TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SIAVASH TAVAKOLI

SOFTWARE FRAMEWORK FOR STATE ESTIMATION

Master of Science thesis

Examiners:
Prof. Robert Piché,
MSc Mathias von Essen,
Prof. Pasi Kallio
Examiner and topic approved by the
Faculty Council of the Faculty of
Automation Science and Engineering
on February 2nd, 2015

NOMENCLATURE

BP Belief Propagation

BUGS Bayesian inference Using Gibbs Sampling

DSL Domain Specific Language

EKF Extended Kalman Filter

GaBP Gaussian Belief Propagation

JAGS Just Another Gibbs Sampler

JSON JavaScript Object Notation

KF Kalman Filter

MBE Model Based Engineering

MCMC Markov Chain Monte Carlo

MDA Model Driven Architecture

MDE Model Driven Engineering

PaBP Particle Belief Propagation

PF Particle Filter

PGM Probabilistic Graphical Model

RLS Robotic Localization Service

RoIS Robotic Interaction Service

RTC Robotic Technology Component

UML Unified Modelling Language

ABSTRACT

SIAVASH TAVAKOLI: Software framework for state estimation
Tampere University of Technology
Master of Science thesis, 70 pages, 15 Appendix pages
January 2017
Master's Degree Programme in Machine Automation
Major: Mechatronics and Micromachines
Examiners: Prof. Robert Piché, MSc Mathias von Essen, Prof. Pasi Kallio
Keywords: Bayesian filtering, Sensor fusion, Model driven engineering, Probabilistic graphical models

Over the past decade, robotics has seen tremendous increase in complexity and variety of applications. The key area in the robots seeing rapid evolution is the software. However, usually the software developed for robots has been limited to a specific application and/or a specific hardware. Unfortunately most of the software developed for robotic applications are not easily re-usable in another project. Very little effort has been done to tackle this issue and the software is developed on an ad-hoc basis.

In this work, a framework for developing sensor fusion software is proposed that is based on practices of model-driven engineering. A small domain-specific language is developed that effectively hides the lower level implementation details and makes the software development more structured and easier to re-use.

It is also discussed how graphical models can be used as computational framework for performing the statistical inference in filtering problems. It is shown how a simple estimation problem can be solved using graphical models.

PREFACE

This thesis work was done partly in robotics lab of University of Leuven. I would like to express my great gratitude to professor Herman Bruyninckx who was a great source of advice and support throughout my thesis. I am very grateful to Nico Hüebel for all of his help and the great discussions we had during my visit to Leuven.

This work wouldn't have been possible without the help of professor Robert Pichè and faculty of Automation Sciences and Engineering at TUT who supported my trip to Leuven. I am thankful to professor Pasi Kallio and Mathias Von Essen for evaluating my thesis work. I deeply appreciate professor Piché's support and guidance all through my thesis work was performed.

March 2016,
Siavash Tavakoli

TABLE OF CONTENTS

1. Introduction 1
  1.1 Sensor Fusion 3
  1.2 Probabilistic Graphical Models 4
  1.3 Sensor fusion software 5
  1.4 Thesis outline 6
2. Theoretical Background 7
  2.1 Robot and The Surrounding Environment 7
  2.2 Recursive State Estimation 8
    2.2.1 Bayesian Filtering 8
  2.3 Bayes Filters 10
    2.3.1 The Markov Assumption 11
    2.3.2 The Kalman Filter 12
    2.3.3 The Extended Kalman Filter 14
    2.3.4 Nonparametric Filters 16
  2.4 Graphical Models 18
    2.4.1 Factorgraph 19
  2.5 Inference On Graphical models 20
    2.5.1 Marginal Inference 21
    2.5.2 Variable elimination and message passing 21
    2.5.3 The sum-product algorithm 23
  2.6 Model Driven Engineering and Domain Specific Languages 24
    2.6.1 Model Driven Engineering 24
    2.6.2 Domain Specific Languages 25
3. Bayesian Estimation and Probabilistic Modeling Software 27
4. Research methodology and materials 33

4.1 Factor graphs as Bayesian estimator 33
4.1.1 Factor graphs for Kalman filtering 36
4.1.2 Factor graphs for Particle filtering 38
4.2 Tools and Frameworks used 38
4.2.1 Probabilistic programming 39
4.2.2 Software Models 40
4.3 Modeling 43
4.4 Code generation 45
5. Results and discussion 47
5.1 An example problem 47
5.1.1 Filter Model 48
5.1.2 Model Checking and Code Generation 49
5.1.3 Results 51
5.2 Future Work 51
6. Conclusion 53
Bibliography 54
A. JSON Schema models 59
B. Matlab code 67

LIST OF FIGURES

2.1 Robot Environment Interaction 7

2.2 Factor graphs representing different factorisations 20

2.3 An example of Markov chain 21

3.1 Inference algorithms supported by OpenGM 28

3.2 comparison of features supported by libDAI 29

4.1 Batch vs. recursive estimation 35

4.2 Realisation of a linear system 37

5.1 Filtering performance comparison for an example problem 52

1. INTRODUCTION

A robot can be described as the device for manipulating the physical world based on perceptions from the outside world and processing the data by a computer. The perception part is a key part which has great impact on the following actions performed by the robot. The more precise data is available to the processor of a robot, the more accurate actions will be. There has been many robotic systems in use in vast fields of applications such as mobile platforms for planetary exploration, robotics arms in assembly lines, autonomous vehicles, actuated arms that assist surgeons.

The major reason of developing robots have been to have a machine "intelligent" enough to do some tedious tasks of the humans such as cleaning, labour work, driving, etc. One of the big challenges in this sense is that our needs and environment has been evolving rapidly during the past decades. Today's application domains differ from yesterday's and so will tomorrows' application domain from today's.

With robotic systems gaining popularity for in-house and outdoor use over the past years, the most striking characteristic of these new robots is that the environments that they operate in are unpredictable and unstructured. In comparison to classic use of robotic systems which was in production assembly lines, predictability of environment has changed considerably. Environment of of a private house is far more unpredictable and uncontrollable.

As a result, robots are moving towards a direction where sensory input data has become increasingly important and the software used on the robot has to be reliable enough to cope with all of aforementioned problems. In this regard, robotics is increasingly becoming more of a software science field with the goal of developing a software robust enough to withstand unstructured, unpredictable, and dynamic environments.

With all being said so far, one of key elements in robotics and the main focus of this

work is *Uncertainty*. There are many factors that give rise to uncertainty. Some of sources of uncertainty are:

1. **Environments:** Physical environments are in nature unpredictable. Though the level of unpredictability can greatly differ between environments. A car driving in a highway and a robot doing indoor cleaning or a robotic arm performing assembly in a production all face some level of uncertainty but with a great difference in level.

2. **Sensors:** Sensors are bounded by what they can perceive and how well they can perceive physical entities. These limitations arise from two facts. First, sensors are intended to measure physical entities and are subject to physical limitations in range and resolution. For example, a camera cannot see through walls. Second, build quality and working principles of sensors greatly affect the resolution of the sensor. For instance, due to the optical properties of photographic lenses, only objects within a limited range of distances from the camera will be reproduced clearly. Also, sensors are subject to noise. Sensor hardware can never be ideal which results in data obtained from the sensor always associated with some level of uncertainty.

3. **Robots:** Robots themselves are never accurate. Robot actions are performed by using mechanical actuators which always have some level of uncertainty because of internal structure (gear backlash, wear and tear, noise in control signal, etc.).

4. **Models:** Models are formalisation of real world phenomena in mathematical notation. Most often, physical relationships being described in the models are very complex and are always simplified by making assumptions. Therefore, the underlying physical interactions of the robot and the environment is only partially modelled.

5. **Computation:** Because of uncertainty in the environment where the robot is performing its operations, most often robots are developed as a real-time system where computations are done online and during the operation. Therefore, the amount of computation that a robot can carry out is limited by hardware and many algorithms have been developed to approximate the results in exchange of less processor load and faster computations.

All the factors above result in uncertainty. Up until past decade, uncertainty in robotics had been mostly ignored (and rightfully so due to use cases being in more predictable environments). However, as robots gain diverse uses in increasingly unpredictable environments the ability to robustly manage uncertainty becomes vital.

1.1 Sensor Fusion

To make up for the aforementioned uncertainties, one very common way is to augment information obtained from various sources that are available on the robot. Robots are often equipped with multiple types of sensors. Information obtained from various sensors are combined ("*fused*") in order to increase belief of perception data. Sensor fusion is widely used in many areas in robotic application such as localization, object recognition, and environmental mapping.

Correct and robust way of integrating the observation data obtained from different sources with different characteristics is a challenge that calls for precise mathematical methodology. There are already well established mathematical tools to perform sensor fusion. The most widely used methods for sensor data fusion have their root in probability theory. Probabilistic data fusion methods are based on Bayesian framework of statistics. Other non-probabilistic methods have also been proposed such as theory of evidence and interval methods but they are not as commonly used as probabilistic methods.

In Bayesian statistics, an initial belief of a random variable called *prior* is updated according to observation data to give *posterior* belief. This inference is performed according to a mathematical relationship connecting two sets of probabilistic variables called the *Bayes rule*. In its simplest form, the Bayes rule can be described as a mathematical formulation of relationship between two probabilistic variables x, z such that:

$$P(x|z) \propto P(z|x)P(x) \tag{1.1}$$

where $P(x|z)$ is conditional probability distribution of x given z.

This rule is in the heart of all of the methods used in probabilistic sensor fusion used in robotics and many many other fields of application. As mentioned above, robot software is most of the time real-time software and computations are carried out while the robot is navigating through the environment. In practice, the Bayes

rule is used recursively over time when new measurement data is present. This process, called recursive Bayesian estimation (or recursive Bayesian filtering), is a general approach common in all of the methods of probabilistic estimators that allow a robot to continuously update its beliefs from the environment. At each step of operation, new observations are performed and fused with previously acquired data to obtain the most probable state of the robot. This information is then stored and used at the next step again as an initial *prior* belief.

There are various types of Bayesian estimators with distinct characteristics. Some of these methods make certain assumptions about the system and the nature of noise present in the system that result in limiting applicability of these estimators to specific systems. There are also methods that pose no restriction and can be used for any arbitrary system with any type of noise. Former type of Bayesian estimators are generally simpler, easier to implement, and less computationally demanding while the latter type require more computation power and time. In this work, one example from each of these categories has been chosen and worked on. These estimators, Kalman filter and the particle filter, are the most widely used family of Bayesian estimators. In Kalman filters it is assumed that there exists a linear system with Gaussian noise whereas particle filters use a Monte Carlo sampling method and can be used for any type of system. A more detailed overview of these filters is presented in Chapter 2.

1.2 Probabilistic Graphical Models

Graphical models are a combination of probability theory and graph theory. Although the theory has been established for quite a time already, graphical models have gain considerable popularity over the past decade with increase of computing power leading to more widespread use of probabilistic models. Factor graphs offer a compact representation of probabilistic models. In graphical models, probabilistic variables are defined as nodes in a graph and relationship between these variables is defined as links (arcs). Inference is performed on graphs to calculate most probable value of a node.

Different types of graphical models have been proposed and analyzed. A general categorization divides graphical models into *directed* graphs (also called Bayesian networks) and *undirected* graph (also called Markov networks). Each of these types has its own characteristics and requires different methods for carrying out inference.

However, a third (and a more general) type of graphical models is often used, called *factor graphs*. Factor graphs are bipartite graphs where two types of nodes are present in the model, probabilistic variables and the function ("factor") specifying the relationship between these variables. Variable nodes can only be connected to factor nodes and vice versa. Due to the generality of factor graphs, inference can be done in an efficient and easy way. A more detailed overview graphical models and some common inference algorithms is presented in Chapter 2.

1.3 Sensor fusion software

There have been vast efforts in the area of software framework for sensor fusion using Bayesian framework of inference. Although areas of application for these efforts might seem different, the core functionality and theoretical grounds are the same. However, up until now most of efforts made in the aspect of Bayesian estimation software have lead to ad-hoc solutions which are either hard to reuse in a different scenario or hard to integrate with an external software. In this thesis project the groundwork for a clean and reusable software framework for Bayesian estimation is laid. More specifically use of *Domain-Specific Languages* (DSLs) is emphasized.

DSLs conform to practice of knowledge representation and model-driven engineering. Model-driven engineering and domain modeling is an approach in software development where general aspects of a domain are encapsulated in a well defined language. Model-driven engineering results in a structured and formalized knowledge from a specific domain. MDE is particularly important in robotic applications where knowledge from different domains is integrated and utilized.

A domain-specific languages is a special-purpose computer language oriented towards a specific application domain. DSLs are developed with the intention of formalizing knowledge in a particular domain in a way that best captures domain's semantics. DSLs are widely used in different computer software domains and are either as an stand-alone language or developed inside a general purpose language. Using DSLs greatly enhances software reuse as they offer small building blocks that can be utilized to build the application logic on top of them.

In robotics DSLs have been developed for various purposes such as overall software architecture design and deployment or defining physical specifications of a robot. Section 2.6.2 gives an overview of some DSLs defined for robotic applications.

In this thesis work, a domain-specific language intended to be used in sensor fusion tasks in robotic applications is proposed. The work is done collaboration with Robotics research group of university of Leuven.

1.4 Thesis outline

This thesis report is structured as follows. In Chapter 2 an overview of the theoretical concepts used is given and the mathematical framework for this work is explained. Also, an overview of related works done regarding domain specific languages is presented. In Chapter 3 some of the currently available software developed for Bayesian estimation are introduced. In Chapter 4 research methodology and the tools used are explained. It is also explained why these tools are chosen to perform the work. In Chapter 5 it is shown how a sample Bayesian estimation problem can be performed using the tools and methodology presented. Finally, in Chapter 6 the conclusion is given.

2. THEORETICAL BACKGROUND

This thesis work makes use of theoretical knowledge from many different fields. Therefore, it is necessary to present a brief overview of theoretical concepts used. In this chapter, some basic ideas and concepts that are utilised in this work are explained.

2.1 Robot and The Surrounding Environment

Robots interact with the surrounding environment in two distinctive ways. They observe and acquire information about the environment through the sensors mounted. They also perform actions and manipulate objects of the environment by using the on-board actuators. Figure 2.1 illustrates the interaction of a robot with its environment.

