Lehrstuhl für Steuerungs- und Regelungstechnik
Technische Universität München

Algorithms and Hardware for Reasoning in Factor Graphs

Indar Sugiarto

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr.-Ing. W. Hemmert

Prüfer der Dissertation:

1. Univ.-Prof. Dr.sc.nat. J. Conradt
2. Prof. St. Furber, University of Manchester/UK
3. Univ.-Prof. Dr.-Ing. K. Diepold

Die Dissertation wurde am 12.06.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 15.11.2015 angenommen.

Foreword

This dissertation summarizes my PhD research at Technische Universität München (TUM) with support from many people. I would like to express my sincere gratitude to them although I would not be able to mention all of them in this short page. First of all, I would like to express my gratitude to my supervisor, Prof. Jörg Conradt, and to my research sponsor, the DAAD. Jörg has guided me through the research and has showed me how to become an independent researcher with enthusiasm. Not only he has given me the inspiration for the topic of this dissertation, he also has given me the freedom to extend the initial research that I am interested in as well as exploring many things even beside the domain of my research area. I am also deeply appreciate his support beyond the research matter, which eases my family life in Munich - the most expensive city in Germany. I also thank DAAD, from whom I get the scholarship that cover mostly our living cost and also by them I was introduced to the Germany legal system.

I would also like to thank my external colleagues and collaborators during my research. I am very grateful for the initial discussion with Dr. Matthew Cook and his team from the Institute of Neuroinformatics (INI) at ETH-Zurich, with whom I have learned the basic principle of factor graphs. One of my research ideas comes as the result of our discussion during my visit to ETH-Zurich in 2012. I also address my thanks to the Advanced Processor Technology (APT) research group at the University of Manchester, in which I learned for the first time the use of their SpiNNaker system. I met them once again during CapoCaccia Neuromorphic workshop in 2012, and I was immersed in a deep discussion and worked together with Evangelos, Luis Plana, Sergio, Francesco, and Simon, just to name a few. My thank goes also to Paul Meier from the Center for Sensorimotor Research at Ludwig-Maximilians-Universität (LMU) Munich, with whom I met several times in various BCCN events and also had fruitful discussions on many conceptual topics related with my research.

My research life in TUM would not be cheerful without my "room-mates". I would like to thank NST-ers for "horsing around", especially to Cristian, Mohsen, Nicolai, Lukas, Marcello, Christoph, and Viviane. I have spent so many hours every week with them to figure out, sometimes in a never-ending discussion, what is actually "our purpose" in the NST world. I am glad that I've found co-workers that share our struggle with the same spirit. I also extend my sincere thanks to Susanne for taking care a lot of administrative tasks for me during my research and my work at NST and LSR.

Finally, I would like to thank my friends, Indonesian community, who live in Munich and offers my family the warmness and kindness in an Indonesian atmosphere. Especially to PRII/MRII München, with whom we have spent many cheerful Sundays in reformational enlightenment. And last, but not least, I thank to God for my family: Astri and Ben. Only for them, my life and my future are dedicated.

Munich, May 2015 Indar Sugiarto

Abstract

The goal of this thesis is to investigate inference schemes in the domain of graphical models and to develop new minimalistic inference schemes which can be implemented in dedicated hardware. This low-level hardware implementation will be very useful for example in the case of autonomous robotics, where hardware resources are limited and power consumption is one of the main constraining factors. We focus on the factor graph which is one of the probabilistic graphical models known as having structural representation similar to an undirected graph but can also be used to represent a directed graph. A factor graph is a bipartite probabilistic graphical model which is composed of two types of nodes: variable nodes and factor nodes. A factor node can represent a conditional probability distribution or simply a functional relationship between variables nodes connected to it. One important task that can be performed in a factor graph is an inference using a message-passing mechanism in a belief propagation scenario. The factor graph performs this inference mechanism by exchanging messages between nodes via the sum-product algorithm. Using the sum product algorithm, a probabilistic query such as marginalization can be performed efficiently. In a discrete factor graph, such a message is represented as a vector of probabilistic values. This is a common implementation of a factor graph and its belief propagation on a digital system. However, this sum-product algorithm might run slowly in a sequential system (i.e. standard PCs). In this thesis, we strive to find a good solution not only to speed-up the computation, but also to find the most efficient message representation. We propose to implement the message representation in a form of population code. This idea comes from computational neuroscience which suggests that a population of neurons can represent a probability distribution resulting from an ensemble of all neuronal activation levels in the population. We argue that factor graphs with population code message encoding can be used to mimic the brain's operation: they can be used to describe massively parallel distributed systems, where overall performance comes from concurrently communicating local computations of many individual units. With this view, we implement such an inference engine in dedicated hardware to mimic the brain's nature on distributed computation. This thesis presents the evolution of our factor graph from a PC-based framework to an embedded factor graph which runs on dedicated hardware. We have developed factor graph models for some important application domains such as machine learning and robotics. The implementation results of our PC-based factor graph framework for those applications show that the framework is mature enough to be taken to hardware instantiations. We started exploring the hardware implementation using a SpiNNaker (Spiking Neural Network Architecture) system. This neuromorphic system shows a potential benefit for implementing factor graphs due to its well defined routing mechanism. From this first embedded factor graph version, we gained insight for developing the second embedded factor graph on an "empty" device. The second hardware implementation is based on SoC (System on Chip), which can be fully optimized to achieve the goal of this thesis. This SoC device contains an FPGA (Field Programmable Gate Array) in which we have developed the core modules for an embedded factor graph from a scratch. This second embedded factor graph gives us flexibility and opens the possibility to extend it into a more powerful system. With these achievements, we have provided the foundation to extend factor graphs into a fully reconfigurable computing machine that will ultimately be capable of performing brain-style information processing.

Zusammenfassung

In dieser Dissertation werden Inferenz-Methoden aus dem Bereich der Graphischen Modelle untersucht, sowie minimalistische Inferenz-Schemen entwickelt, die sich mit dedizierter Hardware implementieren lassen. Die Hardware-nahe Umsetzung wird sich beispielsweise für autonome Roboter als nützlich erweisen, denen begrenzte Hardware-Ressourcen zur Verfügung stehen und bei denen der Energieverbrauch einen limitierenden Faktor darstellt. Die Arbeit konzentriert sich auf Faktorgraphen (FG), eine besonders vielseitige Klasse probabilistischer graphischer Modelle. FGen sind bipartit, und bestehen aus zwei Arten von Knoten: Variablen- und Faktorknoten. Ein Faktorknoten kann hierbei eine bedingte Wahrscheinlichkeitsverteilung oder einfach ein funktionaler Zusammenhang zwischen den mit ihm verbundenen Variablenknoten sein. Eine wichtige Anwendung eines FG ist Inferenz mittels eines Message-Passing-Algorithmus in einem Belief-Propagation-Szenario. In diesem sog. Sum-Product-Algorithmus geschieht die Inferenz, indem Nachrichten zwischen den Knoten ausgetauscht werden. Mit dem Sum-Product-Algorithmus lassen sich wahrscheinlichkeitstheoretische Aufgaben wie Marginalisierung effizient ausführen. Die Nachrichten bestehen im Fall eines diskreten FG aus einem Vektor probabilistischer Werte. Dies ist eine übliche Implementierung eines FG mit Belief-Propagation auf digitalen Systemen. Allerdings ist dieser inhärent parallelisierbare Algorithmus auf sequentiellen Rechnern langsam und ineffizient. Es ist Gegenstand dieser Arbeit, diese Situation grundlegend zu verbessern - nicht nur um Hardware-Implementierungen zu beschleunigen, sondern auch um die effizienteste Form von Nachrichten-Messages zu finden. Wir stellen eine effiziente Implementierung vor, die auf neuronalen Population Codes beruht. Diese Idee aus der Neurowissenschaft impliziert, dass eine Population von Neuronen, bzw. deren Aktivität, eine Wahrscheinlichkeitsverteilung repräsentieren kann. Wir zeigen, dass ein Population-Code-basierter FG im Wesentlichen die Arbeitsweise des Großhirns imitiert: Ein massiv-paralleles verteiltes System, dessen Leistung wie seine Rechenergebnisse sich aus gleichzeitig nebeneinander kommunizierenden lokalen Rechenoperationen vieler individueller Einheiten ergeben. In diesem Sinne implementieren wir also eine Inferenz-Maschine in dedizierten Schaltungen, die dann die Funktionsweise des Hirns imitieren. Diese Dissertation stellt die fortlaufende Weiterentwicklung unseres Faktorgraphen von einer PC-basierten Anwendung hin zu eingebetteten FG dar, die auf dedizierter Hardware laufen. Wir haben Faktorgraphen für einige wichtige Anwendungsbereiche wie Robotik oder Maschinenlernen entwickelt. Die Resultate unseres PC-basierten FG-Frameworks für diese Anwendungen belegen dessen Leistungsfähigkeit und motivieren dessen Umsetzung auf Hardware-Niveau. Wir beginnen mit einer Implementierung auf SpiNNaker (Spiking Neural Network Architecture). Die speziellen Routing-Mechanismen dieses neuromorphen Systems bedeuten potentielle Vorteile für FG-Implementierungen. Die zweite Hardware-Implementierung basiert auf einem System-on-Chip (SoC), das sich für die Ziele dieser Arbeit vollständig optimieren lässt. Das SoC beinhaltet ein Field Programmable Gate Array (FPGA), in dem wir Kernmodule unseres eingebetteten FG umsetzen konnten. Diese Implementierung eines eingebetteten FG gibt uns viel Flexibilität und eröffnet zahlreiche Möglichkeiten für weitreichende Weiterentwicklungen. Mit diesen Errungenschaften haben wir das Fundament von Faktorgraphen in eine vollständig rekonfigurierbaren Rechenmaschine gestellt, das Informationen in einer ähnlichen Art und Weise wie das menschliche Hirn verarbeiten kann.

Contents

List of Figures

List of Tables

List of Algorithms

Notations

Abbreviations

AI	Artificial Intelligence
ARM	Advanced Reduced Instruction Set Computing (RISC) Machines
AXI	Advanced eXtensible Interface
BN	Bayesian Network or Belief Network
BP	Belief Propagation
BRAM	Block Random Access Memory
CPT	Conditional Probability Table
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DBN	Dynamic Bayesian Network
DMA	Direct Memory Access
DOF	Degree of Freedom
EM	Expectation Maximization
FF	Flip-flop
FFG	Forney-style Factor Graph
FG	Factor Graph
FPGA	Field Programmable Gate Array
GMM	Gaussian Mixture Model
GMR	Gaussian Mixture Regression
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
HMM	Hidden Markov Model
IP	Intellectual Property
KL	Kullback-Leibler
LBP	Loopy Belief Propagation
LMS	Least Mean Squares
LUT	Look-up Table
MAP	Maximum A Posteriori
MC	SpiNNaker Multicast Packet
MCMC	Markov Chain Monte Carlo
MDL	Minimum Descriptive Length
MDP	Markov Decision Process
MLE	Maximum Likelihood Estimation
MRF	Markov Random Field
MSE	Mean of Squared Error

NoC	Network on Chip
NST	Neuroscientific System Theory
PACMAN	SpiNNaker Partition and Configuration Manager
PbD	Programming by Demonstration
PC	Personal Computer
PDF	Probability Density Function
PGM	Probabilistic Graphical Model
PL	Programmable Logic
PMF	Probability Mass Function
PS	Processing System
RMSE	Root-Mean-Square Error
RV	Random Variable
SDP	SpiNNaker Datagram Protocol
SLAM	Synchronous Localization and Mapping
SoC	System-on-Chip
SOM	Self-Organizing Map
SpiNNaker	Spiking Neural Network Architecture
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration

Conventions

Scalars, Vectors, and Matrices

Scalars are denoted by upper and lower case letters in italic type. *Vectors* are denoted by underlined lower case letters in italic type, as the vector $\underline{x}$ is composed of elements x_i. *Matrices* are denoted by underlined upper case letters in italic type, as the matrix $\underline{M}$ is composed of elements M_{ij} (i^{th} row, j^{th} column).

x or X	Scalar
$\underline{x}$	Vector
$\underline{X}$	Matrix
$\underline{X}^T$	Transposed of $\underline{X}$
$\underline{X}^{-1}$	Inverse of $\underline{X}$
$\underline{X}^+$	Pseudoinverse of $\underline{X}$
$f(\cdot)$	Scalar function
$\underline{f}(\cdot)$	Vector function
$\hat{x}$	Estimated or predicted value of x
$\tilde{x}$	Estimation error: $\tilde{x} = x - \hat{x}$
$\overline{x}$	Average value of x
$\|\cdot\|_p$	p-norm
$\nabla f(\underline{x}) = \frac{\partial f(\underline{x})}{\partial \underline{x}}$	Gradient vector

Graph Theory

$\mathbf{G} = (\mathbf{X}, \mathbf{F})$	factor graph $\mathbf{G}$ with a set of variable $\mathbf{X}$ corresponds to a set of factor $\mathbf{F}$
$\deg(i)$	degree of node i in the graph
ne(x)	neighbour of x

Probability Theory

$p(A_1, A_2, \cdots, A_n)$	Joint probability of $A_1, A_2, \cdots, A_n$
$p(A \mid B)$	Conditional probability of A given B
$\mathcal{N}(x; \mu, \sigma)$	Normal distribution on X with mean μ and variance σ^2

Symbols

General

$x(t)$	State of a dynamic system in a continuous domain
x_k	State of a dynamic system in a discrete domain
$y(t)$	System output in a continuous domain
y_k	System output in a discrete domain
σ	Standard deviation
σ^2	Gaussian Variance
μ	Gaussian Mean or average
$\mathcal{N}$	Normal distribution
$\frac{1}{Z}$	Factor normalizer
$\mu_{x \to f}(x)$	Message from a variable- to a factor node
$\mu_{f \to x}(x)$	Message from a factor- to a variable node
θ	Network parameter
λ	Lagrange multiplier
x	not x, or exclude x

1 Introduction

1.1 From Artificial Intelligence to Probabilistic Graphical Models

1.1.1 Cognition for Technical Systems

For more than 50 years since its initial booming, Artificial Intelligence (AI) has attracted a lot of interdisciplinary researchers to uncover one of the most fundamental challenges in science and technology: how does intelligence form? Within this domain, many AI researchers explore the cognitive capabilities such as perception, reasoning, planning, and learning with a long term goal of turning technical systems into systems that "know what they are doing." Technical systems with cognitive capabilities will be much easier to interact and cooperate with humans, and are expected to be more robust, flexible, and efficient when working in a dynamic environment such as human living space. Not so surprising, this in turn leads to the development of abstract concept of embodied mind.

Historically, the progress in AI fluctuates. In earlier decades, much of the efforts were dedicated to programming high-level reasoning tasks such as chess-playing, which surprisingly led to an unjustified optimism that all AI problems would be solved. AI sparked the demand of creating a machine exhibiting human-like behaviour or intelligence by media and entertainment business. In the 50s, many programs simulated intelligence using logic and high-level abstract symbols (hence it is called symbolic logic). However, this approach has many problems and researchers discovered in the 1970s and 80s that abstract symbolic reasoning was very inefficient and would never achieve human-levels of competence even on a simple task. In fact, many researchers began to doubt that high level symbolic reasoning could ever perform well enough to solve simple problems. Hence, the ultimate goal of having a machine with human-like intelligence, which is termed strong AI, unfortunately seemed to be out of reach by the symbolic approach.

In the domain of robotics, researchers argued that the "intelligence" carried on symbolic reasoning will definitely fail because this approach neglected the importance of sensorimotor skills for the development of the intelligence in general. The robotics researchers began to direct their attention to another approach such as statistical AI which achieved high levels of success in industry without using any symbolic reasoning, but instead using probabilistic techniques to make "guesses" and improve them incrementally. This process is similar to the way human beings are able to make fast, intuitive choices without stopping to reason symbolically.

In line with this "embodiment" approach, researchers also look into neural network approaches which are based on the actual structures within a human brain that are responsible for intelligence and learning. Many AI-origin robotic experts brought up the embodiment theory and have argued that a machine may need a human-like body to

enable it to think and react as a normal human being. In 1950, Alan Turing wrote [5]

> *It can also be maintained that it is best to provide the machine with the best sense organs that money can buy, and then teach it to understand and speak English. That process could follow the normal teaching of a child.* (Alan Turing, 1950)

One source of inspiration for the embodiment theory has been research in cognitive neuroscience. The idea of the embodied mind starts to grow and becomes one of the intense debates within the field. Neuroscientists discuss how both our neural and developmental embodiment shape our mental and linguistic skills. The degree of thought abstraction has been found to be associated with physical distance which then affects associated ideas and perception of risk [6]. They also explain the idea of embodied cognition in terms of dynamical systems theory which leads robotic experts to explore the domain of imitation learning [7][8].

The progress in computational and cognitive neuroscience also leads to the development of "artificial brain" (also called artificial mind). Research investigating artificial brains strives to understand how the human brain works by simulate the biological processes within to any degree of accuracy. This in turns leads to an attempt of emulating the whole brain using collaborative informatics infrastructure, including the interconnected standard PCs, mainframe computers and also neuromorphic hardware [9]. This effort is supported by thought experiment in the philosophy of artificial intelligence, demonstrating that it is possible, at least in theory, to create a machine that has all the capabilities of a human being.

In the lowest level of hardware, neuromorphic engineering, also known as neuromorphic computing, uses very-large-scale integration (VLSI) systems containing electronic analog circuits to mimic neuro-biological architectures present in the nervous system. It might also use nanotechnology [10][11]. Nowadays, the term neuromorphic has been used to describe analog, digital, and mixed-mode analog/digital VLSI alongside software systems that implement models of neural systems, including for perception, motor control, or multisensory integration [12]. The key insight of neuromorphic systems which inspire this thesis is how the morphology of individual neurons, circuits and overall architectures creates desirable computations. With these neuromorphic systems, researchers explore and try to mimic the neural computations that affect how information, which are usually noisy, are represented robustly, and how learning and development are incorporated. Chapter 4 of this thesis explains how our proposed method is implemented using the neuromorphic device called SpiNNaker.

1.1.2 The Emergence of Graphical Models

Many tools are used in AI, and a currently popular approach includes the statistical and probabilistic method. This method is favourable among others because many problems in AI require the capabilities of reasoning under uncertainty as well as incorporating the prior knowledge to update the overall knowledge representation. The probabilistic graphical models (PGMs) arise as a convenient tool not only for analysis purpose but also for

performing complex intelligent tasks. The new paradigm of probabilistic reasoning in intelligent system proposed by Judea Pearl (1988) become a hallmark of probabilistic method domination in AI on uncertain reasoning and expert systems [13]. Due to its close historical relationship with fundamental probabilistic methods, PGM have gained almost universal acceptance in a wide range of communities [14].

The PGM can be viewed as unification of graph theory and probability theory into a new formalism for multivariate statistical modeling [15]. This formalism provides a convenient way of integrating perception and action as well as learning and planning which are primarily required by almost all AI manifestation. Particularly, it provides the following useful properties [16] :

1. Easily visualize model structure which can inspire a new better model upon it
2. The independency between elements/variables can be easily recognized in the model structure
3. The inference and learning task can be expressed in terms of graphical operations

In modern control systems, engineers use State Space Modelling in time domain for developing a control system. One of the most advanced state variable feedback systems that can deal with uncertainty is the Robust Control System. However, this method relies on the exact mathematical model of the system. For a complex system, where the exact mathematical model is difficult to obtain, engineers use an intelligent approach which usually provides a heuristic and a low cost solution that leads to the development of Intelligent Control Systems [17]. Although many methods have been proposed and can be considered as intelligent approaches, many AI practitioners initially consider neural network and fuzzy system as the prominent constituents of the Intelligent Control System. However, with the emergence of Machine-Learning and Soft-Computing, many researchers have been starting to deploy PGMs for solving the optimization problem such as in motion control and planning area [18].

