Algorithm 3.2: Extended Kalman Filter.

```
input  : state transition model $f$ and its derivative (to $x$) $f'$,
         measurement model $h$ and its derivative (to $x$) $h'$,
         process and measurement noise covariance $Q$, $R$,
         previous state $\hat{x}_{k-1}$ and covariance $P_{k-1}$,
         control input $u_{k-1}$, measurements $z_k$
output: state $\hat{x}_k$, covariance $P_k$ after sensing measurements $z_k$

  // time update
1 $A_k = f'(\hat{x}_{k-1}, u_{k-1})$;   // calculate the Jacobian of $f$ w.r.t. $\hat{x}_{k-1}$
2 $\hat{x}_k(-) = f(\hat{x}_{k-1}, u_{k-1})$;
3 $P_k(-) = A_k P_{k-1} A_k^T + Q$;
  // measurement update
4 $H_k = h'(\hat{x}_k(-))$ ;     // calculate the Jacobian of $h$ w.r.t. $\hat{x}_k(-)$
5 $K_k = P_k(-) H_k^T (H_k P_k(-) H_k^T + R)^{-1}$;
6 $\hat{x}_k = \hat{x}_k(-) + K_k(z_k - h(\hat{x}_k(-)))$;
7 $P_k = P_k(-) - K_k H_k P_k(-)$;
```

measurement update Like A, H varies over time (because the state varies over time). So this matrix H has to be calculated in every estimation cycle too.

The formulas in line 5 to 7 are again the same as in the original Kalman filter. But instead of the nonlinear measurement model the approximation of h namely H is used.

The models and noise covariances may change over time. So one could also (mentally) add an index k to each function and variable f, f', h, h', Q and R.

Note that the process and measurement noise is modeled purely additive (see definition of the model in Section 2.2.1) which is often the case (w or v are not included in f or h respectively). For the general case additional Jacobians (W and V) have to be calculated, and the formulas slightly adapted ($Q \rightarrow W_k Q W_k^T$ and $R \rightarrow V_k R V_k^T$) [31].

3.2.2 Pros and Cons

Advantages.

- Optimal, under the conditions listed for the KF (Gaussian RVs, independent zero-mean Gaussian noise), although its only an approximation of the optimality of the Bayes rule [32].
- Simple.
- Computational complexity is the same as the linear Kalman filter.
- Suitable for nonlinear models.

- Large state space possible - compared to Bayes filters using a sampling approach, e.g. the particle filter (Section 3.4).

Disadvantages.

- The state is assumed to be Gaussian distributed. This works well for unimodal state distributions [27].
- Only first order approximation of the nonlinear model. When the belief is additionally multimodal, i.e. the state's PDF has more than one peak, the approximation can be poor. The more nonlinear the model the more important is a small covariance of the state, a wider distribution is affected more by non-linearities [27].
- Derivatives of the state-space model needed.
- Restriction to zero-mean Gaussian noise. This is not a big problem at all because in most cases the distribution of the noise is unknown and can be assumed to be Gaussian.
- Computational effort and memory requirements higher than KF.

3.2.3 Applications

The range of applications increases considerably compared to the original Kalman filter, because the system is allowed to be of nonlinear type. But the fields stay the same, i.e. wherever a state should be tracked. Concrete examples of nonlinear applications are pose estimation and navigation.

3.3 Unscented Kalman Filter

The unscented Kalman filter is another technique to estimate the state of a nonlinear system. As within the EKF the state is assumed to be a Gaussian random variable (GRV) and therefore is also fully described by its mean and covariance. But unlike the EKF the state is propagated through the original nonlinear system. Hence a better approximation of the GRV's distribution is reached without sustaining a loss of performance.

3.3.1 Theory

Basically the predict and update cycle of the KF/EKF remains the same. Instead of linearizing the state space and observation model to get the a posteriori state, a sampling technique called unscented transform (UT) is used.

Unscented Transform

The unscented transform calculates the statistics of a random variable which is transformed by a nonlinear function [13]. In particular the UT chooses samples which completely feature a Gaussian random variable (e.g. state x to estimate). So a smaller number of samples is mostly sufficient to describe the true probability of the random variable (cf. particle filters, see next section).

Assume the Gaussian random vector x of size L has mean $\hat{x}$ and covariance P_x. $2L + 1$ samples called *sigma* vectors χ_i are chosen according a scaling parameter λ [31]:

$$\chi_0 = \hat{x} \tag{3.5a}$$
$$\chi_i = \hat{x} + (\sqrt{(L+\lambda)P_x})_i \qquad i = 1, ..., L \tag{3.5b}$$
$$\chi_i = \hat{x} - (\sqrt{(L+\lambda)P_x})_{i-L} \qquad i = L+1, ..., 2L \tag{3.5c}$$

Hence the samples are chosen around the mean w.r.t. the square root of the covariance, which corresponds to the standard deviation (see Figure 3.3).

Next the samples are propagated through the *true* nonlinear function (*transformation*)

$$\xi_i = f(\chi_i), \qquad i = 0, ..., 2L \tag{3.6}$$

to get the posterior mean and covariance of the random variable calculated by averaging the samples or squared deviations respectively.

$$\hat{y} = \sum_i W_i^{(m)} \xi_i \tag{3.7}$$
$$P_y = \sum_i W_i^{(c)} (\xi_i - \hat{y})(\xi_i - \hat{y})^T \tag{3.8}$$

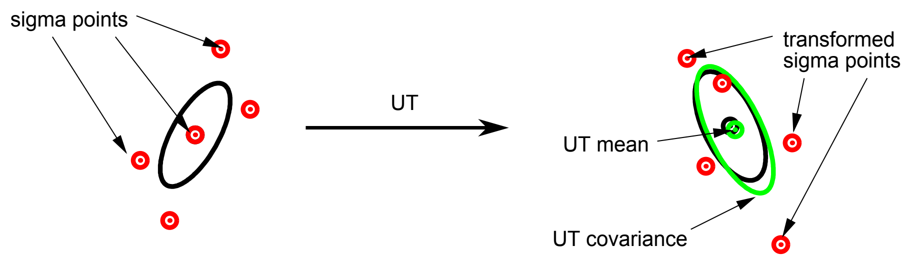

Figure 3.3: Mean and covariance propagation using UT [31].

Additionally the cross-covariance can be calculated (e.g. for the measurement update in the UKF).

$$P_{xy} = \sum W_i^{(c)} (\chi_i - \hat{x})(\xi_i - \hat{y})^T \tag{3.9}$$

The weights for the mean $W_i^{(m)}$ and covariance $W_i^{(c)}$ are defined by

$$W_0^{(m)} = \frac{\lambda}{L+\lambda} \tag{3.10a}$$

$$W_0^{(c)} = \frac{\lambda}{L+\lambda} + 1 - \alpha^2 + \beta \tag{3.10b}$$

$$W_i^{(m)} = W_i^{(c)} = \frac{1}{2(L+\lambda)} \qquad i = 1, ..., 2L \tag{3.10c}$$

The scaling parameter λ is given by

$$\lambda = \alpha^2(L+\kappa) - L$$

where α defines the spread of the samples and is usually set to a small positive value (e.g. $10^{-4} \leq \alpha \leq 1$). κ is typically set to $3 - L$. β incorporates knowledge about the distribution, e.g. for a Gaussian distribution $\beta = 2$ is optimal [31].

Lets abbreviate the UT with the expression

$$[\hat{y}, P_y] = UT(\hat{x}, P_x, f)$$

or

$$[\hat{y}, P_y, P_{xy}] = UT(\hat{x}, P_x, f)$$

when the cross-covariance is needed too.

The UKF Approach

Like the Kalman filter and extended Kalman filter, the UKF can be split into two parts, namely *time update* (“predict”) and *measurement update* (“correct”). The UKF approach is summarized in Algorithm 3.3 and described below.

time update Within the unscented Kalman filter, the UT selects sigma points χ_i w.r.t. the previous mean $\hat{x}_{k-1}$ and covariance $P_{x,k-1}$ with associated weights W_i. Each sample is transformed by the state transition model f and weighted to form the new a priori state estimate $\hat{x}_k(-)$ (line 1).

To get the approximation of the a priori covariance, the covariance of the noise has to be added (line 2).

measurement update In line 3 the a priori state $\hat{x}_k(-)$ is propagated through the observation model h by the unscented transform to get the expected measurements.

The Kalman gain is defined by the measurement covariance and the cross-covariance of state and measurement (P_{xz} is the approximation of $P_x H^T$, P_z^* corresponds to HP_xH^T, cf. KF). Out of the Kalman gain and the new measurements represented by z_k the estimate for time step k is evaluated as with KF and EKF.

The UKF algorithm proposed is valid for system models with purely additive noise. For universal noise the formulas for the covariance and Kalman gain have to be slightly adapted [31].

Algorithm 3.3: Unscented Kalman Filter.

```
input  : state transition model $f$,
         measurement model $h$,
         process and measurement noise covariance $Q$, $R$,
         previous state $x_{k-1}$ and covariance $P_{k-1}$,
         control input $u_{k-1}$, measurements $z_k$
output: state $x_k$, covariance $P_k$ after sensing measurements $z_k$

  // time update
1 $[\hat{x}_k(-), P_x]$ = UT ($\hat{x}_{k-1}$, $P_{k-1}$, $f$);
2 $P_k(-) = P_x + Q$;
  // measurement update
3 $[\hat{z}_k, P_z^*, P_{xz}]$ = UT ($\hat{x}_k(-)$, $P_k(-)$, $h$);
4 $P_z = P_z^* + R$;
5 $K_k = P_{xz} P_z^{-1}$;
6 $x_k = x_k(-) + K_k(z_k - \hat{z}_k)$;
7 $P_k = P_k(-) - K_k P_z K_k^T$;
```

3.3.2 Pros and Cons

Advantages.

- Optimal.
- Simple.
- Good accuracy (third order, cf. EKF) at low computational complexity (cf. other sampling techniques, e.g. particle filters). The system model is not linearized, but used in its original form which justifies the improved accuracy compared to the EKF.
- State-space model used “out of the box” (cf. EKF which needs the Jacobians of the state-space functions).
- Best choice Kalman filter when models are highly nonlinear.

Disadvantages.

- The state is assumed to be Gaussian distributed. This works well for unimodal state distributions but can be poor with multimodal distributions [27].
- Restriction to zero-mean Gaussian noise. This is not a big problem at all because in most cases the distribution of the noise is unknown and/or can be assumed to be Gaussian.
- Computational effort and memory requirements higher than KF.

3.3.3 Applications

The unscented Kalman filter has the same underlying theory and restrictions as the KF and EKF, so the applications are the same, e.g. tracking, navigation and parameter estimation.

Regarding nonlinear models the UKF is an improvement to the EKF w.r.t. configuration and accuracy, although the EKF works well in most applications. The problems show up when trying to implement this filter e.g. on a microcontroller where time and space is rare (the effort for computation and requirements for memory are higher compared to the KF). Then the model should be tried to held linear, e.g. one could possibly filter with a "reduced" state-space model, i.e. a model containing only state variables which are linked linearly. Other entities are calculated outside of the filter.

3.4 Particle Filter

Unlike the methods stated above the particle filter is a nonparametric implementation of the Bayes filter, i.e. the estimated distribution $p(x_k)$ of the state is not described by parameters, e.g. mean and covariance. In case of particle filters, $p(x_k)$ is described by samples so-called *particles*.

3.4.1 Theory

An advantage of a nonparametric method is that the distribution is not bounded to be of specific type, e.g. Gaussian. The particles fully approximate the distribution, i.e. all characteristics of the PDF like mean and covariance can be calculated out of the particles. As a consequence a particle x_k^i, with $i = 1..N$ where N is the number of particles, used for estimation should ideally fulfil

$$x_k^i \sim bel(x_k)$$

I.e. each particle simulates the current belief. So all particles together describe the current state as PDF, e.g. the mean can be calculated by averaging the particles.

Each particle can be seen as a possible state and is therefore a concrete instantiation of x_k, i.e. the state at time step k. To get a good approximation the number of particles N is relatively large, e.g. $N = 1000$.

The algorithm of a basic particle filter known as *Sampling Importance Resampling* (SIR) applying the state-space model (introduced in Section 2.2.1)

$$\begin{aligned} x_k &= f(x_{k-1}, u_{k-1}) + w_{k-1} \\ z_k &= h(x_k) + v_k \end{aligned}$$

is stated in Algorithm 3.4. SIR has three main stages: sample, weight and resample. The first stage (sample) can be seen as the *time update* of a Bayes filter. The samples are updated to represent the (a priori estimated) next state. The latter stages correspond to the *measurement update* of the filter. The particles are assessed with a weight (the better the sample meets the

Algorithm 3.4: SIR Particle Filter.

```
input  : particles $x_{k-1}^i$ with associated weights $w_{k-1}^i$,
         state transition function $f(x_{k-1}, u_{k-1})$,
         PDF of process noise $p_w$,
         likelihood PDF of measurement w.r.t. state $p(z_k|x_k)$,
         control input $u_{k-1}$, measurements $z_k$
output: particles $x_k^i$

// draw particles and assign weights
1  for $i = 1$ to $N$ do
2      draw $n_i \sim p_w$;                     // sample from the process noise PDF
3      $x_k^i = f(x_{k-1}^i, u_{k-1}) + n_i$;          // time update of particle
4      $w_k^i = w_{k-1}^i p(z_k|x_k^i)$;            // calculate weight of particle
5  end

// normalize weights
6  $W = \sum_{i=1}^N w_k^i$;
7  for $i = 1$ to $N$ do
8      $w_k^i = w_k^i / W$
9  end

// calculate effective number of particles
// and resample when $N_{eff}$ falls below a threshold
10 $\hat{N}_{eff} = 1/\sum_{i=1}^N (w_k^i)^2$;
11 if $\hat{N}_{eff} < N_{th}$ then
12     $[\{x_k^i, w_k^i\}_{i=1}^N]$ = Resample($[\{x_k^i, w_k^i\}_{i=1}^N]$)
13 end
```

measurements the higher the weight) and resampled to concentrate on particles with higher weights, i.e. on states with higher probability [3].

sample Line 2 in Algorithm 3.4 updates the particles. The process noise is simulated by drawing a sample from the process noise PDF and adding it to the state. Adding random noise establishes the diversity of particles and features the uncertainty of the process.

This step is similar to the time update of the Kalman filters. Note that the process noise sample n_i could be included into f (line 3), but in the proposed state transition model the noise is assumed to be purely additive.

weight Line 4 assigns a weight or so-called *importance factor* to each particle which represents how *likely* this sample state is. The weight is evaluated out of the likelihood function $p(z_k|x_k)$, i.e. the likelihood of a measurement considering a specific state. The likelihood of a measurement vector z_k can be evaluated by comparing z_k with the expected measurement vector

$$\hat{z}_k^i = h(x_k^i)$$

E.g. when the measurement noise is Gaussian $\mathcal{N}_{(0,R)}$ with zero mean and covariance R, the weight can be calculated by

$$w_k^i = w_{k-1}^i \mathcal{N}_{(z_k^i,R)}(z_k)$$

With the mean set to the expected measurement, the returned probability corresponds to the likelihood of the deviation between $\hat{z}_k^i$ and z_k.

This step is similar to the measurement update in the Kalman filters, where the Kalman gain corresponds to the weight of a particle. The weight represents the probability that the particle is the actual state.

Line 6 to 9 normalize the weights, so that the sum of all weights equals 1.

resample The idea of resampling in line 12 is to choose particles for the next iteration which are more likely, i.e. have a higher weight. A common problem of particle filters is the effect of *degeneracy*. After some iterations all but one particle will have negligible weight. So it is the task of resampling to drop the particles with negligible weights. Sometimes this is not good at all, but there are also other possibilities to avoid the degeneracy phenomenon [3].

Because resampling is a time consuming job it should only be done when necessary and not in every iteration. The degeneracy can be measured by the effective number of particles N_{eff} which can be approximated with following formula.

$$\hat{N}_{eff} = \frac{1}{\sum_{i=1}^{N}(w_k^i)^2} \tag{3.11}$$

An efficient resampling technique is given with Algorithm 3.5 and is illustrated in Figure 3.4 [3, 27]. The proposed resampling algorithm uses the *cumulative distribution function* (CDF) of the current belief, to select the particles for the next iteration. The CDF $f_X(x)$ gives the probability that a random variable is below or equal a specific value x, i.e. $p(X \leq x)$. The CDF is formed by integrating the PDF. In case of particle filters the weights of the samples (forming the PDF) are summed up to get the CDF (line 1 to 4 in Algorithm 3.5).

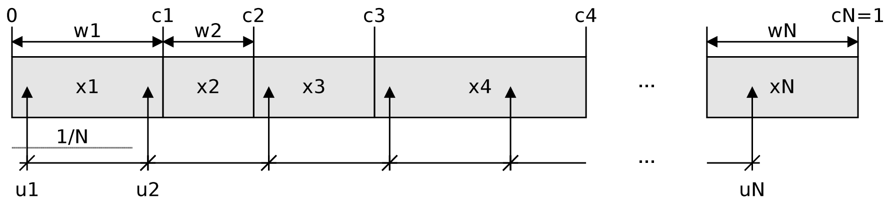

Figure 3.4: Illustration of resampling [27].

Algorithm 3.5: Resample.

```
input  : $[x_k^i, w_k^i]_{i=1}^N$
output: $[x_k^j, w_k^j]_{j=1}^N$
// construct the CDF
$c_1 = w_k^1$;
for $i = 2$ to $N$ do
    $c_i = c_{i-1} + w_k^i$;
end
// draw particles
$i = 1$;
$u_1 \sim \mathbb{U}[0, N^{-1}]$;                    // draw a starting point
for $j = 1$ to $N$ do
    $u_j = u_1 + (j-1)/N$;                    // move along the CDF
    while $u_j > c_i$ do
        $i = i + 1$;
    end
    $x_k^j = x_k^i$;                               // assign particle
    $w_k^j = 1/N$;                                  // assign weight
end
```

In Figure 3.4 the CDF is depicted as a sequence of blocks, where each block corresponds to a specific particle and the length of the block equals the weight of the particle. N particles are selected by equidistant pointers u_1 to u_N. The first pointer is selected randomly between 0 and $1/N$ (line 6 in the algorithm). In the example of Figure 3.4 the pointer u_1 selects the particle x_1. A particle with higher weight is selected with higher probability and might be selected more often, e.g. particle x_1 and x_4 in the example above. Particles with small weights are more likely to be dropped, e.g. particle x_2.

Note that there are various possibilities how to sample and weight. The particle filter in this thesis is used for state estimation so the state transition function f and measurement function h is used to sample and weight the particles. For other possibilities see e.g. [3].

3.4.2 Pros and Cons

Advantages.

- Simple.
- With particle filters the restriction to Gaussian random variables is removed. As a consequence the state may have a multimodal distribution.

- Highly nonlinear systems can be handled.
- Accuracy can be simply increased by a higher number of particles. But a trade-off has to be made between the number of particles and the computational effort.

Disadvantages.

- Full probability density functions of process and measurement noise are needed which are often unknown or assumed to be Gaussian anyway.
- This filter is only an approximation and not optimal cf. Kalman filter. A small number of particles can cause a high approximation error. But a high number of particles leads to a high computational effort. However the number of particles must be significantly higher then the size of the state, i.e. the number of random variables to estimate, otherwise not all relevant regions of the state-space can be covered [27].
- Bad for static states, i.e. $x_k = x_{k-1}$. Resampling causes the diversity to decrease until a single state survives. So resampling should be turned off in static states, e.g. when a robot stops. This is also the reason why particle filters cannot be used for parameter estimation.
- Bad for accurate sensors. The proposed particle filter (SIR) is not suitable for very accurate sensors. With low measurement noise it is possible that all weights of the particles will be almost zero. The particles are generated randomly, so the state corresponding to the accurate measurement may never occur in the set of particles. Resampling does not work any more because all particles have equal and almost zero weight. Hence the SIR particle filter gets inapplicable [27].
- High computational complexity and memory requirements compared to the Kalman filters, but it is in general parallelizable.