As already mentioned in the section for introduction, the information acquired by robot sensors is noisy and not all environmental entities can be captured by the sensors. In order to compensate for the sensor data shortcomings, the robot keeps an internal "belief" of its current state and the state of its environment. Throughout

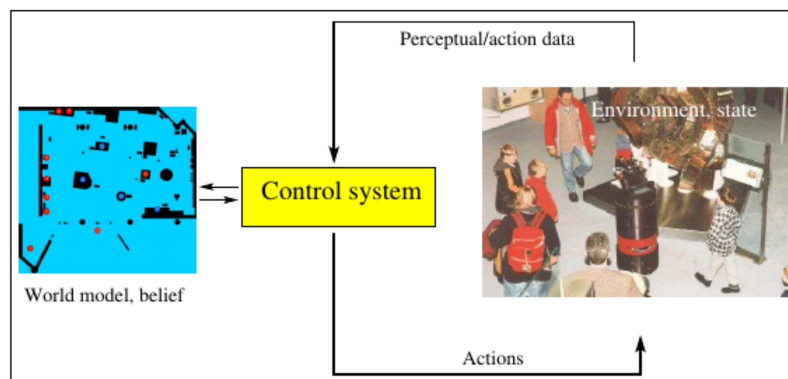


Figure 2.1 *Robot Environment Interaction [38]*

the robot's operation this internal belief is updated whenever new information is obtained by robot sensors, resulting in improved accuracy of beliefs. This is often referred to as "fusion of sensory data over time".

2.2 Recursive State Estimation

As any other dynamical system, a robot's current working condition can be represented by a set of variables describing the current "state" of the robot. The smallest subset of these variables that is sufficient to fully describe the robot's future and past state is called *state variables*. Among these state variables, there are the ones that are possible to measure directly by sensors and there might be others that are not directly observable and should be inferred from the sensor data. For example to be able to navigate a mobile robot through a terrain it is obviously desirable to know the exact position of the robot and the obstacles around it. However, absolute location of the robot is often not measured (e.g. when relative sensors such as encoders are used or position of the robot is obtained by integrating acceleration obtained from accelerometers). Therefore, the robot has to rely on its sensors to gather the data and extract useful information from them.

As discussed in introduction chapter, sensors are limited by physical and technological factors. Sensors often obtain partial information about quantities and the data obtained by sensors is contaminated by noise. Mobile robot localization is a typical example where state estimation is used to recover state variables from noisy sensor data. Probabilistic state estimation is an approach for state estimation where probabilistic methods are used to compute belief distributions for state variables over possible values in the surrounding world.

2.2.1 Bayesian Filtering

In optimal estimation and Bayesian filtering, all state variables of the system as well as sensor measurements and control inputs are all considered to be random variables. Then, governing equations that describe the model of the system dynamics as well as relationships between observations of state variable and the state variables are specified in terms of probabilistic equations and laws. The process of deriving random variables for each state from the observed sensor data which themselves are modelled as a random variable is called probabilistic inference.

If X is a random variable, probability of X assuming a specific value x within its scope is written as

$$p(X = x) \tag{2.1}$$

This notation is usually simplified and instead of writing the random variable name explicitly, the notation is abbreviated as $p(x)$.

Random variables often are correlated to other random variables and some information is shared between them. For example, let's assume that a random variable Y is known to have value y and probability of X having value of x conditioned on value of Y is desired. This probability is given

$$p(x|y) = p(X = x|Y = y) = \frac{p(x, y)}{p(y)} \tag{2.2}$$

and is called *conditional* probability.

The *Bayes rule* is one of principal rules in probabilistic theory that relates odds of an event to odds of their "inverse" event. For the probabilistic variables X and Y in equation 2.2, the Bayes rule is stated as below with one condition that $p(y) > 0$:

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \tag{2.3}$$

If x is to be inferred from y, then probability of x, $p(x)$, is called *prior probability.* y is referred to as the *data* (e.g. an observation obtained from sensor measurement). $p(x)$ encapsulates the available knowledge regarding X before taking the data y into account. Then, the conditional probability of x given y ($p(x|y)$) is called the *posterior probability distribution* over X. In robotics sensor fusion, the conditional probability $p(y|x)$ is the probability of the sensor data y assuming that state value x. This distribution is obtained using well known models such as kinematics, dynamics, sensor physics, etc.

Bayes rule plays an important role in robotics sensor fusion. Using Bayes rule, it is convenient to compute the posterior distribution $p(x|y)$ by using the conditional probability $p(y|x)$ and the prior distribution $p(x)$. In probabilistic sensor fusion, the objective is to infer a state variable x from sensor data y. Using the Bayes rule, it is possible to do so by taking the inverse probability of the sensor data y assuming

that state value x.

2.3 Bayes Filters

In its most general form, the method to recursively calculate beliefs using the Bayes rule can be expressed by the *Bayes filter* algorithm. In this algorithm it is shown how the belief distribution bel can be calculated from observation and control data.

Pseudo-algorithm of the general Bayes filter is shown in Algorithm 1. In order to prevent the problem space to increase which results in drastic increase of demand for more computation power, the Bayes algorithm is utilised recursively. This means that at each time step, the belief is updated according to the belief calculated at previous time step as well as the inputs given to the system and observed data. To put it more formally, $bel(x_t)$ at time t is calculated based on the belief $bel(x_{t-1})$ at time $t-1$ along with the control input u_t and measurement z_t.

Algorithm 1 The general algorithm for Bayes filtering [38]

1: **procedure** BAYES_FILTER($bel(x_{t-1}), u_t, z_t$)
2: $\quad \overline{bel}(x_t) = \int p(x_t|u_t, x_{t-1}) bel(x_{t-1}) dx$
3: $\quad bel(x_t) = \eta p(z_t|x_t) \overline{bel}(x_t)$
4: $\quad$ **return** $bel(x_t)$
5: **end procedure**

As can be seen in algorithm 1, in Bayes filtering there is two main operations being performed at each time step. In line 3, an "intermediatory" belief is calculated with control input and the belief calculated at the previous step. This belief which is a transformation of the previous belief solely based on what has been inputted to the system is called the *prediction step*. In the prediction step, it is calculated that with the control input u_t given to the robot, what is the probability of transition from x_{t-1} to x_t. This probability is then integrated (summed) with the prior distribution assigned to x_{t-1}. Outcome of this step is a "predicted" belief assigned to state x_t.

At the second step of Bayes filters, the "predicted" belief is *updated* with the observation data. In this step (line 4), the intermediatory belief $\overline{bel}(x_t)$ is multiplied by the probability that the measurement z_t has been observed. This multiplication does not necessarily integrate to 1 so in order to get proper probability distribution it should be normalized. Therefore, it is multiplied by a normalization constant η. This step of the Bayes filter is called the measurement update step.

This general form of Bayes filter involves integrating and multiplication of probability distributions. These calculations can be computed in closed form for either very simple cases or if we only restrict ourselves to discrete state space where the integral in line 3 is simplified to summation.

2.3.1 The Markov Assumption

The Markov assumption (also called the complete state assumption) plays a vital role in probabilistic robotics. In a nutshell, the Markov property is a property of stochastic systems where the future state depends solely on current state of the system and not the past. In other words, in order to know the future state of the system x_{t+1} it is sufficient to know the current state x_t and the past states (x_{t-1}, x_{t-2},...) can be ignored.

In the context of mobile robot localization, x_t is the robot's current pose in relation to a fixed map. The pose is estimated by utilising Bayesian filters. However, the Markov assumption is true for ideal systems where there is no error present in the system. Many factors affect conformity of a system to Markov assumption:

- Dynamics in the environment which are not modelled and included in system model (e.g. people moving around the robot and the effect it has on sensor measurements),
- inaccuracies in the measurement and system dynamic models ($p(z_t|x_t)$ and $p(x_t|u_t, x_{t-1})$),
- errors when representing the belief functions using approximate representations (e.g., grids or Gaussians), and

Although it is possible to take all of the above factors into account while defining state representations, doing so is not favorable due to significant increase in computational complexity of the filtering algorithm. Fortunately, Bayesian estimators have been found to be tolerant towards such deviations from Markov assumption. The best practice is to define x_t so that the state variables that are not modeled have close-to-random effects.

2.3.2 The Kalman Filter

Linear Gaussian Systems

The most famous and well studied technique for Bayesian filtering is the *Kalman filter* (KF). The Kalman filter [19] was first introduced in 1960 by Rudolph Emil Kalman, as a method for filtering and estimation in linear systems. The Kalman filter is used to compute continuous state beliefs. In Kalman filtering beliefs are represented by their moments. At each time step t, the belief is represented by its first moment, the mean μ_t, and second central moment, the covariance Σ_t. In addition to the Markov assumption discussed above, if certain conditions are met, the posteriors computed will be Gaussian. These conditions are as follows:

1. In Kalman filters, system dynamics is assumed to be *linear*. Probability of next state given current state, $p(x_t|u_t, x_{t-1})$ must also be a linear function with additive Gaussian noise as shown in the following equation.

$$x_t = A_t x_{t-1} + B_t u_t + \epsilon_t. \tag{2.4}$$

   In this equation, x_t and x_{t-1} are state vectors at times t and $t-1$ (they are assumed to be column matrices), and u_t is the control input at time t. The matrix A_t is a matrix of size $n \times n$ and size of B_t is $n \times m$. Here n is the number of state variables (i.e. dimension of the state vector x_t) and m is the dimension of the control vector u_t. When multiplying the state vector with matrix A_t and control vector with matrix B_t, the state transition function is *linear* in its arguments.

   The variable ϵ_t in 2.4 is a vector of scalar random Gaussian variables which specifies the randomness in state transitions. Dimension of ϵ_t is the same as the state vector and it has zero Mean and covariance Q_t. State transition model of the form as in equation 2.4 is called *linear Gaussian* since it is linear and it has added Gaussian noise.

2. Apart from the state transition probability, the observation probability $p(z_t|x_t)$ should be also linear with additive Gaussian noise:

$$z_t = C_t x_t + \delta_t \tag{2.5}$$

In this equation C_t is a matrix with size $k \times n$, where k is the dimension of z_t, the observation. δ_t is a vector of Gaussian variables with zero mean and covariance R_t which describes the observation noise.

3. Lastly, the belief at time 0 (initial belief), $bel(x_0)$ should also be a normal distribution with μ_0 mean and Σ_0 covariance.

If the assumptions listed above are met, then it is guaranteed that the posterior distribution $bel(x_t)$ is always Gaussian.

The Kalman Filter Algorithm

As discussed above, in Kalman filtering Gaussian beliefs are represented by their mean and covariance. Algorithm 2 illustrates algorithm of Kalman filtering. At each time step t, belief of previous time step $t-1$ is taken as input of the algorithm (with mean μ_{t-1} and covariance Σ_{t-1}). Then, by incorporating current control input u_t and measurement z_t into belief of previous time step, belief at time t represented by μ_t and Σ_t is calculated.

Algorithm 2 The Kalman filter algorithm for linear Gaussian state transitions and measurements [38]

1: **procedure** KALMAN_FILTER($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$)
2: $\quad \overline{\mu}_t = A_t \mu_{t-1} + B_t u_t$
3: $\quad \overline{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$
4: $\quad K_t = \overline{\Sigma}_t C_t^T (C_t \overline{\Sigma}_t C_t^T + Q_t)^{-1}$
5: $\quad \mu_t = \overline{\mu}_t + K_t(z_t - C_t \overline{\mu}_t)$
6: $\quad \Sigma_t = (I - K_t C_t)\overline{\Sigma}_t$
7: $\quad$ **return** μ_t, Σ_t
8: **end procedure**

Similar to what was presented in general algorithm in algorithm 1, in Kalman filtering a predicted belief $\overline{bel}(x_t)$ is computed first which is represented by its mean $\overline{\mu}$ and covariance $\overline{\Sigma}$. This belief represents the predicted belief one time step later, when only input signal is considered and before incorporating the measurement z_t.

This "predicted" belief $\overline{bel}(x_t)$ is then updated with taking the measurement data into account, resulting in the desired belief $bel(x_t)$ (lines 4–6). K_t which is calculated in Line 4 is called the *Kalman gain*.

2.3.3 The Extended Kalman Filter

In practice, assumption of linear state transition and measurement models with added Gaussian noise seldom holds. Therefore, plain Kalman filters are applicable only in trivial applications. The *Extended Kalman filter* tackles one of the assumptions made in plain KF and tries to overcome restriction of linear systems at the price of some computational overhead.

In EKF, it is assumed that the process and measurement models are nonlinear functions as described by g and h in following equations:

$$\begin{aligned} x_t &= g(u_t, x_{t-1}) + \mu_t \\ z_t &= h(x_t) + \delta_t \end{aligned} \tag{2.6}$$

This model is a generalization of the model used in Kalman filter (described in equations 2.4 and 2.5) where matrices are replaced by functions g and h. As discussed earlier when a linear function performs on Gaussian distribution, the resulting belief is also Gaussian. However, since here the functions are no longer linear the belief is also non-Gaussian. In fact, calculating exact values in update step is usually impossible for nonlinear functions (in which case there is no closed-form solution for the Bayes filter).

In EKF, the true belief is approximated by a Gaussian distribution. At each time step t the belief $bel(x_t)$ is represented by a mean μ_t and a covariance Σ_t. Therefore, EKF and KF share the way beliefs are represented but differ in the sense that in KF the belief is exact where in EKF an approximation of the belief is calculated.

The key idea in EKF is linearization of nonlinear state and measurement model equations by utilising the Taylor expansion. In this linearization a nonlinear function g is linearized (approximated by a linear function) at the mean of the Gaussian. Transforming a Gaussian through this linear function yields a Gaussian posterior. In fact, when this linear version of g is used, the mechanics of belief propagation are equivalent to Kalman filter. The same applies for measurement function h.

The EKF Algorithm

Algorithm 3 summarizes the steps in EKF algorithm. As can be seen, the algorithm is very similar to the one in KF with differences in state and measurement predictions (lines 2 and 5 of the algorithm).

Algorithm 3 The Extended Kalman filter algorithm [38]

1: **procedure** EXTENDED_KALMAN_FILTER($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$)
2: $\quad \overline{\mu}_t = g(u_t, \mu_{t-1})$
3: $\quad \overline{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$
4: $\quad K_t = \overline{\Sigma}_t H_t^T (H_t \overline{\Sigma}_t H_t^T + Q_t)^{-1}$
5: $\quad \mu_t = \overline{\mu}_t + K_t (z_t - h(\overline{\mu}_t))$
6: $\quad \Sigma_t = (I - K_t H_t) \overline{\Sigma}_t$
7: $\quad$ **return** μ_t, Σ_t
8: **end procedure**

In contrast to linear predictions in Kalman filters, in EKF the predictions are replaced by their nonlinear generalizations. The EKF estimator uses Jacobians (G_t and H_t in the algorithm 3) instead of the matrices used in linear systems.

EKF, Pros and Cons

EKF is one of the most popular tools for state estimation in robotics. Simplicity and computational efficiency are two important factors leading to popularity of EKF. Computational efficiency of EKF is due to the fact that beliefs are represented by multivariate Gaussian distributions. Moreover, in many practical problems, Gaussians are robust estimators and in many estimation problems that violate the assumptions EKFs have been applied successfully.