It turns out that the Dynamic Bayesian Network, one form of PGMs, is a general case of the well-known Kalman Filter, which is a powerful algorithm mainly utilized by control engineers for developing a state estimator for their state variable feedback control system. The other field of control, where probabilistic methods play important roles for planning and decision making, is the optimal control. In this field, the Markov decision process (MDP) in conjunction with DBN, can be used to extract the knowledge representation of the environment surrounding the robot agent [19] [20]. To our knowledge, this PGM is best suited for higher level control algorithm, although it offers a convenient method for multi-level integration, from low-level control to high-level rule-based planning in the domain of relational statistics [21]. Some researchers have also tried to implement and to compare the performance of such a probabilistic inference for controlling dynamic systems, and they found out that their probabilistic-inference-based controller matches the performance of the standard conventional controller [22] [23].

Many of the classical multivariate probabilistic systems studied in fields such as statistics, system engineering, information theory, pattern recognition, and statistical mechanics, are special cases of the general graphical model formalism. These include many popular

methods such as mixture models, factor analysis, hidden Markov models, Kalman filters and Ising models [24]. This framework has proved its successful outcomes in engineering applications, ranging from a native motor control to an adaptive reinforcement learning [25]. Even a new area in robotics, which is called probabilistic robotics and concerned with the perception and control in the face of uncertainty, is now growing and gaining more popularity [19].

In the field of signal processing and control theory, statistical and stochastic models have long been formulated in terms of graphs. Algorithms for computing basic statistical quantities, such as likelihoods and marginal probabilities, have often been expressed in terms of recursions operating on these graphs [15]. These models are composed of nodes and edges that are used to represent the organization and functionality of real systems, algorithms, etc. Viewed algorithmically, the nodes of the graph represent transformational processes, while the edges represent information paths between the processing nodes. The graph underlying the graphical method may be directed (such as a Bayesian or Belief network), or the graph may be undirected (which is generally known as the Markov random field). Directed graphs are useful for expressing causal relationships between random variables, whereas undirected graphs are better suited to express soft constraints between random variables [16].

There is an increasing trend in this decade to merge/combine both directed and undirected graphs into one unified formality. This unification offers more prospective treatments for applications, where the intrinsic problem in the applications cannot be solved solely by either directed or undirected graphs. For example in the field of robotics, where the robot needs to interact with the environment while executing some actions, it is very common to use a computer vision technique as a means of gaining information about the state of the world, and to use some form of Kalman filters to infer its own internal states for triggering the robot motion. In this scenario, the undirected graph is the right model for dealing with the vision processing, while the directed graph is the right one for modelling the robot motion through some paths or trajectories. This setting is an excellent example for combining the optimization algorithm with its graphical representation, leading to the advancement of probabilistic robotics [19][26]. This emerging field shows that the "perception-action" cycle of a robot life can be cast as a graph of relations between the involved state variables (i.e. the inferred data) and the observations (i.e. the evidence) [27]. In particular, this unified view has contributed to the success of the well-known algorithm called simultaneous localization and mapping (SLAM) [28][29][30].

In the framework of PGMs, the probabilistic inference methods are used to decompose the overall computation including the whole estimation process. Special case of this graphical model is known as the factor graph, which represents function's factorizations of several random variables [31]. Factor graphs support a general trend in the field of computational intelligence from sequential processing to iterative processing. An example of this trend is the factorial hidden Markov model, where the state space of a traditional hidden Markov model is split into the product of several state spaces [32]. In the field of robotics application, the Gaussian Markov random fields (GMRFs) have also been utilized in the form of factor graphs for solving non linear SLAM problems [30]. This particular type of graphical model will be described in detail in chapter two.

Basically, the inference method is used for solving a probabilistic problem, either in the form of marginal probability or maximum a posteriori (MAP) probability [14][33]. In many situations, marginalization and MAP computations are performed concurrently in order to get the most benefit of certain inference algorithm. One example of the basic algorithm for inference is known as the elimination algorithm. The algorithm complexity will grow exponentially as the structure of the graph becomes more complex and redundant (i.e. many common intermediate terms will exist). However, this algorithm has been successfully implemented and become the core of the framework called the GTSAM toolbox (GTSAM stands for "Georgia Tech Smoothing and Mapping"). It provides excellent solutions to the SLAM (Simultaneous Localization and Mapping) and SFM (Structure from Motion) problems, but can also be used to model and solve logical problems like CSP (Constraint Satisfaction Problem). GTSAM is based on the paradigm of viewing matrix factorization as the way for transforming a factor graph into a Bayesian network, which is a graphical model that uses the square root information matrix. The Bayesian network resulting from the elimination/factorization is chordal, and it is well known that a chordal Bayesian network can be converted into a tree-structured graphical model in which these operations can be performed conveniently. More about this GTSAM framework and its application can be found in [34][35][36][37][38][39].

The more efficient algorithm, known as belief propagation via message-passing method, treats the probabilities as messages propagating through the network. This algorithm is similar to the one usually used in neural networks, but with a different paradigm where, in the neural networks setting, each node generally has a single activation value that it passes to all of its neighbours. This message-passing algorithm can be used efficiently in factor graphs to compute certain characteristics of the function, such as the marginal distribution. The most popular form of the message-passing algorithm is known as the sum-product algorithm. A wide variety of algorithms developed in AI, signal processing, and digital communications can be derived as specific instances of the sum-product algorithm, including the forward/backward algorithm, the Viterbi algorithm, the iterative "turbo" decoding algorithm, Pearl's belief propagation algorithm for Bayesian networks, the Kalman filter, and a certain fast Fourier transform (FFT) algorithm [31]. It is interesting to note that despite the empirical success of the sum-product algorithm in many applications, the original algorithm is not guaranteed to converge. However, a unified framework for the sum-product algorithm and its "sister" the max-product algorithm has emerged which proves the convergence of message-passing algorithms and has showed the importance of enforcing consistency in both the sum-product and the max-product algorithms [40].

Some algorithms that belong to the message-passing category sometimes fall into a class referred as exact inference [41], [42]. Exact inference is usually used for simple cases, where the structure of the network is loopless or contains just a small number of loops. Another class, known as approximate inference, tries to maximize probability computation by approximating marginal distribution using stochastic simulation or sampling algorithms. Examples of this approximate inference approach are importance sampling algorithm and Markov Chain Monte Carlo (MCMC) algorithm. Another well-known approximate message-passing technique is the 'loopy' version of belief propagation, and in-

cludes more recent development such as generalized belief propagation [43] and expectation propagation [44]. It is also possible to use a variational method to enhance the learning capabilities of a Bayesian network, since it approximates computations in the model with latent variables using a lower bound on the marginal likelihood [45]. Unfortunately, this approximate inference method requires a bulky memory space and demands a lot of computational resources. Since we are going to explore and exploit a PGM for a specific low-cost robotics application, we strive to utilize the most effective inference method on restricted hardware. Hence, we focus on the exploration and system development that works best with exact inference in a discrete form of PGMs. More detailed descriptions as well as comparisons for exact and approximate algorithms are given in [46].

Working with discrete form is also inevitable choice when we use a digital machine, especially in low-level-embedded-system hardware. Every numerical computation will be subject to quantization because a number, especially the real-valued one, will be stored in a memory hardware with a limited number of bits. Consider the simple case here for representing a Gaussian distribution:

$$\mathcal{N}(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp(\frac{-(x-\mu)^2}{2\sigma^2})$$

In Matlab, to represent such distribution we have to specify the range and the resolution of x:

```
mu = 1.5;       %Set the mean value
x = -3:0.1:3;   %Set input values to be within [-3,3] with step interval 0.1
y = (1/sqrt(2*pi*var))*exp(-power(x-mu,2)/(2*var))   %Compute the distribution
plot(x,y)       %Plot the distribution
```

It is clear from the above simple script that the resolution (i.e. the step interval) of sampling in x will determine how "smooth" the distribution will be represented. As a consequence, the distribution with high "smoothness" will take longer to be processed if the computation is performed sequentially. However, with this sequential process, the memory consumption can be reduced. This trade-off between fidelity and granularity, usually expressed as a cost function, rises naturally almost in every digital machine involving discrete-event systems and there is no single optimal solution that can be applied for every situation [47][48]. In the end, it is the application designers that decide how to make a balance for those aspects to maintain the optimum cost while delivering the best performance of their system.

Regarding the discretization of a continuous value in a probabilistic computation, statisticians usually use one of two approaches: supervised (dynamic) and unsupervised (static) method [49][50]. The unsupervised method does not make use of class membership information while the supervised method uses the information content (entropy) of the continuous variable for the partitioning, and does not require a comparison with the other variables [51]. Examples of the unsupervised method, which are common in practice, are the equal-interval approach and the equal-frequency approach (both approaches use the binning mechanism). However, the method to implement those approaches varies. In gen-

eral, the discretization process has two goals. The first is to find a set of boundary points to partition the overall range of possible values into a small number of intervals that have a good class coherence, which is usually measured by an evaluation metric. The second is to minimize the number of intervals without a significant loss of class-attribute mutual dependence. The discretization process itself can be direct or incremental. Direct methods divide the range of k intervals simultaneously, while incremental methods begin with a simple discretization and pass through an improvement process until some criteria of optimality have been reached. Another dimension of a discretization method is the coverage of the process; it can be local or global discretization [52]. It can also be optimized for certain applications [53].

In this thesis, we explore a different approach for discretization. Our method is inspired by the idea of message encoding from a population of neurons in the central nervous system. Conceptually, this type of code can be used to encode continuous variables usually originating from a measurement in a physical system. In neuroscience, it is known that any individual neuron is too noisy to faithfully encode the variable using rate coding, but an entire population working in synchrony will ensure greater fidelity and precision. The precision of the population can scale exponentially with the number of neurons [54]. This principle in turn will be a good technique to approximate a continuous value with high resolution as a replacement for the sampling mechanism shown in the above Matlab script.

Many researchers have already proposed methods to improve the performance of a graphical model computation by harnessing parallelism in modern computers [55][56][57]. Also not so surprisingly, the trend of exploiting GPUs (Graphics Processing Units) for general purpose computing attracts many researchers to start deploying their graphical models on computer graphic cards. Indeed, with hundreds of processor cores on a single chip, modern GPUs can be programmed to apply the same numerical operations simultaneously to each element of large data arrays under a single program multiple data (SPMD) paradigm. As the same operations run simultaneously, a GPU computation can achieve extremely high arithmetic intensity, provided that the data are transferred smoothly to and from the host processor (CPU). Computational statistics and statistical inference tools have already begun exploring the benefit of such computing power [58][59]. Silberstein et al. first demonstrated the potential of a GPU computation that impacts the performance of Bayesian networks for statistical fitting tasks using a belief propagation approach [60]. Factor graphs have already been implemented in a GPU as well [61][62]. However, to our knowledge, no exploration has yet been made on factor graphs using any dedicated hardware.

One important aspect that is covered in this thesis is how to efficiently implement a PGM on a hardware with limited resources. Such hardware, which is the main target of embedded system applications, is widely used for low-level and frequently low-cost but efficient controlling machine in robotics application. It is also common to utilize the PGM in a complex computing machine and to our knowledge, it is still rare and uncommon to see the PGM action in an embedded system. In this thesis, we show how such a PGM in the form of factor graphs can be implemented in dedicated hardware and we demonstrate some applications from our methods.

1.2 Motivation and Contribution

Factor graph using belief propagation is not a new topic and some people have already explored this fascinating topic for many applications in specific fields, such as in communication and signal processing [63], operation research [64][65], and many more. In this thesis, we are specifically interested in how to empower a technical system using cognitive capabilities encapsulated in a factor graph framework that enable the system to percept, react, learn and plan in an intelligent way. Hence, we want our factor graph to work as close to the lowest level of the hardware system as possible, where the sensors as well as the actuators are integrated and embedded into a physical system.

While the current computer technology can outperform human capabilities in exact computation, the superiority of the human brain at computing uncertainty is still far beyond the power of computers with strict logical systems. Without a doubt, biologically realized cognitive intelligence is the most complex property of the human brain and can be perceived only by itself. The lack of the reasoning skill of a machine for uncertain behavior in the real world (i.e. noisy, incomplete, or inconsistent input data), leads to the development of an intelligent system that tries to synthesize intelligent behaviour similar to biological organisms. It should be noted that this ability to reason under uncertainty, which is also the main theme of this thesis, is only a small part of human-level intelligence. Other parts of human-level intelligence, such as natural language, self-awareness and consciousness, are beyond the scope of this thesis. Hence, it is likely that our machine is capable of reasoning under uncertainty even though it might not understand what it is doing [66]; something which is common and acceptable for AI researchers who realize that "strong AI" is a progressive work. However, one particular aspect of this intelligent system, which motivates this thesis, is its cognition ability: it tries to understand its environment, perceive it and adjust its own state based on its predefined goal. An examination of cognitive and biological models for human control of systems suggests that they exhibit a declarative, procedural, and reflexive hierarchy of functions. In this thesis, we propose biologically-inspired hardware solutions and strategies that can be extended for a broader class of probabilistic inference based on some principles of information processing in the human brain. Thus, the ultimate goal of this thesis is not to develop a human-level intelligent machine, but to build supportive hardware for more complex systems which can be used to mimic such high level intelligence.

SpiNNaker (Spiking Neural Network Architecture) system, one of the most recent neuromorphic devices available for novel research exploring the domain of embodied intelligence, is the first platform that will be used in this thesis to explore many aspects of low level data transmission in a distributed fashion. Our PC-based factor graph framework described in section 2.4 is our starting point for understanding the core computations required to implement effective factor graphs along with the belief propagation algorithm and messages encoding using population coding principle. To our knowledge, this is the first time the SpiNNaker system is being explored in the domain of probabilistic graphical models. Hence we emphasize the novelty of this research on this aspect.

Next, we propose to use a System on Chip (SoC) device as the continuation from our first effort on implementing factor graphs in embedded hardware. SoC is chosen because it offers integration of both software-based control and real time hardware-based processing.

We use the Zynq-7020 from Xilinx as our SoC platform, which is composed internally of two tightly coupled sub-systems: PS (processing system, i.e. microprocessor core) and PL (programmable logic, i.e. FPGA fabric). The PS sub-system consists of equivalently two ARM Cortex-9 processors and the PL sub-system is equivalent with FPGA Artix-7 from Xilinx. We can also take the advantage of the most important feature of FPGA in that it can be re-configured as many times as possible without changing the hardware structure of the system (low-risk high-impact approach). Comparing to the conventional/standard computer hardware based on von Neumann processor at the core, where the algorithm is executed sequentially (even with multithreading feature of multi-cores processor), SoCs offer much more flexibilities due to the fact that their FPGA resources can be structured and organized to mimic the true parallelism in a complex computations. For example, for calculating a factor graph iteratively in a standard computer program, it will consume quite amount of computer resources (i.e. task scheduler) which are very difficult to be maintained without sacrificing load sharing between processes in the CPU and it also introduces delays between threads which affects overall algorithm performance [67][68][69].

This sequential approach is worsened by the fact that it often performs the product operation on a set of values, for example in the use of the sum-product algorithm for implementing the message-passing algorithm. Multiplication, by default, is always an intense resource consuming process. By utilizing the FPGA of the SoC, we can distribute the computation into concurrent calculations effectively in every slice of the FPGA chip. This might be very useful, for examples when calculating Factorial Hidden Markov Models, where the state space of a traditional Hidden Markov Model is split into the product of several state spaces.

The novelty of this research is the exploration of the implementation of probabilistic inference using message-passing-based methods for factor graphs natively in low-level hardware. Such fundamental probabilistic inference hardware, which takes into account the uncertainty and randomness into its computation platform, will produce more powerful, flexible and efficient building blocks for more complex computational intelligence machine. Similar work has also been accomplished by Mansinghka, who created stochastic digital circuit using FPGA to build massively parallel, fault-tolerant machines for sampling, which allow one to efficiently run MCMC [70]. The main difference of this thesis with Mansinghka's work is that we implement the discretization for continuous variables using population coding principle developed by the computational neuroscience community. Furthermore, Manshinghka concentrates on the sampling algorithm and gives a little detail on the higher level of intelligence abstraction. In our method, we keep working in the discrete domain because working with propagation of continuous variable distribution may result in multidimensional integration which leads to intractable operation, especially for embedded systems with limited resources [71]. Furthermore, we address the challenge of implementing such probabilistic reasoning hardware to support the cognitive development of a real technical system (e.g. robotics application).

We are also aware that there is an attempt to implement belief propagation algorithm inspired by neural information processing which harnesses the biophysical properties of neuronal networks. Work by Andreas Steimer (in [72]) focuses on different levels of abstraction, ranging from abstract, spike-based principles that deal with the problem at

the level of neural coding, to more concrete approaches for implementing BP in neural substrates. Although his work is based on the similar belief propagation mechanism in our framework, it is different with our approach developed in this thesis, because Steimer implemented his belief propagation algorithm with message encoding using Liquid-State Machine (LSMs) and implemented it on Forney-style factor graphs. Our approach, on the other hand, uses belief propagation on ordinary but arbitrary factor graph with tuning-curve-based population coding. Furthermore, Steimer developed an abstract idea of hardware implementation called Interspike-Intervals-based processor; while we implement our factor graph in real hardware using two different platforms: SpiNNaker system and SoC (System-on-Chip).

Regarding the population coding principle for encoding messages in factor graph, this thesis offers a better understanding and implementation of such principle compared to the work of Dennis Göhlsdorf (in [73]). The comparison between Göhlsdorf's population coding and our proposed population coding is described in section 2.3. Furthermore, Göhlsdorf's work focuses on developing a factor graph framework which only runs on standard PCs. On the other hand, we develop our own PC-based factor graph framework and then extend it so that we can implement it in embedded hardware efficiently.

Finally, we introduce also the comparison with different hardware platforms, which might be useful for consideration in the future development of robotics system with probabilistic graphical models as the main framework for its cognitive intelligence engine. Throughout this thesis, we use the term "embedded factor graph" to refer to the implemented factor graph framework on a dedicated hardware.

In summary, our contributions are depicted in Fig. 1.1. The relation between technical systems and computational neuroscience give influential insight for the core element of our proposed method. We believe, the cognitive capabilities inspired by the neural processing mechanism in the brain will empower our future technical system (robotic systems in particular), while reciprocally giving valuable feedback for neuroscientists in their understanding of the physical impact of their theories about the neural system.

1.3 Organization

This thesis is organized as follows.

Ch.1 Introduces the motivational background with a brief overview on the state-of-the-art of the related research area, presents a brief overview of our proposed methods, and points to several related works.

Ch.2 Presents the modeling principle using a factor graph. It starts with the basic concept of probabilistic graphical models with focus on factor graphs historical background, and continues with the model building mechanism and the parameter learning. In this chapter, we describe our proposed method of using population codes for encoding messages in a factor graph. The chapter is closed with the description of our PC-based factor graph framework.