3.4.3 Applications

Particle filters are suited for nonlinear non-Gaussian systems with dynamic states (particle filtering cannot be used for parameter estimation) [3, 27].

An example where the Kalman filter fails is data association, e.g. tracking of many objects in computer vision applications [26]. Another one is global localization, i.e. where the initial position of an object is unknown. In localization applications unimodal probability distributions are inappropriate [27].

3.5 Comparison

The following sections compare the features and the complexity of the estimation algorithms.

3.5.1 Model

The proposed algorithms are all able to handle nonlinear systems except for the original Kalman filter.

However the extended Kalman filter linearizes the system, i.e. approximates it with the first order Taylor series, which gives poorer results than the unscented Kalman filter and particle filter, see pros and cons of the EKF (Section 3.2.2). The UKF and particle filter approximate the distribution not the function and therefore perform better than the EKF for nonlinear systems.

Multimodal distributions can only be handled by the particle filter. However when the assumptions for the Kalman filters hold, the KF, EKF and UKF are the optimal methods.

3.5.2 Computational Complexity

The computational complexity of the Kalman filter and extended Kalman filter is the same. The most complex parts of the estimation is the matrix inversion and the matrix multiplication. An efficient algorithm inverting a matrix of size $n \times n$ has the complexity $O(n^{2.8})$. Hence the complexity can be approximated with $O(m^{2.8} + n^2)$ where m is the size of the measurement vector and n the size of the state [27].

The most complex part in the UKF is the calculation of the matrix square root (e.g. Cholesky decomposition) in the unscented transformation with a complexity of $O(n^3)$. In the model proposed at the beginning of this chapter the noise is assumed to be additive, if this is not the case the complexity increases to $O((2n + m)^3)$ [31].

The particle filter's sample and weight part depends on the distributions of process and measurement noise. Assuming these to be Gaussian for reasons of comparison, drawing a noise sample costs $O(n^2)$ or $O(m^2)$ respectively because of matrix multiplications. Resampling with the proposed algorithm is relatively efficient with $O(N)$ (N is the number of particles). Hence the overall complexity is $O((n^2 + m^2) \cdot N)$ because noise sampling has to be done for every particle. However N is selected much bigger than n or m to cover all possible regions of the state.

3.5.3 Memory Usage

The memory usage in the Kalman filters is defined by the used matrices, e.g. covariance matrix. The complexity is $O(m^2 + n^2)$ where the dominant part is n^2 because in most cases the size of the state n is bigger than the number of measurements m.

The extended Kalman filters uses very similar variables, hence the complexity equals that from the original Kalman filter. The sampling technique in the UKF does not worsen the memory usage either. The unscented transformation uses $2n + 1$ samples where n is the size of the state and so the size of each sample. Hence the complexity remains at $O(m^2 + n^2)$.

Particle filters use lots of samples (e.g. $N \geq 100$) to represent and approximate the state's distribution, c.f. the Kalman filters only needs the mean and covariance. The number of samples or particles N in particle filters will be (much) higher than the size of the state n to avoid the particle deprivation problem (Section 3.4.2). So the complexity is $O(N)$ or $O(n \cdot N)$ with $N >> n$ to compare the complexity with the Kalman filters.

3.5.4 Summary

The comparison is summarized in Table 3.1. Note that N is the number of particles and is an order of magnitude bigger than the size of the state vector n and size of the measurement vector m.

	KF	**EKF**	**UKF**	**PF (SIR)**
model	linear	(somewhat) nonlinear	(highly) non-linear	(highly) nonlin-ear
distribution	unimodal	unimodal	unimodal	multimodal
accuracy of lin-earization	-	first order	third order	higher
computational complexity	$O(m^{2.8}+n^2)$	$O(m^{2.8}+n^2)$	$O(n^3)$	$O((n^2+m^2)\cdot N)$
memory usage	$O(m^2+n^2)$	$O(m^2+n^2)$	$O(m^2+n^2)$	$O(n \cdot N)$

Table 3.1: Comparison of features and complexity of the stated estimation algorithms.

Note the UKF does not linearize the model, but uses a sampling technique to estimate the belief. However the accuracy of the UKF corresponds to a third order linearization of the model. The accuracy of the particle filter can be even higher considering a high number of particles.

3.6 Others

The purpose of the developed framework is *low-level* sensor fusion. So basically the tasks of alignment and estimation are implemented through various algorithms. In the literature [12, 14, 21] several techniques are stated to fulfill these tasks (Table 3.2).

The filters provided in the developed library handle one-dimensional input values, i.e. a single value describes a property or attribute of an entity. E.g. in contrast to image fusion algorithms the methods in this framework are not suitable to handle point clouds efficiently. So fusion algorithms performing coordinate transforms are not implemented. Neither are time warping algorithms explicitly programmed which are needed e.g. in video compression. However, temporal alignment is needed to pass the sensor data to the fusion algorithms proposed in the previous chapters.

The estimation techniques are chosen to be based on recursive state estimation. These methods are more efficient w.r.t. memory usage and computational complexity in contrast to batch filters (e.g. maximum likelihood [22] or moving horizon estimator [23]). Batch estimation uses a specific amount of measurements for filtering and the higher the number of measurements, the

Processing Function	**Method**
alignment	coordinate transforms
	time warping
	units adjustments, value normalization
estimation	Bayesian inference:
	- Kalman filters
	- maximum likelihood
	- least squares
	- particle filters

Table 3.2: Fusion methods for alignment and estimation.

higher the computational effort. Further the application fields are more specific, e.g. a moving horizon estimator may work better in processes with slow system dynamics. The least squares method is a predecessor of the Kalman filter and therefore not implemented.

Among the state estimation methods introduced in this chapter, an average and median filter is implemented. Further the algorithm confidence-weighted averaging [8] is also included in the framework. These algorithms do not correspond to the family of state estimation algorithms, but can be used for sensor fusion too. However these algorithms work very simple. An averaging method collects several inputs and outputs the average. The confidence-weighted averaging calculates the average w.r.t. the variance of the inputs, which results in a more reliable output [8]. Furthermore the average and median filter are only suitable for measurements representing the same property. The results will show that the state estimation algorithms have significant advantages over these simple methods (see Section 5.2). Hence these sensor fusion algorithms are not discussed further.

CHAPTER 4

Framework Implementation

The formal framework assumed for the implementation, is that of a distributed network of so-called *nodes*, i.e. components of the system with a specific purpose. This adoption is also used in [15, 21] and applied in the (popular) *Robot Operating System* [25]. The nodes implementing a fusion algorithm are henceforth called *fusion nodes*.

In the following section the requirements for the framework are collected. Considering these requirements the interface of the fusion node and the representation of the system variables or properties are developed, see Section 4.2. Finally the actual implementation is proposed in Section 4.3.

4.1 Requirements

The following requirements reflect the purpose and features of the framework and are partitioned into two sections. The functional requirements specify *what* the framework should actually provide. This includes how the interface, i.e. the inputs and outputs, of a fusion algorithm should look like and what the fusion algorithm should do. Further the usage is specified. The nonfunctional requirements cover "global" properties of the framework, e.g. usability and reliability.

4.1.1 Functional Requirements

fusion algorithm A sensor fusion algorithm should process an input vector of system variables or properties. A sensor fusion algorithm shall provide an output vector of new system variables or properties.

interface Each property (fused or not) of the system shall include an ID, the value of the property, the confidence value and a timestamp.

- The ID of a property should uniquely define the property itself in the whole system.

- The value of the property should represent the current state of the entity in the system.
- The confidence value should specify how much the user can trust the value of an entity.
- The timestamp shall indicate when the value has been provided, i.e. how up-to-date the value is.

usage The outcomes of a fusion node should be made available for the whole system. The user should be able to parametrize the fusion node to optimize the results of the fusion. The same fusion algorithm should be usable for different purposes simultaneously, i.e. there may exist several nodes implementing the same fusion algorithm. So the user can apply the algorithms e.g. for different purposes or simply to structure the fusion. Hence the interconnection of these nodes should be possible, i.e. the output of a fusion node may be used as input for another fusion node.

4.1.2 Nonfunctional Requirements

- The output property's value of a sensor fusion algorithm shall be more trustworthy than the input property's value. Otherwise the fusion would be pointless (the input property could be used right away).
- The integration of a sensor fusion algorithm or fusion node should be simple.
- The interfaces (input and output) of all sensor fusion methods should be universal, i.e. a fusion node should abstract the concrete method away (like a black box).
- The fusion nodes should be able to handle asynchronous measurements. Asynchronous measurements are measurements which do not arrive periodically to be fused, i.e. the time instant of the measurement reception is unknown.
- The fusion nodes should be able to handle multirate measurements. Sensors may provide their measurements with different rates.
- The fusion nodes should be able to handle missing measurements. Components and sensors may fail or messages may get lost. A reliable fusion node should be able to handle such failures.

The collection of the requirements above is the basis for following considerations w.r.t. the interface of the framework and the representation of system variables given in the next section.

4.2 Basic Considerations

As already mentioned the system is distributed and consists of several components called nodes. The next sections describes the interface of a fusion node in general. Section 4.2.2 explains the need for a common representation of the input and output data of a fusion node.

4.2.1 Interface

The basic interfaces of a fusion node are the sources of information, auxiliary information and external knowledge and the results of the fusion (see Figure 4.1).

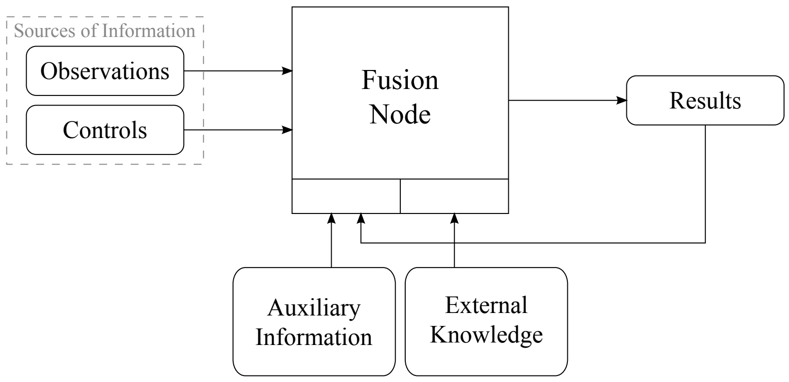

Figure 4.1: A fusion node with its inputs and outputs [21]. The results may be used as auxiliary information for the next calculations. Auxiliary information and external knowledge corresponds to the configuration of a fusion node.

The inputs and outputs of a component in a system can be generally distinguished according to [16]:

data input As explained in Chapter 3 a fusion algorithm takes data to generate an output which is in some sense better than the input. The data input or *observation* is the information to form the belief of a property in the system. W.r.t. sensor fusion this input corresponds to sensor observations or measurements respectively.

An observation is often represented as a tuple of entity name or ID E, spatial location $\boldsymbol{x}$, time instant t, value $\boldsymbol{y}$ and uncertainty $\boldsymbol{\Delta y}$ [21].

$$O = \langle E, \boldsymbol{x}, t, \boldsymbol{y}, \boldsymbol{\Delta y} \rangle \tag{4.1}$$

The source of observation is possibly a single sensor or multiple sensors. When multiple sensors or *redundant* sensors respectively observe a single entity, the measurements may be concatenated to form the overall data input or observation (cf. the measurement vector z for a Kalman filter).

control input The fusion methods proposed in Chapter 3 additionally have the possibility of *control inputs* to better model an interacting system. The majority of systems do not simply observe the environment, rather there exists a loop of measuring and acting, i.e. a *control loop*. E.g. when a robot is controlled to move forward with a specific velocity, the measured velocity (observation) may be filtered considering the desired velocity (control input).

The data input and control input correspond to the *Service Providing Linking Interface* (SPLIF) [16], the primary interface of a component in a system.

data output The estimate is the data output of a fusion node and represents the current belief of an entity. Similar to the data input the estimate can be represented as the very same tuple and may act as an input for another fusion node. The results, i.e. the fused data, may also be used as auxiliary information for the next estimation cycle. E.g. a recursive state estimation algorithm uses the previous fused state to calculate the current one. So the feedback loop features a recursive fusion algorithm.

The data output is also a part of the SPLIF.

configuration The fusion nodes are configured to model a specific application. This type of input may either be auxiliary information, i.e. additional information derived from input data, e.g. variance of the sensor, or external knowledge like the physical model, probability distributions of noise and likelihood functions [21].

These inputs to the fusion node correspond to the *Configuration Planning Interface* (CP) and make sure that the node provides the stipulated services in a specific environment [16].

Packing the task of sensor fusion into a *fusion block* separates the sensor processing of the control application. The fusion block is an additional abstraction layer and therefore decreases the cognitive complexity of the application. It contains an arbitrary number of fusion nodes. An example integration of sensor fusion into the overall application is given in Figure 4.2.

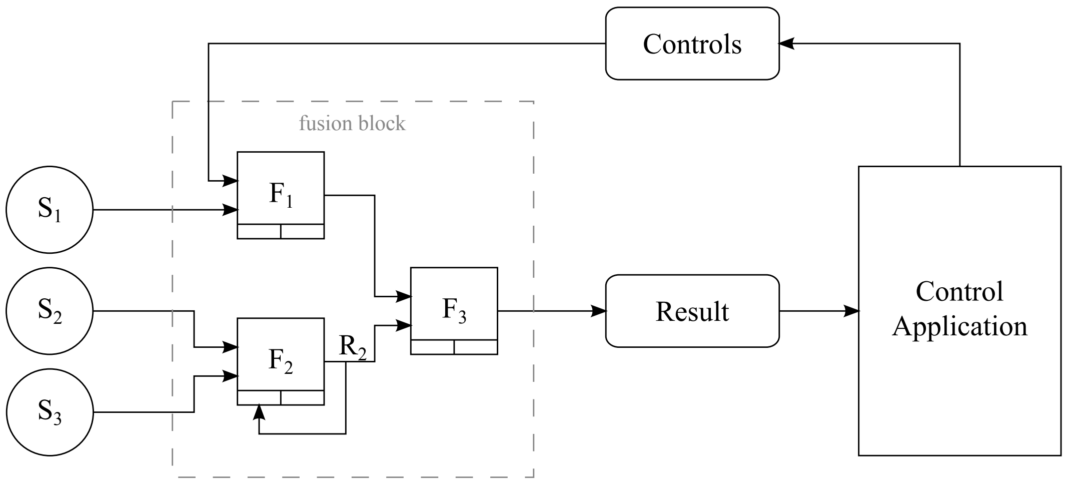

Figure 4.2: An example fusion block and integration into a control system [21].

Each fusion node may implement a different sensor fusion method. Furthermore each fusion node can have different inputs and outputs. The fusion methods, inputs and outputs and interconnections are chosen considering the application.

E.g. in Figure 4.2 the fusion node 1 takes the observation of sensor 1 and additional control inputs to estimate the input for fusion node 3. The fusion node 2 calculates parallel to F_1.

Often such an arrangement is used when a single property is measured with different sensors (cf. competitive sensor configuration). Fusion node 3 would then fuse these two estimates of F_1 and F_2, e.g. by voting or averaging. The reintroduction of R_2 into F_2 features an iterative fusion method, e.g. a recursive state estimation algorithm.

4.2.2 Representation of Inputs and Outputs

Establishing a common representation in data fusion means to transform the spatial location into a common coordinate frame (*spatial alignment*) and to align the time instants onto a common time axis (*temporal alignment*). The units of value and uncertainty should be normalized to a common scale [21].

spatial alignment Defining a common spatial location plays a crucial role in image fusion. The fusion nodes implemented in this framework only handle single numeric values and are therefore not suitable for image fusion. There are other examples where spatial alignment is necessary, e.g. the acceleration on different points on a robot is not equal when driving a curve or rotating. Such coordinate transforms are left to the user. Hence spatial alignment won't be discussed here in more detail.

temporal alignment Measurements are assumed to occur on a *dense time base* [15], i.e. the exact point in time of a message reception cannot be defined. When the estimation algorithms are called periodically, the observations must be transformed (*alignment*) before passing them to the estimator with an agreement protocol (see Figure 4.3).

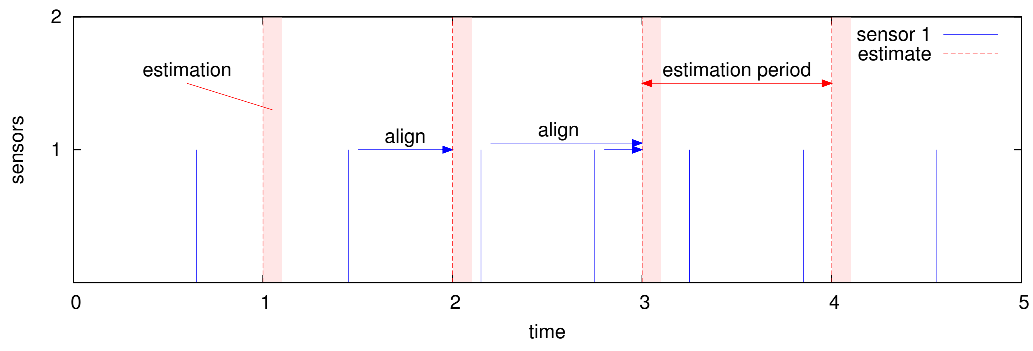

Figure 4.3: Temporal alignment of measurements. The estimation (dashed line) is called periodically.

Hence fusion nodes processing inputs periodically, acting in a system communicating on a dense time base, have to do temporally align inputs. The temporal alignment or agreement protocol implemented in this framework is discussed in Section 4.3.4.

Note that the methods need not to be called periodically. E.g. when the state estimation is called sporadically the elapsed time since last estimation can be calculated and used for the time update. So theoretically the estimation may be called whenever a measurement is received (see Figure 4.4).

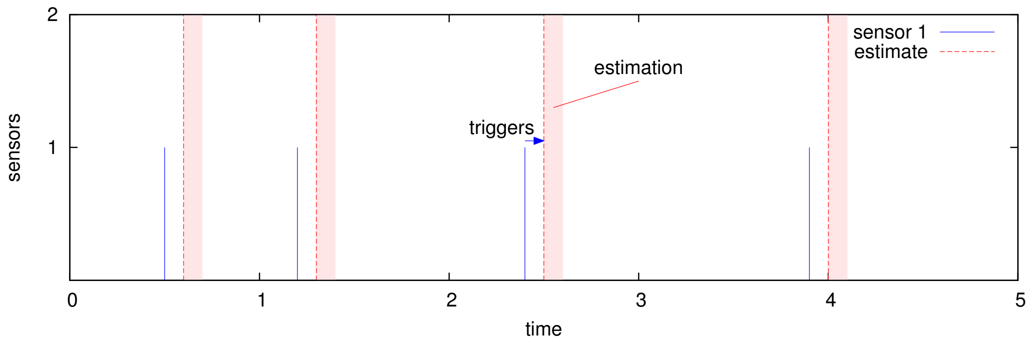

Figure 4.4: Measurements trigger estimations. As soon as a measurement is received the estimation is started. The duration of the estimation is depicted as a shaded area.

But in practice the duration of the estimation must be taken into account too. The measurement may occur *during* an estimation (the user does not know when the message arrives!), hence temporal alignment is needed too (the measurements occurring during estimation have to be aligned for the next estimation). When the measurements arrive more frequently than the estimation can be performed, the measurements occur *always* during estimation. Figure 4.5 depicts the problem of slow estimation and high measurement rates. The estimated state will be evaluated similar to the time-triggered approach, i.e. the estimation will be called periodically with aligned measurements.