On the other hand, inability of representing multimodal beliefs (as in case of KF) is an important limitation of EKF. As an example, in many situations a robot might have two distinct hypotheses of its whereabouts (and a mean middle-ground hypothesis is not likely). In such cases, in EKF since the distributions are all assumed to be Gaussian, and hence uni-modal, EKF might produce erroneous posterior. A common extension of EKF to tackle this represents posteriors using *mixtures* or *sums* of Gaussians [1].

Another important limitation of EKF is inherited from Taylor series expansion of state transition and measurement models. Two factors affect how well this linearization captures the true belief. First, the degree of nonlinearity of the functions and second, the degree of certainty. The more nonlinear the state transition and measurement functions are, the more likely it is that the linear approximation has bigger error. Also, the more a robot is uncertain about its state, the wider its Gaussian belief. Therefore, it is critical to keep the uncertainty of state estimate as low as possible when applying EKF.

It is also beneficial to note that Taylor expansion is one way of linearizing the system. Other methods have been proposed which often produce better results. As an example, *unscented Kalman filter* approximates the function based on function values at some "probe" points. Another method is known as *moment matching*. In this method the linearization is done in such way that the true mean and covariance of the posterior distribution is preserved.

2.3.4 Nonparametric Filters

Nonparametric filters are a widely used alternative to the Gaussian estimators. These filters do not make any assumption (or rely on) a fixed form of the distribution for the posterior (e.g. Gaussian distribution as in Kalman filters). Instead, in nonparametric filters the posterior is approximated by a finite number of point values. As a consequence the accuracy of approximation increases as the number of parameters to represent the distribution is increased (with the ideal case where there is infinite number of parameters in which case the approximated posterior converges to the true posterior). In this work, one of these nonparametric estimators is focused on and discussed. This technique, known as *particle filter* represents the posterior by finitely many samples. The particle filter has gained immense popularity in robotic applications.

Nonparametric filters in general (and consequently particle filters) do not make any strong parametric assumption on the posterior distribution function. Therefore, they are well suited to represent multi-modal beliefs and they are often used when level of uncertainty is high and when the robot needs to deal with data association problems where separate, distinct hypotheses are produced. However, this benefit comes with a price of computational complexity as will be shown in the next section.

THE PARTICLE FILTER

The particle filter is a nonparametric implementation of the Bayes filter. In particle filters, the posterior is approximated by a (finite) number of parameters. There are different types of particle filters. The difference between these kinds of particle filters is the way the approximation parameters are generated, as well as the way they are spread out across the state space. The main idea in particle filters is to work with an approximation of the posterior $bel(x_t)$ which is constructed by a set of random samples drawn from the posterior. Although this alternative representation of the distribution (instead of the parametric form) is an approximation, it is nonparametric, and therefore can be used to represent broader types of distributions.

Each of the samples drawn from the posterior distribution are called *particles* and are defined as

$$X_t := x_t^{[1]}, x_t^{[2]}, \dots, x_t^{[M]} \tag{2.7}$$

Each of the particles $x_t^{[M]}$ (where $1 \leq m \leq M$) is a hypothesis of what is value for the state at each time step t. M is total number of particles (i.e. the size of the particle set). It is ideal to have a large number of particles as it yields better approximation to the true posterior distribution.

The logic behind PFs is the same as Monte Carlo method in which instead of using the full belief distribution, the distribution is approximated by a set of particles X_t. In ideal case, the likelihood of the state hypothesis x_t being in the particle set X_t is proportional to its posterior $bel(x_t)$:

$$x_t^{[m]} \sim p(x_t | z_{1:t}, u_{1:t}) \tag{2.8}$$

Just as other recursive Bayesian filters, the particle filter computes the belief $bel(x_t)$ based on belief of the previous time step $bel(x_{t-1})$ but with the difference that the belief is represented by a set of "particles". This means that particle filters constructs particle set X_t recursively from the set X_{t-1}. Basic algorithm of the particle filter is presented in algorithm 4. The algorithm takes the particle set X_{t-1}, current control command u_t, and current measurements z_t as inputs. Then, each of the particle $x_{t-1}^{[m]}$ is processed by the algorithm and a temporary particle set $\bar{X}$ is constructed which is similar (but not identical) to the belief $\overline{bel}(x_t)$.

Algorithm 4 The particle filter algorithm [38]

```
1: procedure PARTICLE_FILTER(𝒳_{t−1}, u_t, z_t)
2:     X̄_t = 𝒳 = ∅
3:     for m = 1 to M do
4:         sample x_t^[m] ~ p(x_t | u_t, x_{t−1}^[m])
5:         w_t^[m] = p(z_t | x_t^[m])
6:         X̄_t = X̄_t + ⟨x_t^[m], w_t^[m]⟩
7:     end for
8:     for m = 1 to M do
9:         draw i with probability ∝ w_t^[i]
10:        add x_t^[i] to 𝒳_t
11:    end for
12:    return 𝒳_t
13: end procedure
```

The most important part of the particle filter is called *resampling* or *importance resampling* (performed in lines 8 through 11 in algorithm 4). In importance resampling, the particle set of M particles drawn from temporary set $\bar{X}_t$ is transformed into another particle set of the same size but with importance weights taken into account. By this, distribution of the particles change and they are distributed according the posterior rather than the initial $\overline{bel}(x_t)$.

The resampling step in particle filters shares the same nature with the Darwinian idea of *survival of the fittest*. In this step, the focus of particles is shifted to the regions where posterior probability is high. Doing so, computational resources are allocated to the regions where they matter the most.

2.4 Graphical Models

One might expect to feed a mass of data and probability distributions into a computer and get good predictions in a timely manner. However, this is a naive assumption and is very likely to fail in complex situations. By increasing size and complexity of the problem, the time needed to perform the processing increases considerably. Therefore, it is necessary to structure the problem into a "model" so that processing a large data set would take less time.

The goal of a *probabilistic graphical model* is to encode a probability distribution over a set of random variables $\mathcal{X} = X_1, \ldots, X_N$. PGMs help to organize and de-

pict dependence/independence assumptions made in a distribution. Advantage of using PGMs is that they offer a framework for studying a broad range of probabilistic problems and their corresponding algorithms. Particularly, PGMs explicitly state modeling assumptions made and offer a unified framework where inference algorithms developed in different communities can be related and aggregated.

There are many types of graphical models that are best suited for representing different assumptions. However, all forms of graphical models have some level of the ability to express conditional dependence (or independence) statements. For example, "belief networks" (also called "Bayesian networks") are useful for modeling ancestral conditional independence. There are many other types of GM such as Markov networks, chain graphs, and factor graphs. It is not possible to introduce all of these types of GMs here since it would be irrelevant for this thesis. However, since factor graphs are used to implement the Bayesian estimators, a brief overview of this kind of GM is presented later on in this chapter.

In general, problem solving when probabilistic models are utilised can be divided in two high level steps.

- **Modelling** The first step in modelling a problem is to identify all of relevant variables of the problem domain at hand. The purpose is to determine how these variables interact. This can be achieved by exploiting the structural assumptions according to how they form the joint probability distribution.
- **Inference** After structure of the probabilistic model is developed the next step is to perform inference on the distribution to find answer for questions of interest. This step is a very important step as it can involve high level of computational complexity. Therefore, successful graphical modelling needs accurate inference algorithm.

2.4.1 Factorgraph

A factor graph $\mathcal{F}$ is an undirected graph containing two types of nodes: variable nodes (denoted as ovals) and factor nodes (denoted as squares). The graph only contains edges between variable nodes and factor nodes. A factor graph $\mathcal{F}$ is parameterized by a set of factors, where each factor node V_ϕ is associated with precisely one factor ϕ, whose scope is the set of variables that are neighbors of V_ϕ in the graph.

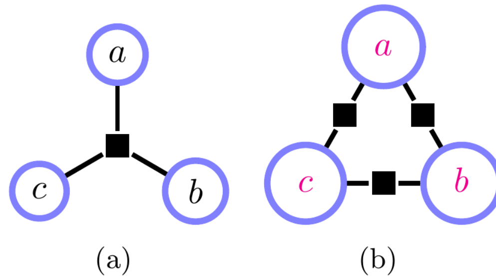


Figure 2.2 *Factor graphs representing different factorisations (a) $\phi(a, b, c)$ (b) $\phi(a, b)\phi(b, c)\phi(c, a)$*

A distribution P factorizes over $\mathcal{F}$ if the distribution can be represented as a set of factors of this form (i.e. can be written as the product of such factors).

For the function f is given as:

$$f(x_1, \ldots, x_n) = \prod_i \psi_i(\mathcal{X}_i) \tag{2.9}$$

In factor graphs, for each of the factors ψ_i a node (shown by a square) is represented in the network and for each of the variables x_j a variable node (shown by a circle) is represented. Also, an undirected link between the variable x_j and factor ψ_i is represented in the graph for each $x_j \subset \mathcal{X}_i$. Figure 2.2 represents two examples of factor graphs.

When used to represent a distribution, a normalization constant is assumed.

$$p(x_1, \ldots, x_n) = \frac{1}{Z} \prod_i \psi_i(\mathcal{X}_i) \tag{2.10}$$

2.5 Inference On Graphical models

Once structure of the model is encapsulated in a graphical mode, we then turn to the problem of inference. Inference in a graphical model is computing the posterior distributions of one or more subset of other nodes when some of the nodes are clamped to their observed values. The structure of the graph can be exploited to find efficient algorithms and to make the structure of those algorithms transparent. Specifically, many algorithms can be expressed in terms of the propagation of local messages around the graph. There are two classes of algorithms when performing inference on graphical models. The first class performs exact inference on the model and the second class performs approximate inference.

2.5.1 Marginal Inference

Marginal inference can be defined as computing probability distribution of a subset of variables, optionally conditioned on another subset. If a joint distribution $p(x_1, x_2, x_3, x_4, x_5)$ is given and evidence $x_1 = \mathrm{tr}$ is defined, marginal inference is calculated as

$$p(x_5|x_1 = \mathrm{tr}) \propto \sum_{x_2,x_3,x_4} p(x_1 = \mathrm{tr}, x_2, x_3, x_4, x_5) \tag{2.11}$$

2.5.2 Variable elimination and message passing

Message passing is the activity of summarizing information from the graph by information of local edges. Message passing is a crucial for efficient inference. Considering the four variable Markov chain

$$p(a, b, c, d) = p(a|b)p(b|c)p(c|d)p(d) \tag{2.12}$$

as presented in Fig 2.3. Here, we are asked to compute the marginal $p(a)$. To simplify the problem, we assume that all variables are binary and have the domain $0, 1$. Then

$$\begin{aligned} p(a = 0) &= \sum_{b\in\{0,1\},c\in\{0,1\},d\in\{0,1\}} p(a = 0, b, c, d) \\ &= \sum_{b\in\{0,1\},c\in\{0,1\},d\in\{0,1\}} p(a = 0|b)p(b|c)p(c|d)p(d) \end{aligned} \tag{2.13}$$

One way of performing the computation would be to sum each of the probabilities for all of the $2 \times 2 \times 2 = 8$ states of the variables b, c and d. This would therefore require 7 addition-of-two-numbers calls.

However, a more efficient way is to shift the summation over d to the right as much

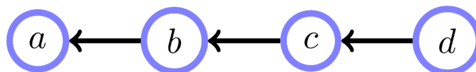


Figure 2.3 *Markov chain of the form equation 2.12*

as possible:

$$p(a=0) = \sum_{b\in\{0,1\},c\in\{0,1\}} p(a=0|b)p(b|c) \underbrace{\sum_{d\in\{0,1\}} p(c|d)p(d)}_{\gamma_d(c)} \tag{2.14}$$

In a similar way, the summation over c can be shifted to the right as much as possible:

$$p(a=0) = \sum_{b\in\{0,1\}} p(a=0|b) \underbrace{\sum_{c\in\{0,1\}} p(b|c)\gamma_d(c)}_{\gamma_c(b)} \tag{2.15}$$

Then, finally,

$$p(a=0) = \sum_{b\in\{0,1\}} p(a=0|b)\gamma_c(b) \tag{2.16}$$

By distributing the summations we have made $3\times2-1=5$ addition-of-two-numbers calls, compared to $2^3-1=7$ from the naive approach. The important point is that by following this procedure, the number of calculations for a chain with length $T+1$ would be linear, $2T$, as opposed to exponential, 2^T-1 for the first approach.

This process is referred to as *variable elimination*. Every time a sum over the states of a variable is computed, it is effectively eliminated from the distribution. It is always possible to effectively perform variable elimination in a chain due to the fact that summations can naturally be distributed from edges inwards.

We can see the variable elimination as a *message* (information) being passed to a neighbouring node in the graph. It is possible to compute a univariate-marginal of an arbitrary tree (i.e. a singly connected graph). To do so, we can start at a leaf (outermost node) of the tree and eliminate the local variable at each node and continue working inward from the leaf. By performing the elimination inwards (starting from the leaves), it is guaranteed that we are able to calculate any marginal $p(x_i)$ by using a linearly scaled number of summations (linear to number of variables).

When using continuous (parametric) distributions $(x|\theta_x)$, message passing is done by passing around parameters of the distribution θ. Doing so, we able to implement the sum-product algorithm by passing the parameters of the distribution, e.g. mean and covariance. It should be noted that this requires that multiplication and integration

should be closed operations with respect to the family of distributions. For Gaussian distributions for example, this is valid as the product of two Gaussian is again Gaussian and the marginal (integral) of a Gaussian is also Gaussian.

2.5.3 The sum-product algorithm

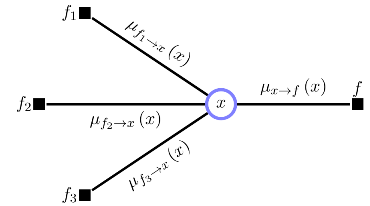
The *sum-product* algorithm, also referred to as *belief propagation* is algorithm of computing marginals by distributing the sum of the variable states over the product of factors. The sum-product algorithm can be performed on all types of graphical models. However, since the focus of this thesis is on factor graphs this algorithm is presented in the framework of factor graphs.

In sum-product algorithm, messages are updated according to the incoming messages from neighbouring nodes (as a function of incoming messages). Computations proceed according to a schedule that allows to compute the new outgoing message based on previously calculated messages. This process continues until all of outgoing messages from factors to variables as well as messages from variables to factors are computed.

Initialisation: If the leaf node is a factor, messages coming from the leaf node are initialised to the factor. Otherwise (the leaf node is a variable), messages coming from the leaf node initialized to unity.