Ch.3 Demonstrates many applications of our factor graph framework. Two application domains are explored: machine learning and robotics. In this chapter, the extension

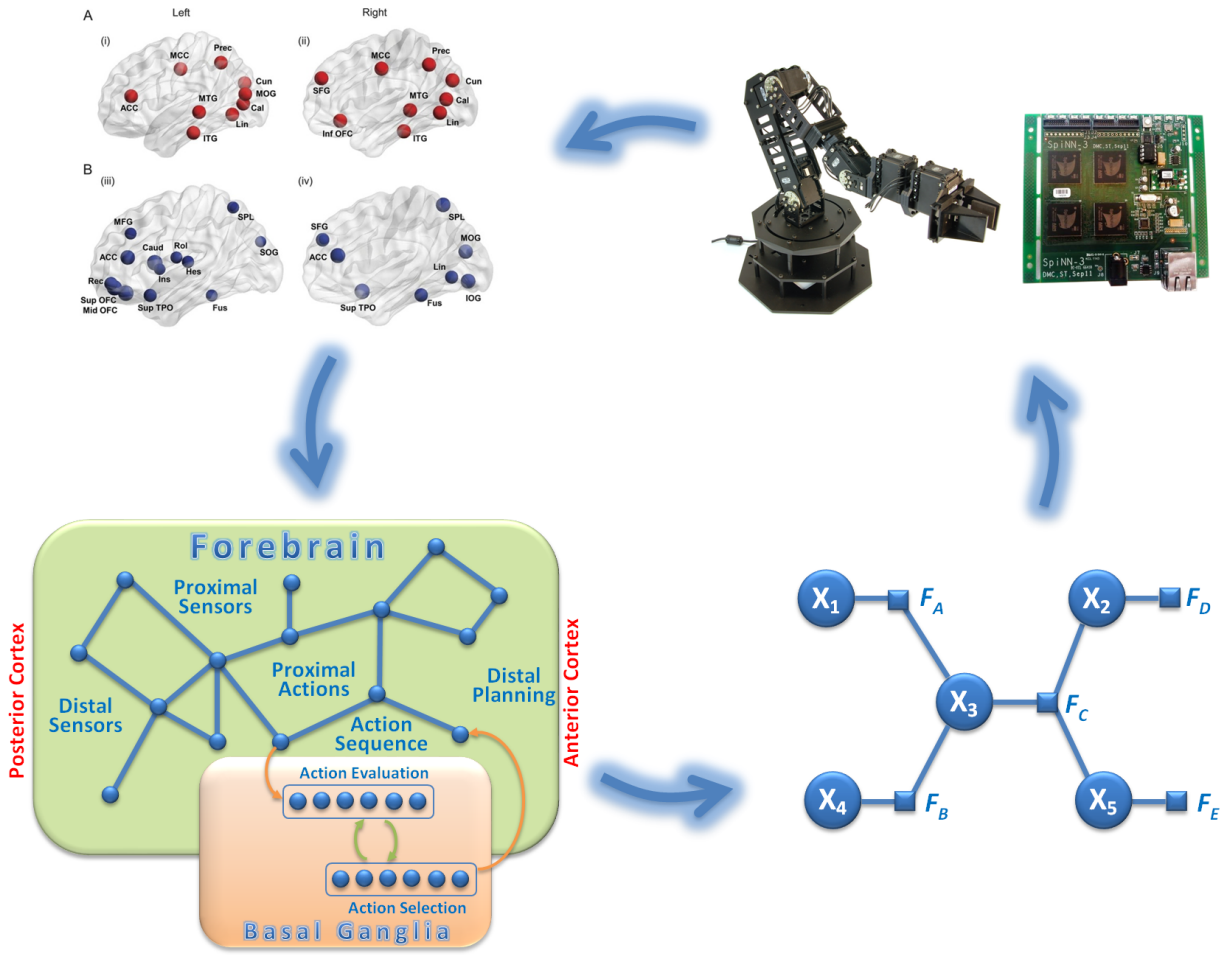


Fig. 1.1: Representation of the theme of this thesis as an interrelation between neuroscience, artificial intelligence (machine learning) and robotics. We learn from neuroscience, how the cognitive intelligence is emerged; we learn from artificial intelligence (machine learning), how we can model it; we then apply our model in robotics applications; finally we inform our colleagues in neuroscience about our experiments using technical systems (robots) which hopefully provide valuable feedback and insight for them into how the cognitive intelligence should be studied further.

of a static factor graph into a dynamic one in our factor graph framework is also presented.

Ch.4 Presents our implementation of factor graphs in a SpiNNaker system. This chapter demonstrates our proposed method for mapping factor graph's core elements into SpiNNaker's resources. It also presents the performance evaluation and our proposed optimization strategy.

Ch.5 Presents our implementation of factor graphs in Xilinx Zynq-7000 SoC. This chapter describes our two implementation methods: FPGA as an accelerator, and FPGA as a full factor graph engine. The chapter is closed with a thorough evaluation on our methods and also with our proposed improvement strategy.

Ch.6 Describes the overall evaluation and our vision on further applications of our factor graph framework.

Ch.7 Summarizes this thesis work.

2 Modeling in Factor Graphs

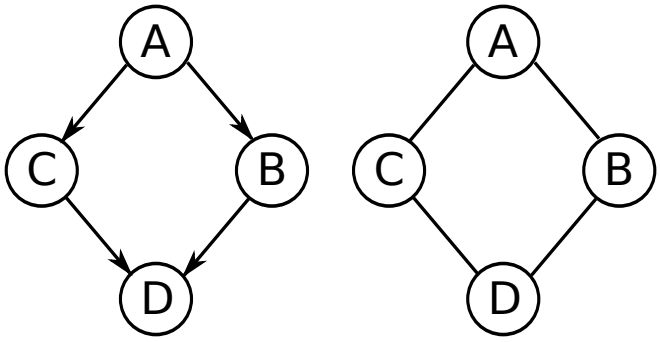
Factor graph is a graphical model which unifies the directed and undirected modes and provides a convenient way for performing inference in order to compute the marginal probabilities of variables involved in the graph. In this chapter, we describe the basic mechanism of belief propagation using message-passing algorithm and how our framework deals with several issues arising during the inference as well as learning the parameters of the network. Our contribution in this chapter lies on the improved parametrization of the network using the population coding principle (section 2.3). We also explore the combination of sum-product and max-sum algorithm to produce an alternative method for learning the parameters for a network with hidden variables (section 2.2.3). We present our framework in section 2.4 as a means for us to understand the core principle of belief propagation in factor graph along with all its challenges, where we gain insights for later implementation in dedicated hardware.

2.1 Probabilistic Graphical Models

Probabilistic graphical modeling (PGM) is a powerful method in artificial intelligence and machine learning which emerges from the combination of graph theory and probability theory. It combines graph theory and probability theory in such a way that it can represent a complete probabilistic distribution over a multi-dimensional space in a factorized representation using a graph. This factorization is the successful key of this method since it provides the modularity mechanism where a complex system can be built by combining its simpler parts while ensuring the consistencies during the process. Its graphical representation makes it an appealing visualization which also provides an intuitive way to interface models to data. These fascinating properties of PGM attracts many researchers and engineers to study and apply PGM to broader applications. After a decade of exploration in many studies, it turn out that many of the classical artificial intelligence and machine learning algorithms in the flavour of multivariate probabalistic/stochastic systems are special cases of PGM [1]. Many well-known frameworks such as mixture models, factor analyses, hidden Markov models (HMM), Kalman filters and Ising models can be represented generally using PGM formalism [74][75]. Another advantage of PGM is that, a specialized technique that has been developed in one field can be transferred between research communities and exploited to gain more applicabilities. This in turn provides a natural framework for the design of new systems [16][14][24][76].

In general, probabilistic graphical models can be grouped into two main classes. The first are so-called directed acyclic graphical models (DAG), which are also popularly known as Bayesian Network; and the second are called undirected graphical models, also commonly referred to as Markov Random Fields (or simply Markov network). Historically, undirected graphical models are more popular by the physics and vision communities

[77][78][79][80][43][81][82], and directed models are more popular with the artificial intelligence and computational method society [83][84][85][13][86]. People who work with DAG are mainly interested in the causality relation between variables; hence, the parameters of the network are more interpretable and computationally easier to be estimated [87][88]. On the other hand, people are interested in the undirected graphical model due to its symmetrical characteristic such that it is more "natural" to represent spatial or relational data, which are commonly found in computer vision applications. Both families encompass the properties of factorization and independences, but they differ in the set of independences they can encode and the factorization of the distribution that they induce. This difference is reflected on the "direction-ness" in the model: the direction in the Bayesian Network reflects the conditional relation between variables tied by the directed-link between them, while the non-direction in the Markov Network simply reflects the potential of maximal cliques within the graph, where the ordering of variables is not important (hence it requires no direction on the link). The same difference, which is exemplified in Fig. 2.1, will have a big impact and lead into different treatment when building a model and determining the parameters of that model. For example, in a Markov network, two sets of variable nodes A and B are conditionally independent given a third set, C, if all paths (in any direction) between A and B are separated by a node in C. By contrast, in Bayesian networks, this dependency will not hold for all directions.



(a) Bayesian network **(b)** Markov network

Fig. 2.1: An example showing the different perspective of Bayesian and Markov network. Although both models have the similar structure, the treatment of probability distribution between variables might be different. In (a), the probability distribution factorizes into $p(A, B, C, D) = p(D \mid B, C) \cdot p(C \mid A) \cdot p(B \mid A) \cdot p(A)$. While in (b), the probability distribution factorizes into $p(A, B, C, D) = \frac{1}{Z}\phi_1(A, B) \cdot \phi_2(B, C) \cdot \phi_3(C, D) \cdot \phi_4(A, D)$. Hence, both models encode different independencies of their variables.

In addition to these two main graphical models, there exist several other models which are particularly useful when bringing those two models into a specific application, such as clique tree (or junction tree), chain graph, conditional random field, restricted Boltzmann machine, etc. Some of these additional models try to exploit both directed and undirected model characteristics in order to find the relationship between those two models. One another model which tries to unify both class PGM while preserving the original advantage of each class is so-called factor graph. This factor graph model, which is also the main theme

of this thesis, provides a unification approach such that we can make further analyses while the network preserves more information about the form of the distribution than either the Bayesian network or the Markov network can do alone [89]. In its original form, the factor graph is an undirected graphical model but in its inference, it can also behave the same way as a directed model depending on the underlying Bayesian network. For a factor graph which has underlying functionality originating from one of those Bayesian networks, it is straightforward to use the standard exact inference such as the belief propagation. However, working with a factor graph which has a cyclic topology (such as those that resulting from Markov Network transformation) will be very challenging and requires special treatment/tuning in order to avoid oscillation in the network [90][91][92] [93][94][95][96]. In the next section, a more in-depth explanation of how a factor graph works will be given.

2.2 Inference and Learning in Factor Graphs

2.2.1 Building a Factor Graph Model

Although the origin of factor graphs lie in coding theory, it has similar instantiations used in machine learning community [31]. It is natural to treat a factor graph (with extension) to express a Bayesian network or Markov network [89].

A factor graph has two important properties: network parameters and network structure. Although learning a network structure is very interesting, it is very challenging and out of the scope of this thesis. The work from Abbeel et.al. shows that learning network structure from data which is not generated by its own distribution class will degrade the accuracy and efficiency [97]. Instead of learning the network structure, in this thesis we assume that the structure of the network will be given (or can be inferred directly) as a part of the task's description in the application.

In general, a factor in a factor graph is a function over some inter-related variables. A factor is nonnegative if all its entries are nonnegative. The variables which become the argument of the function are called the "scope" of the factor. Since a factor is basically a function, it can also be used to represent any logical predicate. This means, in a binary-valued factor, which has a value 1 for certain variable assignments and 0 otherwise, the factor can represent any logical relationship among its scope variable. Extending the concept of a factor in the probabilistic perspective, a factor can also capture the statistical relationship between random variables. In a Bayesian network, factors in the graph represent joint probability and/or conditional probability of interconnecting variables. In a Markov network perspective, however, factors in the graph represent potentials of the corresponding cliques. For example, given a model with five variables (A, B, C, D and E) and the following relation:

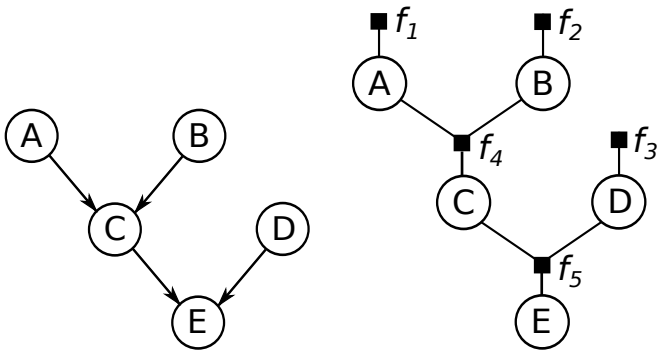
$$p(A, B, C, D, E) = p(E|C, D) \cdot p(C|A, B) \cdot p(A) \cdot p(B) \cdot p(C) \tag{2.1}$$

A Bayesian network expresses this conditional probability by linking all child variables to their parent using a parent-to-child arrow. For the given relation, the resulting graph is shown in Fig. 2.2a. By definition, a factor graph is a bipartite graph that expresses the structure of factorization, which is commonly expressed in a 2-tuple $\mathbf{G} = (\mathbf{X}, \mathbf{F})$. In a

factor graph, expression (2.1) is factorized as:

$$p(A, B, C, D) = f_1(A) \cdot f_2(B) \cdot f_3(D) \cdot f_4(ABC) \cdot f_5(CDE) \tag{2.2}$$

and the resulting factor graph is shown in Fig. 2.2b.



(a) Bayesian network representation **(b)** Factor graph representation

Fig. 2.2: Constructing a factor graph as a structured factorization of the conditional probability.

The structural representation of the factor graph shown in Fig. 2.2a is called the standard/ordinary factor graph. There is, however, another representation called Forney-style factor graph (FFG) which is quite popular in the field of communication and signal processing. This different notation for factor graph is briefly introduced in Appendix-A. In this thesis, we use the standard factor graph representation for the following reasons:

1. The standard factor graph has a structural representation which is similar to two other important graphical models: Bayesian network (BN) and Markov Random Field (MRF). Hence, people who are already familiar with either BN or MRF will not encounter difficulties in understanding the main idea presented in this thesis.

2. In a FFG each variable may only be connected to two factor nodes at most. This strong restriction reduces the simplicity of the graphical model representation; hence, the resulting FFG structure will grow enormous.

3. In a FFG, there exist several additional specific types of factor nodes such as elementary nodes as well as composite blocks, which require additional treatment in the inference process beyond message-passing itself. Hence, it potentially increases the latency in the hardware implementation of a factor graph in generic applications beyond the field of communication and signal processing.

Fig. 2.2 also shows that the factor graph preserves the structural information from its underlying Bayesian network. Since the Bayesian network is an acyclic network, the resulting factor graph also has an acyclic structure, which is favourable especially when dealing with exact inference.

However, when transforming a Markov network into a factor graph, it might yield a cyclic structure. The cyclic structure usually happens when we first transform a Bayesian

network into a Markov network through a moralization procedure before finally transforming it into a factor graph. For example, we can transform the Bayesian network shown in Fig. 2.2a into a Markov network and later into a factor graph as follows.

First the original directed graph is transformed into an undirected graph through a moralization procedure. In this step, all parent nodes of a child node will be connected together, which usually produces a cycle. Next, the decision must be made to either create a factor node for each edge connecting two variables, or to create a factor node only for maximal cliques. If we choose to create a factor node for each edge, the factor graph may end up cyclic. But if we choose to create a factor node only on the maximal clique, cycles may be eliminated resulting in a cycle-free factor graph. Both the resulting factor graphs have their own advantages and disadvantages. For example, in a cycle-free factor graph, we do not need to worry about the loop problem and we can run the inference algorithm in a convenience way which is favourable for an exact inference. However, the size of the factor might be big and the computation on that factor will create a bottleneck for the overall computation (i.e. it will become a subject to a phenomena similar to the **curse of dimensionality** problem). On the other hand, if we choose the factor graph with cycles then we will have a problem with loopy messages which are very difficult to manage. Yet, the size of the factor node is relatively small, and the computation on that factor node will be very fast[1]. Another interesting feature of this single factor node for each edge is that it will satisfy the pairwise Markov property directly, which resembles the classical but powerful pairwise graphical model in machine learning called the restricted Boltzmann machine [98][99][100]. We will cover this loopy phenomenon in more detail in the next section. The result of a model transformation is shown in Fig. 2.3.

A factor graph has a variable node for each random variable of the system being modelled, and a factor node for each local function which takes variable nodes as its argument. Edges only exist between a subset of variable nodes; factor nodes are never connected to other factor nodes, and variable nodes are never connected to other variable nodes. A factor node F_j is connected to variable nodes X_i if and only if X_i are arguments of F_j. Given a factor graph $\mathbf{G} = (\mathbf{X}, \mathbf{F})$, the joint probability of all variables is a product of all factorization by factor nodes in the graph:

$$p(\mathbf{X}) = \frac{1}{Z} \prod_j f_j(\mathbf{X}_i) \tag{2.3}$$

Working with factor graphs to solve a probabilistic inference problem means that variable and factor nodes in the graph also have probabilistic representations: this is how the network will be parameterized. In general, we can represent a value by: a categorical (ordinal) or a numeric representation. The categorical representation is useful when dealing with categorical variables such as blood type, weather condition, etc. The numerical representation is useful when dealing with applications that include real numbers such as someone's weight, student grade, or temperature measurement. For applications

[1]In literatures, the number of scope variables of a factor determines the "order" of that factor. A factor with only two scope variables, such as the one which is created by assigning a factor for each edge in the graph, is commonly referred to as a **binary factor**. Likewise, a factor with a single scope variable is commonly referred to as a **unary factor**, which plays an important role as an input/output point of the system being modelled by the factor graph.

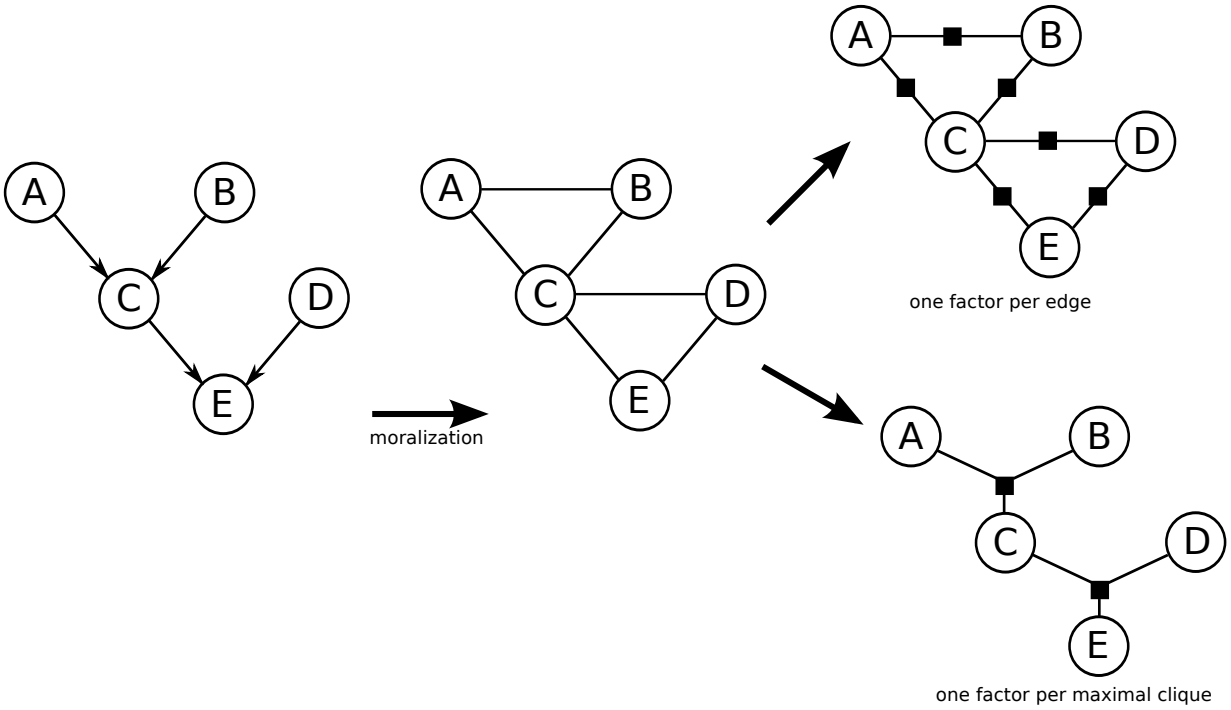


Fig. 2.3: Transforming a Bayesian network first into a Markov network and then into a factor graph. The final transformation produces two different factor graphs: the first one is with cycles and the second one is without any cycle.

in technical system, generally the network will involve numerical representation. Usually, the numerical variables take continuous values, and it will be very challenging to propagate them in a network because of multimodality and multidimensional integration [71]. Although working with continuous variables seems to be more natural, but at the end, it still needs to employ some sampling steps for approximating the integral [101]. Hence, we need to find a better discretization strategy for those continuous values and treat the network as a network with completely discrete parameters.