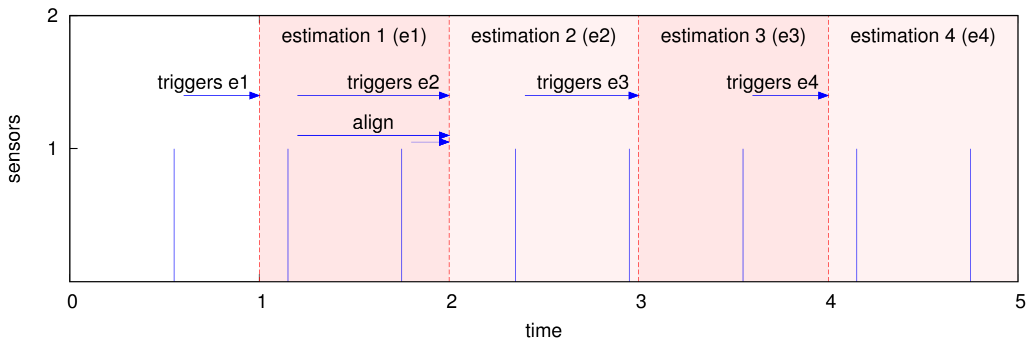

Figure 4.5: The measurements arrive faster than the estimation can be executed. The width of the shaded areas features the duration of the estimation (cf. Figure 4.3 and Figure 4.4).

normalization Inputs and the output of a fusion node are represented by a vector of numeric values (see the definition of an observation in the last section). Hence the input values should conform to a common scale w.r.t. the property it is representing to avoid scaling errors. E.g. all velocities should be described by the same unit (e.g. m/s). If a fusion node outputs the velocity v_1 in mm/s, fusion nodes using v_1 must distinguish between v_1 and other velocities.

The uncertainty should have an appropriate unit too. E.g. when the uncertainty of a velocity in m/s is described by the standard deviation (i.e. the root of the variance), the unit of the standard deviation should be m/s too. A different scale in uncertainty confuses the user of fusion node inputs and outputs.

Fusion nodes are able to transform the inputs to the common scale, e.g. with the measurement model of a state estimation algorithm.

4.3 Implementation

A main goal of the thesis is to be able to use the fusion algorithms within the *Robot Operating System* (ROS) [25]. So a short introduction to ROS is given in the next section.

The implementation of the sensor fusion framework or library respectively is organized in a package. An overview to this sensor fusion package is given in Section 4.3.2.

The three main implementation parts of the framework follow. In Section 4.3.3 the library collecting the sensor fusion algorithms is described. The ROS node for sensor fusion or ROS wrapper respectively, i.e. the code for using the fusion methods within the Robot Operating System, is proposed in Section 4.3.4. Finally the configuration of a fusion node within ROS is explained in Section 4.3.5.

4.3.1 Introduction to ROS

The Robot Operating System is a framework consisting of libraries, tools and conventions which simplifies the development of robotic systems [25]. Basically it is a meta-operating system providing services like hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management [24]. ROS can be installed and run on several platforms including Linux, MacOS and Windows.

The basic architecture is that of a distributed system. ROS may run on multiple computers each holding several processes called *nodes* which are able to communicate with each other.

A node may communicate over so-called *topics*, i.e. messages are published to the whole system through a topic A and all nodes subscribing to topic A receive these messages. This communication style is asynchronous. Nodes may also communicate through services, i.e. a node responds with the service requested by another node. The last possibility is to communicate over the parameter server, i.e. nodes can get and set values on this global server. It is basically a dictionary and is used to provide static data, e.g. configuration parameters.

ROS nodes can be collected into namespaces, so several fusion nodes may be summarized to a fusion block (proposed in Section 4.2).

4.3.2 Package Overview

The overall sensor fusion package `sf_pkg` includes three sub-packages containing the sensor fusion algorithms, code needed to be able to use the fusion methods within ROS and the message definitions of the ROS node implementing a fusion algorithm (e.g. the definition of the output message). Further the documentation of the code and examples are collected in separate folders. Figure 4.6 shows the structure of the sensor fusion package.

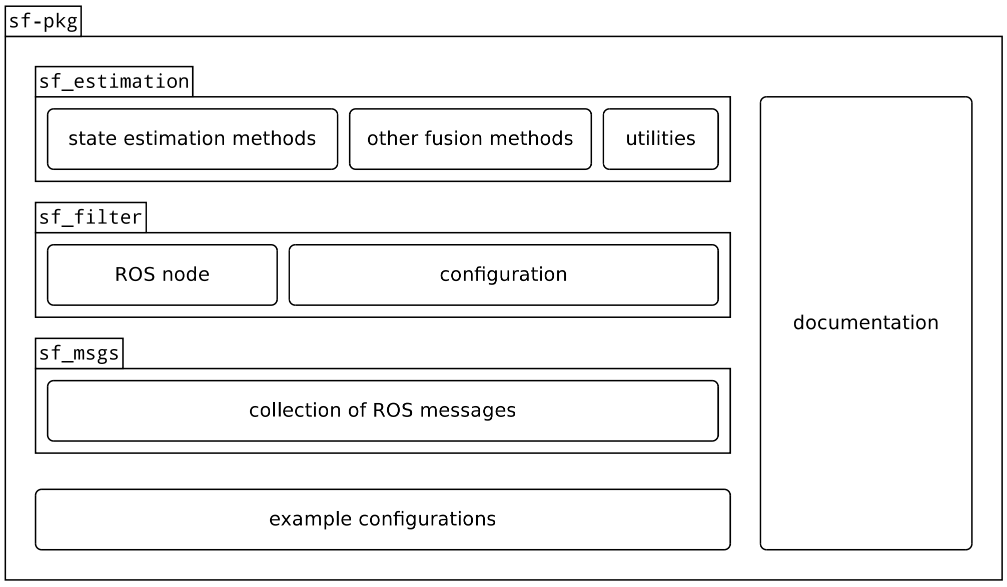

Figure 4.6: Package overview.

Sub-package `sf_estimation` contains an object-oriented C++ library implementing the sensor fusion algorithms described in Chapter 3, among others (e.g. averaging). The estimation library `sf_estimation` is completely independent of other packages in `sf-pkg` or other ROS packages, e.g. it is unaware of the sensors or where the input for the fusion is coming from. This part of the framework can also be used outside a ROS system.

The code for the configuration of a fusion node is strongly connected to the node itself. So these parts are collected in another package called `sf_filter`. The fusion node is implemented to run in the *Robot Operating System*. So sub-package `sf_filter` holds the ROS node, i.e. the executable, and its individual configuration.

The output messages of a sensor fusion node are defined in `sf_msgs`. The documentation of all packages is collected in a separate folder contained in the overall package `sf_pkg`. Additionally some examples for configuring a sensor fusion method are given.

4.3.3 Library

The algorithms estimating a single sensor's value are implemented in C++ classes collected in the `sf_estimation` package. The majority of the estimation methods implemented basically perform matrix manipulations, for which the math library Eigen [7] is used.

In the following sections the parts of the library are described. Figure 4.7 shows the main parts of the sub-package. `sf_estimation` has two namespaces named `estimation` and `probability`. The former contains the fusion algorithms, its interfaces and a factory creating instances of fusion methods w.r.t. given parameters. The latter namespace holds functions for sampling e.g. from a Gaussian distribution or calculating the probability of a concrete event considering the PDF of a random variable.

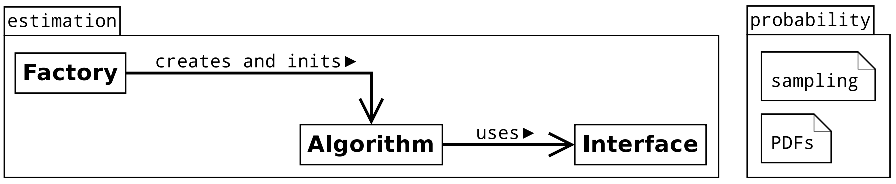

Figure 4.7: Overview of sub-package `sf_estimation`.

First the implementation and class hierarchy of the state estimation algorithms is presented. Like in the previous chapters the focus lies on the recursive state estimation algorithms. So basic parameters for state estimation algorithms follow (e.g. model). Then the interface regarding observations, controls and results is discussed. Regarding the inputs the approach for handling missing measurements is proposed. Finally the collection of sampling functions and probability density functions are given. The last part gives a summary of the library implementation and depicts the whole class diagram.

Algorithms

The algorithms differ in their behavior and possible parameters such that additional unique methods are needed (see important parameters in the next section). However a general interface can be defined which is valid for every class in the C++ estimation library. This interface is realized by an abstract class `IEstimator` with pure virtual or empty methods, see Figure 4.8.

With `IEstimator` the basic interface (for measurements, controls and the resulting estimate) of a fusion node is realized. The configuration of a method depends on the particular algorithm and is therefore implemented in the concrete classes.

The Kalman filters are grouped together with `AbstractKalmanFilter` because the configuration and the procedure is basically the same. The implementation of the unscented transform is separated in another class. A UT instance generates the samples and passes the samples through a transform function of a class implementing `ITransformer`, e.g. the UKF. Finally the posterior mean and covariance is calculated and provided by the UT instance.

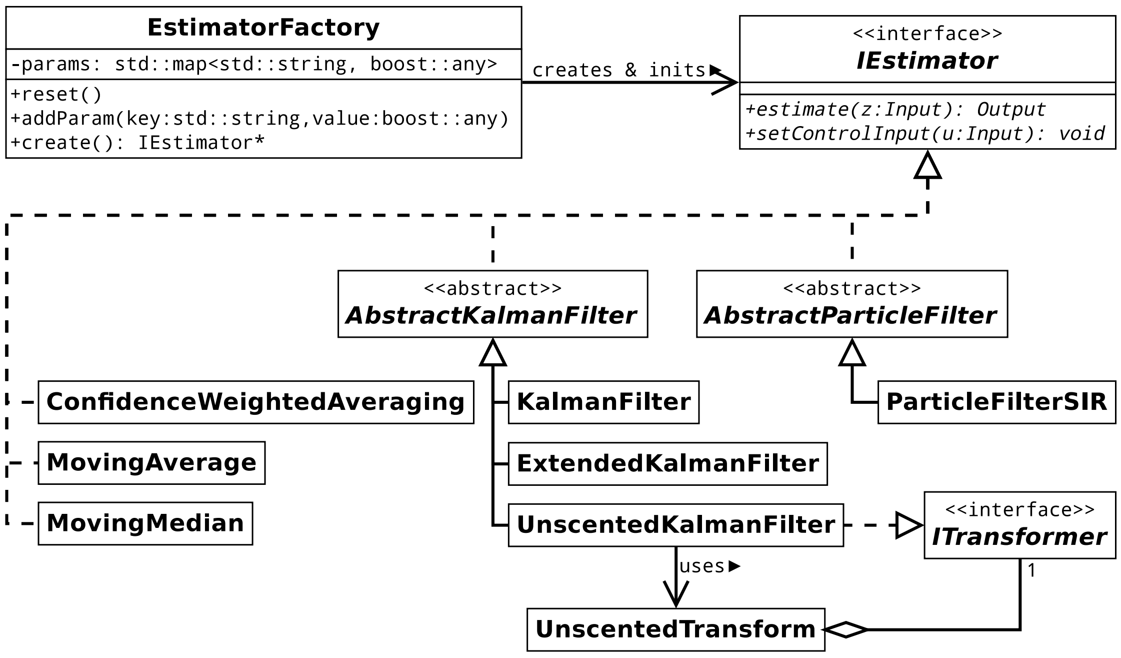

Figure 4.8: Class diagram of the fusion methods and the factory.

Although a single particle filter is implemented an additional abstract class is introduced because the approach is equal in most variants of a particle filter. The SIR algorithm implemented in the framework uses Gaussian distributions, but particle filters are able to handle each type of measurement or state distribution. The class `AbstractParticleFilter` should ease up the development of an individual particle filter.

The algorithms may be parametrized in several ways, e.g. parameters from a configuration file. For that purpose a separate class `EstimatorFactory` is available. The factory takes a map of parameter names and values to initialize a specific fusion method. So instead of configuring a fusion algorithm through the particular methods, the configuration can be passed to the factory which responds with the initialized estimator.

This factory is used by the configuration described in Section 4.3.5.

Parameters

The estimators or filters respectively can be configured by various methods to add auxiliary information and external knowledge. E.g. a Kalman filter needs, beside the measurements and controls, noise covariances and the system model. Optionally the initial state and error covariance may be set.

The most important parameter for state estimation algorithms is the model. The recursive state estimators KF, EKF, UKF and PF of the library use the state-space model introduced in Section 2.2.1. Another essential parameter is the *estimation period*, i.e. the period in which the estimation is executed.

linear model Linear models are described by matrices. So instead of coding a function, a matrix with numeric values has to be passed to the estimator.

The size of the matrices vary depending on the application, so the Eigen matrix type `MatrixXd` is used. `MatrixXd` can hold matrices of arbitrary size where each coefficient is a value of type `double`. Considering e.g. the "train example" in Section 2.2.1 the state transition matrix must be initialized with

```
MatrixXd A(2,2);
A << 1, 0.1, \
     0, 1;
```

nonlinear model Nonlinear models are described by functions. The state transition model must be converted to a C++ function mapping the previous state to the a priori state. So the time update equations are collected in a function of type

```
/**
* @brief Pointer to a function representing the state transition model.
*
* @param[in/out] x The state x at time k-1 (input) used to calculate
*   the estimated a priori state vector at time k (output).
* @param[in] u The control input at time k-1.
*/
typedef void (*func_f)(VectorXd& x, const VectorXd& u);
```

At the end of this function vector x represents the a priori state $\hat{x}_k(-)$.

The measurement update equations are similarly collected in a function of type

```
/**
* @brief Pointer to a function representing the observation model.
*
* @param[out] z The estimated measurement vector.
* @param[in] x The a priori state vector.
*/
typedef void (*func_h)(VectorXd& z, const VectorXd& x);
```

which maps the a priori state to the estimated measurement vector $\hat{z}_k$.

Considering the "train example" once more (Section 2.2.1) the corresponding state transition function states

```
void func_f(VectorXd& x, const VectorXd& u)
{
  VectorXd x_apriori;

  x_apriori[0] = x[0] + x[1]*0.1;    // v = v + a*T with T = 100ms
  x_apriori[1] = x[1];               // a = a

  x = x_apriori;
}
```

The EKF additionally needs the Jacobians of the state transition and measurement functions for that reason similar function pointer exists.

estimation period Almost every model will contain the time since last estimation T, i.e. the time which elapsed from estimating x_{k-1} to x_k. T is constant when estimating periodically. So the *estimation period*, i.e. the period in which the estimation should be executed, is defined by the model. When using $T = 100ms$ the estimation period must be set to $100ms$ and vice versa.

So the algorithms only give correct results when called periodically with the period used in the model. Hence the estimation period of the filter must not be changed without adapting the model.

During initialization the parameters have to be passed to the estimator. In general all parameters of an estimator can be changed during runtime too, including the models. In C++ the functions cannot be set like variables, so the functions (for nonlinear models) must be compiled before usage. But pointers can be changed. So when using several models for one filter all models must be specified before compilation. During runtime the model can be adapted by changing the function pointer to the address of another function representing a model.

Interface

In Section 4.2 the terms *observation*, *control input* and *estimate* have been discussed. In this section these inputs and outputs are concreted for implementation. An observation corresponds to a measurement or a collection of measurements. A single measurement can be described by <*ID, value, timestamp*> [15]. Some methods may need the variance of the observation too, e.g. Confidence-Weighted Averaging [8], so the measurement has been extended to

$$M = \langle ID, value, variance, timestamp \rangle \tag{4.2}$$

Above representation is simpler to that given in Section 4.2 (Definition 4.1), in particular value and variance are scalar and the position is dropped. But regarding the position, one can argue that the ID uniquely defines the sensor and therefore the position of the measurement source. Furthermore the uncertainty is represented by the variance of the measurement. When extending the measurements to a collection of measurements the representation is quite similar (except the position) to Definition 4.1.

A fusion method generally outputs the fused value, i.e. the estimate, and the variance indicating the confidence of the value. To be able to use an output as an input for another filter, the specification of the output equals that of the input, namely <*ID, value, variance, timestamp*>.

Figure 4.9 shows the class diagram regarding the inputs and outputs of a fusion node. A single measurement is described by an instance of `InputValue`. A single estimate, e.g. a variable of the estimated state, is represented by the class `OutputValue`. The common base class `Value` features the similar representation of input and output of a fusion algorithm.

To meet the requirement of missing measurements (see Section 4.1) `InputValue` has an additional member to indicate a missing measurement. No further distinctions exist between `InputValue` and `OutputValue` except the type and use of these values.

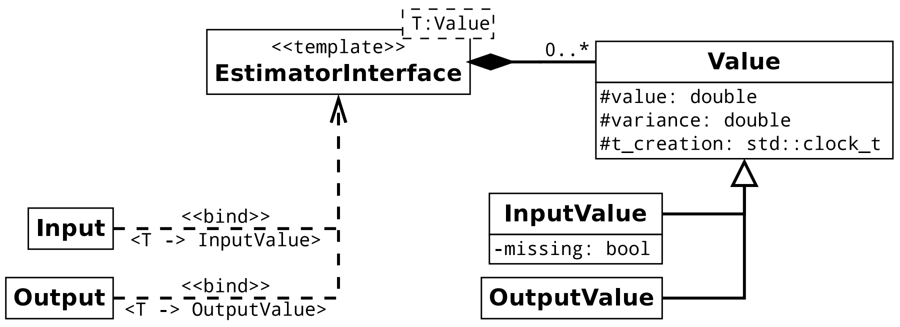

Figure 4.9: Inheritance diagram of the inputs and outputs.

Controls are very similar inputs. One can also think of it like a special kind of observation, so the simple form <*ID, value, timestamp*> is used. Control inputs may be missing as measurements (or even more often), e.g. the control signal opening or closing a valve may be conditioned on a state variable and therefore be received only sometimes. So controls are represented by the same class as measurements, namely `InputValue`. The variance of the controls is not needed in the sensor fusion methods implemented.

The single inputs and outputs can be collected by the template `EstimatorInterface`, e.g. several measurements can be collected into an instance `Input` to form a measurement vector. There is no restriction to the number of inputs or outputs, but the size of an input collection must meet the configuration of the fusion algorithm. E.g. the measurement model of a Kalman filter depends on the number of measurements, hence these two sizes must always match. See Figure 4.10 for an example ("moving train" example in Section 2.2.1).

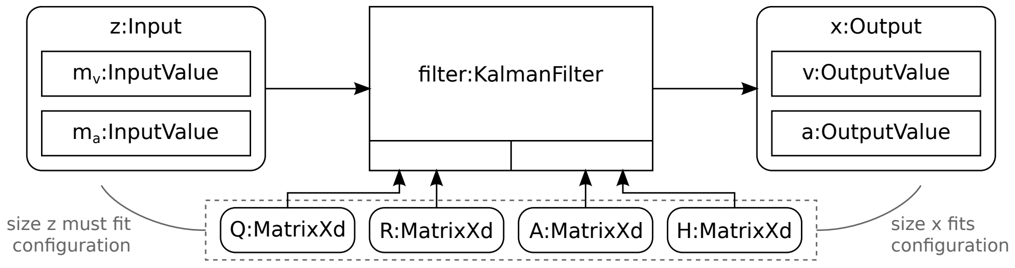

Figure 4.10: Inputs and output of a filter estimating the speed of train.

One may notice that there is no attribute in class `Value` corresponding to the ID of a measurement. The order of the inputs is relevant in state estimation algorithms because the input collection is directly mapped to the measurement vector. So the *position* of an input value in the collection also specifies the observed property (regarding state estimation methods), i.e. the position in an input collection of measurements corresponds to the ID of the measurement. Similar is valid for the output. In state estimation methods the output collection corresponds to the state vector.

Each estimation algorithm accepts the same type of input and output (c.f. `IEstimator` interface). So the user can use the very same input for different fusion algorithm. E.g. an unscented Kalman filter can be exchanged by a particle filter, it remains to adapt the configuration.

The possibility to be able to estimate with an incomplete measurement vector is vital for systems using event-triggered communication like ROS. Hence the next section proposes the implemented approach of handling such measurement vectors.

Handling Missing Measurements

The fusion methods are implemented to work only with fixed estimation periods by reasons given in Section 4.2.2 regarding temporal alignment.