Message from Variable to Factor

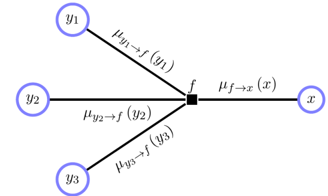
$$\mu_{x \to f}(x) = \prod_{g \in \{ne(x) \backslash f\}} \mu_{g \to x}(x)$$



Message from Factor to Variable

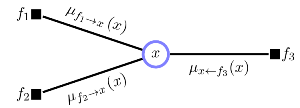
$$\mu_{f \to x}(x) = \sum_{\mathcal{X}_f \backslash x} \phi_f(\mathcal{X}_f) \prod_{y \in \{ne(f) \backslash x\}} \mu_{y \to f}(y)$$

Here $\sum_{\mathcal{X}_f \backslash x}$ denotes summation over all states in the variables set $\mathcal{X}_f \setminus x$



Marginal

$$p(x) \propto \prod_{f \in ne(x)} \mu_{f \to x}(x)$$



The important information in marginal inference is the relative size of the message states. This is important since we may want to renormalise the messages. The marginal of a variable is proportional to the messages sent to that variable from other nodes. Therefore, we can easily obtain the normalisation constant by utilizing the fact that the sum of the marginal should be 1.

2.6 Model Driven Engineering and Domain Specific Languages

2.6.1 Model Driven Engineering

Model Driven Engineering (MDE) or *Model Based Engineering* (MBE) is a methodology in software development that tries to improve the developed software by defining a *modeling language* (also called a *meta-model*). The meta-model captures the basic ideas and aspects in a particular domain. This language is then used to specify concrete models that can then be analyzed, validated, transformed or even executed. The latter activities are greatly facilitated by having a formalization of the meta-model available. The main benefit of this approach is the clean separation of the *domain knowledge* from technical implementation details.

MDE has been specified and standardized by different organizations. One of the mostly used is the *Model Driven Architecture* (MDA) initiative [25] by the Object Management Group (OMG). Several modeling levels have been described by MDA:

- The computation independent model (CIM) is the most high level and informal description
- The Platform Independent Model (PIM) specifies the software system independently of the platform where the software will later run on;
- Platform Specific Model (PSM) is the model can then be transformed to programming language code where information about the platform is added to the PIM model.

A thorough introduction to MDA can be found in [5].

MDA is developed along other OMG standards: the Meta-Object Facility (MOF) [28] describes the meta-meta modeling architecture and language that is used to specify meta-models. The Query View Transformation (QVT) [30] standard specifies several languages to support describing model transformations. The Unified Modeling Language (UML) [32] specifies a variety of standard diagrams for modeling different aspects of software (these diagrams themselves are described using MOF). The Object Constraint Language (OCL) [29] is used to define additional constraints on models which can't be expressed by UML alone.

A shortcoming of the approach proposed by OMG is the lack of a rigorous semantic formalization or a reference implementation. This leads to vendors making incompatible implementation choices. Executable UML (xUML) [23] tries to address this issue by formalizing the semantics of a subset of UML. Basic idea of xUML evolved out of the Shlaer-Mellor method [22] and has been adopted by OMG, giving birth to foundational UML (fUML) [31]. According to the fUML standard specification, the intention is "to encourage use of the broadest possible subset of UML constructs that can be reduced to a small set of elements" and to provide a "precise definition of the execution semantics of that subset."

In robotics, OMG's robotics domain task force (DTF) promotes and is extending OMG standards for developing component based robotics systems. Several standards have been specified, including the *Robotic Technology Component* (RTC) which has been used as the basis for the OpenRTM framework [2] as well as the *Robotic Localization Service* (RLS) and *the Robotic Interaction Service* (RoIS). Burmester et al. [7] have introduced Mechatronic UML as an extension to UML for modeling hybrid real-time systems.

2.6.2 Domain Specific Languages

A *Domain Specific Language* , is a language that, in contrast to a general purpose language, has been specifically tailored to express the concepts of a particular domain. DSLs have been in use for decades, especially in Unix operating system and were initially described as *little languages* by Bentley [4]. Some well-known examples of DSL are the "make" language to specify software builds, "sed" and "awk" for text processing or XML to describe hierarchically structured data. An extensive

overview of research preformed on DSL can be found in [39]. [36] and [24] discuss patterns and trade-offs involved in developing a DSL.

DSLs are central to MDE. Any meta-model can effectively be understood as a DSL; however the term DSL seems to be more associated with textual than graphical modeling languages. Generally, DSLs are categorized into two major types [9]:

- *External DSLs* are developed and constructed from scratch, usually with help of a language constructor tool such as lex or yacc.
- *Internal DSLs* or *embedded DSLs* are developed inside an already available general purpose "host" language. Many general purpose languages have been used to develop DSLs such as Haskell, Ruby, and Lisp.

Both approaches have their own advantages and disadvantages: external DSLs offer more flexibility for the syntax, but are more complex to implement since a parser for that particular syntax must be developed. On the other side, internal DSL are bound by limitations and syntax rules of the host language, but in return can reuse the host language's facilities for parsing, computing, and error reporting.

DSLs have been in use in robotics for a long time ([18]; [17]; [11]). MAESTRO [8] is a language developed for specification, validation and control of robotic missions. Frob [33] and AFRP [14] are internal DSLs built on top of the Haskell programming language for programming robots using the Functional Reactive Paradigm. Bjarnason et al. [6] have described a toolchain to interactively develop DSLs. In this work, a case study is discussed for industrial robot programming language and the need for a parametrizable and composable DSL is emphasized (such as composing DSLs for specifying the application level as well as motion control level). Then, two solutions are proposed based on multi-layered grammars and procedure inheritance.

3. BAYESIAN ESTIMATION AND PROBABILISTIC MODELING SOFTWARE

As will be discussed later in the next chapter, the scope of this thesis is a blend of Bayesian estimation software and Domain-Specific languages. While DSLs play an important role, in the end it is the graphical models that are used as the framework to compute the Bayesian estimation. There have been various projects to implement graphical modeling and inference.

As will be discussed later, in this project, one of already available software engines to specify the modeling and performing the inference is chosen and used. Therefore, it is beneficial to briefly present some of these projects which and specify the merit for decision made for the probabilist programming stack of choice.

OpenGM

OpenGM[3] is a library implemented in the C++ programming language for discrete factor graph models and distributive operations on these models. It allows for saving of models in a file in an open (HDF5) format. OpenGM imposes no restriction on factor graphs and can be used to construct a factor graph with arbitrary structure. However, it only supports tabular conditional probability distributions (sparse and dense tables). Many inference algorithms are supported by OpenGM. Figure 3.1 depicts an overview of the these algorithms.

OpenGM architecture is designed so that it can deal with large scale problems efficiently. OpenGM imposes no restrictions on the factor graph and what operations can be performed on the model. User extensions are automatically handled by the file format. OpenGM is modular and easily extendible. In OpenGM internal modules of the software such as inference algorithms, the graphical model data structure, and different encodings of functions interoperate through well-defined

Message passing	Graph cut	Search	Sampling	LP / ILP
Loopy BP	α-expansion	ICM	Gibbs	Dual decomposition
TRBP	$\alpha\beta$-swap	LazyFlipper	Swendsen-Wang	Branch & cut
TRW-S	QPBO	LOC		A*

Figure 3.1 *Inference algorithms supported by OpenGM*

interfaces.

However, the library's limited functionality to discrete factor graphs as well as tabular CPDs render this framework unusable for Bayesian state estimation in robotics where continuous state variables and probability distributions are present.

MRF-lib

MRF-lib[37] is a software API for performing inference based on energy minimization in Markov random fields (MRF). MRF-lib is restricted to min-sum semi-ring and second-order grid graphs. Although it is highly efficient in those, it is not easily extendible. Optimization algorithms implemented in MRF-lib are Iterated Conditional Modes (ICM), Graph Cuts, Max-product loopy belief propagation, and Tree-Reweighted Message Passing (TRW).

libDAI

LibDAI[26] is an open source library for inference in graphical models with discrete variables. LiDAI provides implementations of various exact and approximate inference methods for these models. libDAI support max-product and sum-product algorithms which are hard-coded in the library. LibDAI Supports only dense value tables in order to encode functions. Inference methods supported by libDAI are: junction-tree method and brute force enumeration for exact inference; mean field (loopy) belief propagation, fractional belief propagation, tree-reweighted belief propagation, generalized belief propagation, tree expectation propagation, double-loop generalized belief propagation, loop-corrected belief propagation, a Gibbs sampler, conditioned belief propagation and a decimation method for approximate inference methods for calculating MAP states, marginals and partition sums. In addition,

	libDAI	BNT	PNL	FastInf	GRMM	Factorie	JProGraM
Language	C++	MatLab, C	C++	C++	Java	Scala	Java
License	GPL2+	LGPL2	BSD	GPL3	CPL 1.0	CC-by 3.0	GPL3
Junction tree	+	+	+	+	+	-	+
Belief propagation (BP)	+	+	+	+	+	+	-
Fractional BP	+	-	-	-	-	-	-
Tree-reweighted BP	+	-	-	+	-	-	-
Generalized BP (GBP)	+	-	-	+	+	-	-
Double-loop GBP	+	-	-	-	-	-	-
Loop-corrected BP	+	-	-	-	-	-	-
Conditioned BP	+	-	-	-	-	-	-
Tree expectation propagation	+	-	-	-	-	-	-
Mean field	+	-	-	+	-	-	-
Gibbs sampling	+	+	+	+	+	+	+
Other sampling methods	-	+	+	+	+	+	-
Decimation	+	-	-	-	-	-	-
Continuous variables	-	+	+	-	-	+	+
Dynamic Bayes nets	-	+	+	-	-	+	-
Conditional random fields	-	-	-	-	+	+	-
Relational models	-	-	-	+	-	+	-
Influence diagrams	-	+	-	-	-	-	-
Parameter learning	+	+	+	+	+	+	+
Structure learning	-	+	+	-	-	-	+

Figure 3.2 *comparison of features supported by libDAI with some open-source libraries for approximate inference on graphical models*

parameter learning of conditional probability is possible in libDAI by maximum likelihood or expectation maximization (in case of missing data).

Figure 3.2 depicts a comparison of features supported by LibDAI with various open source software packages for approximate inference on graphical models:

FastInf

FastInf[16] is a library for memory and time efficient approximation of inference in large relational undirected graphical models. FastInf is focused on message passing and imposes no restriction on the factor graph structure. Contrary to libDAI, FastInf supports different function types and shared functions in a so-called relational model which follows the same design principles as OpenGM. However, only sum-product is supported in FastInf and unlike what is possible in OpenGM, no generic template abstraction of semi-rings is available in FastInf. Inference methods implemented in FastInf are Loopy Belief Propagation, Junction-Tree algorithm, Generalized Belief Propagation, Tree Re-weighted Belief Propagation for exact inference and propagation based on convexification of the Bethe free energy, Mean field, Gibbs sampling.

Libra

The Libra[20] toolkit is a framework for performing inference and structural learning for in discrete domains. It is developed in OCaml language with some memory-intensive routines implemented in C. For factors, Libra supports tables, trees, and arbitrary conjunctive feature functions. Many inference and learning algorithms are implemented in the Libra toolkit. For exact inference, the most common algorithms are junction tree, enumeration, and variable elimination. For approximate inference, Libra provides Gibbs sampling, loopy belief propagation, and mean field, all of which are optimized for structured factors. For learning, Libra supports maximum likelihood parameter learning and pseudo-likelihood optimization. Structure learning is one of Libra's greatest strengths.

Grante

Grante [27] is a proprietary library for modelling, inference, and learning using discrete factor graphs. Grante provides different function types and shared functions. Moreover, a set of learning methods are implemented in Grante. However, unlike OpenGM Grante is limited in its inference methods due to the fact that is not template based.

For factors Grante supports unary, pairwise, and high-order factors; linear data-dependent factors; non-linear data-dependent factors; and sparse factor data and for shared data among multiple factors. For inference many algorithms are implemented such as Sum-product and max-product Loopy Belief Propagation; MAP for exact inference in tree-structured factor graphs; and Approximate MAP-MRF Linear Programming inference. For learning Maximum (Conditional) Likelihood Learning for tree-structured factor graphs; Maximum Pseudolikelihood Learning for general factor graphs; and Expectation Maximization (EM) for partially observed data are supported.

BUGS, OpenBUGS, JAGS

BUGS [35] (Bayesian inference using Gibbs Sampling) is a language for Bayesian analysis of probabilistic models using Markov Chain Monte Carlo (MCMC) . The

BUGS project began in Cambridge, England in 1996 and later on the efforts were focused on the more modern implementation of the language, WinBUGS[21] and then later on the OpenBUGS project which is an open source version of package.

JAGS[34] (Just Another Gibbs Sampler) is a clone of the *classical* BUGS language used widely in many fields. One of the main advantages of JAGS is platform independence. JAGS is compatible with the original family of the BUGS language and can be controlled from within another program such as R.

In the family of BUGS languages, it is assumed that the model is specified in the form of a DAG (directed acyclic graph), and Gibbs sampling is used for inference. A large number of different conditional distributions (node types) are supported. Internally, various algorithms are used to sample from the full conditionals.

BFL

The Bayesian Filtering Library (BFL) [10] is a Bayesian estimation library included in the OROCOS framework for robotic software development. BFL provides an API for state estimation using the Kalman filtering family including the standard KF, EKF, IEKF and Non-minimal State KF; as well as the particle filtering family including standard Particle filter with arbitrary proposal, the Bootstrap filter, Auxiliary Particle filter, and Extended Kalman Particle Filter.

In implementation of filters in BFL no graphical model is used and the closed form solutions to the filters are implemented. The library provides a set of functions to "construct" the needed structure for the system dynamics and measurement models.

Dimple

Dimple [13] is another open-source software framework for probabilistic modeling, inference, and learning. Dimple provides facilities to define models in form of factor graphs and infer on those models using a variety of algorithms. These algorithms include sum-product and Gaussian belief propagation, min-sum BP, particle BP, linear programming, and Gibbs sampling.

For factors, it is possible to define arbitrary factor functions as well as to use a

library of mathematical functions. Many standard distributions are also supported by Dimple and can be used in defining the probabilistic models.

One of great features of Dimple is that it provides tools to easily specify complex models. In particular, support for nested graphs and rolled-up graphs (repeated structures inside a graph) will be discussed again later on in the next chapter.

4. RESEARCH METHODOLOGY AND MATERIALS

In this chapter research methods and materials used in this research work is presented. As stated in the introduction chapter, aim of this research work is to present a methodological approach to modeling knowledge about software implementations in sensor fusion applications in form of a domain specific language. This approach leverages software development for robotic applications by introducing a unified design for implementations. On the other hand, modeling of available knowledge allows sharing this information between machines more efficiently. Although this will be a long term goal, it is hoped that this work contributes towards achieving it.

In this chapter, a general overview of tools and frameworks used in this research is presented and their use is justified. Then, the modeling work conducted is explained, and lastly it is shown how this modeling is utilized to get a concrete implementation for a certain problem.