Also when working with discrete parameters, one must consider the numeric encoding of data since it can have a significant effect on the performance. In this thesis, the term discrete parameter reflects the fact that the variable nodes in the factor graph are discrete ones. By definition, a discrete random variable (RV) X is a measurable function $X : \Omega \to S$ from the finite/countable sample space Ω to another measurable finite space S called the state space. Each state has a deterministic value which represents a probability of a certain event that occurs randomly. Subsequently, parameter learning simply means how to update the states representation of a factor in a factor graph. In general, learning the parameter of a factor is basically a task of approximating the probability mass function. For a factor graph without any cycle, one can use an algorithm that exploits dependence trees to approximate the parameters [102]; however, such an algorithm cannot be used for general graphs. Instead, we will use a more generic approach called the expectation maximization for learning the parameter of a network.

Once we have done with defining the factor graph structure and network parameters, we can continue with performing the inference. If the structure of the factor graph does not

contain any loop then we can use an exact inference algorithm conveniently. One popular method of exact inference is the belief propagation mechanism. In references, the Bayesian network that uses this mechanism is known as Belief Network [83][103]. There is a similar mechanism in factor graphs popularly known as the sum-product algorithm [31], which relies on message-passing among nodes in the graph. A belief is the marginal distribution of a node, and a message is a representation of such distribution which is interchanged between corresponding nodes. Section 2.2.2 explains in detail how the belief propagation works and how we implement this algorithm in our system.

Given a factor graph, the next task before we can do any inference on it is to determine all nodes parameters. However, learning the parameters is crucial since it involves the decision of how to discretize RV's values (such as data from a robot sensor which only takes discrete range between [-500, 500]). Working in direct probabilistic computation means that one needs to provide an array with the number of elements as many as 1000 elements to cover all possible values in that range. This number of elements (or states), which is termed "cardinality" in PGMs, will increase exponentially as the number of RVs involved in the structure is also increased. Our discretization strategy to reduce the number of states needed to represent a real-valued number is based on the positional coding principle in population coding theory (see section 2.3). In the population coding theory, a collection of neighboring neurons in the brain, which might has similar characteristics, will react in synchrony after the stimulus [104]. The combined activation levels of those neurons resemble the probability density function of a certain multinomial distribution.

2.2.2 Inference Through Belief Propagation

This subsection describes the basic principle of belief propagation. First we describe the belief propagation mechanism on a network without any loop and then extend/modify the same algorithm for more generic networks which might contains a loop.

Belief Propagation on Factor Graph without Loop

In general, belief propagation is a class of inference algorithm commonly used in probabilistic graphical models, such as Bayesian networks and Markov random fields. It works on the factorized joint probability distribution by passing messages along the edges of the graph according to a set of message-passing rules which exploits the graphical structure of the network. The algorithm was first proposed by Judea Pearl in 1982 [105][83], who formulated this algorithm on trees and was later extended for general graphs. During the inference, each message will be updated consecutively from the previous value of the neighbouring messages. The "neighbouring messages" means all messages that come from all connected neighbour except the one to which the output message will be computed. Updating the message requires different scheduling. In the case where the network is a tree, the belief propagation algorithm will reach convergence after computing each messages only once; hence an optimal scheduling is always guaranteed. This is also valid for a chain graph which involves forward and backward phases.

Two types of messages are transmitted within the factor graph: the message sent by a variable node to a factor node (denoted as $\mu_{x \to f}(x)$ and the message sent by a factor node

to a variable node (denoted as $\mu_{f\to x}(x)$). These messages are computed according to the following equation.

$$\mu_{x\to f}(x) = \prod_{h\in n(x)\setminus\{f\}} \mu_{h\to x}(x) \tag{2.4}$$

$$\mu_{f\to x}(x) = \sum_{\sim\{x\}} \left(f(X) \prod_{y\in n(f)\setminus\{x\}} \mu_{y\to f}(y)) \right) \tag{2.5}$$

where $X = n(f)$ is the set of arguments of the function f and $\sim\{x\}$ is the "not-sum" or *summary* indicating the variables being summed over.

The belief propagation running on an acyclic factor graph usually has two phases: from leaves to root and then from root to leaves. We can think of these two steps as a similar process to the forward-backward procedure of the Baum-Welch algorithm used in hidden Markov model (HMM) [106][107]. A leaf node is a node which is connected only to a single neighbour node. For a single-inference mechanism, an arbitrary node is selected as the root in the beginning. However, for multiple-inference mechanism, the root selection is unnecessary. The message-passing is started on every leaf by computing and propagating messages to its neighbours. The message propagation continues until all nodes receive the messages in both directions. Fig. 2.4 illustrates this basic principle of belief propagation.

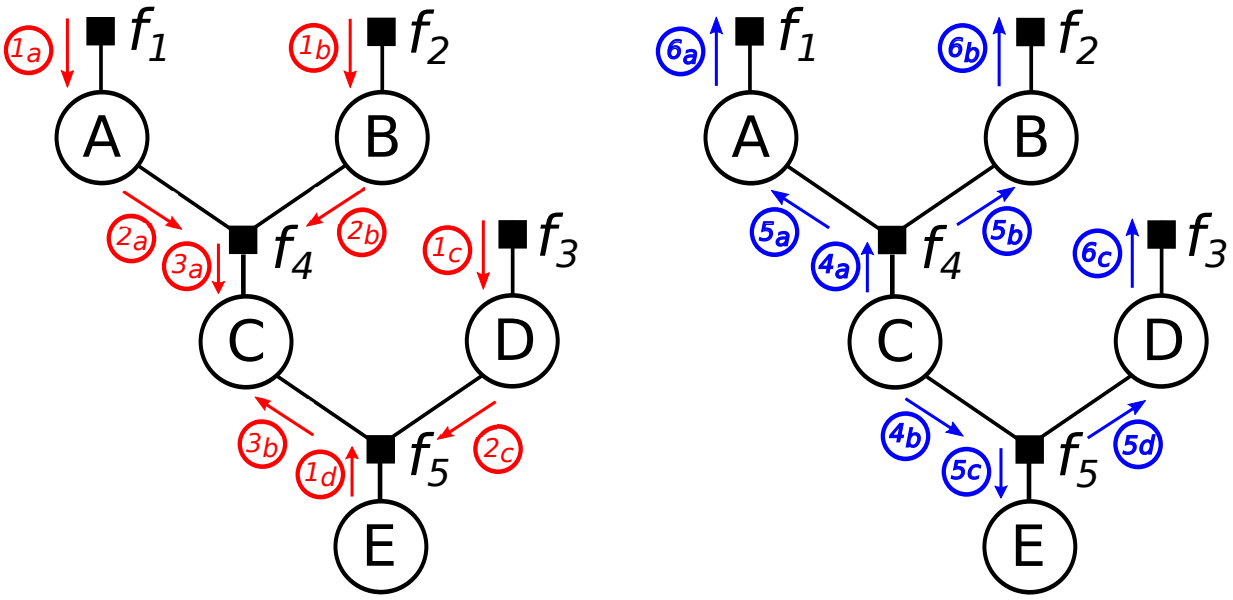


Fig. 2.4: Illustration of belief propagation using message-passing on a factor graph. Each circled number represents the iteration step. The red-coloured arrows and their corresponding numbers indicate the "forward" phases where the messages propagate in the "network-entering" direction, while the blue-coloured arrows and their corresponding numbers indicate the "backward" phases where the messages propagate in the "network-exiting" direction.

This belief propagation can be interpreted intuitively as follows. The belief is all the information currently available about a variable or a factor, and this information is the product of the initial information (prior belief) and the information passed on from the

messages. This is analogues to the standard Baye's rule where the messages operate as likelihoods that will be multiplied with a prior to get the posterior. Belief propagation algorithm ensures that a node will update its belief after receiving messages from all neighbouring nodes:

$$b(X_i) = \prod_{ne(X_i)} \mu_{f(X)\to(Xi)}(Xi) \tag{2.6}$$

$$b_F(X_F) = f_F(X_F) \prod_{ne(F)} \mu_{X_i\to f(X)}(Xi) \tag{2.7}$$

The consistency of the belief propagation is established when the propagated messages to the variable agree in terms of the marginal of the factor over the corresponding variable:

$$b_F(X_i) = b_i(X_i)$$

In the termination step, the posterior probability of a variable $p(x_i)$ can be computed as the product of all messages directed toward x_i. This means, since the message passed on any given edge is equal to the product of all but one of these messages, $p(x_i)$ can be computed as the product of the two messages that were passed (in opposite direction) over any single edge incident on x_i.

Belief Propagation on Factor Graph with Loop

The same belief propagation mechanism can also be applied to graphs with cycles and commonly referred to as loopy belief propagation (LBP). To our knowledge, however, there is no proof that LBP can be used generally (in any case). It is shown that LBP can produce a good result for some cases, but sometimes it does not even converge. Theoretical understanding of how actually LBP can converge in a network with a single loop is given by [93] and [108]. Analysis on how the behaviours of the LBP algorithm are affected by the discrete geometry of the factor graph, and the relation between the LBP, the Bethe free energy and the graph zeta function, are given by [109]. Further analysis on a network with arbitrary topologies and a Gaussian joint distribution shows that the belief propagation will converge to the correct marginals [110]. Based on empirical study in [90], if the algorithm converges then the results are usually close to the true marginal. If the algorithm oscillates, the correct marginal values seem to lay in the interval defined by the oscillation. This could not be shown in all cases. Furthermore, it is not the structure of the network but the parameter values of the network itself that determines if the convergence will be achieved. The choice of the initial messages, however, does not have any impact on the convergence of the algorithm.

Regarding the scheduling for updating messages, when the factor graph has cycles, an optimal scheduling does not exist, and the proper scheduling strategy must be found based on the characteristic of the application itself[90]. Basically, the scheduling for cyclic factor graphs can be done synchronously or asynchronously. In the synchronous mode (also called flooding schedule), all messages are updated in parallel, while in the asynchronous mode (also called serial schedule), an update is only applied to one message at a time. There is

no consensus on how to apply those scheduling on a cyclic graph; however, for a graph with a grid structure (e.g. in computer vision application such as stereo on pixel lattice), people often sweep in an "up-down-left-right fashion" [111][112]. Unfortunately, choosing a good schedule requires some experimentation in most applications. Elidan et.al analyse in [94] of how to best schedule the messages. Experiences lead to the conjecture that asynchronous message-passing works better for loopy networks although many previous convergence analyses have mostly been done in synchronous/parallel message updating. They assume that every message will at least be updated once within a finite interval of time, which is also a similar condition for convergence with the parallel update version. They show that asynchronous message-passing converges at least as fast as parallel updating. To improve the convergence rate, they propose a message scheduling scheme called "Residual Belief Propagation". A residual in this scheme is defined as the difference between a message and its update. The main idea of this approach is to update only the message with the largest residual, i.e. the message whose update will have the biggest effect on the network. Experiments show that Residual Belief Propagation converges faster and more often than other approaches.

In a cyclic factor graph, the initialization and message scheduling of the algorithm are defined differently than for the case with an acyclic factor graph. A cyclic factor graph illustrated in Fig. 2.5 shows that neither variable node z nor factor node f_A are able to compute an outgoing message as there are not enough incoming messages. Therefore, instead of starting the propagation chain by sending messages from the leaves, the message on each link in every direction is initialized with an initial value. Then each node can start computing outgoing messages using its own local information and the initialized messages. The freshly computed outgoing message on a link will then replace the old one.

It can be seen in Fig. 2.5 that the messages may be passed around the loop endlessly. By analysing the maximal difference between a new message update and the previous message at some iteration i, one can define a threshold for this difference below which the algorithm can be considered converged. The message values often converge after some iterations. However, it might occur in some cases that the message values oscillate and never converge. In the case of convergence, the result does not correspond to the exact marginal. Nevertheless, some experiments have shown that the approximation is often very good [90].

There are two kinds of error that occur when applying a standard belief propagation to a factor graph with loops. The first is the cycling error which arises due to the fact that messages are being passed around in the loop. It happens when a node computes an outgoing message based on the incoming messages, assuming they contain new information. However, some of the incoming messages will contain the same information they have accumulated during the previous iteration within the loop. The second type of error is called convergence error, which arises especially in the case where the factor graph originates from a Bayesian network. It happens when a child node, which is a part of the loop, receives messages from its parents. When computing the standard outgoing message, it assumes that those messages are stochastically independent. Since the child node and its parents are part of the loop, they are in fact connected by another path, and are therefore not independent [92].

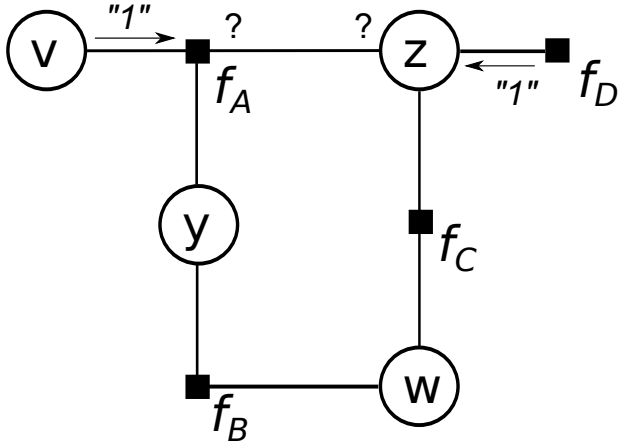


Fig. 2.5: Example of a cyclic factor graph. Factor node f_A and variable node z cannot compute outgoing messages because there are not enough incoming messages from the other neighbouring nodes. Starting from the node v (by sending a value "1" that means it is observed), the message goes to node f_A and waits for other messages (either from node y or node z) to appear. However, node y cannot send a message while it does not receive any message from node f_B. The same also happens with node z. Let's assume that node f_D sends an uninformative message (denoted by a uniform distribution "1") to node z, that message will also get stuck at node z because node z has not received any message from the other neighbors other than from node f_B. Therefore, it is intuitive to solve this problem by assuming initial messages at every links to avoid the deadlock. This assumption, later on, will require a specific mechanism of scheduling. Otherwise, the network will easily get trapped in an oscillatory state.

Our factor graph framework uses the same belief propagation and can be extended easily to perform the loopy belief propagation. However, we do not put to much attention for solving the convergence problems of running loopy belief propagation on an acyclic factor graph. In section 3.1.3 on page 52 we give a simple example to demonstrate that our framework is capable of handling a factor graph with a single loop. It can be inferred from [93][94] that a single loop factor graph usually does not have many problems with the cycling error; hence, it is easier to converge. This explains why our loopy factor graph in section 3.1.3 can produce convergence result. We also facilitate this loopy belief propagation in our embedded factor graph version using a SoC by providing a dedicated module for managing the scheduling of the message-passing algorithm (see chapter 5).

2.2.3 Parameter Learning

Working with the probabilistic graphical models is usually preceded by the selection of what kind of parameters that will be used for the model. In statistical terminology, there are basically two different types of model: parametric and non-parametric model. In parametric model setting, it is assumed that the model has a fixed number of parameters. In non-parametric model, the number of parameters is not fixed and might grow with the amount of training data. The parametric model has an advantage such that it is simpler to analyse and sometimes also faster but it also has the main drawback as it

usually uses a strong assumption about the nature of the data distribution which can lead to wrong decision. The non-parametric model is usually more flexible but often requires more computation resources. It does not mean that non-parametric model should be avoided when trying to implement it in low-level hardware because it is still possible to use the infinite Gaussian mixture model which is particularly well-suited for resource-limited hardware [113][114]. It turns out that our proposed method, at some extent, mimic this characteristic as well. However, in this thesis we prefer to call our work as parametric models. As described in chapter 1, we always have to bring any numerical value into digital data than can be processed by digital hardware in discrete processing. Hence we have to discretize the fix-size parameter of our model into discrete values. In this setting actually we mix between the parametric and non-parametric in the sense that we use a number of parameters which describes our model but we limit the size of those parameters and also we assume that those parameters actually come from a family of probabilistic distribution (e.g. Gaussian, Beta, etc.). Later we will describe how to implement such discretization of parameters using population coding principle.

Regarding the parameter learning scenario, there are two conditions that need different approaches. First, if the datasets from which the parameters can be learnt are complete (i.e. there are neither missing values nor hidden variables), then the standard maximum a priori (MAP) query can be used. In this thesis, instead of using the full MAP computation, we use maximum likelihood estimation (MLE) for estimating the parameters. Second, if the datasets have missing values or hidden variables, then we need to apply an iterative conditional modes (ICM) based approach. There are exists several algorithms for this second case such as Gradient Ascent and Expectation Maximization (EM), but we prefer to use the EM approach since it is more generic and appropriate in graphical modelling framework.

Maximum Likelihood Estimation

In common setting, one uses learning algorithm for estimating network parameters from observed data and uses inference algorithm to make prediction about data and then perform reasoning based on the result of prediction. There is a close relation between inference and learning in probabilistic graphical model. It turns out that in MLE setting we can use the standard belief propagation also for performing the learning task. Later in this section as well, we will show that in the Expectation Maximization (EM) algorithm, the result of inference step for calculating the expectation value has crucial effect in the next step for maximizing the expected log-likelihood [76] [14].

If we work with fully observed data and the structure of the network is given, then it is sufficient to use Maximum Likelihood Estimation (MLE). In fact, MLE is one of preferred methods for parameter estimation in statistics, particularly in nonlinear modeling with non-normal data [115]. It is also closely related with the concept of information bottleneck in the information-theoretic formulation for clustering problems which tries to construct a new variable T given the joint probability $p(x, y)$ that defines partitions over the values of X that are informative about Y [116].