So the user of this library must collect the observations and call the estimation algorithm in fixed intervals. This task is known as temporal alignment introduced in Section 4.2. But it may happen that not all sensors sent a measurement to the filter since last estimation, i.e. a complete measurement vector with new sensor values cannot be created. In real world systems sensor values may get lost, e.g. caused by disturbance. But missing observations can also occur when a sensor's sample rate is lower than the filter rate (= 1/estimation period). This is often the case in systems using different kinds of sensors (c.f. the possible sample rates of a gyroscope and a sonar sensor).

The estimation algorithms implemented are aware to handle missing measurements. The proposed method is based on the recalculation method [9], where missing measurements are ignored and only a time update is performed, i.e. the measurement update of a filter is omitted.

The estimation is adapted to prepare the measurement vector before the measurement update is executed. In particular the missing measurements are replaced by the expected measurements. Recalling the theory, the expected measurement vector is calculated out of the a priori state $\hat{x}_k(-)$ given by the time update and the measurement model.

$$\hat{z}_k = h(\hat{x}_k(-))$$

Assuming the measurement vector contains only missing measurements, i.e. $z_k = \hat{z}_k$, the measurement update is without effect.

Kalman Filters. The measurement update in the Kalman filter is reduced to $\hat{x}_k = \hat{x}_k(-)$ because the error $z_k - \hat{z}_k$ evaluates to 0. The a priori state is not corrected by the error multiplied with the Kalman gain.

Particle Filters. The measurement update of the particle filter corresponds to the recalculation of the weights of the samples. A new weight is evaluated out of the probability density of the measurement vector. When the measurements are missing and replaced by their expected values, i.e. $z_k = \hat{z}_k$, the returned probability will result in the highest probability value possible regardless which particle is processed. Hence the weight of *each* particle will be multiplied with the *same* value. After normalization the weights equal the values of the previous cycle.

A similar consideration can be made when only parts of the measurement vector are missing. E.g. state variables depending on missing measurements are only evaluated through the time update.

Without a measurement the error covariance cannot be corrected, i.e. the covariance of the state increases when only a time update is performed. However when the measurement arrives, the covariance after e.g. two time updates is decreased to the same value as with a single time update (assuming appropriate models and process noise covariances).

Probabilities and Sampling

The Kalman filters are restricted to Gaussian distributions. State and measurements are assumed to be Gaussian. But particle filters are able to work with completely different probabilities and unlike Kalman filters they need the complete probability density function, e.g. for calculating the likelihood of a measurement or sampling from a specific PDF.

For that reason an additional namespace `probability` was created to provide place for further PDF's and sampling methods, e.g. for individual particle filters.

The Gaussian probability density function and sampling from a Gaussian distribution has been implemented because the libraries Boost [4] and Eigen [7] only provide 1-dimensional PDF's and sampling methods. This would be sufficient if measurements are assumed to be independent, i.e. the variance of each entry of the measurement vector z is provided instead of the overall covariance R. E.g. in the test application proposed in Section 5.1, the measurements are assumed to be independent, i.e. all non-diagonals of the measurement noise covariance are zero. Often we don't know the overall covariance and it remains to estimate at least the variance of each sensor separately. However the covariance can be evaluated by sampling the measurements of all sensors and therefore left as part of the representation of the measurement vector's Gaussian distribution.

Summary

In the last sections the parts of the basic sensor fusion library have been described. All classes and additional files are depicted in Figure 4.11.

The function pointers for the models are located in the header file `models.h`. The definition of the sensor fusion algorithm classes are collected in the header file `methods.h` needed e.g. by the factory.

Although the user has to know the types of the configuration parameters, e.g. the linear model must be a matrix of type `MatrixXd`, the user only needs the factory and the interface classes (`Input`, `InputValue`, `Output`, `OutputValue`) to work with the sensor fusion library.

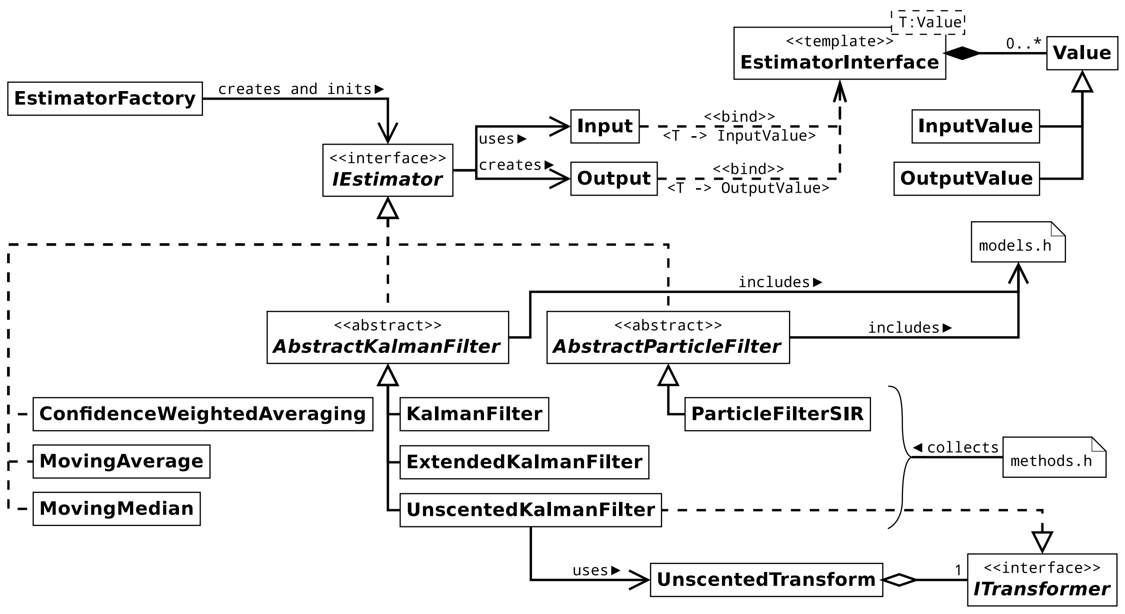

Figure 4.11: Classes, files and its dependencies in namespace `estimation`.

4.3.4 Node

In the previous section the basic algorithms were packed into classes and the interfaces were defined. A goal of the thesis is to be able to use these algorithms within the *Robot Operating System* (ROS) in a simple and generic way. The next part will cover the development of the ROS node. The structure and procedure is proposed. Then the approach for temporal alignment, the main task of the node, is presented in detail. The section concludes with the limits and drawbacks of the ROS node.

Structure

The ROS node developed in the sensor fusion framework subscribes topics containing measurements and controls, calls the estimation algorithm to update the belief and publishes the estimate, i.e. the result of estimation. Hence it is basically a wrapper around the estimation algorithm to be able to use it within ROS (see Figure 4.12).

Which observations and controls are subscribed, are specified during compilation and is part of the next section. The observations and controls may be messages of any type, but the field of a topic must be a numeric value (it is casted to `double`). However the measurements have to be aligned and the estimation has to be called.

The state machine of the ROS node is depicted in Figure 4.13. After initialization of the estimator and the node itself the main loop is entered, which alternates between checking for received messages and estimating.

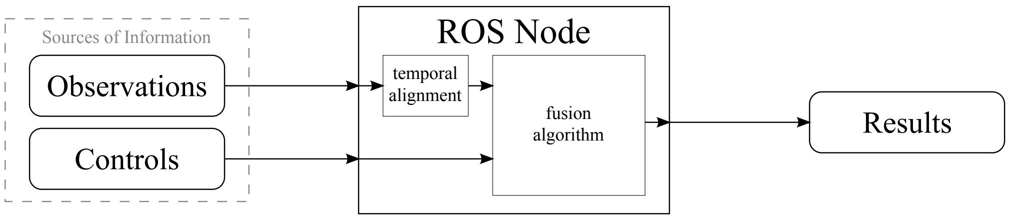

Figure 4.12: ROS node as a wrapper of the fusion algorithm.

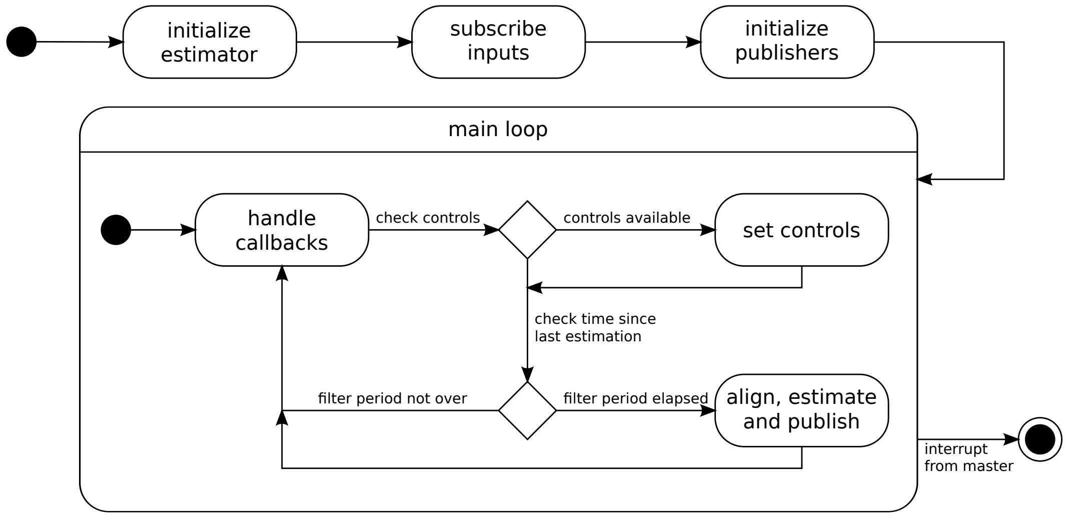

Figure 4.13: The state machine of the ROS node.

initialization First the node is initialized like every other ROS node, i.e. the node sets its name in the system and ROS specific arguments are handled (e.g. renaming of topics). Next an estimator factory is instantiated and initialized with the user specific configuration. The estimator factory then returns an initialized estimator. The last part of the initialization is to create the main access point for communication over ROS, the *node handle*. With the node handle the topics with relevant measurements and controls are subscribed and the publishers for the result, i.e. the state variables, are initialized.

main loop The time, when ROS should notify a node about new messages, can be chosen. In this ROS node implementation the messages are handled at the beginning of the main loop (state *handle callbacks*). Incoming messages are then received through the callback functions specified when subscribing to the topics. In the callback functions the received messages are saved into separate sets of structures, one set for the measurements and one set for the controls.

Back in the main loop the set of control structures are checked for new messages. Flags indicate when messages have been received. New control inputs are immediately passed to the estimator (state *set controls*).

Measurements are collected (in the measurement structures) to pass the measurements to the estimator exactly when the filter period elapsed. The time is checked as often as possible to call the estimator periodically. When a period runs out the measurements from the receive structures are temporally aligned (see next section for details) and passed to the estimator in an input collection. The result of the estimation is published a moment later (state *align, estimate and publish*).

Controls may be passed before or after estimation. It might depend on the application, the estimation period and the rate of control input messages. Once again note that the communication is event-triggered, the impact of the time instant *when* the control input is passed to the estimator should be held low, e.g. by choosing a small estimation period. Similar is true for measurements. The smaller the estimation period, the more actual is a measurement for the estimation, so the better the result of the estimation will reflect the actual state.

Temporal Alignment

ROS nodes generally communicate with the publish/subscribe method, so the measurements cannot be *sampled*, i.e. at a specific time instant, e.g. directly before the estimation. This would only be possible with services, which are not commonly used for providing sensor values (most ROS nodes communicate over the publish/subscribe method). However ROS doesn't ensure the arrival of a message at a specific time instant nor within a specific duration, neither with publish/subscribe nor with services.

Hence, when using ROS we must be aware that the measurements arrive at different time instants and may be delayed. Furthermore ROS nodes may publish with different rates.

This section describes a straight forward method for temporal alignment, other possibilities to handle asynchronous multirate measurements can be found in [9,21,33]. All measurements of a *single* topic received in a specific interval, namely the estimation or filter period respectively, are combined to result in *one* observation for the current estimation.

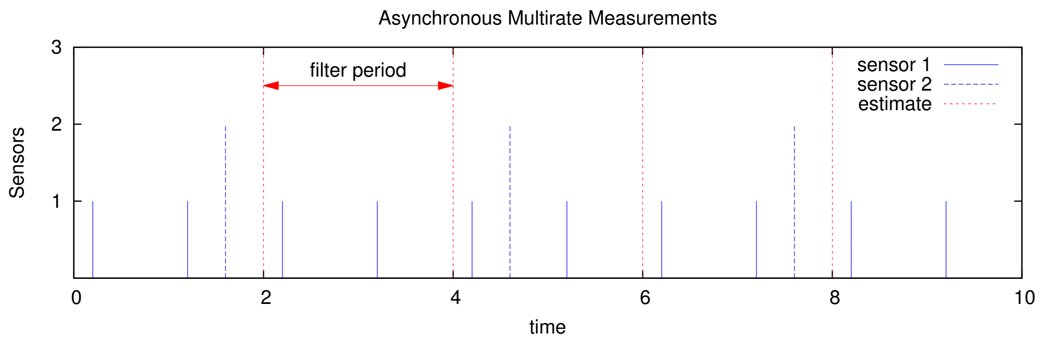

Figure 4.14: Arrivals of messages or measurements respectively in contrast to the filter's period.

In Figure 4.14 a typical situation is depicted. Sensor 1 publishes the measurements with higher rate than the filter rate. There are two measurements of sensor 1 available in each estimation cycle. Sensor 2 has a very low sample rate w.r.t. the filter rate, so it regularly happens that the observation is missing.

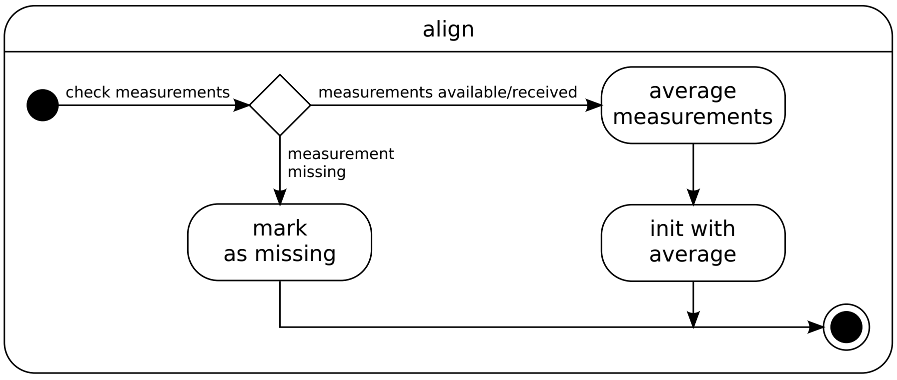

Figure 4.15: Temporal alignment of measurements.

The state machine of temporal alignment is given in Figure 4.15. Measurements received in the last filter period are averaged and filled into the measurement vector. The missing measurements, i.e. measurements which haven't been received since the last estimation, are indicated to be missing and added to the measurement vector too. Note that the size of the measurement vector does not change when missing measurements occur.

E.g. when packing the measurement vector for estimation at time 4 of the example in Figure 4.14, first two measurements from sensor 1 have to be averaged. A measurement from sensor 2 is missing in the filter period from time 2 to 4. So first an instance of `InputValue` (recall: an `InputValue` represents a single measurement) is created and initialized with the average of the sensor 2 measurements. Than another instance `InputValue` is initialized to be missing, representing the missing measurement of sensor 2. Finally the two value instances are packed into an instance `Input` representing the measurement vector for estimation.

The implementation of temporal alignment in the ROS node is summarized as follows:

- The estimation is done every T seconds, i.e. the filter rate $f_e = \frac{1}{T}$ is fixed. The filter period T can be specified by the user within the configuration header (see next section).
- Measurements of sensors which publish their value with a lower rate than the filter rate, i.e. $f_s < f_e$, are not available in every filter cycle. Hence sometimes these measurements appear to be missing.
- For measurements with a higher sample rate than the filter rate, i.e. $f_s > f_e$, the average of the measurements received in a period T is calculated. The average value represents the observation for the next estimation.

So handling missing measurements is part of the estimation algorithms and not that of the ROS node. The estimators implemented in the framework handle the missing measurements, see Section 4.3.3.

The approach described is not an optimal solution but handles asynchronous multirate and possibly delayed and missing measurements. In some applications it would be better to use the last measurement instead of an average. E.g. consider following model of position estimation and two sensors measuring the actual position (1-dimensional for simplicity) and velocity.

$$\begin{aligned} s &= s + v \cdot T \\ v &= v \end{aligned}$$

The last measurement from the position sensor should be used to update the position estimate, because it best reflects the actual position. On the other hand the average of the measurements from the velocity sensor should be taken. When the velocity changes, the distance moved since last estimation ($v \cdot T$) is best represented by the mean of the velocity.

However the error made by temporal alignment can be held low when choosing a small estimation period.

Drawbacks and Limits

The ROS node should be generic, i.e. applicable for different types of messages or measurements respectively and different applications. From this requirement follows that more complex inputs cannot be used with this node, e.g. a point cloud or an image. However this framework is designed for *low-level* sensor fusion, hence the possibility to handle simple observations is sufficient.

The node is designed to accept only single measurements. So when the observations of a 3-axis accelerometer should be filtered, all 3 measurements of the acceleration on each axes must be subscribed *separately*. But this does not reduce the node's performance. Although the same topic (representing e.g. the 3-axis acceleration) is subscribed three times, the message is received only once by the node and distributed to the 3 subscribers which then process the same message.

ROS is an event-triggered system. The estimation may not be called exactly every period. However a bigger problem may be the asynchronous and multirate measurements described above. The user must be aware that the estimate is not optimal because the measurements arrive at different time instants, i.e. are uncorrelated. Further the estimate may be delayed to its actual value, because measurements may arrive much earlier (but maximal the estimation period) than the estimation is performed.

4.3.5 Configuration

Simple filters like moving average, moving median and confidence-weighted averaging [8] could be parametrized by arguments. Even a state transition matrix for a Kalman filter could be read from arguments or a configuration file. But the physical model of state estimators are described by functions which have to be compiled and cannot be added to the filter during runtime in C++.

Another difficulty of configuration is the need for a generic input interface. Whatever topic is used as an input, the algorithm should calculate the output. A *generic* fusion method is fully decoupled of the application, i.e. it doesn't know the sensor to improve. In ROS every package can define its own messages, so in general the message structure is unknown to the fusion node. But without knowing the structure we cannot extract the sensor value out of a message.

Several possibilities have been analyzed to overcome the problem that arises with the requirement that the node should be generic.

- The user transforms the topic for the ROS fusion node and the ROS fusion node subscribes always to the topic of a specific type (i.e. the topic type of the fusion node is always the same). The transformation of messages has to be done in an additional node created by the user. So the usability decreases when using this approach. Further the overall system and communication is loaded with an additional node and additional topics.

- A template for using the C++ estimation framework is provided. The message containing the entity to fuse is subscribed by the user itself and the input for the estimation is set by the user whenever a message arrives. The user must collect messages on its own and make sure that the estimation is executed periodically. Here too, the usability decreases, i.e. the effort for configuration is even higher than in the previous option.

- A meta-topic class (e.g. `topic_tools::ShapeShifter`) is able to receive every type of topic. The message callback could be created for the class e.g. `ShapeShifter`. But to extract a specific field of a message a parser is needed which would result into a big overhead in implementation and during runtime.

- The fusion node may be written in Python or the relevant code could be embedded. But the performance of the estimation would suffer (the execution time of the estimation should not be ignored, see experimental results in Chapter 5).

- The topic name and type to subscribe to and the topic field containing the value to estimate may be put into a header file. The header file is always included in the ROS node. Only one sensor fusion node can be generated out of this header file, hence the code has to be copied for each desired fusion node.

All previous stated possibilities include to make compromises, but still the simplest and most suitable solution is to use a header file. The state space models may be defined by functions which can only be defined within header files, so a henceforth called *configuration header* is needed anyway.

Configuring the ROS fusion node includes specifying the interface for the node and the parameter map for the estimator factory (which creates the appropriate estimator). The syntax of the header is proposed in the next section followed by a section listing the parts of the configuration header. The configuration header only includes the configuration specified by the user, e.g. parameters and topics to subscribe. The actual code, e.g. initializing the appropriate estimator, is generated by several other files described in the following section "Structure". Finally this section is summarized by a configuration use case.