4.1 Factor graphs as Bayesian estimator

A short explanation of graphical models was presented in Chapter 2.4. As it was discussed, graphical modeling is a mixture of graph theory and probability. It is an established framework for probabilistic reasoning and provide a powerful and flexible framework of encapsulating probabilistic models in a concise manner. Inherent benefit of using factor graphs as a structural framework of modeling is the possibility of defining multiple layers of abstraction and building the model on top of simpler and smaller sub-models. This way, complex models are built and represented more efficiently. In this work, *factor graphs* have been chosen as the framework of modeling. This hybrid, bipartite graph offers good flexibility in modelling with defining explicit "factors" representing relationships between different probabilistic variables of the model represented as nodes in the graph.

In general, graphical models can be used as a general formalism for different fields of probabilistic modeling and reasoning. Graphical models enable us to represent probabilistic problems as common framework. This approach is very beneficial since developed methods of reasoning can be used regardless of the model being developed for a specific field of research. Methods and knowledge from a different domains can be shared and transferred easily once a common underlying infrastructure is used. These reasons, along with the computational benefits introduced in Section 2.4 were the reasons graphical models were chosen as formalism for modeling used in this thesis research.

Optimal filtering and estimation is used in various areas and levels of perception and action in robotics, from visual recognition of objects to low level control of joints. Therefore, it is beneficial to have a common ground for all of these problems and utilize framework of graphical modeling. Another natural gain of using graphical models is that often in robotics there are multiple layers of abstraction for a given problem. This work is focused on sensor fusion and in this area it is important to have multiple levels of abstractions. Problem of sensor fusion is subdivided in different layers of data association, data fusion and (sometimes) multi-sensor fusion. Each of these sub-problems can be tackled using graphical models.

Two main aspects of a graphical model (and hence a factor graph) are the structure of the model and the inference algorithm used which specifies how the mathematical calculations lead to an inference on the model. Bayesian estimators are indeed a form of solution to a statistical problem. This problem can be encapsulated and formulated in a graphical model like any other statistical inference problem. Various combinations of model structures and inference algorithms can be used to formulate Bayesian estimators.

In general, the structure of a factor graph for state estimation problem consist of three main elements: the *prior* state (state computed at the previous step), an observation of current state variables, and estimated ("inferred") state variables at the current time step considering the observations and the prior. It should be mentioned here that when laying out the structural design of the model, one can consider the whole time series of steps and corresponding state variables and construct a model with a relatively big skeleton (number of time steps * state + number of time steps * observation in each time step). This approach, as demonstrated in figure 4.1 leads to a very large model and by doing so, all observations of the state variables are

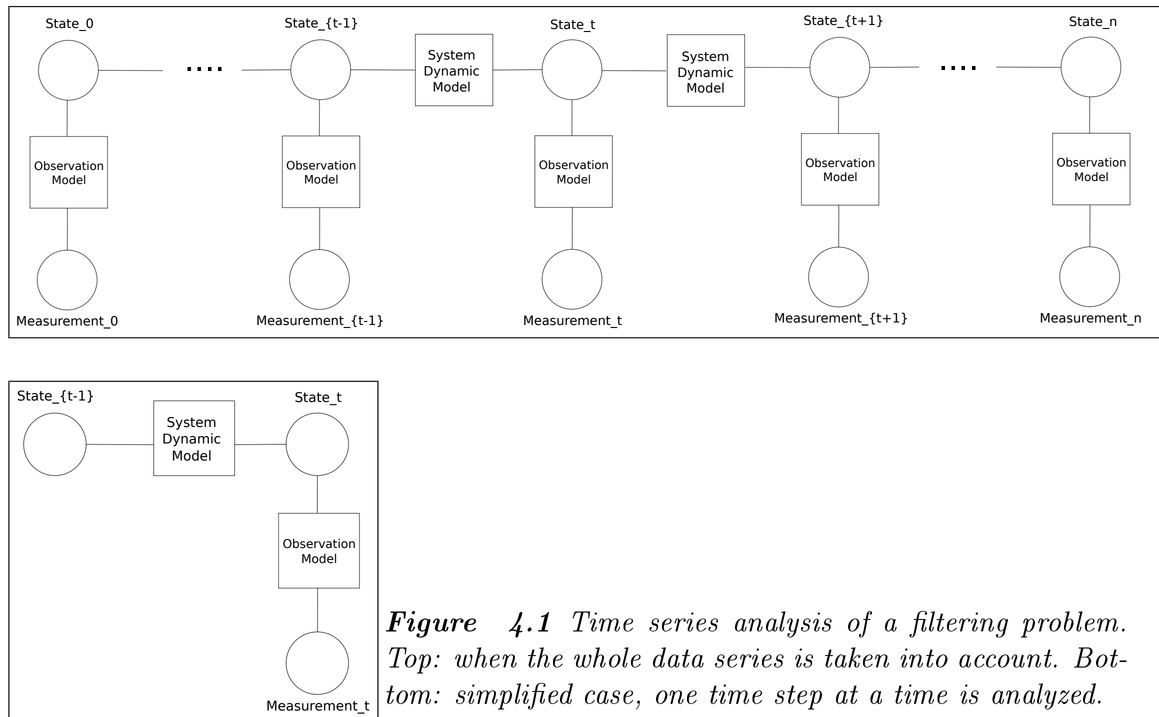


Figure 4.1 *Time series analysis of a filtering problem. Top: when the whole data series is taken into account. Bottom: simplified case, one time step at a time is analyzed.*

considered when each of the states at time step t are estimated.

However, this is not necessary in most of the cases and as it was introduced in Section 2.3.1 the Markov assumption can be used most of the times to simplify the solution. According to the Markov assumption, state of the system at the next step in time only depends on the current state. What this means in terms of state estimation is that we can subdivide the problem and take each time step individually rather than take the whole time series of state evolution. Figure 4.1 demonstrate the difference between modelling the whole time series (i.e. batch solution) vs the one-step at a time (i.e. recursive solution) approach.

The simpler (and more intuitive) approach of recursive state estimation can be realized using "*rolled-up*" graphs. A rolled-up factor graph is a factor graph where a smaller sub-graph is repeated many times inside the parent graph. This approach is also in-line with step-wise nature of robotic applications where a processing unit polls for data input from the surrounding environment, processes the data, and then perform an action. The rolled-up factor graph is specified only once in the model and when inference is performed, the result is a factor graph that is implicitly unrolled.

At each step of calculations, a sub-graph with a structure similar to the simplified structure shown in figure 4.1 is considered. The process starts with observed values

of state variables as an evidence of state of the current time step in the graph, as well as the prior belief of the state set as fixed values which is the posterior computed at the previous step. At the other side of the graph is the variable node which is the current state of the system. The inference engine starts with passing messages between all these nodes. Message passing between the nodes continues until either the model reaches equilibrium and values for next state are stable or a predefined maximum number of iterations is reached. In the latter case the inference does not produce the correct result and calculations are terminated.

One of the key areas when building a graphical model for a given problem is how this model should be reasoned about. Inference scheme of a graphical model specifies how message passings are performed and what kind of information each of the messages contain. Various inference schemes have been developed and used to solve graphical models of different types. Here, we focus on those specialized to use for factor models. Filters of interest here fall in two types of inference scheme families introduced in Chapter 2.5, sum-product and particle belief propagation. More details about this is given in the next subsections where Bayesian estimators are explored and it is discussed how factor graphs can be utilized to perform such estimations.

In order to understand better how optimal filtering can be achieved using graphical models (and more specifically factor graphs), it is essential to analyze the behavior of such inference algorithms and see how Bayesian filters can be “reconstructed” in a factor graph solver. In what follows, it is explained how one can specify a common behavior between specific solvers of factor graphs and Bayesian estimators.

4.1.1 Factor graphs for Kalman filtering

As stated in Chapter 2.3.2, main characteristics in a Kalman filtering problem are presence of a linear system as well as assumption of Gaussian probability distribution for state variables and additive Gaussian noise for the process and measurement noise i.e. the Kalman filter is the closed form solution to the Bayesian filtering equations for the filtering model, where the dynamic and measurement models are linear Gaussian.

Therefore, two key distinctions that should be taken into account for constructing the model for KF is the linear model structure and Gaussian distributions. As for the model structure, the system propagation and measurement models can easily

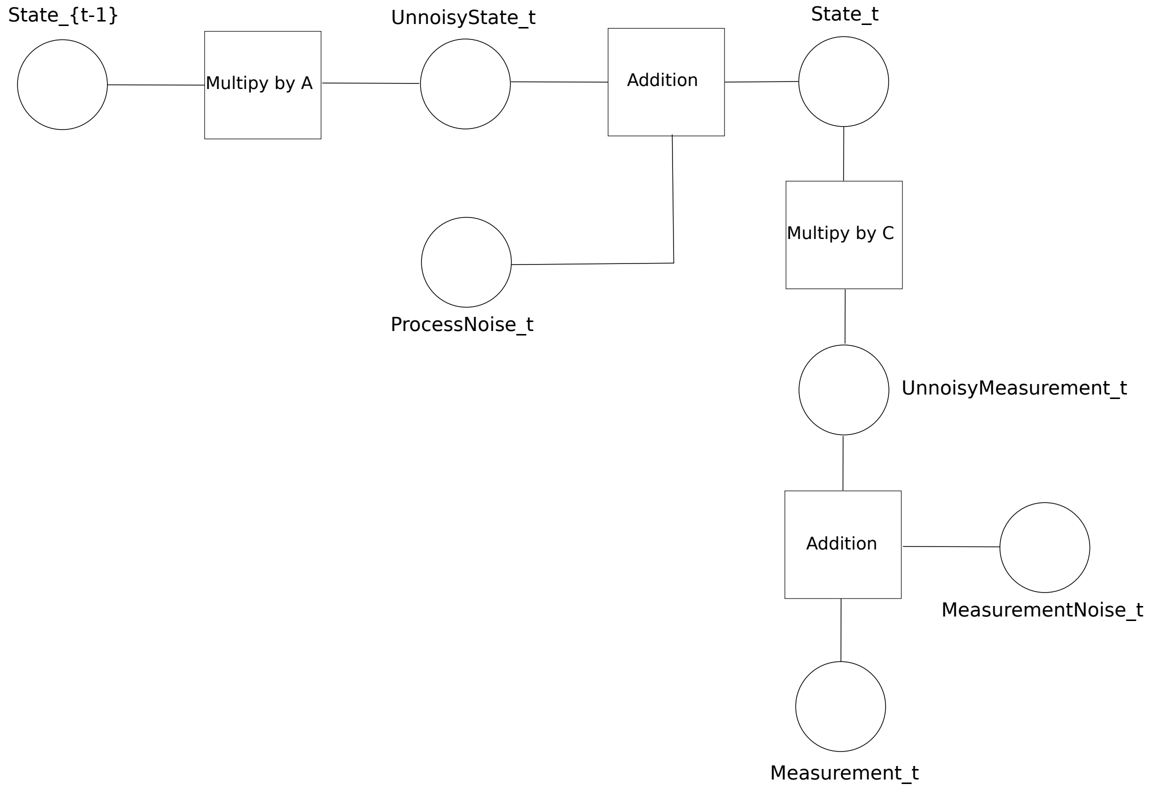


Figure 4.2 *Realisation of a linear system*

be constructed by defining linear factors between the state posterior and the prior saved from the previous step on one end and the current observation on the other. On the other hand, additive Gaussian noises can be modeled as separate nodes with a predefined distribution. As we will see later, when inference is performed on the model these nodes are sampled and a value is assumed which pertains to the designated distribution.

Figure 4.2 shows a realization of a linear system as defined in equations 2.4 and 2.5. As shown in the figure, probabilistic model of a linear system is easily constructed by combining sufficient variables (as nodes in the model, represented by circles) and functions (“factors”, represented by squares).

On the other hand, for solving the model for KF a variation of belief propagation algorithm introduced in Chapter 2.5.3 named Gaussian Belief Propagation (GaBP) [40] can be used. Gaussian BP i.e. is a special case of continuous BP, where the underlying distribution is Gaussian i.e. messages passed to and from continuous variables are represented in a Gaussian form. This ensures that the Gaussian representation of nodes is always kept since the belief of each of node is proportional

to sum of messages received and sum of Gaussians is Gaussian.

It is worthy to add here that this procedure can be used for implementing extended KF as well as the original filter. By adding n-th order derivative term of the nonlinear functions to the graphical model as factors it is possible to achieve the family of EKF filters (EKF, EKF2, etc).

4.1.2 Factor graphs for Particle filtering

Contrary to what is available in a Kalman filtering problem, in particle filters there is no assumption on the distribution of variables or the system model. The system can have arbitrary structure for observation and state evolution models. Therefore, there can be no "general" form of structure for the system model and consequently developing structure of a factor graph encapsulating a particle filter depends on the dynamic and observation model of a particular system and needs ad-hoc solution. The model can be highly non-linear with variables distributed with any distribution.

In Chapter 2.3.4 an overview of particle filters was presented. In order to capture behavior of particle filters in graphical models, a solver that uses *Particle Belief Propagation* (PaBP) [15] is used. This algorithm introduces a sample-based belief propagation in graphical models with encapsulate particle values in messages passed between nodes of graphical model. In principle, PaBP in graphical models and particle filters in Bayesian estimators share the same workings such as sampling of proposal kernels, particle weights, etc thus it is very convenient to implement particle filters by using graphical models and particle belief propagation message passing method.

4.2 Tools and Frameworks used

In this section different tools and software frameworks used to perform the work are presented. This thesis work deals with different areas ranging from probabilistic inference and mathematical modeling to software modeling and design. Proper tool needs to be selected for each of these areas in order for the project to be in harmony. In interdisciplinary projects such as this work if tools selected are incompatible and poorly selected it can result in the whole project to be either hard to use or hard to re-use.

The tools used in this work fall into two major categories, probabilistic programming and software modeling. This work is targeted at software development for sensor fusion in robotic applications. The sensor fusion part deals with probabilistic models and optimal filtering theory and the software development part deals with software modeling and domain-specific languages.

4.2.1 Probabilistic programming

In order to define probabilistic models using computers, various programming frameworks and languages have been developed. A wide range of possibilities is available for defining a probabilistic problem with various special purpose languages or frameworks written for general purpose languages. Some of these projects were introduced in Chapter 3. There have been frameworks developed using an already available general-purpose programming language and also there have been specialized languages developed from scratch for this specific purpose. These two categories of software are called *probabilistic programming languages*.

It is worth mentioning here that although this part of the work is done using a specific software framework, the main point of this effort has been towards what we believe to be a better approach to software development which results in better structured and more flexible software. The probabilistic modeling tool used here serves as a demonstration of how DSLs can be used to generate concrete implementation needed for sensor fusion applications and should be seen as a use-case example. The framework chosen here has been based on needs and criteria of a specific use-case which will be presented shortly. Others may find other frameworks or probabilistic programming languages more suited to their usage and should be able to choose freely their tools with as least effort as possible for adapting the software.