We start exploring MLE by considering Bayes' rule formula:

$$p(\theta|D) = \frac{p(D|\theta) \cdot p(\theta)}{p(D)} \tag{2.8}$$

The denominator *p(D)* does not do anything with parameter θ and it just reflects the normalization of distribution. For observed data D and parameter $\theta, p(D|\theta)$ is the likelihood of the data D generated by the model with parameter $\theta, p(\theta)$ is the prior knowledge about the underlying parameter θ, and $p(\theta|D)$ is the posterior. Thus, the learning process is to maximize the posterior, which is called maximum a posteriori (MAP):

$$\theta^{MAP} = arg\ max_\theta\ p(\theta|D) \tag{2.9}$$

In one iteration only learning process, it is safe to assume that we know nothing about the prior and we make the prior to be uniform. In this setting, we will get the maximum likelihood:

$$\theta^{ML} = arg\ max_\theta\ p(\theta|D) \tag{2.10}$$

It is also important to assume that there is no dependency between observations:

$$p(d^1, ..., d^N|\theta) = \prod_{n=1}^{N} p(d^N|\theta) = L(\theta : D) \tag{2.11}$$

Equation (2.11) is called likelihood function and we want to maximize it. Using empirical distribution (with Kronecker-delta function) for discrete variable with k number of states, (2.11) becomes:

$$L(\theta : D) = \prod_{i=1}^{k} \theta_k^{N_k} \tag{2.12}$$

In our population coding representation, N_k is the number of neuron-k^{th} fires while the other neurons keep silent.

It is more convenient if we maximize the log of (2.12) and call it log-likelihood function $l(\theta : D)$. When we maximize $l(\theta : D)$, we also need to apply the sum-to-one constraint to θ. For discrete variable with k number of states, the constraint is:

$$p(\Theta) = \sum_{i=1}^{k} \theta_i = 1 \tag{2.13}$$

Adding this constraint to (2.11) will result in Lagrangian objective function:

$$l(\theta : D) = \sum_k N_k\ log\ \theta_k + \lambda(1 - \sum_k \theta_k) \tag{2.14}$$

with Lagrange multiplier λ. Maximizing (2.14) by taking its derivatives with respect to θ_k

to zero yields:

$$\frac{\partial l}{\partial \theta_k} = \frac{N_k}{\theta_k} - \lambda = 0 \tag{2.15}$$

So that $N_k = \lambda\, \theta_k$
Solving λ by using (2.13) yields:

$$\lambda = \sum_k N_k \tag{2.16}$$

And thus:

$$\theta_k = \frac{N_k}{\sum_k N_k} \tag{2.17}$$

Expression (2.17) basically is the probability of each value of θ corresponds to its frequency in the training data (i.e. the firing rate of neuron k^{th} in the population). Expression (2.17) can be adapted in a generic setting where parameter representations such as our proposed population coding needs to be learned from data before it can be utilized in factor graphs. In the belief propagation setting, we simply summarize every neuron response using sum-product formula. It means that for all samples in the dataset, we summing up the probability of each state as neuron's firing rate. This is a generalization procedure of empirical distribution with Kronecker-delta function; where for each sample, only one state has probability 1.0 and 0.0 for the rests. To train our network using (2.17) in order to get the factor value θ, we use the following algorithm.

Algorithm 1 Estimate factor parameter θ

```
Θ ← uniformly distribute
for all sample in X do
  for i = 0 to k do
    compute product(θ_k)
    Θ ← ∑ Θ_k
  end for
end for
return Θ
```

Expectation Maximization

In the situation where the datasets are incomplete (i.e. there are missing values) or contain hidden variables, one must use an iterative approach such as EM algorithm. Sometimes, the hidden variables emerge when we try to reduce the complexity of a network. For example, if we have a four variables network, then the joint probability $p(A, B, C, D)$ expressed by a factor graph will require the presence of a factor node with four scopes $f(A, B, C, D)$. If each variable is has 10 states, then the factor node f will have an internal function with $10^4 = 10000$ states. The factor f can be split into two factors f_1 and f_2 while keeping the consistent overall joint probability such that $p(A, B, C, D) = f_1 \cdot f_2$. Each factor f_1 and f_2 will now have only $10^3 + 10^3 = 2000$ states, much lower than the previously factor

f with 10000 states. Since the factor node cannot be connected to another factor node, we have to add a new variable. Let's call this new variable H, then the joint probability distribution of the network can be expressed as: $p(A, B, C, D) = f_1(A, B, H) \cdot f_2(H, C, D)$. This new variable H is hidden and will not appear in the dataset. The EM algorithm then can be used to learn these new factor f_1 and f_2. This situation is depicted in Fig. 2.6.

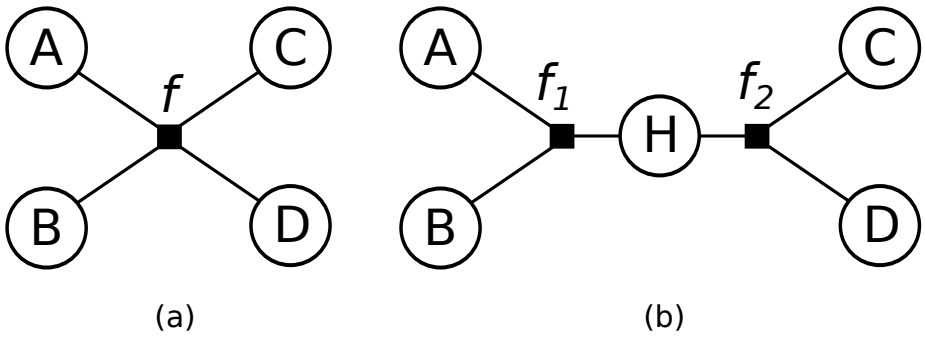


Fig. 2.6: Reducing the complexity of the network by splitting the factor f in (a) into two factors f_1 and f_2 shown in (b). Splitting the factor f will introduce a new hidden variable H which connects the factor node f_1 to f_2. This in turn will reduce the memory consumption for storing the internal function of the factors but with a cost of slowing down the inference speed due to the increasing number of messages that need to be passed from f_1 to f_2 through the hidden variable H.

The MLE algorithm described in the previous sub-section tries to maximize the log-likelihood of observed data in which all variables are observed is basically a convex optimization problem. Maximizing the log-likelihood with datasets containing hidden variables, however, is a non-convex problem. Forcing the the log-likelihood maximization on this case might produces many local maxima. Also, the standard gradient-based methods usually found in optimization literature can converge to a local maximum and very difficult to find the learning rate for which it will not become too slow nor oscillate[117][118].

In PGM, and expectation-maximization (EM) algorithm is an iterative method for finding maximum likelihood or maximum a posteriori (MAP) estimates of parameters, where the model depends on unobserved latent variables. It was first introduced by Arthur Dempster in 1997 and many papers covering the convergence analysis of EM were published since then. The EM iteration alternates between performing an expectation (E) step, which creates a function for the expectation of the log-likelihood evaluated using the current estimate for the parameters, and maximization (M) step, which computes parameters maximizing the expected log-likelihood found on the E step. These parameter-estimates are then used to determine the distribution of the latent variables in the next E step. Comparing EM algorithm with gradient-based algorithm such as Gradient Ascent, the EM algorithm does not require learning rate. However, a careful treatment must be taken because a single update during the training can cause large jumps in parameter space. Despite of these possibly large update steps, the algorithm is guaranteed to converge to a stationary point which most likely will be a local maximum [119].

The initial implementation of EM algorithm in this thesis is motivated by the work of Göhlsdorf [73]. In his thesis, Göhlsdorf uses the following formula to learn the parameters

of factors in a factor graph:

$$f_{\acute{a}}^{t+1}(X_{\acute{a}} = x) = f_{\acute{a}}^{t}(X_{\acute{a}} = x) \cdot \frac{\langle p(X_{\acute{a}} = x \mid Y^i, \theta)\rangle_i}{p(X_{\acute{a}} = x)} \tag{2.18}$$

Here $f_{\acute{a}}$ denotes an internal function of a factor node in a factor graph and $X_{\acute{a}} = x$ indicates a specific variable configuration (i.e. state) for this function. Hence, $f_{\acute{a}}(X_{\acute{a}} = x)$ corresponds to a single parameter of that function. The EM update rule above was derived under the assumption that the partition function $Z = \sum_X \prod_a f_a(X_a)$ would only be subject to small changes during an update. Hence, this formula is useful to change only a few model parameters in each update. Comparing equation (2.18) to (2.17), both have similar underlying mechanism which express the approximation to the first moment of the probability distribution. This is because both equations are derived using expectation formula.

The update rule in (2.18) is used as follows. First, the intrinsic marginal probability $p(X_{\acute{a}} = x)$ and the marginal probability given the observed data $p(X_{\acute{a}} = x \mid Y^i, \theta)$ must be provided in advance. In an acyclic factor graph, $p(X_{\acute{a}} = x)$ can be determined by fixing the messages from all observed variables to non-informative messages (i.e. uniform messages) and running the belief propagation algorithm until convergence. The marginal probability $p(X_{\acute{a}} = x \mid Y^i, \theta)$ can be obtained similarly, but with the observed nodes fixed to the observation Y^i. Using the above update rule, the model parameters can change dramatically within a single update. Therefore, in Dennis' work, it is necessary to present a large batch of data points in order to determine a good approximation of $p(X_{\acute{a}} = x \mid Y^i, \theta)$ before performing an update. Normally, the complete batch of data should be presented before each update. As soon as the marginal probabilities are determined, the factor entries can be updated. Also, in order to ensure numerical stability, the factor entries should be normalized after each update. Scaling the factor entries will not have any effect on the overall probability distribution represented by the factor graph because the partition function will scale as well.

The EM algorithm using the update rule in (2.18) has two limitations:

1. It uses standard averaging technique for all points in the dataset at once in order to avoid the large jump in state space during the training. We argue that this is not the optimal solution because some important information regarding the unequal state distribution might be lost due to overall averaging. A better approach would be to iterate per data point until it converges and then add-up to the previous result. At the final phase, the factor just needs to be normalized again.

2. It updates a single state of the factor node's function in each iteration which reflects the use of Kronecker delta function for the discrete factor graph. Hence, we need to extend it so that it can be used for discrete factor graph with population code.

This thesis provides an improvement for the EM algorithm using the update rule in (2.18) by introducing the population code as the argument for the update rule. It is inefficient to calculate $f_{\acute{a}}^{t+1}(X_{\acute{a}})$ by enumerating all configurations in the arguments of $f_{\acute{a}}$. Furthermore, we are also interested to exploit the inference in the belief propagation mechanism for learning the parameters, similar to the previous treatment for the MLE

approach. Hence, we need to find the configuration with maximum probability value and then spread the distribution according to the population code's variance. Since the EM is an iterative approach and we use population coding instead of Kronecker delta function, we use Kullback-Leibler (KL) formula to measure the divergence level of the new learned parameters. This KL divergence has the following basic form:

$$D_{KL}(p \parallel q) = \sum_x p(x) \ln \frac{p(x)}{q(x)}$$

$$D_{KL}(p \parallel q) = \sum_x p(x) \ln p(x) - \sum_x p(x) \ln q(x) \tag{2.19}$$

By using Jensen's inequality theorem, we know that $D_{KL}(p \parallel q) \geq 0$ with equality *iff* $p = q$. We apply the KL measure on the difference between the new probability distribution ($p(x)$) and the old probability distribution ($q(x)$) about some threshold value as the stopping criteria. The iteration will stop when this KL measure is fulfilled or when the $MAX\_ITERATION$ value is reached. The algorithm for learning the parameters using this EM approach is shown below.

Algorithm 2 Estimate factor parameter θ using EM

```
Θ ← uniformly distribute
for all sample in X do
  φ, φ_old, φ_new ← uniformly distribute
  for i = 0 to MAX_ITERATION do
    for j = 0 to k do
      compute product(φ^k_new)
      φ ← Σ φ^k_new
    end for
    compute diff = KL(phi_old, phi)
    if diff ≤ THRESHOLD then
      φ_old ← Σ φ
      break
    end if
  end for
  Θ ← φ_old
end for
normalize Θ
return Θ
```

Our parameter learning approach is still based on frequentist paradigm on probability. There is another way of learning which uses complete Bayesian treatment. One important aspect of Bayesian learning concept in this way is that it can be used for learning with very little amount of data or when the data is sparse. However, this Bayesian learning concept also has a drawback in determining the prior probability. Without any information about prior probability, the Bayesian learning concept will lose its generality. Justin Dauwels et.al. show that it is possible to utilize the message-passing algorithm for EM in Bayesian

setting [120]. They implement the EM with Bayesian setting in FFG which is out of the scope of this thesis.

We are also aware that there is an effort to use EM algorithm for estimating Bayesian network parameters based on factor graphs in a distributed computing framework [121]. They use libDAI [122] for implementing discrete factor graphs and speed up the computation by using parallelism framework called MapReduce[2] [123]. Unfortunately we couldn't test their method on a discrete factor graph using population code since libDAI does not support our discretization technique. Rather than reimplementing libDAI for supporting population code technique, we prefer to continue working on embedded factor graph.

Another option that can be used to enhance the EM algorithm is by using the max-product algorithm alongside the sum-product algorithm in the message-passing framework. This is not a new idea and in fact the similar idea was used by Zhao Song for a specific application [124]. The basic idea of using max-product for EM can be traced back from many literatures on probabilistic graphical models which explain that the max-product algorithm is a proper query to find the state's configuration for maximizing the posterior probability. In this approach, the sum-product algorithm will be used for inference in E-step of EM process and the max-product algorithm will be used for maximization in the M-step of the EM process. The max-product algorithm (or max-sum if we work in logarithmic domain) can be viewed as an application of dynamic programming in the context of graphical models [16]. We also gain a benefit such that our EM approach using sum-product and max-product algorithm might be used as an alternative approach for the Viterbi algorithm used in common HMM. This method can be described as follows.

Basically, what we want to find the set of values (not only a single configuration) that jointly have the highest probability $\mathbf{x}^{max} = \arg\max_{\mathbf{x}} p(\mathbf{x})$ for it will produce the joint probability distribution with highest value as $p(\mathbf{x}^{max}) = \max_{\mathbf{x}} p(\mathbf{x}) = \max_{x_1} \cdots \max_{x_M} p(\mathbf{x})$. Since products of many small probabilities can lead to underflow problems, it is convenient to work with the logarithm version in which $\ln\left(\max_{\mathbf{x}} p(\mathbf{x})\right) = \max_{\mathbf{x}} \ln p(\mathbf{x})$. This is the max-sum version of the max-product in logarithmic domain. Using the similar approach to the sum-product algorithm, the max-sum can be written in terms of message-passing simply by replacing 'sum' with 'max' and replacing products with sums of logarithms:

$$\begin{aligned} \mu_{f\to x}(\mathbf{X}) &= \max_{\mathbf{x}} \left(\ln f(\mathbf{X}) + \sum_{\sim x} \mu_{x\to f}(\mathbf{X}) \right) \\ \mu_{x\to f}(\mathbf{X}) &= \sum_{\sim x} \mu_{f\to x}(\mathbf{X}) \end{aligned} \tag{2.20}$$

The maximum probability and the corresponding states which generate that maximum

[2]libDAI is a free and open source C++ library for discrete approximate inference in graphical models developed by Joris Mooij. MapReduce was introduced by Google as a simplified software framework for parallelizing computation across large clusters of standard computers

probability value are given below:

$$
\begin{aligned}
p^{max} &= \max_{x} \left(\sum_{s \in ne(x)} \mu_{f_s \to x}(\mathbf{X}) \right) \\
x^{max} &= \arg\max_{x} \left(\sum_{s \in ne(x)} \mu_{f_s \to x}(\mathbf{X}) \right)
\end{aligned}
\tag{2.21}
$$

We have implemented such approach in our PC-based factor graph framework using Kronecker delta discretization technique but we do not implement them in our current embedded hardware for the following reasons:

1. Although the max-product can run in the same message-passing scenario as the sum-product algorithm, it cannot be used directly to get the final result of the parameters without going into the "back-tracking" procedure which requires more additional memory resources. This is because without back-tracking (where all previous values are stored), due to maximization rather than summation, there are several multiple configurations of $\mathbf{x}$ all of which give rise to the maximum value for $p(x)$. As the consequence, it is possible that the maximized configuration is biased. Eventually, if we use the back-tracking mechanism, it will consume a considerable amount of memory resources.

2. It is unclear yet how to efficiently handle the population coding in the scheme of max-product algorithm. In our current implementation, we just spread out the state which produces the maximum probability to emulate a population. But it might be inefficient because we neglect the importance of previously found state. We believe that this task is a challenging one and need deeper exploration. We will consider this in our future work.

Since our current hardware memory is limited, we keep this EM approach using sum-product and max-product algorithms to run only on a PC. The resulting parameters then can be sent to the hardware for normal operation of message-passing algorithm.

2.3 Population Coding Representation

In this section, our method for descretizing continuous variables, which is based on the population coding theory, is presented. To our knowledge, this approach is not entirely new and we can find similarities on some principles to other discretization techniques described in statistical literature. In fact, the population coding principle has also been used for instance by Göhlsdorf [73]. However, in this thesis, we propose an improved version by introducing an adaptive partitioning scheme based on the data distribution.

As we have mentioned briefly in section 1.1.2 on page 2, our method is inspired by the idea of stimuli encoding from a population of neurons in the central nervous system. From literature in the field of neuroscience, it is known that any individual neuron is too

noisy to correctly encode a probabilistic distribution using the spike rate. However, an entire population works in synchrony will ensure greater fidelity and precision of such an encoding. Two motivations for using a population code in our approach for discretizing continuous values are:

1. By using a small number of neurons with certain activation function, the entire space can be represented compactly so that the loss of information due to quantization can be minimized.

2. The probability distribution produced by the population of neurons can be used to represent the uncertainty of sensory information. It is known that the reliability of a sensor reading depends on many aspects ranging from the internal characteristics of the sensor itself to the noise present in the environment. Hence it is beneficial to read the sensor data with some level of confidence encapsulated in the probability distribution.

2.3.1 Encoding and Decoding Principle

Encoding Continuous Values using Population Code

A framework of information encoding known as the population code has been proposed in computational neuroscience to reflect the fact that in many regions in the brain, a group of neurons are activated in a way such that they produce neural responses with certain probabilistic distribution when given stimuli [125][126][127]. In general, the combined response of those neurons is favourable for reducing the uncertainty due to the variability of each neuron. Interestingly, experimental studies in neuroscience also show that this coding paradigm is commonly used in the sensorimotor cortex [128][129]. It is also argued that these paradigms support the inference mechanism for cognitive tasks [130]. In fact, the brain shows a mixture model which integrates many aspects of neural computation including sensory processing, motor command generation, and cognitive planning as well as decisions making [131]. This is a superiority that no current technology can match: the brain shows robust computations in the presence of noise. Hence, this phenomenal performance leads scientists to believe that the brain behaves as a complex stochastic optimal controller [7][132].

This astonishing robustness often gives clues on explaining how stimulus-driven attention is an emergent property of a neural population, which is very robust and able to track a static or moving target in the presence of strong noise or with many distractions, even more salient than the target itself [131]. In perceptual systems, a stimulus parameter can be extracted by determining the center-of-gravity of the response profile of a sensory neurons population. Likewise at the motor end of a neural system, center-of-gravity decoding generates a movement direction from the neural activation profile [133]. Based on these findings, we develop experimental and simulation studies using such a profound theory to demonstrate its consistency and applicability in a broad application area.