Syntax

Simple methods like averaging have simple parameters. E.g. the window size (of an average filter), i.e. the number of measurements which should be averaged to form the output, is of type integer. The window size of an average filter is specified by

```
#define WINDOW_SIZE 5
```

Most parameters needed by state estimators are no more primitive data types (like integers) but rather collections of numbers or even formulas, e.g. matrices with unknown size or a set of functions. ROS is delivered with the popular Boost library [4], which provides macros solving these configuration needs [5].

In particular the construct named *sequence* is used. A sequence may have arbitrary size. The size can be evaluated and each element is accessed by its position. Vectors are represented by sequences, e.g.

```
#define INITIAL_STATE    (0)(0)
```

specifies the initial state to be the vector $(0\ 0)^T$.

An example for specifying a matrix through a sequence of sequences is given below. Each matrix element is wrapped by parentheses to form a sequence element. Several elements are concatenated to form the matrix row, a sequence. Several rows, i.e. sequences, form the overall matrix.

```
#define PROCESS_NOISE_COVARIANCE \
  ( (1) (0) ) \
  ( (0) (2) )
```

The above matrix is represented by a sequence of rows, whereas each row is a sequence of the elements in a row.

Inputs, outputs and functions can be defined similarly. During compile time the size and elements are extracted with special Boost preprocessor macros and used to initialize variables, e.g. covariances, and generating functions, e.g. receive methods for ROS topics.

Parts of the Configuration Header

The concrete estimator, or filter respectively, can be created by specifying the parameters through macros in an appropriate header file named `config.h`.

The parts of the configuration header are summarized in Figure 4.16 and explained in the list below.

- Define the method. Available methods are listed in a separate header file (`pre_config.h`), these are:

  ```
  KALMAN_FILTER
  EXTENDED_KALMAN_FILTER
  UNSCENTED_KALMAN_FILTER
  PARTICLE_FILTER_SIR
  MOVING_MEDIAN
  ```

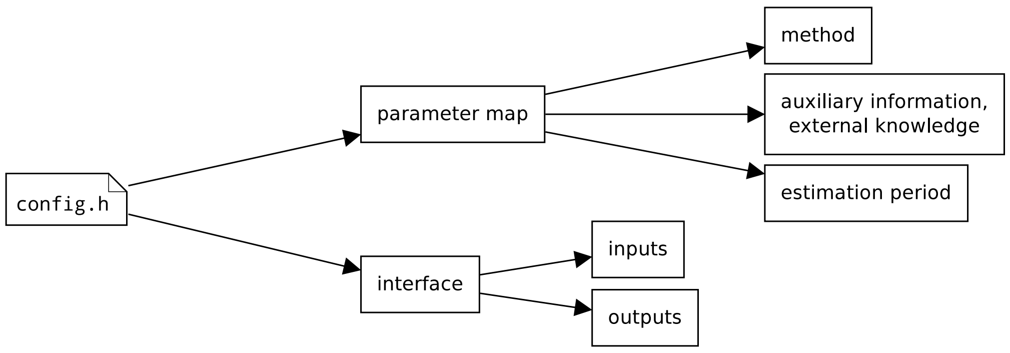

Figure 4.16: Configuration parts extracted from the configuration header.

```
MOVING_AVERAGE
CONFIDENCE_WEIGHTED_AVERAGING
```

A macro defining a method is provided as an integer to be able to compare such macros, e.g. to choose which estimator should be initialized. The user is able to select the method with

```
#define METHOD        KALMAN_FILTER
```

- Define the method specific parameters, i.e. auxiliary and external information. These macros depend on the chosen method. Such parameters often initialize matrices, vectors, sizes or flags. A more complex parameter is a set of functions, e.g. the state transition model of the extended Kalman filter. Considering the method, some parameters are required (e.g. the model of a KF), some may be optional (e.g. the initial error covariance). E.g. the state transition model for a UKF estimating the state of a moving train (example in Section 2.2.1) is configured with

  ```
  #define FILTER_T     0.1
  #define STATE_TRANSITION_MODEL \
    ( x[0] + x[1]*FILTER_T ) \
    ( x[1]                 )
  ```

  Because the state consists of two variables, namely velocity and acceleration of the train, the model is specified by two functions. One formula for each state variable is required.

  The state variables `x[0]` (velocity) and `x[1]` (acceleration) in the formulas correspond to the posteriori state of the last estimation. The variables are numbered from 0 to $n-1$ (n is the size of the state).

  The configuration generates the state transition function, required by a state estimator. In this function the formulas from the macro are evaluated and assigned to the a priori estimated state in the order they are given, i.e. the first item in the sequence will be applied to the a priori `x[0]`, the second item to the a priori `x[1]`, and so on.

- Define the estimation period, i.e. the time between successive estimations. The rate of publishing the estimated state can be specified with this parameter.

Parameters may have default values. The default value of the estimation period is $100ms$. To change the estimation period to e.g. 50ms one must use (note the period is given in ms)

```
#define ESTIMATION_PERIOD       50
```

- Define the inputs of the ROS node, i.e. measurements or controls. Topic name, topic type and the field in the topic containing the value to estimate are needed to specify an input. Additionally the field containing the variance may be specified too. Further the include files of the message types have to be defined, otherwise the fusion node cannot be compiled successfully. An input must be specified with

  ```
  #define TOPICS_IN \
    ( (name1) (type1) (value1) (variance1) ) \
    ...
    ( (nameN) (typeN) (valueN) (varianceN) )
  ```

  This macro is already a more complex one. The number of inputs may vary. Further the variance is optional. Configuring the ROS node to accept the acceleration into x-direction given in a topic named `acceleration` of type `Vector3Stamped` (from the geometry package provided by ROS) has to be defined with

  ```
  #include <geometry_msgs/Vector3Stamped.h>
  #define TOPICS_IN \
    ((acceleration) (geometry_msgs::Vector3Stamped) (vector.y))
  ```

  For every input in the `TOPICS_IN` macro a subscriber is produced and initialized. Further a callback function for the ROS node is generated.

  The control inputs have to be defined within the macro `TOPICS_IN_CTRL` similar to `TOPICS_IN` above. Control inputs are optional, so this macro is not required in the configuration header. Control inputs have no variance, this element of the sequence must be omitted.

- Define the output. The outputs can be chosen from the estimated state w.r.t. an index. Further the name of the topic can be specified.

  ```
  #define TOPICS_OUT \
    ((name1) (index1)) \
    ...
    ((nameN) (indexN))
  ```

  The index must not be greater than or equal the number of state variables, the valid range goes from 0 to $n-1$. E.g. the estimated state variable x_0 representing the velocity of an entity can be published with

  ```
  #define TOPICS_OUT \
    ((velocity) (0))
  ```

  For each output a publisher is produced and initialized. Further the code for publishing the appropriate state variable is generated.

The complete list of macros with additional description can be found in the documentation of the sensor fusion package. Which parameters are required or optional are stated in the code documentation of `sf_pkg`.

Structure

There are lots of header files for "parsing" the configuration. Additionally some validation is done w.r.t. the chosen method. Figure 4.17 shows the dependency of the header files.

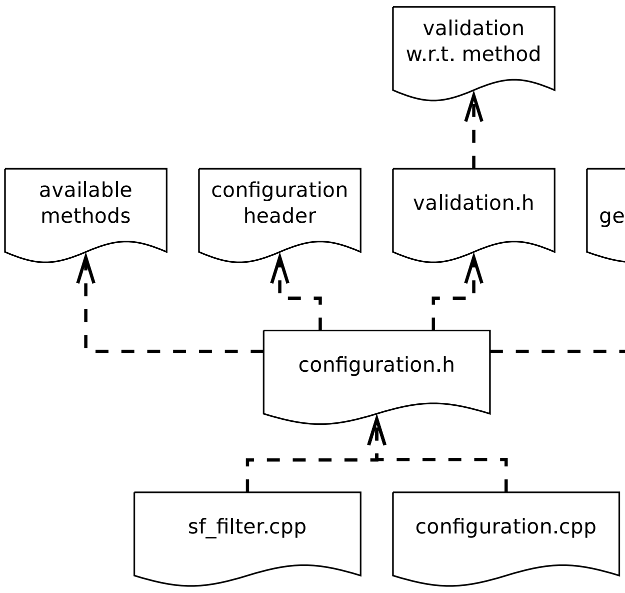

Figure 4.17: Dependency of header files for configuration.

The generated code is located in two files which together form the fusion node, namely `configuration.cpp` and `sf_filter.cpp`. The former contains the initialization of the estimator factory, i.e. adding the macros as parameter map to the factory. Functions applying the models of state estimators are located in this file too. The latter is the actual ROS node performing the estimation. This part subscribes to topics, receives the messages, collects and passes them to the estimator. Further the estimated state variables, defined as node outputs, are published.

The header `configuration.h` collects the headers needed for generating the code. First the available methods are included. In the configuration header, provided by the user, one of these methods is selected and the parameters for method and node specified. It follows a validation of the parameters, e.g. the size of the state transition matrix is checked to conform the size of the process noise covariance. The validation depends on the method (averaging has other parameters than a particle filter), so `validation.h` selects the appropriate validation w.r.t. the method. Finally the macros generating the code, e.g. receive functions for inputs, are included.

The problem with (these) header files is that a typo in the definition of the parameters may produce strange errors and is difficult to locate. The generation of the code is done with deeply nested macros. Hence the error messages during compilation often contain macros which were not even used directly. So special care must be taken when writing a configuration header. Some typical error messages are collected in the code documentation.

Use Case

The common approach for creating a fusion node is depicted in Figure 4.18. `sf_filter` contains the sources and headers to build a fusion node, i.e. it can be used as a template for each fusion node to create. So to use a ROS fusion node and configure it w.r.t. an application, the sub-package `sf_filter` has to be copied (it may remain in the overall package `sf_pkg`).

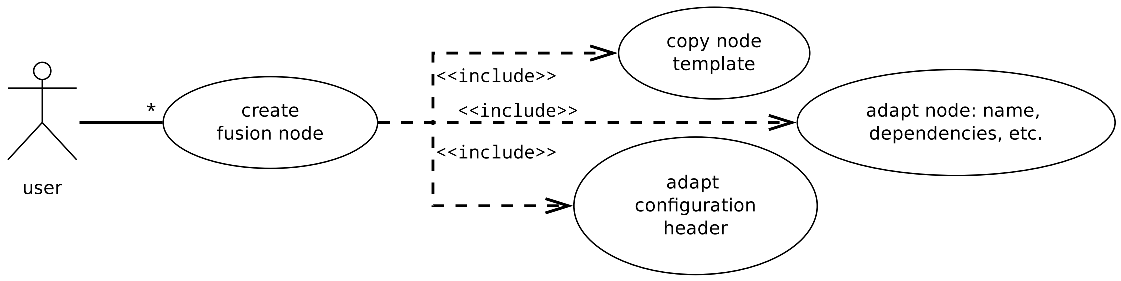

Figure 4.18: Using the sensor fusion package to create a ROS fusion node.

First the name of the new ROS node should be specified, i.e. the name must be adapted in the file `package.xml` (contains the meta data of the node for ROS) and `CMakeLists.txt` (specifies how to build the node). Next additional dependencies have to be specified in these two files. E.g. when subscribing to a topic of type `Vector3` of the package `geometry_msgs` (contains message types concerning geometry), this dependency on `geometry_msgs` should be listed. It remains to adapt the configuration header.

CHAPTER 5
Evaluation

The accuracy and performance of the state estimation algorithms are tested with a navigation application on a robotic platform.

The application, robot and configuration are described in the next section. The results of various configurations and optimizations are presented in Section 5.2. It follows a section containing guidelines for setting parameters of the state estimation methods, summarized with a short tutorial how to set up the configuration of an estimator.

5.1 Application

To test the filters the framework has been installed on a Pioneer 3-AT from MobileRobots [1] running the Robot Operating System [25]. The application is position estimation by dead reckoning, i.e. the position is calculated out of a previously determined (initial) position by considering speed, heading and elapsed time. This kind of position estimation is often executed only with wheel encoders. The accuracy of the estimated position suffers from slipping of wheels, different kinds of driving surface, mechanical parameters, encoder resolution and radius variations of the wheels [11]. In addition to the wheel encoders the robot is equipped with an accelerometer and a gyroscope to evaluate the benefits of sensor fusion.

For an experiment the robot is placed on a marked starting position. The P3-AT is set up and several filters are started to estimate the position. The robot is then controlled to drive on an arbitrary path finally returning to the starting position. The deviation to the starting position is measured (final actual position) and compared with the final position logged by the filters (final estimated position).

The following sections deal with the setup of the test application. The next section describes the hardware, i.e. the robot and the sensors. Section 5.1.2 proposes the basic configuration for the state estimation methods which are used in the experiments. The models for the navigation application are stated and the noise covariances are defined. In Section 5.1.3 the setup of the ROS system on the robot and the interaction of the ROS nodes are described.

5.1.1 Robot and its Sensors

The Pioneer 3-AT is a four-wheel drive robot. Wheels and sensors are controlled by a microcontroller which is connected to an on-board CPU running ROS.

The basic version is already equipped with several sensors including front and rear sonar arrays to measure distances, bumper arrays to detect collisions and wheel encoders. To reach a better accuracy of the estimated position, additional sensors (accelerometers and gyroscopes) have been mounted onto the top of the robot. For the application the wheel encoders, the accelerometer on the *base link* (the rotational center of the robot) and a gyroscope is used (see Table 5.1).

Wheel Encoder. Encoders can be used to measure the angular speed of a wheel. Considering the radius of a wheel the encoder is used to calculate the linear speed of a wheel. Furthermore the linear and angular speed of the vehicle can be calculated w.r.t. the geometry of the driving wheels and the traction scheme of the robot.

Accelerometer. An acceleration sensor measures the linear acceleration typically in 2 or 3 axes. When the sensor is accelerated in +x, +y or +z direction, the corresponding output will increase (and vice versa for the negative direction). So the sensed values can be accumulated to estimate the linear translation into a specific direction.

Gyroscope. A gyroscope measures the angular velocity typically in 3 axes. So the measurements can be accumulated to obtain the heading, or orientation respectively, of an object (considering an initial heading).

Name	Type	Description
wheel encoder left	encoder	speed of robot's left wheels
wheel encoder right	encoder	speed of robot's right wheels
KXTF9	accelerometer	acceleration
IMU3000	gyroscope	angular velocity

Table 5.1: Summary of used sensors for the test application.

The board containing the accelerometer and gyroscope is mounted near the base link. Note that the accelerometer at the base link is *not* mounted according to the coordinate frame of the robot, named *base frame*. In fact the accelerometer's y-axis corresponds to the robot's x-axis (see Figure 5.1).

To be able to fetch the data of the additional sensors a Raspberry Pi running ROS has also been mounted on the top of the robot. Two ROS driver nodes `kxtf9` and `imu3000` sample the data via I2C interface and publish the values in fundamental units (m/s^2 and rad/s).

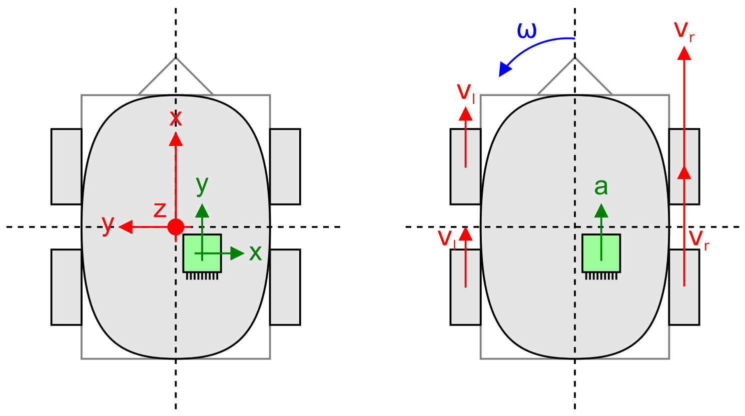

Figure 5.1: The robot and its sensors. The left sketch shows the base frame of the robot (red) and the coordinate frame of the accelerometer (green). The right sketch depicts the properties measured by sensors.

The sensor measurements of the basic version of the robot are published by the ROS package `rosaria` [28] which establishes a connection to the Pioneer's microcontroller through the ARIA library [2] provided with the robot. At the time of writing `rosaria` does not publish the wheel velocities itself. So the package was forked and updated to do so. To match the standards of ARIA and `rosaria` this is done by publishing the speed of wheels in mm/s.

Node	**Topics**	**Message Type**	**Unit**
rosaria	velocity_wheel_left	std_msgs/Float32	mm/s
	velocity_wheel_right	std_msgs/Float32	mm/s
kxtf9	acceleration	geometry_msgs/Vector3Stamped	m/s^2
imu3000	angular_velocity	geometry_msgs/Vector3Stamped	rad/s

Table 5.2: Nodes publishing the topics containing measurements for sensor fusion.

For the test application only specific fields of the topics listed in Table 5.2 are needed. The angular velocity is only needed to estimate the heading of the robot, hence the field `vector.z` of the topic `angular_velocity` is used. Similar applies for the acceleration. The accelerometer is mounted near the base frame of the robot, hence the centrifugal force can be neglected. Therefore only the field `vector.y` is extracted for the fusion which corresponds to the acceleration in x-direction of the robot.

5.1.2 Configuration

This section specifies the initial configuration of the filters. The used formulas of the models and the covariance matrices are listed. Further improvements of the configuration are proposed with the results in Section 5.2 and in the guidelines of Section 5.3.

Model

The differential equations for acceleration, speed and distance are the base for the model. The formulas for the state transition and measurement model needed for an estimation algorithm are created as described in Section 2.2.

State transition model. The state transition model only contains the basic properties position (x,y), orientation θ, distance covered since last estimation Δs, speed (linear v and angular ω) and linear acceleration a.

Each property has been chosen to be included into the state vector either because its a property of interest, e.g. position, or a property which can be measured by a sensor (directly or indirectly), e.g. acceleration, and can be used to derive a property of interest.

The basic properties are acceleration and angular velocity of the robot. Out of the acceleration and angular velocity other properties are calculated. The basic variables (acceleration and angular velocity) are assumed to be constant. In this simple model one doesn't know it better (how the basic properties would change) and therefore models the reality to be sluggish, i.e. assumes that the basic properties remain the same from one estimation to another. The possible change of the variables can be modeled in the process noise covariance. Anyway, the estimates of a and ω are corrected by measurements. The state transition formulas are as follows:

$$x_0 : \quad x = x + \Delta s \cdot cos(\theta) \tag{5.1}$$

$$x_1 : \quad y = y + \Delta s \cdot sin(\theta) \tag{5.2}$$

$$x_2 : \quad \theta = \theta + \omega \cdot T \tag{5.3}$$

$$x_3 : \quad \omega = \omega \tag{5.4}$$

$$x_4 : \quad \Delta s = v \cdot T \tag{5.5}$$

$$x_5 : \quad v = v + a \cdot T \tag{5.6}$$

$$x_6 : \quad a = a \tag{5.7}$$

All values of the state transition model are held in fundamental units, e.g. the velocity is represented in m/s, and correspond to the robot's base frame.

The enhanced state transition model takes the control input into account. The robot is controlled by a teleoperation node which publishes a topic named `cmd_vel`. The messages of this topic contain the desired linear and angular velocity. So during normal operation the linear and angular velocity of the state should correspond to the command given by the teleoperation node `pioneer_teleop` published through the topic `cmd_vel`. There are situations where this model is inaccurate or may fail. E.g. when the robot accidentally drives into an obstacle the proposed model does not reflect the reality. The robot decelerates but the control input remains the same.

$$x_0: \quad x = x + \Delta s \cdot cos(\theta) \tag{5.8}$$
$$x_1: \quad y = y + \Delta s \cdot sin(\theta) \tag{5.9}$$
$$x_2: \quad \theta = \theta + \omega \cdot T \tag{5.10}$$
$$x_3: \quad \omega = \texttt{cmd\_vel.angular.z} \tag{5.11}$$
$$x_4: \quad \Delta s = v \cdot T \tag{5.12}$$
$$x_5: \quad v = v + a \cdot T \tag{5.13}$$
$$x_6: \quad a = (\texttt{cmd\_vel.linear.x} - v)/T \tag{5.14}$$

The difference of desired velocity given with the control input (value of `cmd_vel.linear.x`) and estimated velocity v evaluates the expected acceleration a.