The criteria that were considered to choose the probabilistic modeling software here are as follows:

- Ability define models using factor graphs since this type of graphs are used to implement the Bayesian estimators
- Support as many inference algorithms as possible with reliable computations
- Support for defining hierarchical models. This was one of the major factors in choosing the probabilistic programming framework. This feature enables us

to define models on different levels of abstraction. This is beneficial in robotic applications which deal with different sub-areas which often have very different nature. Abstracting these on different levels can result in cleaner and more structured models.

- Licensed under an open source license to provide possibility of examining and modifying its source codes.
- Cross-platform.

Considering the above reasons, Dimple [1] was chosen as the framework for defining and reasoning on graphical models. Dimple is developed by Analog Devices Inc [2] and is released under the Apache license 2.0[3]. A brief introduction on Dimple was presented in Section 3. Dimple is focused on factor graphs with possibility of choosing various inference schemes including PaBP and GaBP discussed earlier. To the best of writer's knowledge, this set of features is unique among available frameworks. Specially, availability of more complex solvers such as those based on sampling methods such as Gibbs [12] and particle belief propagation for inference on factor graphs is beneficial for implementing sophisticated Bayesian estimators.

Dimple's core software is developed in Java programming language. There is also a Matlab® interface available which can be used to define models and perform inference on them. The Matlab interface exposes all of Java classes and utilities to Matlab. In this work the Matlab API is used because of ease of use and cleaner code output. Another benefit of using the Matlab API is availability of various functions and utilities that can be used to pre- or post-process the data easily as well as easy data visualization using built-in graphs.

4.2.2 Software Models

Another side of this thesis work deals with modeling of software components and domain specific languages. As discussed in Chapter 2.6.2 it is possible to define models in a plethora of different ways. Well known methodologies have been used throughout the years to model and design software and modeling languages have been developed with different characteristics and design goals.

[1] http://dimple.probprog.org/
[2] http://www.analog.com
[3] http://www.apache.org/licenses/LICENSE-2.0

One of well known methods is the framework of *Unified Modeling Language* (UML). A brief overview of UML methodology is presented in Section 2.6.1. One of the main problems with UML is that it is primarily intended for humans to deal with and understand. Using UML diagrams is not very well suited for machine to process and therefore a machine should be able to perform complex processing on the model to be able to understand and analyze the models. As stated earlier, one of the main reasons for this work is to enable robots to share their knowledge about the problem at hand and it is important to choose a tool best suitable for this need.

On the contrary to UML models which are graphical and based on diagrams, there are textual models which represent the model in text files. One of textual modeling formats which was chosen to use in this thesis work is a light-weight and clean modeling notation named JavaScript Object Notation (JSON) [4]. JSON is a general-purpose open standard that introduces simple textual notations to define and transmit data models. JSON models are simple text files containing structure of data. JSON models are easy for humans to read and write and easy for machines to parse and generate.

Listing 1 JSON notation of a simple object

```
1 {
2   "id": 1,
3   "name": "Example_Model",
4   "tags": [
5     "Tag1",
6     "Tag2"
7   ]
8 }
```

Figure 1 demonstrate an example of a model written in JSON format. One of key features of JSON is that it is programming language independent. A model is written in simple textual format and can be easily parsed and used in various languages which either support JSON through their standard library or by using a third-party developed library. JSON models are built on top of two types of structures. The first type is a collection of name-value pairs and the second type is an ordered list of values. A model can be specified by encapsulation of attributes

[4] http://www.json.org/

Listing 2 JSON schema example

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Model",
  "type": "object",
  "required": ["id", "name"],
  "properties": {
    "id": {
      "type": "number",
      "description": "Model identifier"
    },
    "name": {
      "type": "string",
      "description": "Name of the example model"
    },
    "tags": {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  }
}
```

of the model in key-value pairs or by specifying ordinal sequence of values. A lot of advanced features have been added to JSON standard during the past years, allowing to define complex models with interconnected references to other models or foreign types. JSON has been standardized since 2013 and is used widely in diverse application domains.

Using JSON, one is able to start sketching a model for a specific need/problem, populating the "values" in the name-value pair or the ordered list mentioned earlier. However, this is not a very good approach as the model is not explicitly checked against a predefined structure. One of advanced features of JSON is that it is possible to define a "template" or a "blueprint" of the model, referred to as *schema*. JSON schema are themselves a model defined in the JSON notation but instead of specifying specific values, types and rules are filled in the model. In other words, JSON schema are meta-models that define models. These schema are specified once

for a specific application and then for each realizations of the schema (i.e. the actual instantiation of the model) the model is validate against the rules defined in the schema. A simple JSON schema is presented in listing 2. Note that this schema describes the model given in listing 1 and can be used to check the validity of the model.

One the big advantages of JSON models are the fact that due to clean and minimal syntax they are easily understandable by humans as well as machines. Support for JSON format has been embodied in many programming languages which makes it very easy to generate and parse models using already available tools. Although JSON format was inspired first by JavaScript language, it is a language independent format which is not tied to a specific language. This has high importance in this work since as already mentioned the purpose here is to decouple models from actual implementations of the software. By using a tool that is not tied to a specific programming language or software framework this can be easily achieved. It should be possible to choose freely low level underlying software implementations based on needs of a use case and software models should be easy to specify accordingly.

The fact that JSON is an open standard and well supported in various programming languages makes it an ideal tool for our use case. It is worth mentioning that here what is specified in JSON files are meta-model of the Bayesian estimators (and sometimes different models for underlying components) which is then processed to understand about the structure of the estimator and its internals. A more detailed explanation of the workflow is give in the next section.

4.3 Modeling

Up until now it was explained how and by what means the modelling and Bayesian estimator specification is laid out. In this section, the structure and contents of the models are explained.

In general, recursive Bayesian estimators usually consist of a *prediction* component which “blindly” (i.e. without looking at observations of the real word) computes a belief propagation of the system states according to the system model, and an *update* component which takes into account observed values of the state variables in order to improve the predictions. This is a common behavior that can be seen in all of families of Bayesian estimators but with differences in details of how these steps are

taken.

Consequently, one would expect this structure to be applicable in the software developed for Bayesian estimation as well. Although this is true from an algorithmic point of view, this is more applicable to implementation of the software. In this work, emphasis is placed on minimalism and ease of use for the end-user i.e. the engineer dealing with the estimator. From user's point of view, a Bayesian estimator of any type should be fully defined with the fewest parameters possible and in an intuitive and easy to use manner.

In order to have clear and coherent semantics, modeling is performed in different levels of software. At the top most level, there is a "master" filter model which acts as a placeholder for defining each of the Bayesian estimators. This container model then includes reference to a list of possible sub-models (KF, EKF, etc. filter specifications) containing actual definitions of each of the filters. The schema used for the Bayesian filter estimators and an example of such filters is given in Appendix A

One important remark that should be noted here is that these high-level meta-models specify the structure of a filtering scheme and have enough information to infer about the parameters of a certain estimator from algorithmic point of view but they bear no meaningful information regarding actual implementation of the software. The reason for this is that each of these model encapsulate knowledge from a specific domain. In other words, while Bayesian estimator models express what are the parameters of a certain filter, there should be a middle-level mapping between these definitions and the actual pieces of software to be used. In order to be able to so, there should be a meta-model representation of the low level software implementations. As stated in 4.2.1 Dimple is used as the software framework for implementing the Bayesian filers. Therefore, different components of this framework relevant to Bayesian filtering and graphical models were modeled. On top of already available elements in Dimple, some additional models were added to provide more layers of abstraction leading to a cleaner design and more intuitive utilization of the models. These added models include elements such as *Matrix* data type and *function* definitions.

All of the models discussed so far are spread across files each containing a meaningful piece of sub-models. As stated in 4.2.2 models are defined in JSON file format allowing to validate models against a predefined schema. JSON schemas developed are the actual models of the software and examples are also implemented and validated

against the schema definitions.

4.4 Code generation

In the previous section it was explained how meta-models are utilized to represent the knowledge in Bayesian filtering. In an actual use case, these models should be then translated to a level that machines can understand and run. Various low-level implementations of Bayesian estimators exist with different characteristics and design goals.

As stated in previous chapters, Dimple was chosen as the framework of defining probabilistic networks. The work flow of defining the Bayesian estimator to deployment and execution is:

- First a Bayesian estimator is defined by specifying the meta-models in JSON files.
- Then, these models are then processed by scripts and routines defined in Python programming language to validate the filter definition against the previously defined schema.
- Then, the model is parsed to obtain the type and parameters of the estimator.
- Once the type and parameters of the filter is known, depending on the type of the estimator it is decided which of the middle-level filtering model should be used to determine how high level filter definition should be mapped to functions and utilities of the computational framework.
- At the end, the lower level implementation of the estimator is written using the parameters defined by the user and the framework models.

It should be noted that this translation from the meta-models to lower level lines of code is very framework dependent and greatly differs if different programming language/software framework is used. In this thesis work, the low level language of computations is chosen to be Matlab. Dimple offers interfaces for both Matlab and Java programming language but it is chosen to work with the Matlab interface due to ease of use and slightly simpler coding needed to define the factor graph network.

The output of Python scripts after processing the models is a Matlab m-file containing necessary lines of code to define, configure, and solve the factor graph. This approach allows the software developer to freely choose which specific implementation to use for a given problem since it is only needed to modify the intermediate level i.e. the translation layer so that it contain information about how meta-models should be translated to code.

5. RESULTS AND DISCUSSION

In this chapter, a sample problem for Bayesian estimation is discussed and it is shown how models can be used to implement a Kalman filter for the recursive estimation of state variables. It is also shown how graphical models can be used as the underlying computational framework of the solution. Accuracy of the result is compared between the factor graph model solution with GaBP and the closed form solution.

5.1 An example problem

For this section, a simple linear system is considered and used for recursive estimation. The model is a discretized version of the noisy resonator model with a given angular frequency ω. The system has two state variables and one observed state with Gaussian white noise for state propagation and measurements. The system model is stated in equations 5.1 and 5.2.

$$\begin{aligned} \mathbf{x}_t &= \begin{pmatrix} \cos(\omega) & \frac{\sin(\omega)}{w} \\ -\omega\sin(\omega) & \cos(\omega) \end{pmatrix} \mathbf{x}_{t-1} + \epsilon_t \\ z_t &= \begin{pmatrix} 1 & 0 \end{pmatrix} \mathbf{x}_t + \delta_t \end{aligned} \tag{5.1}$$

where $\mathbf{x}_t \in \mathbb{R}^2$ is the state, z_t is the measurement, $\delta_k \sim N(0, 0.1)$ is a white Gaussian measurement noise and $\epsilon_t \sim N(\mathbf{0}, \mathbf{Q})$, where

$$\mathbf{Q} = \begin{pmatrix} \frac{q^c\omega - q^c\cos(\omega)\sin(\omega)}{2\omega^3} & \frac{q^c\sin^2(\omega)}{2\omega^2} \\ \frac{q^c\sin^2(\omega)}{2\omega^2} & \frac{q^c\omega + q^c\cos(\omega)\sin(\omega)}{2\omega} \end{pmatrix} \tag{5.2}$$

The angular frequency is $\omega = \frac{1}{2}$ and the spectral density is $q^c = 0.01$. The problem

was first solved by the closed-form Kalman filter and then the same parameters were used to construct a factor graph model and solved. The data used for measurement and real values for state variables where obtained by simulating the system operation in Matlab.

5.1.1 Filter Model

Estimator models were explained in Section 4.3. For this example problem, system and measurement models are linear with additive Gaussian noise. Therefore, the filtering problem is described as the model shown below:

```
{
  "filter_type": "KF",
  "filter_specs": {
    "transition_matrix": [
      [0.8776,   0.9589],
      [-0.2397,  0.8776]
    ],
    "process_noise_covariance": [
      [0.0032,  0.0046],
      [0.0046,  0.0092]
    ],
    "measurement_matrix": [
      [1, 0]
    ],
    "measurement_noise_covariance": [
      [0.1]
    ],
    "input_matrix": [
      [0]
    ]
  }
}
```

As can be seen, the above representation of the estimator is more compact and

understandable as opposed to the case where the problem characteristics are interleaved with implementation details.

It should be noted here that the above model is the simplest form of a Bayesian estimator. More complex models can be specified for different types of estimators. It is possible to refer to another model defined in another file/url to build more complex filters. Examples of this external reference in models is used for functions in non-linear filtering, models of solvers, distributions, etc.

5.1.2 Model Checking and Code Generation

Model schema were introduced in Section 4.2.2. When the user specifies the characteristics of the estimator as explained above, the models are fed to the python scripts for checking and parsing. Validity of the models are checked against the set of predefined schema for different layers of the estimator and if the model is valid, the equivalent implementation of the filter is generated. As discussed in Section 4.2.1 the framework used for implementing graphical models is Dimple. For this example problem, the equivalent implementation of the filter generated is as below:

```
1  transition_matrix__ = [0.8776 0.9589 ; -0.2397 0.8776];
2  process_noise_covariance__ = [0.0032 0.0046 ; 0.0046 0.0092];
3  measurement_matrix__ = [1 0];
4  measurement_noise_covariance__ = [0.1];
5
6  n_state = size(transition_matrix__,1);
7  n_measured = size(measurement_matrix__,1);
8
9  % Child graph
10 states_prior = RealJoint(n_state);
11 states_prior.Name = 'StPrior';
12 states_nonoise = RealJoint(n_state);
13 states_nonoise.Name = 'StNN';
14 states_posterior = RealJoint(n_state);
15 states_posterior.Name = 'StPosterior';
16 observation_nonoise = RealJoint(n_measured);
17 observation_nonoise.Name = 'ObsNN';
```

```
18 observation = RealJoint(n_measured);
19 observation.Name = 'Obs';
20 nested_graph =
   ↪ FactorGraph(states_prior,states_posterior,observation);
21
22 process_noise = RealJoint(n_state);
23 process_noise.Name = 'PrcsN';
24
25 measurement_noise = RealJoint(n_measured);
26 measurement_noise.Name = 'MsrN';
27
28 % KF specific parts %
29 setSolver('SumProduct');
30 process_noise.Input = FactorFunction('MultivariateNormal',zeros(n_st ⌋
   ↪ ate,1),process_noise_covariance__);
31 measurement_noise.Input = FactorFunction('MultivariateNormal',zeros( ⌋
   ↪ n_measured,1),measurement_noise_covariance__);
32 state_transition_factor = nested_graph.addFactor(@constmult,states_n ⌋
   ↪ onoise,transition_matrix__,states_prior);
33 state_transition_factor.Name = 'StTrF';
34 state_noise_factor = nested_graph.addFactor(@add,states_posterior,st ⌋
   ↪ ates_nonoise,process_noise);
35 state_noise_factor.Name = 'StNF';
36 measurement_projection_factor = nested_graph.addFactor(@constmult,ob ⌋
   ↪ servation_nonoise,measurement_matrix__,states_posterior);
37 measurement_projection_factor.Name = 'MsrPrF';
38 measurement_noise_factor = nested_graph.addFactor(@add,observation,o ⌋
   ↪ bservation_nonoise,measurement_noise);
39 measurement_noise_factor.Name = 'MsrNF';
40 % /KF specific parts%
41
42
43 % Variable streams
44 states_stream = RealJointStream(n_state);
45 measurement_stream = RealJointStream(n_measured);
46
```