We adopt the population coding principle, especially the positional method, to create a compact state representation of a random variable (RV). One method for implementing this principle works as follows. In a discrete network, every discrete RV has a certain

number of states that represents its probability distribution; for example, a binary RV has two states to represents its two possible values. We transform a value in the RV into a vector of some discrete representation based on the predefined cardinality. Consider the network in Fig. 2.2; if we assume that all variables are binaries, then it requires 32 states to represent the joint probability $p(A, B, C, D, E)$. In the numerical representation setting, first we split the interval $I = [\text{min\_val}, \text{max\_val}]$ into k subintervals (called states). Then we assign a probability value for each state and make sure that $\sum_k p_k = 1$. Since we only work with discrete factor graphs, a factor is stored as a conditional probability table (CPT). This standard method easily burdens the computation once the interval I has a long range and thus requires a bigger value of k to properly represents the number. By using the population coding principle, we can think of a state in a discrete variable like a neuron in a small population located somewhere in the brain. A population of several neighbouring neurons with similar characteristics will react in synchrony to the stimulus. For example, in a fully connected homogeneous neuron population, an external stimulus $S^{ext}(t)$ will trigger the population to generate a response $R(t)$ (see Fig. 2.7).

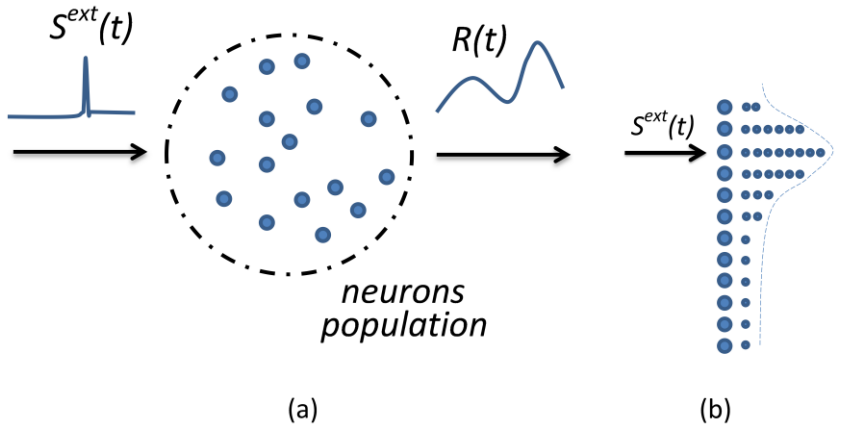


Fig. 2.7: The principle of population coding in a homogenous neurons population. (a) A homogenous neurons population receives an input and generates certain output according to the underlying neuronal model (adapted from [104]). (b) Illustration of an aligned homogenous neurons population that shows the overall response as a Gaussian distribution. In the population, each neuron reacts differently in a form of frequent spike train according to how close the input to each neuron.

Neuroscientists use population codes to decode brain activities that correspond to specific types of perception-action relations. It is common to use bell-shaped (i.e. Gaussian function) tuning curves to encode such relations. The combined activation levels from each neuron then shapes the overall distribution of the corresponding population. In this sense, the usual application of machine learning in neuroscience is to extract information from these population codes. One of preferred methods is the maximum likelihood estimation (MLE) which shows an excellent read-out method, and is regarded as an ideal observer [134]. The main use of the concept of population coding in this thesis, however, is not for estimating the underlying probability distribution of the population, but to compute the activation level of each neuron given the overall prior or posterior distribution of the population alongside the received input stimulus within the receptive field of the neuron

population. Fig. 2.8 shows an example of the tuning curves and illustrates how a real value input is encoded in a population code.

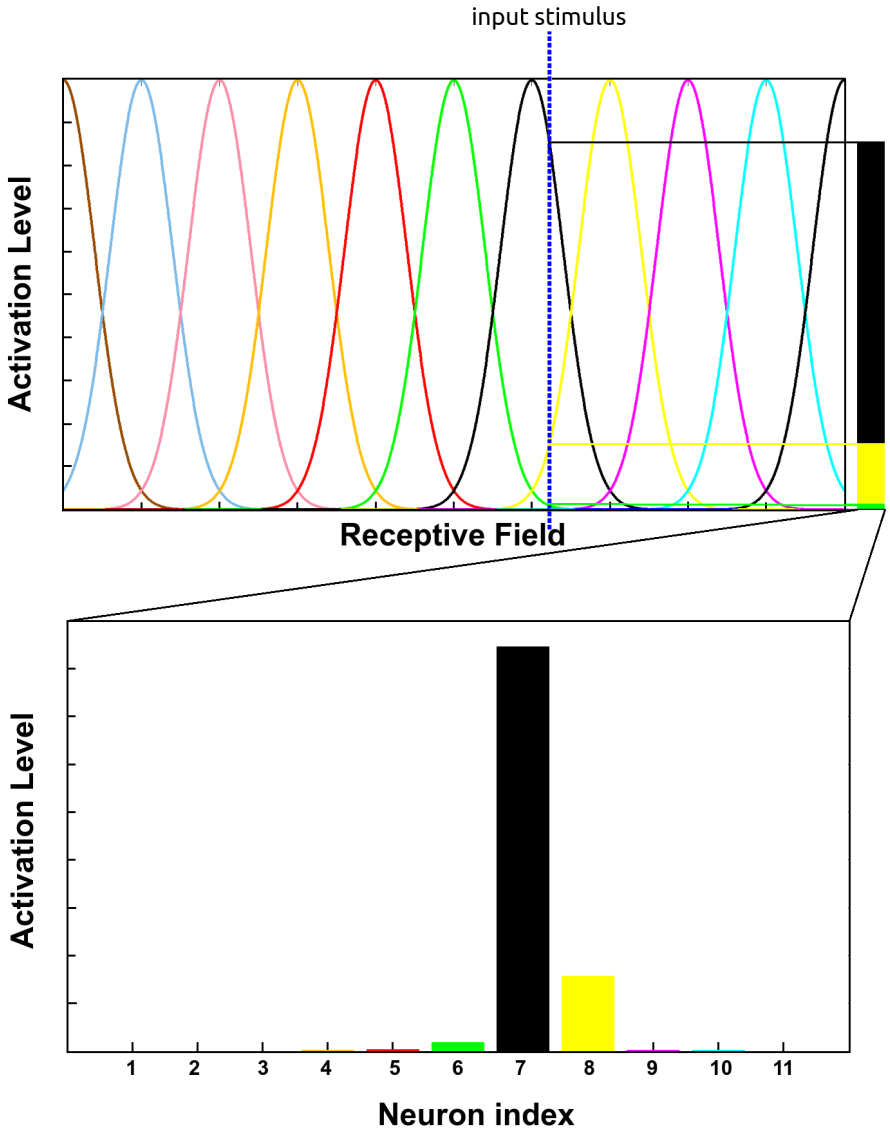


Fig. 2.8: The Gaussian tuning curves for representing neuronal activation levels in a homogeneous population comprising of 11 neurons. When an input stimulus arrives at the receptive field (shown as a blue-dashed line), each neuron fires. The measured activation levels from all neurons (along the blue-dashed line) are then combined to produce the overall probability distribution which is depicted as a discrete probability mass function.

The tuning curves shown in Fig. 2.8 are useful for encoding values where the distribution of the values is uniform. Hence, the tuning curves are equally spaced throughout the accessible space in the domain. In statistical terminology, this is called equal interval binning. The resulting probability mass function (PMF) is then the descretized form of the input value. However, if the distribution of the values is not uniform, then the above tuning curves will fail to produce the correct representation of the discretized value. This usually occurs, for instance, when the population code is used for mapping a non-linear

function. Fig. 2.9 shows an example of a non-linear mapping from the domain A to the domain B where the value distribution in the domain B is not uniform.

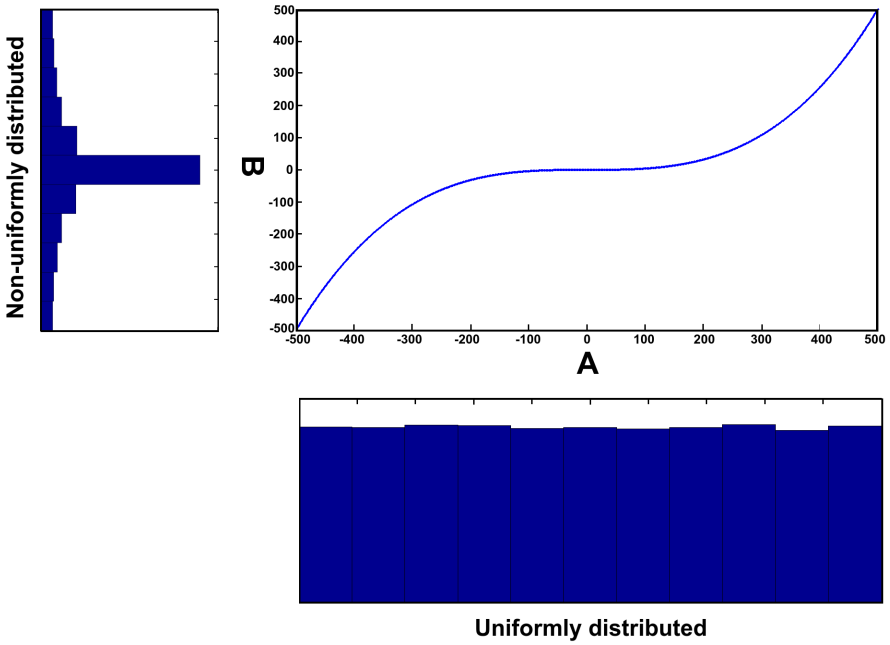


Fig. 2.9: A non-linear mapping between domain A and B. The values in A are uniformly distributed while the values in B are centred around the middle value. In this scenario, uniform tuning curves can only be used to encode the value of A.

For encoding a non-uniform data distribution such as the variable B shown in Fig. 2.9, the tuning curve will be arranged such that more neurons are concentrated in the region where the distribution is denser. To put the words differently, each neuron will be specialized on a certain region such that the number of sampled data in all regions are balanced (in statistics, this is called the equal frequency binning). Fig. 2.10 shows the arrangement of the tuning curves to achieve the equal frequency binning for the variable B in Fig. 2.9. The resulting PMF for the given input stimulus will be different from the previously computed PMF with equally spaced tuning curves shown in Fig. 2.8.

The remaining question for using a population code to generate a PMF that represents a continuous value is how do we determine the center of each Gaussian curve? This is a straightforward procedure for the uniformly spaced curves since we know exactly the width of each region, and the center of the curve lies at the center of each region. We just need to make sure that the distance between successive curves are equal throughout the working space. For non-uniformly spaced curves, however, it is not straightforward and we need some techniques to best allocate the curves' centers. An excellent technique used in statistics to determine such an allocation is based on the criterion called the minimum descriptive length (MDL) [51][135][136][137]. One such an algorithm, which is quite popular and is used also in this thesis, is the K2 algorithm. To achieve such an equal frequency binning with proper number of partitions, the entire working space is partitioned into N-regions (usually starts from a small number and is increased during the iteration), and a counter assigned for each region will count how many data have been placed into that region. The MDL-based algorithm then tries to find the best cutting-points for each region

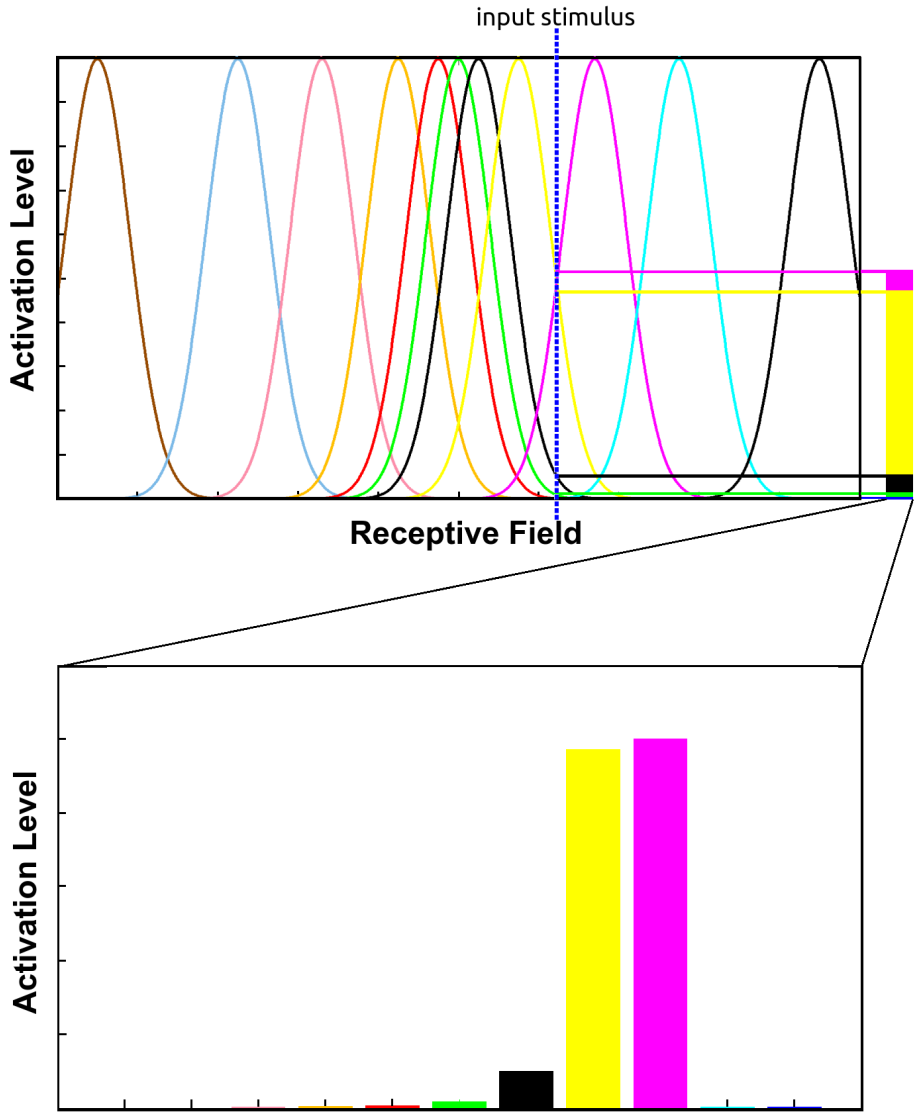


Fig. 2.10: The Gaussian tuning curves for representing a non-uniform data distribution are spaced unequally in order to represent neuronal activation levels in a homogeneous population comprising 11 neurons. The input stimulus (depicted as a blue-dashed vertical line) similar to the one shown in Fig. 2.8 is received at the receptive field. Each neuron then fires and the measured activation levels from all neurons (along the blue dashed line) are combined to produce the overall distribution which is depicted as a discrete probability mass function.

by making sure that the number of data points in successive regions are equal or almost equal. As a consequence of this step, some adjacent partitions might be combined to better represent the value range of the corresponding variable. This is an iterative process and there are several approaches to maximize the entropy with minimal information loss by dynamically repartitioning the continuous space.

The more biological plausible method to achieve similar result is based on the unsupervised training method using the Kohonen's self-organizing map (SOM) network. The idea is straightforward: with a proper training, the SOM will preserve the topological properties

of the input space. In this sense, the SOM is trained to produce a map – the discretized representation of the input space of the training samples [138][139]. In our discretization method, we create a one-dimension SOM network composed of N neurons, and perform the network training based on the Gaussian neighbourhood function. The weight of each neuron will then represent the balanced location of the Gaussian tuning curve. These weights are adjusted according to the data fed into the SOM network. This process is depicted in Fig. 2.11.

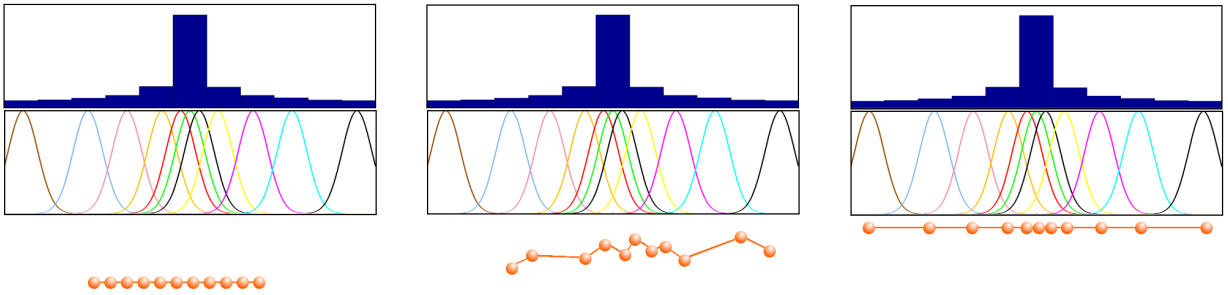


Fig. 2.11: The principle of fitting mechanism for optimally-spaced tuning curves that is based on SOM training. Each neuron's weight in the SOM network will be updated during the training to preserve the topological structure contained in the presented data to the SOM network.

So far we treat the population of neurons as a homogeneous population. Another type of population is the non-homogeneous one, where each neuron in the population has its own characteristics that differ from one neuron to the others. For example, neurons with a bell-shape activation level might have different variance value. However, in this thesis, we do not use this heterogeneous population because we follow the idea that a population of neurons should represent a node consistent with the graphical model theory which states that such a node should be internally coherent [140][141]. Also, we found no firm evidence that exploiting the heterogeneity of a population of neurons will improve the overall encoding performance in the context of probabilistic graphical modelling. However, it is interesting for our future work to include also the heterogeneous population coding since some works suggest that the heterogeneous neurons population might facilitate temporally controlled behaviours [142][143][144].

Decoding the Population Code into a Value

Given a probability message encoded in a population code that circulates around in a factor graph network, the question is how can we decode such a message to get the real value back from the population code representation. One might be tempted to use the mode of the distribution curve since this point intuitively represents the most likely state that contributes the largest portion to the overall density. This idea is a common perspective in decision theory but only valid for multi-modal probability density function such as the mixture Gaussian. For example, in [13] chapter 15, the author gives a valid example of how the flying bird intuitively decides which direction to take for an evasive action in avoiding the collision with an object using its visually generated perceptional belief encoded in a multi-modal distribution. However, this approach does not always work especially in the

case where the density function originates from data with a non-uniform distribution, for instance in the non-linear transformation problem shown in Fig. 2.9. This is illustrated further in Fig. 2.12. If the mode is used to decode the message for variable B, then it will produce a "bias" (shifted value) which is not correct.

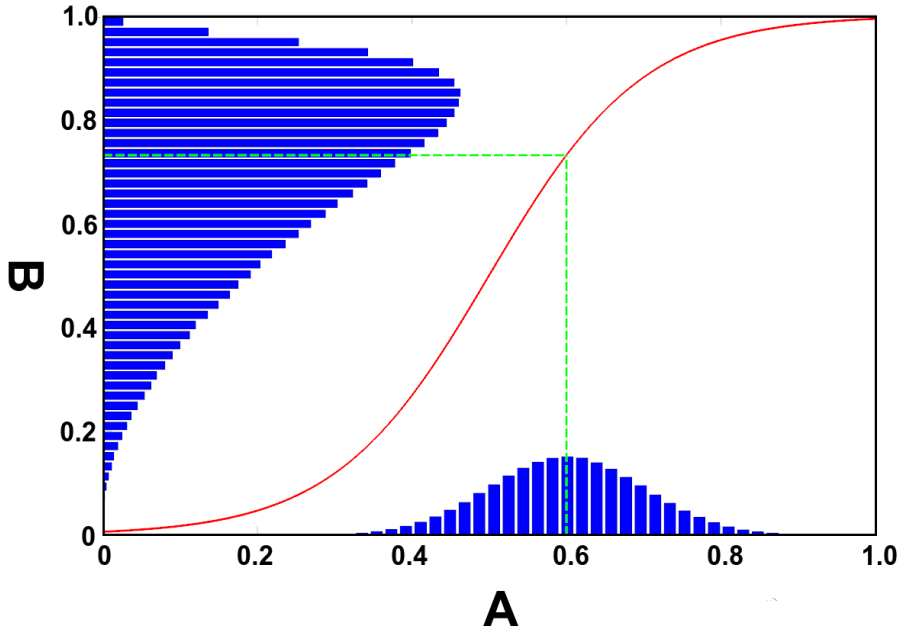


Fig. 2.12: The biased value resulting from an improper computation that is based on the mode of the population code (adapted from [1]).