The state transition models above are not the only solutions. E.g. Formula 5.13 and 5.14 from the enhanced state transition may be replaced by

$$x_5: \quad v = \texttt{cmd\_vel.linear.x}$$
$$x_6: \quad v_{old} = v$$

In comparison to the original enhanced state transition model, the control input is directly mapped to the linear velocity. The variable v_{old} is needed to calculate the acceleration (e.g. in the measurement model or as an additional state variable) which is measured by the accelerometer ($a = (v - v_{old})/T$).

However the first two models (Formula 5.1 to 5.7 and Formula 5.8 to 5.14) are used to be able to log the acceleration directly. Note that Δs isn't necessary too, i.e. $v \cdot T$ could be used directly to calculate x and y, but used for the same reason as the acceleration, namely logging.

Measurement model. The measurement model maps the state variables to the observations given by a sensor. The measurements from the accelerometer and gyroscope directly correspond to the state variables acceleration and angular velocity respectively. The linear and angular velocity on the other side can be derived from the measurements of the wheel encoders.

To derive the linear and angular velocity from the wheel velocities, the traction scheme or driving mechanism respectively of P3-AT has to be considered. The traction scheme of the P3-AT robot is skid-steering, i.e. the velocities of the left and right wheels are controlled similar to differential drive. E.g. when the robot rotates on a spot, the velocities of the left wheels equal the velocities of the right wheels, but are turning in the opposite direction. So for a simple kinematic model of skid-steering, differential drive is assumed [19], depicted in Figure 5.2.

In Figure 5.2 the velocities of the right wheels v_r are higher than the velocities of the left wheels v_l. Hence the robot drives a left curve. Note that the radius R does not equal the half of the track width as it is the case in differential drive. The linear velocity v, w.r.t. the robot's frame, is the mean of the left and right wheel velocities.

$$v = \frac{v_l + v_r}{2}$$

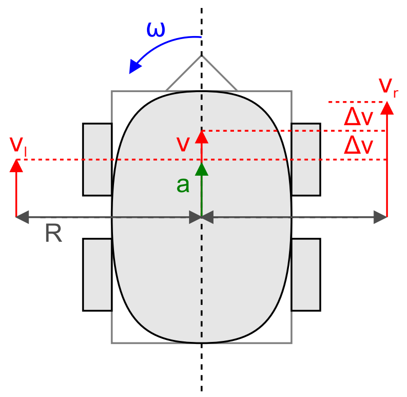

Figure 5.2: Kinematic model of Pioneer 3-AT [19].

The angular velocity ω of the robot can be calculated from the wheel velocities too, but w.r.t. the radius R.

$$\omega = \frac{v_r - v_l}{2R}$$

The radius is assumed to be constant (note, it generally depends on slippage, center of mass of the robot, etc. [19]). An odd parameter R causes a diverging estimated angular velocity w.r.t. the actual ω. In particular when changing the robots orientation (e.g. driving a curve or rotating) the estimated heading θ (evaluated from ω) diverges from the actual orientation and in further consequence the estimated position from the actual position. To match the radius as good as possible, R has been evaluated during a parameter estimation, i.e. parameter R has been estimated with a separate filter using an appropriate model. The parameter estimation evaluates the radius R to equal $0.3m$ (confirmed by [19]).

For the measurement model the above formulas for linear and angular velocity have to be reshaped. The resulting measurement model states:

$$z_0 : \quad \texttt{angular\_velocity.vector.z} = \omega \tag{5.15}$$

$$z_1 : \quad \texttt{acceleration.vector.y} = a \tag{5.16}$$

$$z_2 : \quad \texttt{velocity\_wheel\_left.data} = (v - \omega \cdot R) \cdot 1000 \tag{5.17}$$

$$z_3 : \quad \texttt{velocity\_wheel\_right.data} = (v + \omega \cdot R) \cdot 1000 \tag{5.18}$$

The field `vector.z` of the topic `angular_velocity` represents the angular velocity ω, i.e. the rotational part of a movement. `acceleration.vector.y` gives the acceleration in in x-direction w.r.t. the robot's frame, see Figure 5.1. Note that the units of the wheel velocities given by `velocity_wheel_left.data` and `velocity_wheel_right.data` are not equal to the unit of v (Table 5.2), so the factor 1000 is used for conversion.

Covariances

Process Noise Covariance. The process noise covariance (PNC) reflects the inaccuracy of the state transition model. It is assumed that the differential equations of the model listed in the previous section are complete and correct. But one cannot predict the movement of the robot, i.e. the controls from the user, so the value of a and ω may change with a specific amount. This specific amount is projected into the process noise covariance.

The robot's microcontroller constrains the change of linear acceleration a to about $0.3m/s^2$. The maximum deviation of the angular velocity ω is evaluated considering the maximum rotation in the datasheet and the maximum changes of the user command. The robot is able to rotate with $140°/s$ $(1.2rad/s)$. The commands from the user are bounded to $0.7rad/s$. So a change of $0.7rad/s$ is the worst case deviation. Although the standard deviations or variances would be smaller than these maximum deviations, the variances of linear acceleration and angular velocity are set to $0.3m/s^2$ and $0.5rad/s$ respectively.

These considerations are valid for a continuous time model, but generally not for a discrete time model. The implemented filters running on the CPU and estimating every T seconds need the *discrete* process noise covariance (here named as Q_k or simply Q if constant). As a first-order approximation [10]

$$Q_k = T \cdot Q(t) \tag{5.19}$$

may be used. Note that the discrete process noise covariance depends on the estimation period T. The lower the estimation period the lower the process noise covariance should be, otherwise the system is modeled with more uncertainty than with higher estimation periods.

In the test application the process noise covariance remains constant, so the discrete PNC is simply denoted by Q (without the time index k). The values in the diagonal of Q represent the variances of the state variables from one estimation to another.

$$Q = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.3 \end{pmatrix} \tag{5.20}$$

For testing with different estimation periods, the process noise covariance has been set to a higher value than necessary (for the PNC given above, T is set to $1s$). It is generally better to set the process noise covariance to a higher value when trying the filter for the first time. When the process noise is too low the measurements are not trusted and an error in the state transition model makes the estimation very inaccurate or even unusable.

Particle filters are often not suitable for a low process noise covariance. The diversity of the particles are "generated" by the process noise. Hence when the process noise is low an appropriate diversity cannot be reached which is needed to match the actual state as accurate as possible. So the covariance for the SIR particle filter is set to:

$$Q = \begin{pmatrix} 0.1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.3 \end{pmatrix}$$

Measurement Noise Covariance. The measurement noise covariance reflects the inaccuracies of the sensors. The variance of a sensor may be taken from its datasheet or by sampling several measurements to evaluate the real variance. The variance of the accelerometer and gyroscope correspond to the typical deviations given in the datasheet. The uncertainties of the wheel encoders have been evaluated by sampling, i.e. the wheel velocity given by the encoders during constant driving speed has been logged and the variance has been calculated out of the deviations.

The variances of the sensors are located in the diagonals of the measurement noise covariance, i.e. coefficient in row 1 and column 1 represents the variance of measurement z_1 (angular velocity around the z-axis of the gyroscope). All non-diagonals are set to 0, indicating that the measurements are independent from each other.

$$R = \begin{pmatrix} 0.00017 & 0 & 0 & 0 \\ 0 & 0.25 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

The values are in the units given by the sensors. E.g. the variance of the wheel velocities is $2mm/s$.

5.1.3 ROS Setup

All basic tasks run on the CPU of the P3-AT robot, e.g. controlling the robot, filtering and logging. The Raspberry Pi fetches data from the additional sensors mounted on the top of the robot and publishes these values to the ROS system. Since the Raspberry Pi is connected via Ethernet with the CPU of the P3-AT, published information is available on both computers.

The P3-AT's computer can be accessed through wireless LAN. An SSH connection to the CPU and further an SSH connection from the CPU to the Raspberry Pi is established to start the ROS nodes (the Raspberry Pi and the notebook are not in the same network). In Figure 5.3 the structure of the robot regarding the communication is depicted.

First the ROS master, i.e. the central coordinator of all ROS nodes, is started on the CPU. Then the drivers are started on the Raspberry Pi which perform an offset calibration and then immediately start to publish the sensor values. Finally a launch file is called to create the nodes

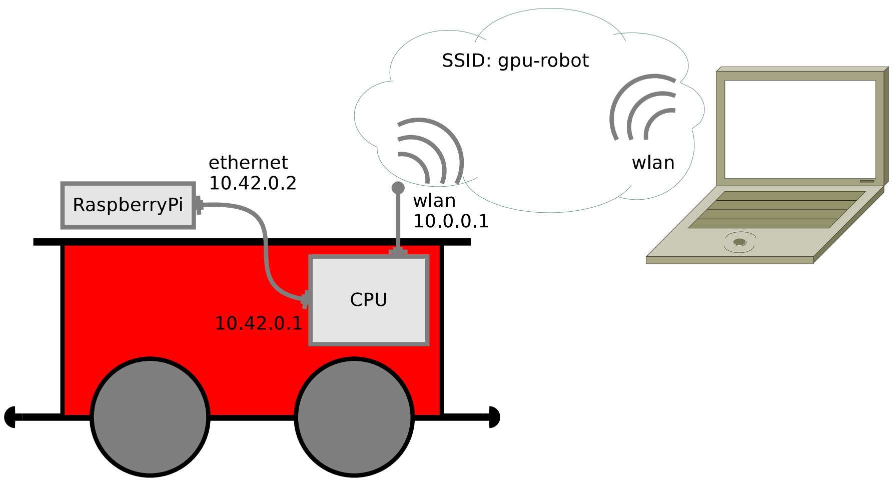

Figure 5.3: Communication structure of Pioneer 3-AT.

for controlling (`/pioneer_teleop`), filtering and logging. The nodes running in ROS are depicted in Figure 5.4.

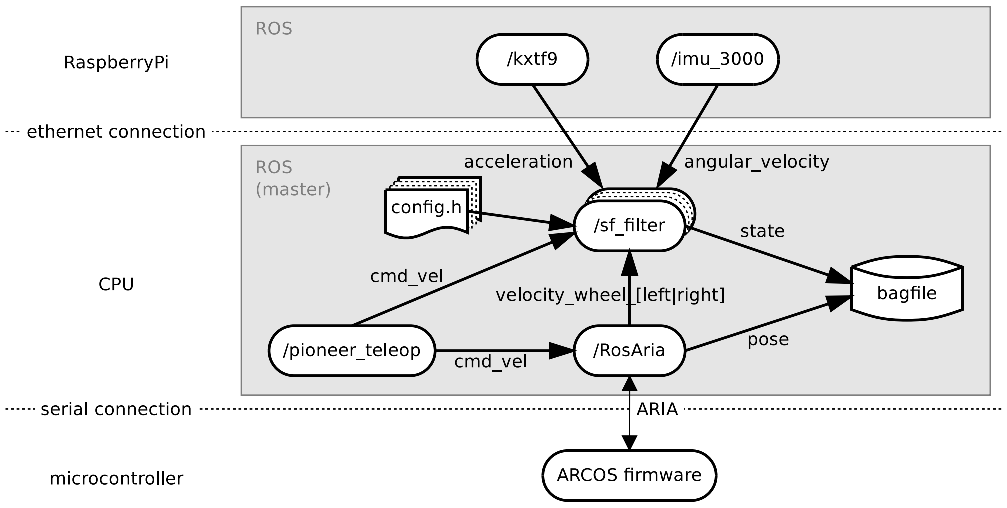

Figure 5.4: Nodes running on the robot during test.

Several filters with different configurations are started in parallel to compare the results of different filter methods, models and other settings. The messages are logged with `rosbag`, a tool provided by ROS, which is able to *record* and *play* back published messages. The generated *bag file* can be converted into CSV (comma-separated values) to plot it, e.g. with gnuplot.

5.2 Experimental Results

For the experiments different models are used to show the improvements in accuracy achieved with additional information (measurements or controls). These models are described in the next section. The first experiments with little information (measurements and controls) follow, which already show the advantage of the state estimation methods compared to averaging. In Section 5.2.3 all models are compared. Then Section 5.2.4 compares the execution time of the filters. Finally some parameters are tuned to give even better results w.r.t. accuracy (Section 5.2.5).

5.2.1 Introducing the Model Variants

Throughout the experiments several variants of the models proposed in Section 5.1.2, regarding the number and type of inputs, have been used. The abbreviations of the variants are described below.

nav1 Uses only the accelerometer and gyroscope to estimate the position. Hence the position is accumulated based on the acceleration and angular velocity. These two basic state variables are assumed to be constant. The used formulas of the model are Formula 5.1 to 5.7 and 5.15 to 5.16.

nav1ctrl In addition to *nav1* the control input `cmd_vel` is used. So in comparison to *nav1* the acceleration and angular velocity is not assumed to be constant, moreover it follows the control from the teleoperation node. So for this model the Formula 5.8 to 5.14 and 5.15 to 5.16 are used.

nav2 Uses only the wheel velocities to estimate the position. The used formulas are Formula 5.1 to 5.6 and 5.15 to 5.18 In this model the state variable for the linear acceleration is not needed and can be removed. There is no need to hold a state variable when the property is of no use. Hence the Formula 5.6 ($v = v + a \cdot T$) is set to $v = v$.

nav2ctrl In addition to *nav2* the control input `cmd_vel` is used. The state transition formulas are Formula 5.8 to 5.13 and 5.17 to 5.18 where the next state equation of v, Formula 5.13, is replaced by $v =$ `cmd_vel.linear.x`.

nav3 Uses accelerometer, gyroscope and wheel velocities to derive the position. The big difference to the above models is that two sensor sources are fused to one property, namely the wheel encoders and (indirectly) the acceleration sensor to the linear velocity, and the wheel encoders and the gyroscope to the angular velocity. This model promises a better accuracy than the previous models. The model formulas are Formula 5.1 to 5.7 and 5.15 to 5.18.

nav3ctrl In addition to *nav3* the control input `cmd_vel` is used. So the equations for the model are Formula 5.8 to 5.14 and 5.15 to 5.18.

nav3red The state vector of the model *nav3ctrl* is reduced to its minimal size of acceleration, linear and angular velocity. The evaluation of the position described by (x,y,θ) is not part of the filter (here the position is offline calculated from the logged reduced state). So only Formula 5.11 to 5.14 and 5.15 to 5.18 are used. This model is used to show the reduction of execution time of the estimation w.r.t. the state size.

5.2.2 From Averaging to Estimation

The accelerometer (KXTF9) mounted on the top of the robot occurred to be quite noisy or inaccurate respectively. Beside noise the accelerometer suffers from an output bias which changes over time (so the offset calibration during start up does not overcome this problem). Therefore the velocity accumulated from the measured acceleration increases steadily. This results in an unusable position estimation.

However when fusing the desired velocity given by the topic `cmd_vel` with the accelerometer data (model *nav1* $\rightarrow$ *nav1ctrl*), the position estimation with this bad sensor results in a good accuracy.

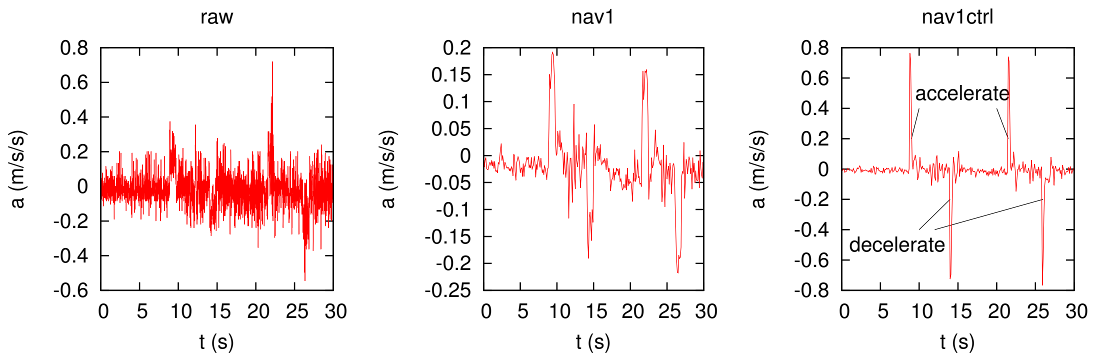

Figure 5.5: Measurements from the KXTF9 (raw) and estimated acceleration (*nav1* and *nav1ctrl*) when the robot is accelerating and decelerating regularly.

The improvement of the estimated acceleration is shown in Figure 5.5. The estimation is executed every 100ms but the accelerometer's publishing period is $10ms$, so 10 measurements are averaged and passed to the estimator. Additionally, the fusion algorithm filters these inputs (*nav1*) and fuses the acceleration with the desired velocity or acceleration respectively (*nav1ctrl*).

Considering the raw measurements of Figure 5.5 it seems surprising that a dead-reckoning navigation with this accelerometer could provide useful information. One can also calculate the path simply with the process model or the desired velocities given by the user respectively, omitting the measurements. But this would only be a guess and would fail as soon as the robot e.g. crashes into an obstacle. State estimation algorithms use both, the process model (in *nav1ctrl* improved by the controls) and the observations and are therefore more accurate than using measurements or desired velocity alone - the more information is fused, the better is the result. The path of the different models are compared in Figure 5.6 using an EKF.

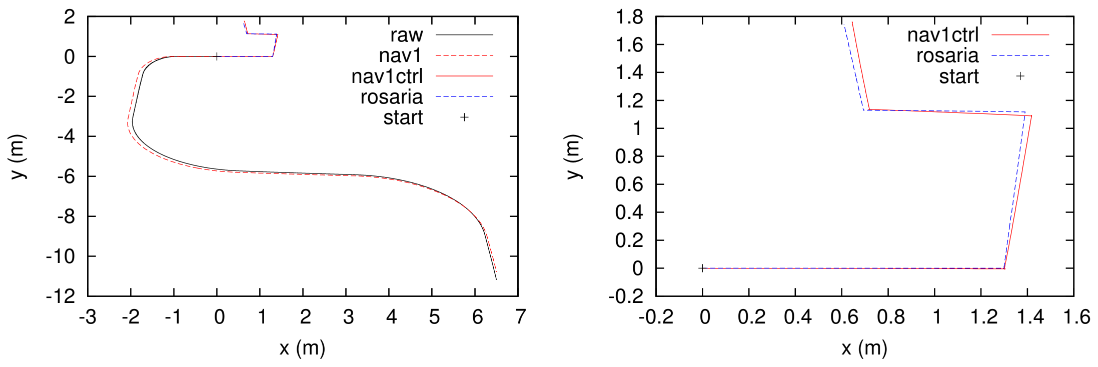

Figure 5.6: Path of robot when using model *nav1* and *nav1ctrl*.

Additionally the position evaluated by `rosaria` using the wheel encoders is plotted in the figures. The path *raw* is calculated using the raw accelerometer and gyroscope data of the sensors. Here the position is accumulated according to the state transition model.

The very similar paths *nav1* and *raw* originates from the fact that an estimation algorithm configured with a model, where the basic properties or variables respectively are assumed to be constant, is the same as an averaging algorithm (the “amount” of averaging is specified by the measurement and process noise covariances). In model *nav1* the data of accelerometer and gyroscope is not substantiated by other inputs, i.e. neither there is a sensor which provides measurements to the same property (a so-called *replicated* sensor) nor there is a control input supplying information to the same property. Hence this filter is no improvement to averaging or no filtering at all.

The averaging property of the estimation with model *nav1* is also visible when examining the peaks of the acceleration in Figure 5.5. One may note that the maximum acceleration with *nav1* is much smaller than the raw sensor data, i.e. the data is smoothed.