```
47 % Parent Graph
48 parent_graph = FactorGraph();
49 parent_graph.addFactor(nested_graph,states_stream.getSlice(1),states
   ↪ _stream.getSlice(2),measurement_stream.getSlice(1));
```

5.1.3 Results

The network obtained in the above section is the equivalent of the estimator specified in JSON.However, there is still some manual work needed in order to solve the problem. First it is assumed that Dimple is initialized and all of the Dimple API calls are available. Second, the input and output of the network is left for the user to manually setup since the inputs may come from various sources such as online stream of sensor data or as in this example, from a series of simulated data residing in Matlab's workspace. In the network generated, the input/output nodes of the netowrk (variable streams in Dimple's terminology) are defined but connecting these nodes to streams of data is left for the user.

Figure 5.1 depicts the results obtained from running the GM network generated. For the sake of comparison, matrix form solution of the KF estimator for this specific problem is included in the graph. The figure also depicts the time series of true values for the simulated observed state as well the (simulated) observed value. The overall script which includes the matrix-form solution and data simulation part can be found in Appendix B.

As can be seen from figure 5.1, the factor graph estimator is able to filter the signal with satisfactory accuracy. The RMS error of the FG estimator was computed to be 0.4 which is higher when compared to the RME of the KF filter, 0.2 but the difference is small. In fact, this slightly higher error is expected since the KF filter is the optimal solution while the factor graph solution is approximated. The Matlab file used to generate the simulated data and performing the KF filtering is included in Appendix B.

5.2 Future Work

The models and framework presented here is a first step towards a better and more structured software for a specific problem, sensor fusion. Although this practice

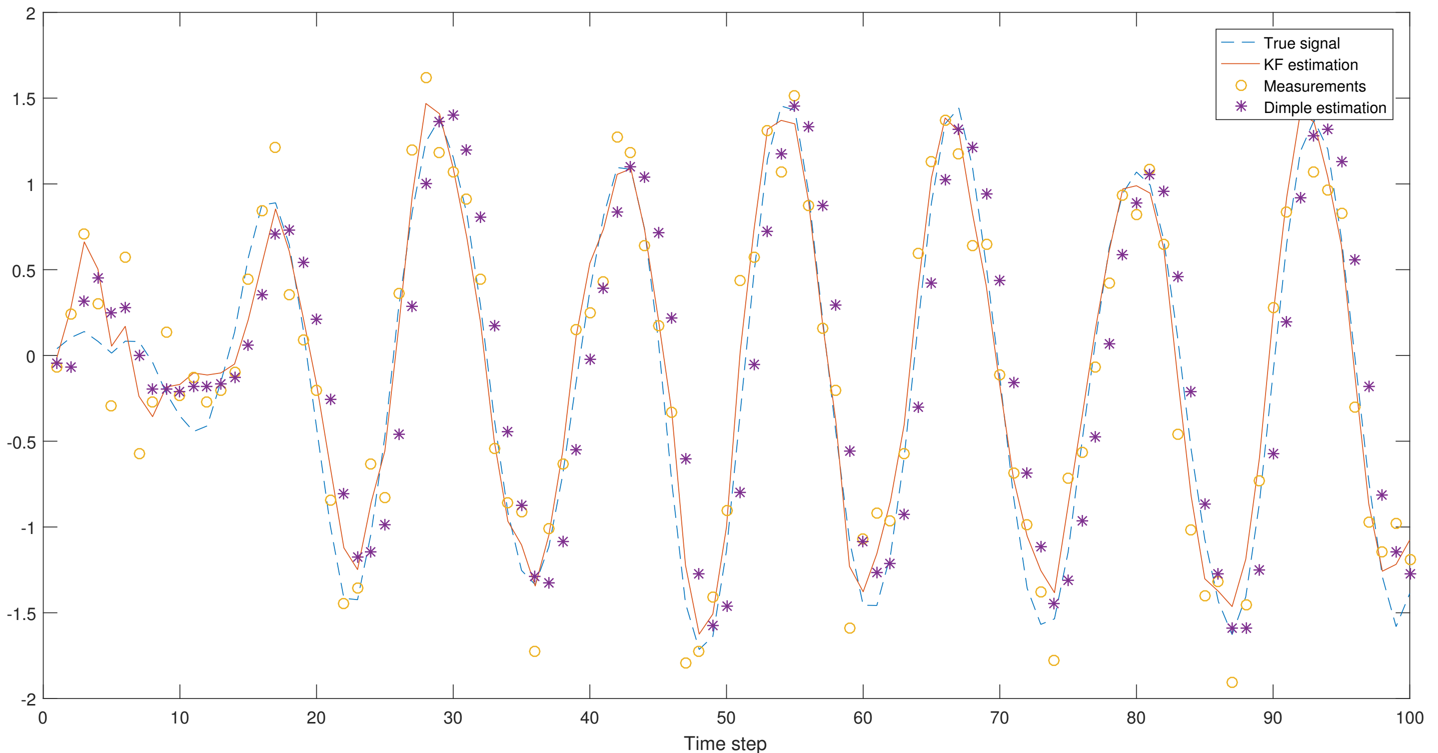


Figure 5.1 *Comparison of KF performance when closed form matrix solution used vs when implemented using factor graph in Dimple*

has been in use for many other application areas in computer science and software engineering domain, in robotics mostly the software developed does not adhere to MDE methodology. This work was performed in the hope that one day all layers of software involved in a Robotic application are developed in a way that is easier to maintain for humans and easier to process and configure for machines.

However, this work can be extended in many ways. The underlying computational framework for factor graph models is one example of such these frameworks. Every framework (and programming language) has its own set of strengths and limitations. It would be better to have as many alternatives as possible so that it can be easily chosen based on the need for a project to choose which framework. The scripts developed to parse and validate models have also limited functionality and are developed for a simple use-case. Also, another one distant goal can be to leave this choice to the machines themselves. When software is intelligent enough to have enough knowledge about estimation algorithms and its own environment, it is not impossible to imagine that one day it can choose the best fitting algorithm for a specific task autonomously.

6. CONCLUSION

In this thesis research, a software stack was developed for Bayesian sensor fusion in robotics applications (though not limited to only robotics). The framework relies heavily on the practices of model-driven engineering and architecture. The software is used as a small domain-specific language where parameters and structure of the Bayesian estimator are written in form of models in JSON schema format.

These models are then processed and a mapping between the high level filter properties and lower level computational stack (which is itself modeled and is processed by the scripts) is constructed. At the end, the final code is written in the syntax of the target framework and programming language which can be used to deploy on the target hardware and perform the filtering.

Graphical models (more specifically, a type of graphical models known as factor graphs) were chosen as the underlying computational framework to implement the estimators. Graphical models offer a concise and structured framework to model and solve a probabilist problem. Using factor graphs, Bayesian estimators can be formulated in form of a network. Structure of the network and the solver used in the model are then chosen according to the specific type of the estimator. At the end, as an example a problem was considered as proof of concept for performing the Bayesian estimation using factor graphs.

BIBLIOGRAPHY

[1] S. Ali-Löytty, *Gaussian mixture filters in hybrid positioning.* Tampere University of Technology, 2009.

[2] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: Openrtm-aist," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots.* Springer, 2008, pp. 87–98.

[3] B. Andres, T. Beier, and J. H. Kappes, "Opengm: A c++ library for discrete graphical models," *arXiv preprint arXiv:1206.0111*, 2012.

[4] J. Bentley, "Programming pearls: little languages," *Communications of the ACM*, vol. 29, no. 8, pp. 711–721, 1986.

[5] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

[6] E. Bjarnason, G. Hedin, and K. Nilsson, "Interactive language development for embedded systems," *Nord. J. Comput.*, vol. 6, no. 1, pp. 36–54, 1999.

[7] S. Burmester, H. Giese, and M. Tichy, "Model-driven development of reconfigurable mechatronic systems with mechatronic uml," in *Model Driven Architecture.* Springer, 2005, pp. 47–61.

[8] E. Coste-Maniere and N. Turro, "The maestro language and its environment: Specification, validation and control of robotic missions," in *Intelligent Robots and Systems, 1997. IROS'97., Proceedings of the 1997 IEEE/RSJ International Conference on*, vol. 2. IEEE, 1997, pp. 836–841.

[9] M. Fowler, "Language workbenches: The killer-app for domain specific languages," 2005.

[10] K. Gadeyne, "BFL: Bayesian Filtering Library," http://www.orocos.org/bfl, 2001.

[11] E. Gat, "Alfa: A language for programming reactive robotic control systems," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on.* IEEE, 1991, pp. 1116–1121.

[12] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 721–741, 1984.

[13] S. Hershey, J. Bernstein, B. Bradley, A. Schweitzer, N. Stein, T. Weber, and B. Vigoda, "Accelerating inference: towards a full language, compiler and hardware stack," *CoRR*, vol. abs/1212.2991, 2012.

[14] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," in *Advanced Functional Programming.* Springer, 2003, pp. 159–187.

[15] A. T. Ihler and D. A. McAllester, "Particle belief propagation." in *AISTATS*, 2009, pp. 256–263.

[16] A. Jaimovich, O. Meshi, I. McGraw, and G. Elidan, "Fastinf: An efficient approximate inference library," *J. Mach. Learn. Res.*, vol. 11, pp. 1733–1736, Aug. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1859908

[17] L. P. Kaelbling, "Goals as parallel program specifications." in *AAAI*, 1988, pp. 60–64.

[18] L. KAELBLING, "Rex- a symbolic language for the design and parallel implementation of embedded systems," in *AIAA Computers in Aerospace Conference, 6 th, Wakefield, MA*, 1987, pp. 255–260.

[19] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[20] D. Lowd and A. Rooshenas, "The libra toolkit for probabilistic models," *CoRR*, vol. abs/1504.00110, 2015. [Online]. Available: http://arxiv.org/abs/1504.00110

[21] D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter, "Winbugs-a bayesian modelling framework: concepts, structure, and extensibility," *Statistics and computing*, vol. 10, no. 4, pp. 325–337, 2000.

[22] S. S. S. J. Mellor and S. Shlaer, *Object-oriented systems analysis: modeling the world in data.* Yourdon, 1989.

[23] S. J. Mellor, M. Balcer, and I. Foreword By-Jacoboson, *Executable UML: A foundation for model-driven architectures.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[24] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[25] J. Miller and J. Mukerji, *MDA Guide version 1.0.1. Technical report*, Object Management Group (OMG), 2003.

[26] J. M. Mooij, "libdai: A free and open source c++ library for discrete approximate inference in graphical models," *J. Mach. Learn. Res.*, vol. 11, pp. 2169–2173, Aug. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1859925

[27] S. Nowozin, "Grante library," http://www.nowozin.net/sebastian/grante.

[28] Object Management Group, "Meta object facility (mof) core specification," http://www.omg.org/technology/documents/formal/data_distribution.htm.

[29] ——, "Object Constraint Language," http://www.omg.org/spec/OCL/.

[30] ——, "Query View Transformation," http://www.omg.org/spec/QVT/.

[31] ——, "Semantics of a Foundational Subset for Executable UML Models (fUML)," http://www.omg.org/spec/FUML/1.0/.

[32] ——, "Unified Modeling Language (UML) superstructure specification," http://www.uml.org/.

[33] J. Peterson, P. Hudak, and C. Elliott, "Lambda in motion: Controlling robots with haskell," in *International Symposium on Practical Aspects of Declarative Languages.* Springer, 1999, pp. 91–105.

[34] M. Plummer *et al.*, "Jags: A program for analysis of bayesian graphical models using gibbs sampling," in *Proceedings of the 3rd international workshop on distributed statistical computing*, vol. 124. Vienna, 2003, p. 125.

[35] D. J. Spiegelhalter, A. Thomas, N. G. Best, W. Gilks, and D. Lunn, "Bugs: Bayesian inference using gibbs sampling," *Version 0.5,(version ii) http://www. mrc-bsu. cam. ac. uk/bugs*, vol. 19, 1996.

[36] D. Spinellis, "Notable design patterns for domain-specific languages," *Journal of systems and software*, vol. 56, no. 1, pp. 91–99, 2001.

[37] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother, "A comparative study of energy minimization methods for markov random fields with smoothness-based priors," *IEEE transactions on pattern analysis and machine intelligence*, vol. 30, no. 6, pp. 1068–1080, 2008.

[38] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics.* MIT press, 2005.

[39] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography." *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[40] Y. Weiss and W. T. Freeman, "Correctness of belief propagation in gaussian graphical models of arbitrary topology," *Neural computation*, vol. 13, no. 10, pp. 2173–2200, 2001.

Appendices

A. JSON SCHEMA MODELS

Schema of the top level Bayesian filter

```
1  {
2    "id": "https://gitlab.mech.kuleuven.be/u0097847/bayesianmodelling/
       ↪  raw/master/models/bayesian_filter_schema.json",
3    "$schema": "http://json-schema.org/draft-04/schema#",
4    "description": "Top level schema for a Bayesian filter",
5    "type": "object",
6    "properties": {
7      "filter_type": {
8        "description": "Enum specifying type of filtering; one of
           ↪  Kalman filtering, Extended Kalman filtering first order,
           ↪  Extended Kalman filtering second order",
9        "type": "object",
10       "oneOf": [
11         {
12           "description": "KF filter model",
13           "filter_type": "KF",
14           "filter_specs": {
15             "$ref": "https://gitlab.mech.kuleuven.be/u0097847/
                 ↪  bayesianmodelling/raw/master/models/kf_schema.json",
16           }
17         },
18         {
19           "description": "EKF filter model",
20           "filter_type": "EKF",
21           "filter_specs": {
22             "$ref": "https://gitlab.mech.kuleuven.be/u0097847/
                 ↪  bayesianmodelling/raw/master/models/ekf_schema.json",
```

```
23            }
24          },
25          {
26            "description": "PF filter model",
27            "filter_type": "PF",
28            "filter_specs": {
29              "$ref": "https://gitlab.mech.kuleuven.be/u0097847/
                 ↪  bayesianmodelling/raw/master/models/pf_schema.json",
30            }
31          },
32        ]
33      },
34      "required": [
35        "filter_type", "filter_specs"
36      ]
37    }
38  }
```

An example of the top level Bayesian filter