A better solution for computing a real value back from a distribution is by using the maximum likelihood inference (MLI) in which the stimulus estimator $\hat{x}$ is obtained by maximizing the log likelihood $p(\mathbf{r} \mid x)$, where $\mathbf{r}$ is the tuning curves function and x is the stimulus. For practical consideration, it is convenient to assume that the correlation between the tuning curves can be neglected. Thus, solving this MLI will be the problem of approaching the stimulus estimator using the center-of-mass method [126]:

$$\hat{x} = \frac{\int_{-\infty}^{\infty} x \cdot p(x)\, dx}{\int_{-\infty}^{\infty} p(x)\, dx}$$

If the message containing the above information is normalized, which is a standard practice, then the denominator part can be removed. In the discrete form, the stimulus estimator is the expected value of the probability mass function:

$$\hat{x} = \sum_{i}^{n} x_i p_i(x) \tag{2.22}$$

where x_i is the center of the tuning curves and p_i is the activation level of the corresponding neuron. In other words, all neurons responses are integrated by using the weighted population average.

The limitation of this approach is that, when the distribution is multi-modal, then the

real value computed using formula (2.22) will not be exact simply because that formula performs an averaging. However, we can still approximate it using the decomposition procedure such as in a Gaussian mixture model (GMM) where we want to estimate the mean for each component of the GMM. This is done usually for analytical reason but, for practical application, we prefer to use formula (2.22) assuming that the message circulating in the belief network is uni-modal.

2.3.2 Performance Evaluation

It is imperative for us to test the performance of our encoding/decoding approach before using it further in more complex applications. To test the performance, a two-variables factor graph shown in Fig. 2.13 is used. A dataset containing two RVs A and B with certain relationship that will be fed into the network is generated. Variables A and B from the dataset are discretized using the population coding approach, and enter the network through factors f_A and f_B respectively as belief messages. The prior belief of the factor node f_{AB} is updated with these messages using (2.7). The posterior of f_{AB} is basically the accumulation of the counting action in (2.17).

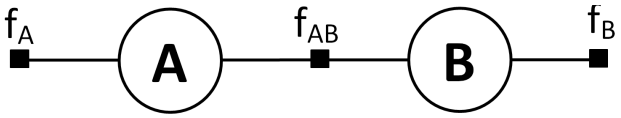


Fig. 2.13: An example network for linearity test of the proposed discretization strategy using the population coding principle.

The network shown in Fig. 2.13 is used for testing against the linearity and non-linearity of the mapping response of the population codes. For the linearity test, we generated two linear data, labelled A and B, in the range [minA, maxA] where we set B simply equal to A times a constant value C. For the non-linearity test, we firstly generated A in the range [minA, maxA] and then computed B using a logistic function from A. The result is shown in Fig. 2.14. For this test, the Gaussian function is used for encoding the value of the network parameter.

Fig. 2.14 is produced by varying the parameter σ^2 of the Gaussian function used for encoding the message and also by varying the number of states for that message. As shown in the figure, the variance value has the same important effect on the smoothness of the curves as well as the number of states used in the population code. This effect can be clearly seen when inspecting the "internal" state distribution of the factor as shown in Fig. 2.15. Giving too low value on the parameter σ^2 will yield a "stair-case" effect (shown by blue curves in Fig. 2.14), which is a natural response by which a Gaussian function will approach a Kronecker-delta function. This in turn will cause the joint probability distribution (i.e the internal function of the factor node) to become "thinner" (see Fig. 2.15a). However, setting too high value on σ^2 (see Fig. 2.15d) will flatten the Gaussian function almost close to a uniform distribution. This action will yield the scaling effect when using the weighted population average to recover the real value back from the distribution. In our experiments, we usually set this parameter σ^2 within the range [1, 10]. In general, our proposed discretization approach using the population coding paradigm with proper

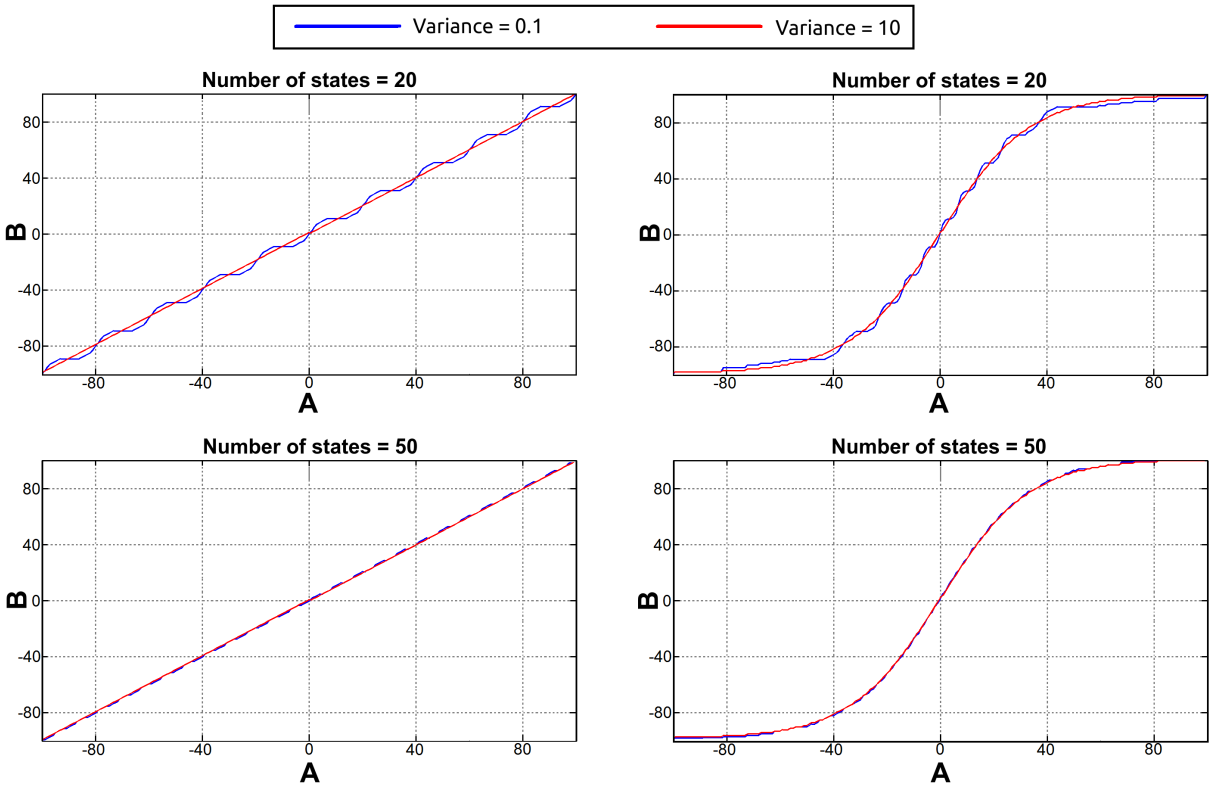


Fig. 2.14: Linearity and non-linearity tests for the proposed population coding as a function of Gaussian's σ^2. Graphs to the left show the result of the linearity test, and graphs to the right show the result of the non-linearity test. Both the variance and the number of states will determine the "smoothness" of the result. In general, using high number of states will diminish the quantization effect. However, using a proper variance value can also improve the performance even when using a small number of states, as shown in the upper graphs.

parameter tuning produces very good result. For example, the linearity test in Fig. 2.13 using 50 states and $\sigma^2 = 10$ produces mean squared error (MSE) = 0.4, while the non-linearity test produces MSE = 0.5.

Regarding the generality of population coding in a belief propagation setting, Göhlsdorf describes in his work that population codes are incompatible with factor graphs [73]. This is true when factor nodes in the factor graph contain the joint probability distribution instead of the conditional probability distribution. In fact, this is valid not only for factor graphs with population codes, but also for all factor graphs which originate from Bayesian networks. It can be easily understood by considering the basic formula of the Baye's rule:

$$p(A \mid B) = \eta \cdot p(B \mid A) \cdot p(A) \tag{2.23}$$

where η is the normalizing constant which is equal to $\eta = \frac{1}{\sum_B p(A|B) \cdot p(B)}$.

To understand the problem, consider the following setting. The belief propagation algorithm is mainly used for computing the marginal probability of a certain variable. This is done by first computing the messages from factor nodes to the respective variable node using equation (2.5). As an example, let's assume that we want to compute $p(B)$

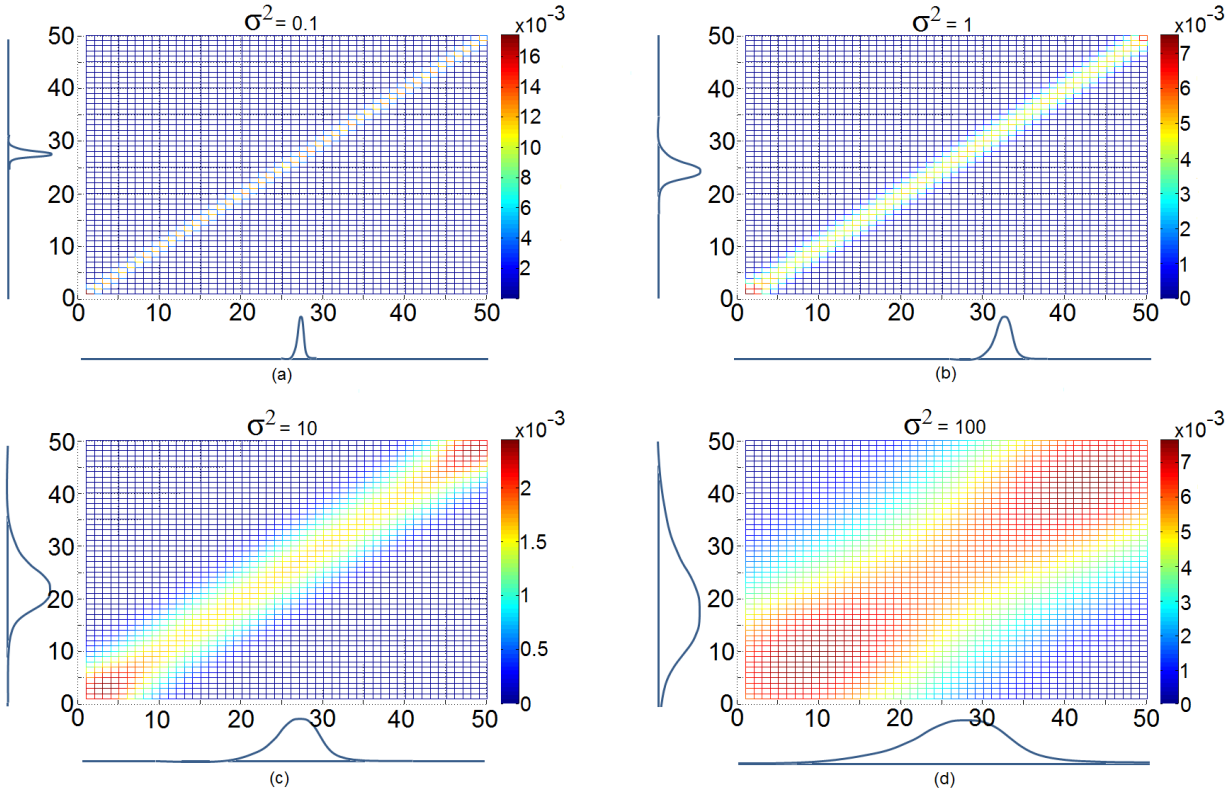


Fig. 2.15: The content of "internal" state distributions of the factor f_{AB} in Fig. 2.14.

using the network in Fig. 2.13 and the factor node f_{AB} contains the joint probability $p(A, B)$. The input A ($f(A)$) represents the prior distribution of variable A ($p(A)$) and will arrive at node f_{AB} as the message $\mu_{A \to f_{AB}}(A)$. Using equation (2.5), the output message $\mu_{f_{AB} \to B}(B)$, which represents the marginal probability $p(B)$, will be computed as:

$$p(B) = \sum_{A} p(A, B) \cdot \mu_{A \to f_{AB}}(A) = \sum_{A} p(A, B) \cdot p(A) \tag{2.24}$$

which is clearly not the correct formula according to the equation (2.23). Instead, we need to compute:

$$p(B) = \sum_{A} p(B \mid A) \cdot \mu_{A \to f_{AB}}(A) = \sum_{A} p(B \mid A) \cdot p(A) \tag{2.25}$$

The equation (2.24) is the biased version of the equation (2.25). This bias effect is illustrated in Fig. 2.16. To overcome this problem, the joint probability distribution needs to be conditioned to get the conditional probability distribution.

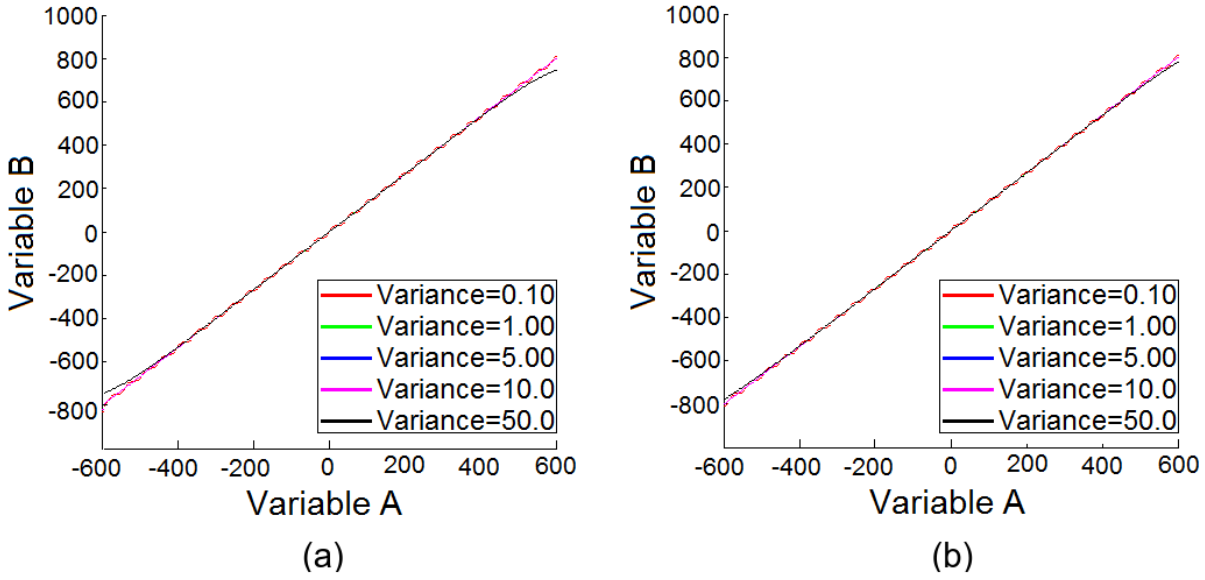


Fig. 2.16: The bias effect is produced when an improperly probability distribution is assigned to the factor node. (a) The joint probability distribution is assigned as the internal function of the factor node f_{AB} in Fig. 2.13. (b) The joint probability distribution is corrected into the conditional probability distribution for the factor node f_{AB}. The result is exactly linear as expected.

The second improvement that we have made for optimally using the population codes is a SOM implementation for adaptively adjusting the mean of each tuning curve in the neurons population. This is done by observing the data distribution of the corresponding dataset, and corresponds to the non-uniformly spaced tuning curve shown in Fig. 2.10. An example of using this non-uniformly spaced population versus the uniformly spaced population is shown in Fig. 2.17. This figure is produced using the same network in Fig. 2.13 where we fed the network with both linear and non-linear data.

In summary, there are two important aspects for the population codes to be used optimally in factor graphs. First, the factor nodes must have valid probability distributions in order to avoid biasing effects due to the prior probability. Second, the tuning curves must capture the "content" of information (i.e. the data distribution), otherwise, it will overfit the data.

2.4 Software Framework Development

Since the PGM is not a new concept, many libraries have been built and published to accommodate the needs for deploying specific algorithms/applications based on the probabilistic reasoning paradigm, either as an open source or a proprietary library. Those libraries usually belong to one of these categorizations: directed or undirected graphical models, and exact or approximate inferences. One reference to these libraries can be found in Murphy's website about software packages for graphical models[3] [145]. More than 50 software packages have been listed so far and this number keeps growing from time to time.

[3]http://www.cs.ubc.ca/ murphyk/Software/bnsoft.html

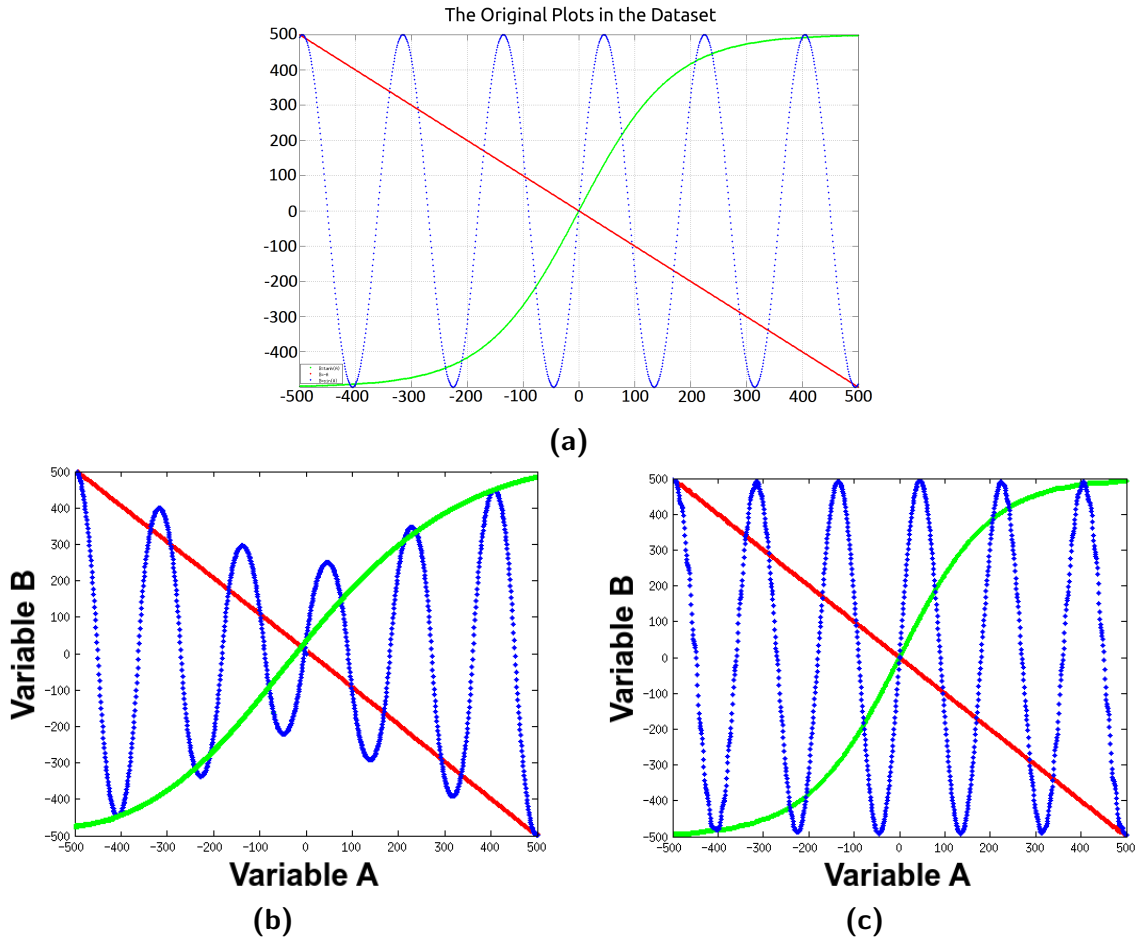


Fig. 2.17: Miscellaneous self-consistency tests involving linear and non-linear data for evaluating the optimality of the population codes. (a) The original dataset that need to be recovered back by inference processes on variable B in Fig. 2.13. (b) The result of using uniformly-spaced tuning curves. The linear data does not have any problem with recovery, but the non-linear data will be increasingly deteriorated (as illustrated by the sinusoidal curve). (c) When the tuning curves are spaced properly by learning the data distribution, the inference process yields much better results.