5.2.3 Model versus Reality

Adding the wheel encoders to the model gives additional information about linear and angular velocity. Accelerometer and wheel encoders are fused to the linear velocity, and the gyroscope and the wheel encoders (w.r.t. to the turning radius R) to the angular velocity (see sketch of the robot in Figure 5.2).

The accuracy depends on the settings of the parameters, e.g. the measurement noise covariance. If the accelerometer is trusted more than the wheel encoders, i.e. the variance of the wheel encoders are proportionally higher (although the wheel encoders are more accurate), the position estimation will be poorer. So, to compare the models equal configurations of covariances and estimation period are used (default settings of Section 5.1.2).

The left graph in Figure 5.7 shows the estimated paths of an experiment running five UKFs parallel estimating the position of the robot. The right graph depicts the final estimated positions of the filters and the final actual position of the robot. The final orientation is indicated by an arrow.

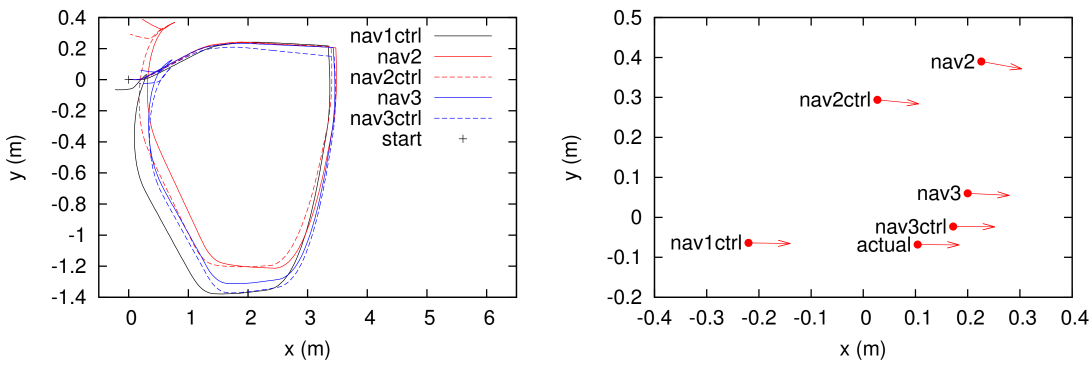

Figure 5.7: Paths and final positions estimated by various UKFs using different models.

As with the model *nav1* and *nav1ctrl* the accuracy increases the more information is fused. Model *nav3ctrl* uses all measurements and controls available and therefore gives the best results, i.e. the final estimated position of *nav3ctrl* is the closest one to the final *actual* position (see Figure 5.7). One may note that the final position of model *nav1ctrl* comes closer than *nav2* and *nav2ctrl*. It appears (regarding several experiments), that the accuracy of the results from the filters using the wheel encoders, depend on the path of the robot. One may also note that the headings of *nav2* and especially *nav2ctrl* differentiate from the other models. This indicates a drift of the wheels and can be calibrated to get better and more reliable results regardless which path has been traveled.

5.2.4 Execution Time and Different Algorithms

The measurements show that the execution time of the estimation must not be ignored when choosing the estimation period. Kalman filters are in general many times faster than a particle filter (regarding the comparison in Section 3.5 and the test results below). Particle filters use a high number of samples to estimate the state which makes the estimation slow. Further the UKF uses some samples for estimation too which makes it slower than a KF or EKF.

The performance of a (non-parallelized) particle filter is highly influenced by the number of particles N used for estimation. The lower N, the poorer the accuracy, but the higher the number of particles, the higher is the execution time. Table 5.3 shows the significant difference w.r.t. the execution time of Kalman filters and the particle filter. The filters are executed on the internal CPU of the robot (Mamba EBX-37: Dual Core 2.26 GHz, 2 GB RAM). Note that the number of particles had to be reduced, to be able to execute the estimation of the particle filter within an estimation period of $100ms$.

	KF	EKF	UKF	PF	N
nav3ctrl	0.9	1.6	5.8	180	700
nav3red	0.8	1.1	4	79	500

Table 5.3: Execution time of model *nav3ctrl* and model *nav3red*. All values in ms.

The models *nav3ctrl* and *nav3red* only distinguish in the number of state variables. Whereas the state in model *nav3ctrl* is of size seven, the model *nav3red* only uses three state variables. Table 5.3 shows that the execution time can be reduced by choosing a smaller state (if possible).

However the large state, including the position, is useless when estimating with a particle filter. The state variables x, y, θ and Δs estimated by a PF in this test application are useless (in every experiment). Figure 5.8 shows the estimated position (x, y) of an experiment using a particle filter. An estimation period of $200ms$ and 700 particles have been used.

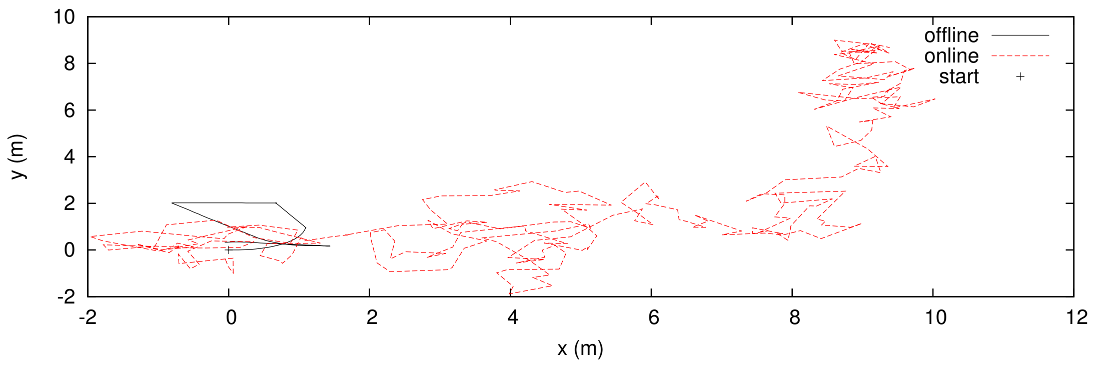

Figure 5.8: Path or position respectively estimated by a particle filter (online) and calculated from the estimated state variables v and ω (offline).

The state variables x and y change with every estimation but cannot be corrected by measurements. Hence the variance of x and y increases with every step, which causes the (few) samples to be spread over and over. Furthermore the particles are never weighted (by measurements), so only the state transition model is applied and random noise added. This randomness, added with the noise, causes a zigzag path. Similar is valid for the state variables θ and Δs, which cannot be corrected by measurements too.

Figure 5.9 shows the results of an experiment estimating the position with different state estimation algorithms (KF, EKF, UKF, PF) but equal configuration. The Kalman filter is not able to calculate the position directly (sin and cos are nonlinear functions), so the position is calculated offline from the logged state variables v and ω. The path of the particle filter is also calculated offline from the estimated state variables v and ω (due to the reasons given in the previous paragraphs). The left graph depicts the estimated path, whereas the right graph shows the final positions estimated by the filters compared to the final actual position.

The final estimated position of the KFs come closer to the final actual position of the particle

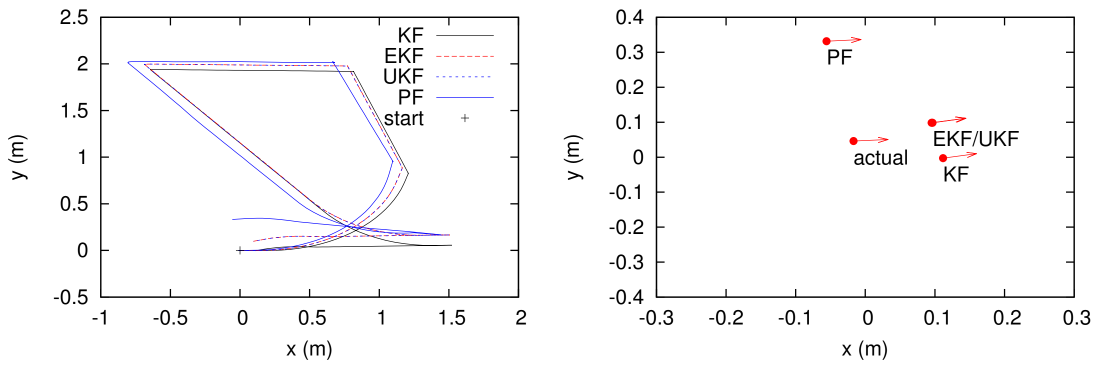

Figure 5.9: Path estimated by different filters using the *nav3ctrl* model (left). Final position, actual and estimated by different filters (right).

filters (right graph of Figure 5.9). For the particle filter too few particles are available to estimate the position as accurate as the Kalman filters. Using the reduced model *nav3red* and decreasing the number of particles to 500, the particle filter could be applied with an estimation period of $100ms$. Although the EKF and UKF include the position in the state, there are no sensors which measure the position of the robot. Hence all KFs (nearly) have the same accuracy. I.e. including the position as properties in the state vector of a filter or not does not make any difference, when there are no sensors correcting the state variables x and y. The path and final position of EKF and UKF cannot be distinguished in Figure 5.9, indicating that the first-order linearization of the EKF is sufficient in this test application.

These results show that for unimodal belief distributions the Kalman filters should be preferred over the particle filters. The Kalman filter is the optimal solution in such cases, however the particle filter is able to come close to the accuracy of the KFs when enough time and performance is available, i.e. the number of particles can be high.

5.2.5 Tuning

Except for the models the covariances and estimation periods may be changed. Several configurations have been tested to increase the accuracy of the estimation.

- Reduction of the estimation period. The estimation period is reduced to $10ms$ such that each measurement from the accelerometer can be passed to the estimation.

- Increase of the measurement noise covariance. The values of a sensor with high variance will be smoothed more than a sensor with lower variance. The measurement noise covariance is changed to

$$R = \begin{pmatrix} 0.0017 & 0 & 0 & 0 \\ 0 & 2.5 & 0 & 0 \\ 0 & 0 & 20 & 0 \\ 0 & 0 & 0 & 20 \end{pmatrix} \tag{5.21}$$

The variances of the sensors are each multiplied by 10. It is important to increase all variances with the same amount. E.g. when the encoder's variance is increased but not that of the accelerometer, i.e. the encoders lose their trustworthiness but not the accelerometer its own too, the estimated velocity gets more influenced by the noisy accelerometer than by the encoders w.r.t. the original configuration. So it is important to leave the proportion of the sensor variances unchanged.

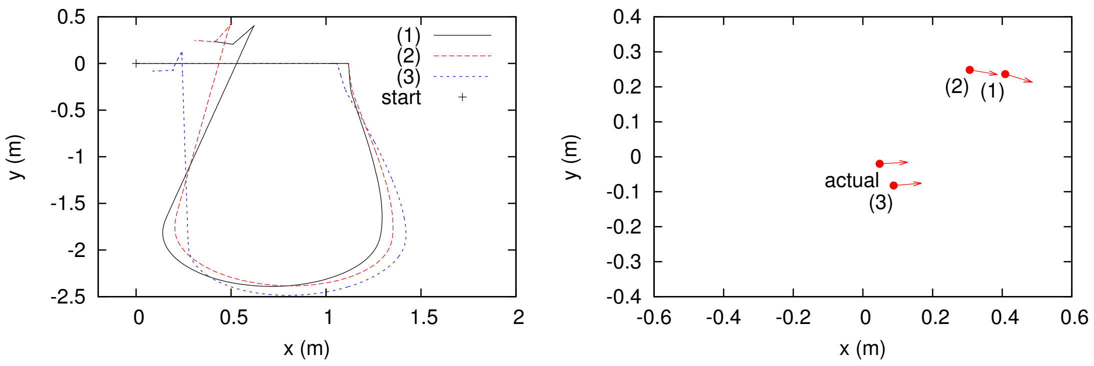

Figure 5.10: Estimated path by UKFs configured with the *nav3ctrl* model. (1) is the original configuration of Section 5.1 and an estimation period of $100ms$. (2) uses the increased measurement noise covariance, see Definition 5.21. (3) is configured with the increased measurement noise covariance as (2) but with an estimation period of $10ms$.

Figure 5.10 shows the comparison of the original configuration and the improvements listed above. A lower estimation period gives a better accuracy of the final position because more measurements "correct" the filter's estimate. The increase of the measurement noise covariance shifts some trustworthiness from the measurements to the process. This may not be optimal although it improves the result in Figure 5.10. E.g. when the robot drives over a slippy ground, the process excepts a higher translation (the desired translation) of the robot than the actual translation, caused by spinning of the wheels on that slippery surface. So the position will diverge by the amount the process is trusted.

- Decrease the process noise covariance. The default process noise covariance has been set to a high value (see Section 5.1.2). The process noise covariance may be adapted to model the possible state changes better, i.e. the possible change of variables is lower when estimating every $10ms$ instead of e.g. every $100ms$.

Figure 5.11 shows that a decreased process noise covariance gives much poorer results than the default PNC. The default process noise covariance has been multiplied by 0.1 or 0.01 respectively. The filters, initialized with different PNCs, estimate every $10ms$.

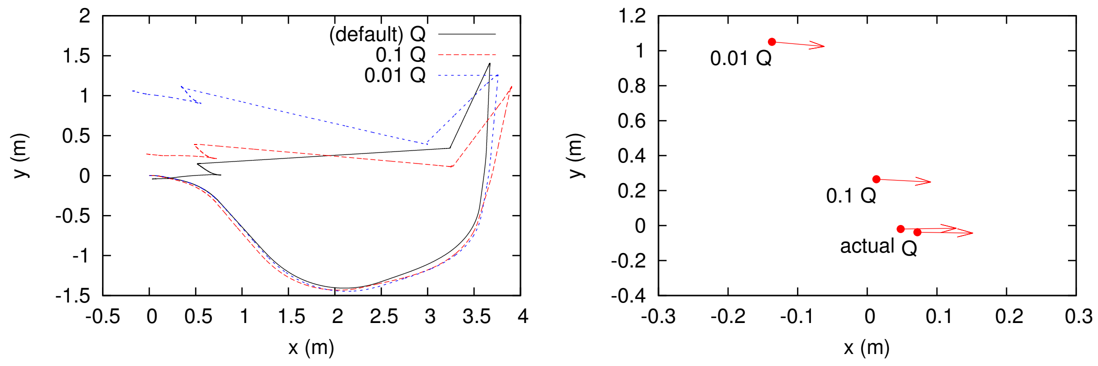

Figure 5.11: Estimated paths of UKF filters with different process noise covariances.

Although the low process noise covariance better fits to the estimation period of $10ms$ it does not fit to the possible change of the control input from the user. From one estimation to another (so in the time between two estimations) the command from the user may change from a velocity of 0 to a velocity of $0.3m/s$. But because this steep change is modeled to be very unprobable, i.e. a lower variance of e.g. $0.003m/s$ is assumed, the control input won't be trusted. Hence when the control from the user changes, the state variables, depending on the user's commands, perform a damped oscillation, i.e. the estimate is not "sure" to trust the measurements or the process. Figure 5.12 shows the impact of a small process noise covariance. The left graph depicts the velocity estimate when using the default process noise covariance Q. The right graph shows the velocity estimate when using $0.01\,Q$.

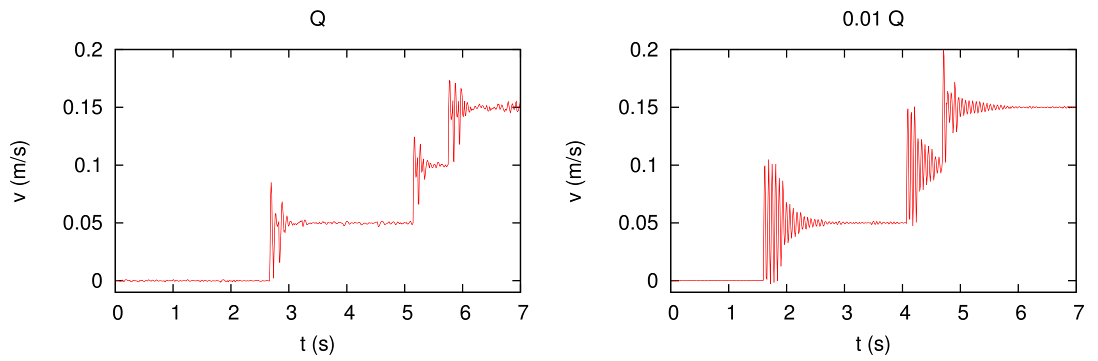

Figure 5.12: Velocity with two different process noise covariances.

So when setting the process noise covariance also the control inputs have to be taken into account (not only the state transition model, i.e. the time update equations).

The best improvement of the accuracy can be reached by the reduction of the estimation period. The process noise covariance has also a big impact on the accuracy of the estimation. The above results show that the results of the filters are highly influenced by the configuration. However care has to be taken when selecting the estimation period and the process noise covariance in particular. Section 5.3 summarizes the issues which may occur during configuration.

5.3 Configuration Issues

A careless configuration could cause errors during estimation. Furthermore the accuracy of the estimate can be increased by an appropriate selection of the parameters of a filter. This section describes possible issues with the parameters of a filter and gives suggestions how to select its values. A guideline for setting up a state estimator is given at the end of this section.

5.3.1 Choosing Parameters

Estimation Period

As described in Section 4.3, measurements are generally asynchronous and multirate. When only one sensor is used as input for estimation, the filter rate should be adapted to meet the publishing rate of this sensor. So each measurement of the sensor can be used to update the estimate.

If the measurements are taken exactly before the estimation is executed, this results in an optimal estimate. However in general this cannot be accomplished. ROS nodes cannot be started on specific time instants. Furthermore received measurements seem to drift (w.r.t. the estimation) because lots of nodes use delays to specify the time when a message should be published (instead of checking the time since the last send operation). The drift of an input w.r.t. the estimation of a ROS fusion node is depicted in Figure 5.13. The node providing the measurements, first samples the measurement, then publishes the value and finally awaits a *delay* of e.g. $100ms$ before sampling again. A ROS node subscribes to a measurement and publishes the last received measurement, whenever e.g. $100ms$ *elapsed*.

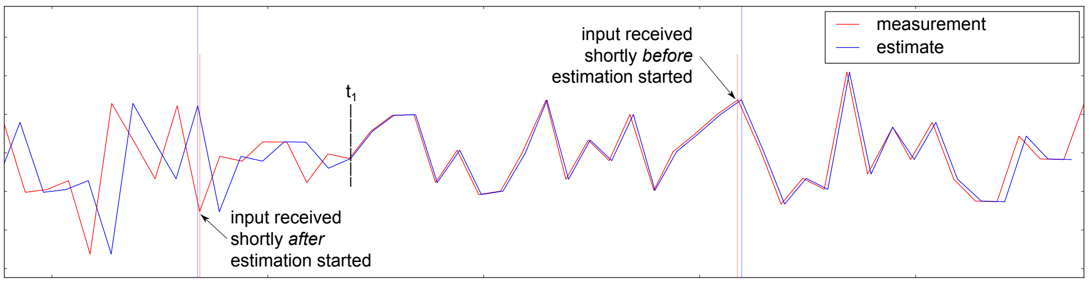

Figure 5.13: Fusion inputs may drift w.r.t. the estimation.

The measurements arrive asynchronous, i.e. the input may arrive shortly before, somewhere in between or shortly after the estimation is performed. The estimation to the left of time instant t_1 of Figure 5.13 uses “old” measurements. Whereas the estimate after time instant t_1 is more actual w.r.t. the measurement.

When measurements are received with different rates choosing the right estimation period may be difficult. The experiments show that it is generally better to choose a low estimation period. Figure 5.14 shows the increase of accuracy when using a low estimation period. The position is estimated by three UKFs each with different estimation period.