```
{
  "filter_type": "KF",
  "filter_specs": {
    "transition_matrix": [
      [0.8776,   0.9589],
      [-0.2397,  0.8776]
    ],
    "process_noise_covariance": [
      [0.0032,  0.0046],
      [0.0046,  0.0092]
    ],
    "measurement_matrix": [
      [1, 0]
    ],
    "measurement_noise_covariance": [
```

```
      [0.1]
    ],
    "input_matrix": [
      [0]
    ]
  }
}
```

Schema of the Kalman filter

```
1  {
2    "id": "https://gitlab.mech.kuleuven.be/u0097847/bayesianmodelling/
      ↪  raw/master/models/kf_schema.json",
3    "$schema": "http://json-schema.org/draft-04/schema#",
4    "description": "Model of the Kalman filter",
5    "type": "object",
6    "properties": {
7      "uuid": {
8        "description": "Unique ID of of the Factor Graph (for now:
          ↪  UUID)",
9        "type": "string",
10       "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-
          ↪  [a-fA-F0-9]{4}-[a-fA-F0-9]{12}$"
11     },
12     "transition_matrix": {
13       "description": "State transition matrix",
14       "$ref": "matrix_schema.json#/matrix_definition"
15     },
16     "measurement_matrix": {
17       "description": "name of the state measurement matrix",
18       "$ref": "matrix_schema.json#/matrix_definition"
19     },
20     "process_noise_covariance": {
21       "description": "State transition noise covariance matrix (noise
          ↪  is assumed to be zero mean)",
```

```
22        "$ref": "matrix_schema.json#/matrix_definition"
23      },
24      "measurement_noise_covariance": {
25        "description": "Measurement noise covariance matrix (noise is
          ↪  assumed to be zero mean)",
26        "$ref": "matrix_schema.json#/matrix_definition"
27      },
28      "input_matrix": {
29        "description": "Input matrix of the linear system",
30        "$ref": "matrix_schema.json#/matrix_definition"
31      },
32    },
33    "required": [
34      "uuid",
35      "transition_matrix",
36      "measurement_matrix",
37      "process_noise_covariance",
38      "measurement_noise_covariance",
39      "input_matrix"
40    ]
41  }
```

Schema of the extended Kalman filter

```
1   {
2     "id": "https://gitlab.mech.kuleuven.be/u0097847/bayesianmodelling/
        ↪  raw/master/models/ekf_schema.json",
3     "$schema": "http://json-schema.org/draft-04/schema#",
4     "description": "Model of the Extended Kalman filter",
5     "type": "object",
6     "properties": {
7       "uuid": {
8         "description": "Unique ID of of the graph (for now: UUID)",
9         "type": "string",
```

```
10        "pattern": "^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-f
          ↪  A-F0-9]{4}-[a-fA-F0-9]{12}$"
11      },
12      "transition_model": {
13        "description": "State transition model specified in form of a
          ↪  function signature",
14        "$ref": "function_schema.json#/function_definition"
15      },
16      "measurement_model": {
17        "description": "Measurement model specified in form of a
          ↪  function signature",
18        "$ref": "function_schema.json#/function_definition"
19      },
20      "linearized_transition_model": {
21        "description": "Linearized transition model specified in form
          ↪  of a function signature, output of this function is
          ↪  equivalent to first derivative of transition model with
          ↪  respect to state variables",
22        "$ref": "function_schema.json#/function_definition"
23      },
24      "linearized_measurement_model": {
25        "description": "Linearized measurement model specified in form
          ↪  of a function signature, output of this function is
          ↪  equivalent to first derivative of measurement model with
          ↪  respect to state variables",
26        "$ref": "function_schema.json#/function_definition"
27      },
28      "input_model": {
29        "description": "Input model specified in form of a function
          ↪  signature",
30        "$ref": "function_schema.json#/function_definition"
31      },
32      "process_noise_covariance": {
33        "description": "State transition noise covariance matrix
          ↪  (noise is assumed to be zero mean)",
34        "$ref": "matrix_schema.json#/matrix_definition"
```

```
35      },
36      "measurement_noise_covariance": {
37        "description": "Measurement noise covariance matrix (noise is
        ↪  assumed to be zero mean)",
38        "$ref": "matrix_schema.json#/matrix_definition"
39      },
40    },
41    "required": [
42      "uuid",
43      "transition_model",
44      "measurement_model",
45      "linearized_transition_model",
46      "linearized_measurement_model",
47      "input_model",
48      "process_noise_covariance",
49      "measurement_noise_covariance",
50    ]
51  }
```

Helper models

```
1   {
2     "id": "https://gitlab.mech.kuleuven.be/u0097847/bayesianmodelling/
    ↪  raw/master/models/matrix_schema.json",
3     "$schema": "http://json-schema.org/draft-04/schema#",
4     "description": "Model of a matrix in json",
5     "type": "object",
6     "properties": {
7       "matrix_definition": {
8         "type": "array",
9         "items": {
10          "type": "array",
11          "items": {
12            "type": "number"
13          }
```

```
14        }
15      }
16    }
17  }
```

```
1  {
2    "id": "https://gitlab.mech.kuleuven.be/u0097847/bayesianmodelling/
     ↪ raw/master/models/function_schema.json",
3    "$schema": "http://json-schema.org/draft-04/schema#",
4    "description": "Model of a function signature",
5    "type": "object",
6    "properties": {
7      "function_definition": {
8        "function_name": {
9          "description": "Function name string",
10         "type": "string"
11       },
12       "interface_description": {
13         "description": "An array of variables this function accepts
           ↪ as arguments, each element in the array is an array in
           ↪ itself with fist element name of the argument and second
           ↪ element type of the argument",
14         "type": "array",
15         "items": {
16           "type": "array",
17           "items": [
18             {
19               "description": "argument name",
20               "type": "string",
21             },
22             {
23               "description": "argument type",
24               "type" : "string",
25               "enum": [
26                 "https://gitlab.mech.kuleuven.be/u0097847/bayesianmo
                   ↪ delling/raw/master/models/variable_schema.json"
```

```
27                 ]
28               }
29             ],
30             "additionalItems": false
31           }
32         },
33         "function_parameters": {
34           "description": "An array of variables this function accepts
           ↪  as parameters",
35           "type": "array",
36           "items": {
37             "oneof": [
38               {"type": "number"},
39               {"type": "string"},
40               {"$ref": "matrix_schema.json#/matrix_definition"}
41             ]
42           }
43         },
44         "required": [
45           "interface_description",
46           "function_arguments"
47         ]
48       }
49     },
50   }
```

B. MATLAB CODE

Matlab program used in Chapter 5 to generate results in the sample problem

```
%% Generate data

clear; clc; close all;
% Lock random seed
rng(12);

% Gaussian ranom draw, m is the mean and S the covariance
gauss_rnd = @(m,S) m + chol(S)'*randn(size(m));

% Calculate root mean squared error
rmse = @(x,y) sqrt(mean((x(:)-y(:)).^2));

% Define parameters
steps = 100;  % Number of time steps
w     = 0.5;  % Angular velocity
q     = 0.01; % Process noise spectral density

% This is the transition matrix
transition_matrix = [cos(w)    sin(w)/w;
                    -w*sin(w) cos(w)];

% This is the process noise covariance
process_noise_covariance = [0.5*q*(w-cos(w)*sin(w))/w^3
 ↪ 0.5*q*sin(w)^2/w^2;
                           0.5*q*sin(w)^2/w^2
                            ↪ 0.5*q*(w+cos(w)*sin(w))/w];
```

```
25
26  % This is measurement model Matrix
27  measurement_matrix = [1 0];
28
29  % This is measurement noise variance
30  measurement_noise_covariance = 0.1;
31
32  % This is the true initial value
33  initial_state = [0;0.1];
34
35  % Simulate data
36  true_signal = zeros(2,steps);  % The true signal
37  measurement = zeros(1,steps);  % Measurements
38  T = 1:steps;          % Time
39  state = initial_state;
40  for k=1:steps
41  state = gauss_rnd(transition_matrix*state,process_noise_covariance);
42  y = gauss_rnd(measurement_matrix*state,measurement_noise_covariance);
43  true_signal(:,k) = state;
44  measurement(:,k) = y;
45  end
46
47
48  process_noise_mean = zeros(2,1);
49  measurement_noise_mean = zeros(1,1);
50
51
52  %% Kalman filter
53
54    % Kalman filter solution. The estimates
55    % of x_k are stored as columms of
56    % the matrix EST2.
57
58    m2 = [0;1];  % Initialize first step
59    P2 = eye(2); % Some uncertanty in covariance
60    EST2 = zeros(2,steps); % Allocate space for results
```

```
61
62    tic;
63    % Run Kalman filter
64    for k=1:steps
65      % Prediction step
66      m2 = transition_matrix*m2;
67      P2 = transition_matrix*P2*transition_matrix'+process_noise_covar⌋
       ↪  iance;
68
69      % Update step
70      v = measurement(k)-measurement_matrix*m2;
71      S = measurement_matrix*P2*measurement_matrix'+measurement_noise_⌋
       ↪  covariance;
72      K = P2*measurement_matrix'/S;
73      m2 = m2+K*v;
74      P2 = P2-K*S*K';
75
76      % Store the results
77      EST2(:,k) = m2;
78    end
79    t_kf = toc;
80    fprintf('Time elapsed for Kalman filtering algorithm:
     ↪  %0.4f\n',t_kf);
81
82  %% KF in Dimple
83
84  n_state = size(transition_matrix,1);
85  n_measured = size(measurement_matrix,1);
86
87  % Child graph
88  states_prior = RealJoint(n_state);
89  % states.Name = 'State';
90  states_prior.Name = 'StPrior';
91  states_nonoise = RealJoint(n_state);
92  % states_next_nonoise.Name = 'StateNextNoNoise';
93  states_nonoise.Name = 'StNN';
```

```
94  states_posterior = RealJoint(n_state);
95  % states_next.Name = 'StatesNext';
96  states_posterior.Name = 'StPosterior';
97  observation_nonoise = RealJoint(n_measured);
98  % measurement_nonoise.Name = 'MeasurementNoNoise';
99  observation_nonoise.Name = 'ObsNN';
100 observation = RealJoint(n_measured);
101 % measurement.Name = 'Measurement';
102 observation.Name = 'Obs';
103 
104 nested_graph =
     ↪ FactorGraph(states_prior,states_posterior,observation);
105 
106 process_noise = RealJoint(n_state);
107 % process_noise.Name = 'ProcessNoise';
108 process_noise.Name = 'PrcsN';
109 
110 measurement_noise = RealJoint(n_measured);
111 % measurement_noise.Name = 'MeasurementNoise';
112 measurement_noise.Name = 'MsrN';
113 
114 % KF specific parts %
115 setSolver('SumProduct');
116 process_noise.Input = FactorFunction('MultivariateNormal',zeros(n_st⌋
     ↪ ate,1),process_noise_covariance);
117 measurement_noise.Input = FactorFunction('MultivariateNormal',zeros(⌋
     ↪ n_measured,1),measurement_noise_covariance);
118 state_transition_factor = nested_graph.addFactor(@constmult,states_n⌋
     ↪ onoise,transition_matrix,states_prior);
119 % state_transition_factor.Name = 'StateTransitionFactor';
120 state_transition_factor.Name = 'StTrF';
121 state_noise_factor = nested_graph.addFactor(@add,states_posterior,st⌋
     ↪ ates_nonoise,process_noise);
122 % state_noise_factor = nested_graph.addFactor(@AdditiveNoise,states_⌋
     ↪ next,states_next_nonoise,process_noise_covariance);
123 % state_noise_factor.Name = 'StateNoiseFactor';
```

```
124 state_noise_factor.Name = 'StNF';
125 measurement_projection_factor = nested_graph.addFactor(@constmult,ob
    ↪ servation_nonoise,measurement_matrix,states_posterior);
126 % measurement_projection_factor.Name = 'MeasurementProjectionFactor';
127 measurement_projection_factor.Name = 'MsrPrF';
128 measurement_noise_factor = nested_graph.addFactor(@add,observation,o
    ↪ bservation_nonoise,measurement_noise);
129 % measurement_noise_factor.Name = 'MeasurementNoiseFactor';
130 measurement_noise_factor.Name = 'MsrNF';
131 figure;
132 nested_graph.plot('labels',true);
133 title('Factor Graph Model Of The Kalman Filtering Problem');
134 % /KF specific parts%
135
136
137
138 % Variable streams
139 states_stream = RealJointStream(n_state);
140 measurement_stream = RealJointStream(n_measured);
141
142 % Parent Graph
143 parent_graph = FactorGraph();
144 parent_graph.addFactor(nested_graph,states_stream.getSlice(1),states
    ↪ _stream.getSlice(2),measurement_stream.getSlice(1));
145
146 % Measurement Data Source
147 measurement_datasource = MultivariateDataSource();
148 for i = 1:steps
149     measurement_datasource.add(measurement(:,i),1e-100*eye(n_measure
        ↪ d));
150 end
151 measurement_stream.DataSource = measurement_datasource;
152
153 % States Data Sink
154 states_datasink = MultivariateDataSink();
155 states_stream.DataSink = states_datasink;
```

```
156
157 % Solve graph
158 parent_graph.setOption('DimpleOptions.randomSeed',10);  %No effect?
159 tic;
160 parent_graph.solve();
161 t_dmpl = toc;
162 fprintf('Time elapsed for factor graph solution: %0.4f\n',t_dmpl);
163
164 % Get results
165 dimple_guesses = zeros(1,steps);
166 i = 0;
167 while states_datasink.hasNext()
168     i = i + 1;
169     m = states_datasink.getNext();
170     dimple_guesses(i) = m.Means(1);
171 end
172 % Last iteration results
173 for j = 1:length((states_stream.Variables)-1)
174     i = i + 1;
175     dimple_guesses(i) = states_stream.Variables(j).Belief.Means(1);
176 end
177
178 %% Compare Results
179
180 % Plot the true signal and its estimates
181 figure;
182 plot(T,true_signal(1,:),'--',T,EST2(1,:),'-',T,measurement,'o',T,dim
    ↪ ple_guesses,'*');
183 legend('True signal','KF estimation','Measurements','Dimple
    ↪ estimation');
184 xlabel('Time step'); title('\bf Filtering Performance');
185
186 % Compute error
187 fprintf('RMS error for KF estimate:
    ↪ %0.4f\n',rmse(true_signal(1,:),EST2(1,:)));
```

```
188 fprintf('RMS error for Dimple estimate:
    ↪ %0.4f\n',rmse(true_signal(1,:),dimple_guesses));
```