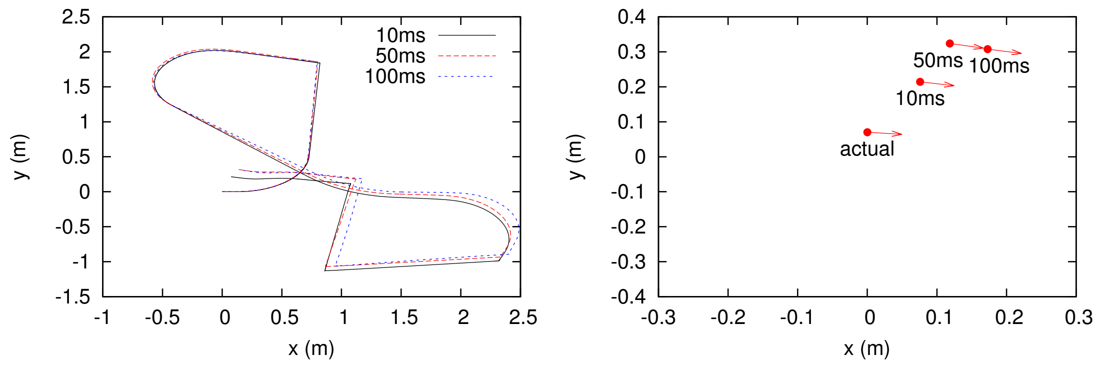

Figure 5.14: Estimated paths with different estimation periods.

When choosing the estimation period one should consider following points:

- Selecting an estimation period lower than the lowest publishing period of a sensor does not improve the estimate. Since the measurements correct the estimated state, at least one measurement should be available in each estimation cycle.

- Selecting an estimation period higher than the highest publishing period should be avoided. Many measurements will be averaged, and the estimate will be updated with less measurements. Furthermore state changes may be overlooked, therefore high estimation periods should not be used in highly dynamic systems. The only reason why a high estimation period may be needed is when the execution time of the estimation is higher than the highest sample period. This problem may occur with (non-parallelized) particle filters.

The best accuracy will be achieved with an estimation period equal to the lowest publishing period. In this case in every estimation at least one measurement will be available to update the filter, and all measurements will be used to correct the filter.

The smaller the estimation period the higher is the resolution of the estimate. W.r.t. to the test application the path of the robot can be calculated in more detail.

When changing the estimation period one must not forget to adapt the timing constants in the model (needed e.g. when integrating a state variable).

Process Noise Covariance

Choosing an appropriate process noise covariance is a bit trickier than choosing e.g. the estimation period or the measurement noise covariance. In general we are not able to observe the process to estimate completely. As a rule of thumb one should pick a higher process noise covariance when the system model is held very simple [32].

For a first step the continuous process noise covariance $Q(t)$ may be considered (instead of the discrete PNC) because it seems more natural to derive (already discussed in Section 5.1.2). The process noise covariance in its continuous time model reflects the variance of the state variables. There are several possibilities to convert $Q(t)$ into the discrete process noise covariance Q_k (or simply Q if constant). Some solutions for typical models and approximations can be found in [10]. A straight forward method is the first-order approximation [10] which was also used for the test application described in Section 5.1.

$$Q_k = T \cdot Q(t)$$

For some applications, e.g. estimation of a harmonic resonator, this approximation is too vague or even fails in specific situations. Then van Loan's method provides a remedy [10].

The effects of decreasing the original process noise covariance has already been discussed in Section 5.2.5. The measurements are not trusted any more and the process is assumed to be sluggish, although the system may be highly dynamic. So when setting the process noise covariance one should think of the possible changes of the system variables w.r.t. the state transition model and control inputs.

For the first test run the continuous process noise covariance should be used. To form the continuous process noise covariance following approach may be useful:

- Basic properties, like acceleration and angular velocity in the test application, should have a variance unequal zero. The change of such properties cannot be foreseen. This unpredictability is represented by a variance unequal zero.
- The variance of other state variables depending on the basic properties should be set to 0. These variables are in contrast to the basic properties predictable, i.e. the state transition formulas give a clue what these properties will look like. When setting the variance of the dependent state variables to 0, means that the state transition formulas are assumed to be correct.
- When the estimation works acceptably, the process noise covariance may be changed to a better approximation of the discrete process noise covariance. Note covariances are symmetric.

Considering the process noise covariance of the test application, the desired angular velocity ω may change by e.g. $0.1 rad/s$. Because this value is lower than the standard deviation (square root of the variance) of the angular velocity, this value seems probable. When also the measurement reflects this change the new angular velocity is trusted. However this new angular velocity causes the heading θ to change by $0.01rad$ with the next estimation after $100ms$. Hence the heading changed *exactly* as expected ("without noise"), i.e. θ increased by the angular velocity multiplied with the estimation period.

Measurement Noise Covariance

The measurement noise covariance should be filled with the actual variance of the sensors evaluated by measuring the outputs with constant inputs [10]. As a starting point the information about typical deviations of a sensor may be taken from its datasheet. Unlike the process noise covariance, the measurement noise covariance does not depend on the estimation period T.

The sensor fusion algorithms in the framework assume *zero-mean* Gaussian noise. When an output bias occurs, i.e. the mean of the sensor's noise is non-zero, the sensor should be calibrated or the bias should be estimated by an additional state variable or a separate filter.

Especially in navigation and tracking applications where state variables are integrated in each estimation cycle, the influence of the bias of a sensor value must not be neglected. Figure 5.15 shows the values of the acceleration sensor mounted at the top of the robot. Although the robot performs an offset calibration at start up, the bias decreases steadily (the average bias is depicted in the left graph).

The robot drives and stops frequently but the velocity never returns to 0 again. The bias causes the speed of the robot to increase with every estimation cycle. In the right graph of Figure 5.15 the robot seems to gain speed into backward direction. The incorporation of the control input `cmd_vel` into the model overcomes this problem.

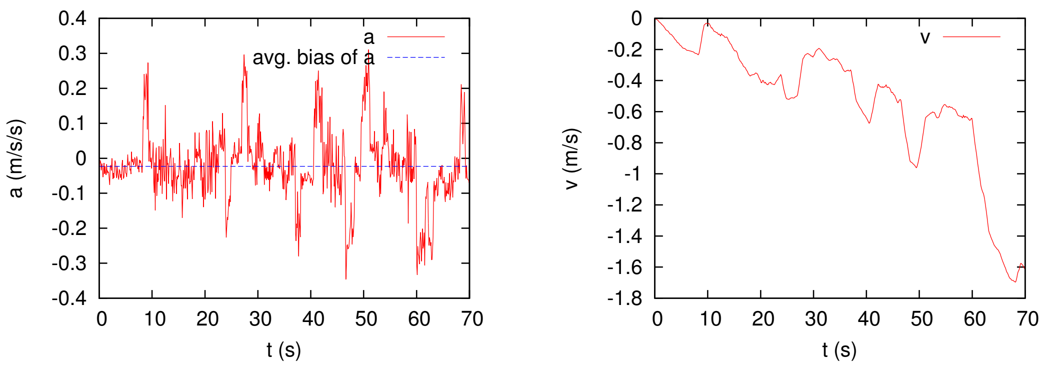

Figure 5.15: Acceleration a with average bias and resulting velocity v.

Hence state variables should not simply depend on biased sensors. A control input or another sensor should be fused to estimate the state variable. It would be even better to add a state variable to estimate the bias too or estimate the bias of the sensor by a separate filter modeling the sensor. Examples can be found in [10] (accelerometer model, gyroscope calibration model).

Initial State

Setting the initial state may be required depending on the application. In case of dead-reckoning it makes sense to initialize the state with the actual starting position and all other variables with 0 (the robot is stopped when the experiment starts).

The state variables which are corrected by measurements don't have to be initialized. The variables quickly converge to the actual values. But there may be variables which are not corrected, which are accumulated from other state variables, e.g. x and y in the proposed application. The error in the initial state of such variables will produce an offset on all estimates over time.

As an example imagine the model

$$\begin{aligned} v &= v + a \cdot T \\ a &= a \end{aligned}$$

where only the acceleration a is measured by a sensor. The velocity v is accumulated from the acceleration. When the initial velocity is wrong, the estimated velocity will be biased all the time by the initial error.

Adding a tachometer overcomes this problem. The initial error will vanish after some iterations because v is corrected by the measurements of this sensor.

Initial Error Covariance

The same as for the initial state is true for the initial error covariance. The initial error covariance is the covariance of the initial state.

As with the initial state the initial error covariance converges to its final value (if there is one). The variance of a state variable, which is not corrected by measurements, may only increase because in every estimation cycle the error covariance of the state is increased by the process noise covariance.

In the dead-reckoning application one is fully aware of the initial state, so the initial covariance is 0, i.e. the initial state is exact.

So when the initial values of state and error covariance are unknown, system variables, which are not corrected by measurements, should not be included into the model.

Number of Particles

The number of samples used to estimate the state in a particle filter should be selected as high as possible to get a better accuracy. The number of particles N is limited by the computational power. The execution time depends on N. So one should check the execution time of an estimation and adapt the number of particles such that the calculation does not last longer than an estimation period.

5.3.2 Guideline for Configuring an Estimator

The following list should simplify the task of setting the parameters.

1. *Specify the model.* List the differential equations and convert it to the state-space model introduced in Section 2.2.

2. *Choose an estimation method.* As long the state is distributed alike an unimodal distribution its a good choice to use the unscented Kalman filter. The configuration of the model is straight forward (no derivatives and no matrix form is needed, i.e. the state transition and measurement equations can be used right away). Put only the variables you really need and which are able to be corrected with measurements into the state vector.

3. *Check the sample rates of your sensors.* For a first step select the estimation period to match the highest sample period, e.g. in a system of two sensors with 10ms and 100ms sample period respectively choose the estimation period to be 100ms.

4. *Investigate the state for possible changes and uncertainty about the variables.* Collect the value a state variable may change. Do this for each state variable.

   Consider the state transition formula of the state variable and decide if its a good model for the property it is representing. E.g. compare the state transition formulas

   $$v = v + a \cdot T \qquad\qquad a = a$$

   The right formula assumes the acceleration is constant. In a dynamic system (e.g. a moving robot) this assumption is very unlikely. So the state variable is marked to be uncertain by setting a process variance unequal zero. The velocity depends on the acceleration as stated in the left formula. So this part of the model is very certain, hence the variance may be set to zero.

   Typically each variance of a basic state variable (basic means here: independent from other state variables) is non-zero.

   Note the particle filter needs variances greater than zero. So set each zero variance (or diagonal) in the process noise covariance to a relatively small value w.r.t. the non-zero variances.

5. *Look up the datasheets of your sensors to get typical deviations* to get a first approximation of the measurement noise covariance. It would be even better to collect the output samples of the sensor with constant input and calculate the variance out of these samples (note that the variance might vary with the input, e.g. a distance sensor may be more inaccurate near the borders of its range). Note the remarks on biased sensors in this section ("Measurement Noise Covariance").

6. *Set the initial state and initial error covariance if necessary.* When all variables in the state vector are somehow corrected by measurements, you don't have to set the initial values at all (the values will converge whatever initial settings are specified). Be aware of accumulated state variables, if not corrected by measurements these values may be faulty.

7. *Apply the filter* and check for reasonableness.

8. *Tune the filter.*

   - Decrease the estimation period. If possible set the estimation rate to the highest sample rate. Don't forget to check the execution time.
   - Test other filters (for linear models use the Kalman filter).
   - Check other approximations of the process noise covariance or increase the measurement noise covariance.

CHAPTER 6

Conclusion

In the last chapters a generic sensor fusion framework for low-level applications was developed and several fusion algorithms implemented. The sensor fusion methods best fit for applications like navigation, tracking and process modeling.

The fusion nodes may be used in multi-sensor multirate dynamic systems with asynchronous and/or delayed measurements. Because these algorithms are held generic the methods are suitable for most low-level applications.

6.1 Summary

The results in Section 5.2 show, that state estimation algorithms significantly increase the accuracy in comparison to averaging (c.f. model *nav1* and *nav1ctrl*). The framework enables an easy and fast configuration of various sensor fusion methods. The configuration is easily maintained and extensible. E.g. to add a new sensor only the measurement model and the measurement noise covariance have to be extended in the configuration header.

The Kalman filters suit best for unimodal probability distributions. It is also observable that the unscented Kalman filter provides general better results for nonlinear models, with the drawback of (slightly) higher execution time.

The particle filter turned out to be a relatively slow filter (in comparison to the Kalman filters) and best suitable for non-Gaussian probability distributions. For most low-level applications the implemented particle filter, Sampling Importance Resampling, is far too slow. The execution time of the PF on a single CPU often exceeds most publishing periods, or sample periods respectively, of sensors. Hence not all measurements can be used which results in a poorer accuracy. To work well, a parallel implementation should be preferred. However for many low-level applications the Kalman filters are sufficient, e.g. physical properties can often be described by an unimodal distribution.

The advantages of this generic framework are extensibility, maintainability, usage and applicability, which manifest in the following facts:

- All algorithms in the framework use the same interfaces and are therefore fully replaceable with other algorithms. Furthermore the fusion nodes can be combined.
- The estimators in this framework can be used in multi-sensor systems, i.e. replicated sensors observe the same entity. The user decides which measurements are fused in which node, so several fusion approaches, centralized or decentralized, may be applied.
- The framework is able to handle multirate and asynchronous measurements without further configuration.
- The filters can be used in various applications without adaption, only the model in the configuration has to be changed.
- The code is well documented and a tutorial helps to implement a new fusion algorithm. Interfaces and abstract classes are provided to speed up the extension of the library.
- Several fusion methods can be tried out and configurations may be tested easily, so the time of development of a fusion block reduces to a minimum.

6.2 Outlook

Possible extensions, which could ease up configuration or make the framework even more applicable, are listed below.

- The particle filter is implemented to work with Gaussian distributions. But the strengths of this filter are multimodal distributions. So more implementations of probability distributions, sample algorithms and other types of particle filters, e.g. Unscented Particle Filter [29], may be helpful.
- For state estimation the filters that are used most often and which are most accurate have been implemented. However there are estimation algorithms which may work better in some situations, e.g. in parameter estimation, and are worth to be investigated too.
- In ROS many topics provide fields to enter the covariance of multidimensional entities. The configuration may be enhanced to not only specify a single value with variance but a vector with appropriate covariance. In the current implementation the value and covariance of a 3-axis accelerometer must be split up into three values and its variances. Hence the dependency between these three properties gets lost (the non-diagonals of the covariance are assumed to equal zero).
- The filter configuration could be made automatic by use of a knowledge base. Due to the common interface the combination of fusion nodes may be automated too.

There is no method or algorithm which is the superior solution for sensor fusion due to the broad field of possible applications sensor fusion can be applied to. The developed framework is a collection of low-level sensor fusion algorithms and is open for the implementation of further fusion algorithms.

Bibliography

[1] Adept MobileRobots. Pioneer 3-AT. http://www.mobilerobots.com/ResearchRobots/P3AT.aspx, 2013. [Online; visited on Jan. 23th 2014].

[2] Adept Technology. ARIA Developer's Reference Manual. http://robots.mobilerobots.com/Aria/docs/main.html, 2014. [Online; visited on Jan. 23th 2014].

[3] M.S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

[4] Boost. Boost C++ Libraries. http://www.boost.org/, 2014. [Online; visited on Jan. 13th 2014].

[5] Boost. The Boost Library Preprocessor Subset for C/C++. http://www.boost.org/doc/libs/1_46_1/libs/preprocessor/doc/index.html, 2014. [Online; visited on Jan. 13th 2014].

[6] H.F. Durrant-Whyte. Sensor Models and Multisensor Integration. *International Journal of Robotics Research*, 7(6):97–113, 1988.

[7] Eigen. Eigen - a c++ template library for linear algebra. http://eigen.tuxfamily.org, 2014. [Online; visited on Jan. 15th 2014].

[8] W. Elmenreich. *Sensor Fusion in Time-Triggered Systems*. PhD thesis, Vienna University of Technology, 2002.

[9] V. Fathabadi, M. Shahbazian, K. Salahshour, and L. Jargani. Comparison of Adaptive Kalman Filter Methods in State Estimation of a Nonlinear System Using Asynchronous Measurements. In *Proceedings of the World Congress on Engineering and Computer Science 2009 Vol II*, October 2009.

[10] M. Grewal and A. Andrews. *Kalman Filtering:Theory and Practice Using MATLAB*. Wiley and Sons, Hoboken, New Jersey, 3rd edition, 2008.

[11] M. Gu, J. Meng, A. Cook, and P. Liu. Sensor Fusion in Mobile Robot: Some Perspectives. In *Proceedings of the 4th World Congress on Intelligent Control and Automation*, pages 1194–1199, 2002.

[12] D.L. Hall and J. Llinas. An Introduction to Multisensor Data Fusion. *Proceedings of the IEEE*, 85(1):6–23, 1997.

[13] S.J. Julier and J.K. Uhlmann. A new extension of the Kalman Filter to Nonlinear Systems. In *Proceedings of AeroSense: The 11th International Symposium on Aerospace=Defence Sensing*, 1997.

[14] L.A. Klein. *Sensor and Data Fusion - A Tool for Information Assessment and Decision Making*. SPIE Press, Bellingham, Wash., 2007.

[15] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[16] H. Kopetz and N. Suri. Compositional Design of RT Systems: A Conceptual Basis for Specification of Linking Interfaces. In *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003*, pages 51–60, 2003.

[17] E.A. Lee and S.A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. `http://LeeSeshia.org`, 2011.

[18] E.A. Lee and P. Varaiya. *Structure and Interpretation of Signals and Systems*. `http://leevaraiya.org/`, 2nd edition, 2011.

[19] A. Mandow, J.L. Martinez, J. Morales, J.-L. Blanco, A. Garcia-Cerezo, and J. Gonzalez. Experimental kinematics for wheeled skid-steer mobile robots. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 1222–1227, Oct 2007.

[20] P.S. Maybeck. *Stochastic models, estimation and control*. Academic Press, 1979.

[21] H.B. Mitchell. *Multi-Sensor Data Fusion - An Introduction*. Springer, Berlin, Heidelberg, New York, 2007.

[22] A. B. Poore, B. J. Slocumb, B. J. Suchomel, F. H. Obermeyer, S. M. Herman, and S. M. Gadaleta. Batch Maximum Likelihood (ML) and Maximum A Posteriori (MAP) Estimation With Process Noise for Tracking Applications. In *Proceedings of Signal and Data Processing of Small Targets*, pages 188–199, 2003.

[23] C.V. Rao, J.B. Rawlings, and D.Q. Mayne. Constrained State Estimation for Nonlinear Discrete-Time Systems: Stability and Moving Horizon Approximations. *IEEE Transactions on Automatic Control*, 48(2):246–258, 2003.

[24] Robot Operating System. ROS Documentation - Introduction. `http://wiki.ros.org/ROS/Introduction`, 2013. [Online; visited on Jan. 14th 2014].

[25] Robot Operating System. ROS Homepage. `http://www.ros.org/`, 2013. [Online; visited on Dec. 17th 2013].

[26] S. Russel and P. Norvig. *Artificial Intelligence - A Modern Approach*. Pearson Education, Upper Saddle River, New Jersey, 3rd edition, 2010.

[27] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, Cambridge, 2006.

[28] UniZG Faculty of Electrical Engineering and Computing - AMOR Group. ROS package rosaria. `https://github.com/amor-ros-pkg/rosaria`, 2014. [Online; visited on Jan. 23th 2014].

[29] R. Van der Merwe, A. Doucet, N. de Freitas, and E. Wan. The unscented particle filter. Technical report, Engineering Department, Cambridge University, 2000.

[30] L. Wald. Definitions and Terms of Reference in Data Fusion. *International Archives of Photogrammetry and Remote Sensing*, 32, Part 7-4-3, W6:651–654, 1999.

[31] E.A. Wan and R. Van der Merwe. The Unscented Kalman Filter for Nonlinear Estimation. In *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pages 153–158, 2000.

[32] G. Welch and G. Bishop. An Introduction to the Kalman Filter. University of North Carolina at Chapel Hill, Department of Computer Science, July 2006. Available online at `http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html`; visited on Mar. 21th 2013.

[33] L. Yan, B. Xiao, Y. Xia, and M. Fu. The Modeling and Estimation of Asynchronous Multirate Multisensor Dynamic Systems. In *Proceedings of the 32nd Chinese Control Conference*, July 2013.