This thesis does not use any of those existing libraries. Instead, we develop our own framework for the following reasons:

1. All of those existing libraries are designed to run on a PC. Even some of those libraries run only on a simulation platform such as Matlab. None of them is designed to be directly implementable in a dedicated hardware such as SpiNNaker or SoC.

2. Some of those libraries support the hybrid models (i.e. mixing the continuous and discrete variables), but none of them supports the implementation of population coding for the discretization phase. Our factor graph framework uses a population coding approach to discretize continuous variables and also to build a hybrid model.

3. By developing our own framework, we gain deep and thorough knowledge about how the fundamental/core of the factor graph algorithm is actually computed and thus

we know exactly where the code should be optimized or be left as it is.

For simulation and investigation purposes of this research, we have developed the data structure and classes for inference in C++ programming language. Fig. 2.18 shows the foundation class CFactorGraph of the factor graph framework which contains two important classes: CFactor and CNode. The class CFactor can be used to instantiate factor nodes and is derived from the class Ccpt (a class for managing Conditional Probability Table, or CPT). The class CNode can be used to instantiate variable nodes and is derived from the class Cpmf (a class for manipulating Probability Mass Function, or PMF). An example of how to use of our factor graph library can be found in Appendix-B.

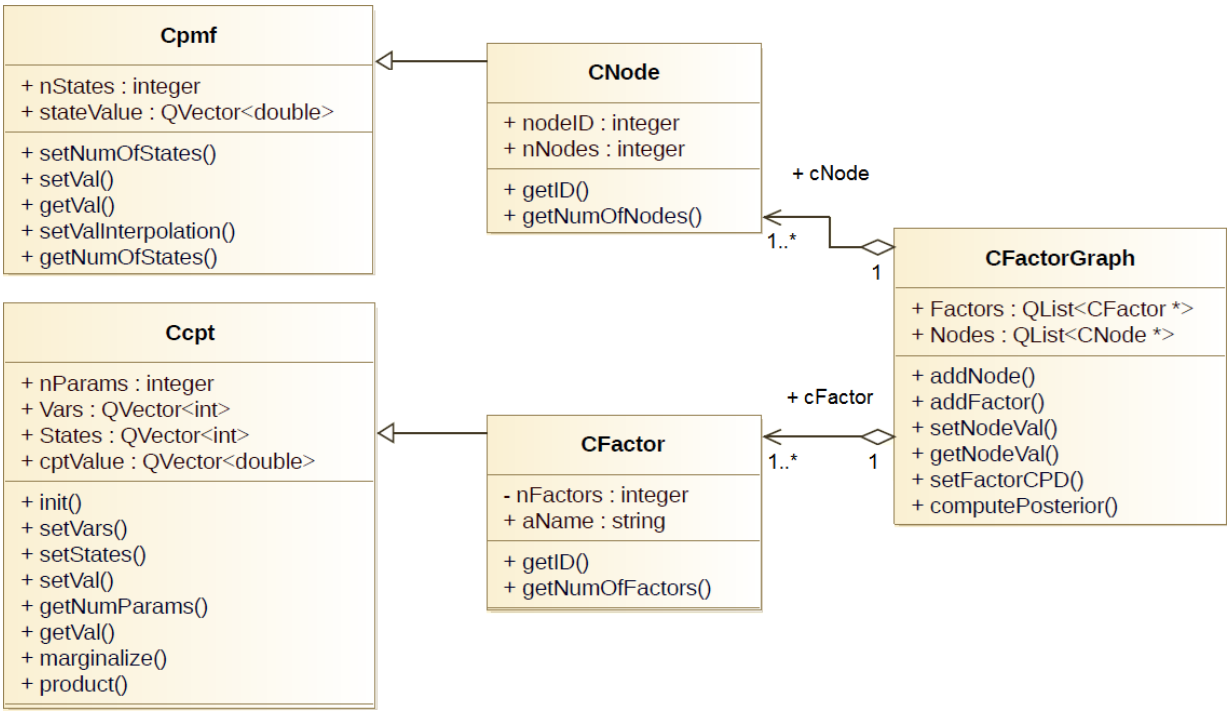


Fig. 2.18: The simplified UML (unified modeling language) diagram of our factor graph framework.

The diagram shown in Fig. 2.18 does not include the user interfaces (GUIs) which are also developed during this research for helping us in many circumstances. More specifically, we create a software suite for simulation and data collection. The simulation platform is extensively used to investigate the behavior of the system in a "clean" environment (i.e. manageable noise level) and help us understanding the nature of our algorithm. Fig. 2.19 shows one of our software modules used for collecting data in an experiment with mobile robot control using a factor graph network (see section 3.3.1 on page 61).

We have tested our framework not only in a conventional workflow but also using several other parallelism techniques. In particular, we are interested in implementing our framework in a multi-core PC as well as a GPU. The current trend in such parallelism platforms enables us to quickly implement our algorithm and gain benefit of parallelism such that we can get the faster result for analysis. However, parallelizing our factor graph framework in a PC-based machine is not our goal. Here we only show a proof of concept of how our

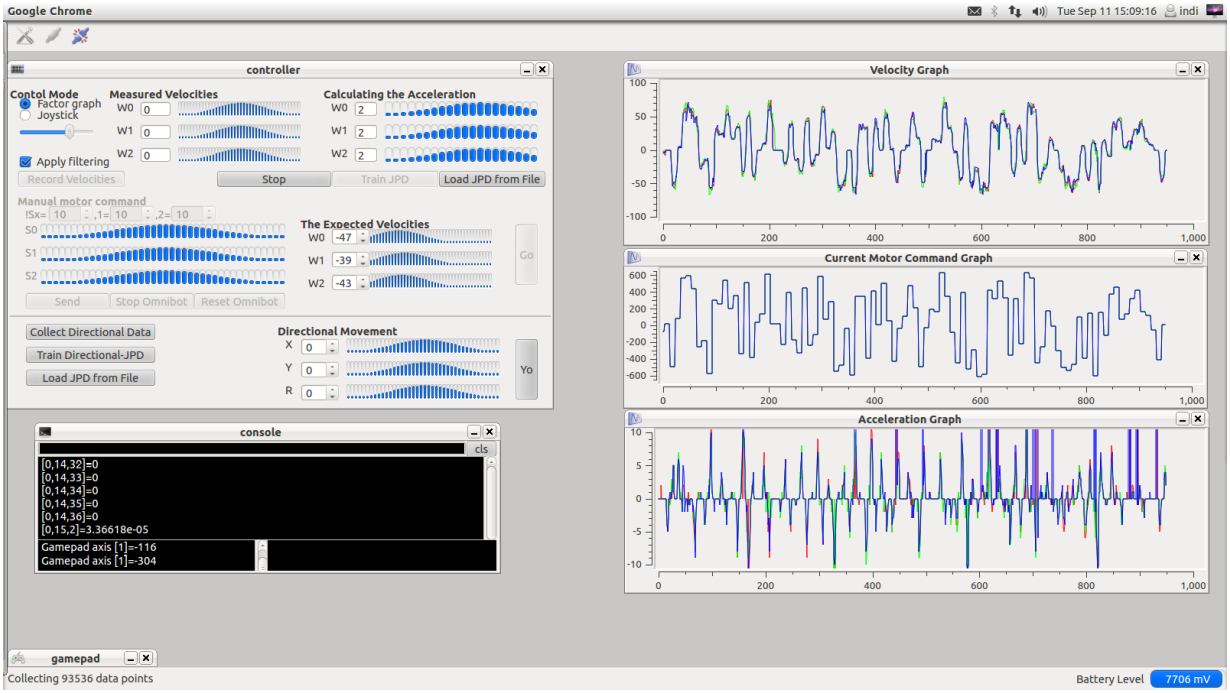


Fig. 2.19: One of our software suite which is a part of our factor graph framework developed for PCs. It is equipped with the data acquisition program to capture data that will be processed by our factor graph framework.

framework can easily be extended in such a parallelism platform. For example, in the GPU version of our factor graph, we do not utilize the full multi-grid threading to implement our factor graph node-by-node. Instead, we just use the GPU as an accelerator for some of the most intensive computations in the belief propagation algorithm. The result of our experiment with parallelism strategy on a PC-based machine is shown in Fig. 2.20.

One important aspect that we observe on using these platforms is about the data preparation. It turns out that the speed-up gain is also heavily influenced by the way we prepare the data apart from the algorithm itself. This is because the underlying parallelism core has its own mechanisms for handling potential software bugs introduced by the concurrency process, and we have to follow its rules. Problems such as race condition and mutual exclusion need to be handled properly in order to make sure that the results are consistent with the standard/normal way of running the algorithm on a PC. Also, communication and synchronization between the different threads are the most difficult tasks to handle in the first place to get the best performance of the program running in parallel. These issues contribute to the phenomenon related to the maximum possible speed-up of a single parallelized program known as the Amdahl's law.

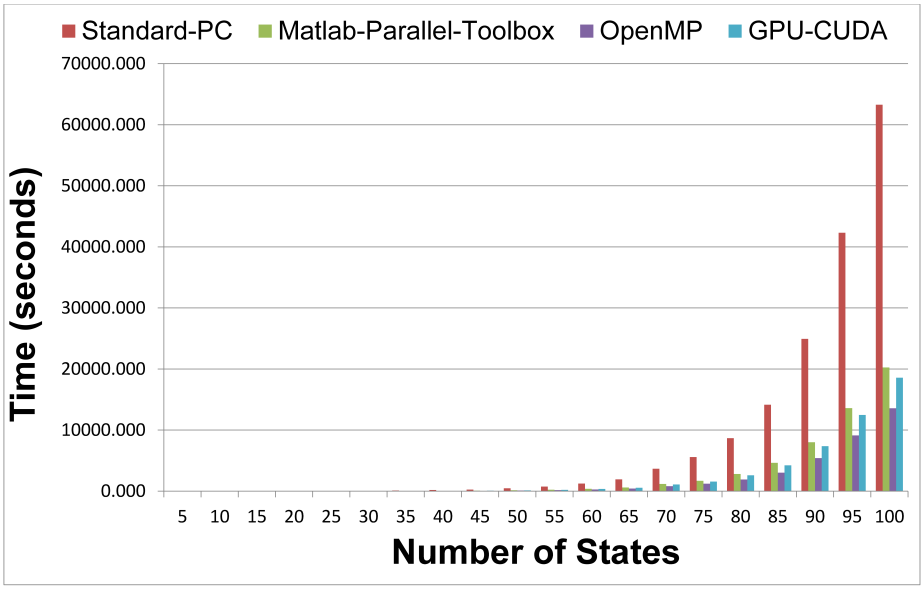


Fig. 2.20: The performance comparison of our factor graphs in a PC that were implemented using Matlab's Parallel Toolbox, OpenMP, and GPU-CUDA. The average speed-up gains for the parallelized factor graphs for example applications described in chapter 3 are: 3.3 for Matlab's Parallel Toolbox, 4.7 for OpenMP and 3.4 for GPU-CUDA.

3 Reasoning in Factor Graphs

Many applications can use factor graphs to perform reasoning on a given task. In those reasoning tasks, the factor graph is used mainly as an inference engine. In this chapter we give examples to show how our factor graph framework can be used in applications, ranging from standard tasks usually found in machine learning literature to the challenging and complex ones in the robotic domain. This chapter has two purposes. First, it serves as a means of proof-of-concept where we apply our factor graph framework in standard but important applications (mainly in the domain of machine learning). Second, it demonstrates our contribution in developing extensible robotic subsystems with embedded cognition capabilities that useful for building more complex robotic applications.

3.1 Application in Machine Learning

Why are we interested to apply factor graphs in the machine learning domain? It is because in machine learning, probabilistic graphical models are used for computing uncertainty and for generating actions based on perception, which provide a unified framework for graph theory and probabilistic reasoning in a complex real-world setting. This in turn will lead to the development of a larger framework that is capable to generate intelligent behaviours in a fashion similar to the neural computation in the brain: a massively parallel distributed computing system, in which the overall performance comes from independent computation of local units that communicate with their neighboring units. Also, machine learning is a vast domain which becomes one of the most progressive fields in recent decade that we believe will open opportunities for our methods to give beneficial contributions to science and engineering.

Although so many branches of application have been developed within this domain, there are two basic categories where those applications can be grouped together: classification and regression [146]. In this section we give examples of how to use our factor graph framework in this field. The purpose of these examples is not just as a proof-of-concept but also to show that our framework has promising prospects for collaborations with other domains to build more complex applications.

3.1.1 Factor Graph for Regression

Basically, regression is a process for estimating the relationships between a dependent variable and one or more independent variables. It is widely used for prediction or forecasting, and, in restricted circumstances, it can also be used to infer causal relationships between independent and dependent variables. Hence, in this scenario, the dependency between two variables, let's say A and B, yields a mapping function $f : A \rightarrow B$ which is depicted in Fig. 3.2a. In probabilistic setting, it can be viewed as a process of estimating the target

variable z given some new values of the input variable x with some degree of uncertainty based on a probability distribution. For example, a dataset which contains N data points might have a Gaussian distribution for its target values $\mathbf{z} = (z_1, \cdots, z_N)^T$ and the given input values $\mathbf{x} = (x_1, \cdots, x_N)^T$:

$$p(z \mid x, \theta) = \mathcal{N}(z \mid y(x, \theta)) \tag{3.1}$$

This is illustrated in Fig. 3.1.

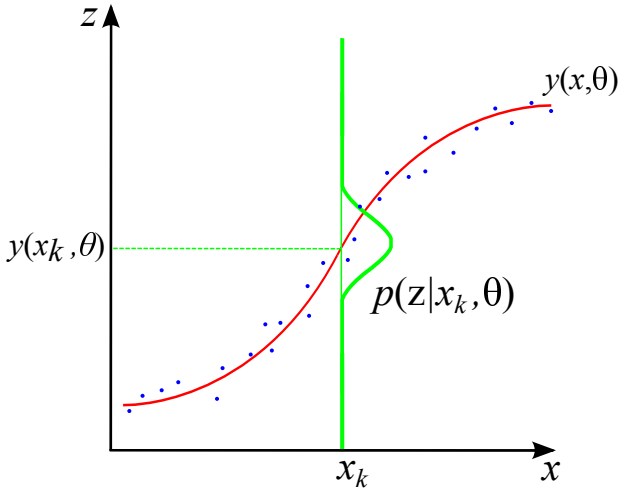


Fig. 3.1: Illustration of a regression technique in a probabilistic perspective. It shows the conditional distribution for y given x in which it is assumed to be a Gaussian.

At some extent, regression can also be used as a performance test of the proposed discretization strategy for our factor graph (see section 2.3.2). For example, we generated 50 samples from a function $f(B) = 500 \cdot sin(A \cdot 0.5 \cdot \pi) + 250 \cdot cos(A \cdot 2 \cdot \pi)$ where variable A has a range [-400,400]. The regression result, shown in Fig. 3.2b, is the updated belief at node B after a message starts propagating from f_A and arrives at B.

It is interesting to note that the applied inference mechanism on the factor graph shown in Fig. 3.2b yields non-overfitting curve since its marginal likelihood basically integrates over all model parameters. To evaluate further the performance of such a regression model, we also add noises on the sampled data and then we perform the inference process once again. The result is shown in Fig. 3.3a, together with its root-mean-square error (RMSE) calculated during the MLE process of the model parameters (Fig. 3.3b). It is also interesting to note that the RMSE gets steady when the model has received enough samples for estimating its parameters (in this case 30% of all points in dataset). This explains the well behaviour of a regression inference in Bayesian settings as shown in Fig. 3.3a.

3.1.2 Factor Graph for Classification

Another application which is quite popular in machine learning is a classification task. For classification in a setting of probabilistic framework, one can use methods such as the popular Naïve Bayes classification algorithm. In fact, many have considered Naïve Bayes classifier as a baseline classifier to try before developing more complex classifiers

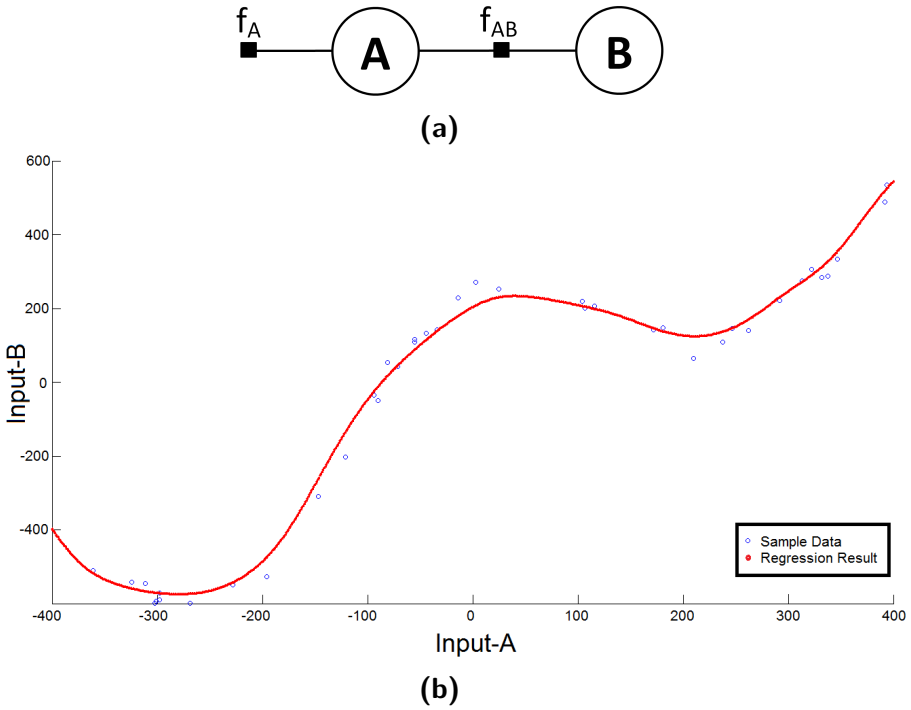


Fig. 3.2: (a) The factor graph network for regression tasks. (b) The regression result using the network shown in (a). The small blue circles are the original data-points in the dataset, and the red plot is the regression curve learned by the network.

[147]. Here we show how to use a simple Naïve Bayes classification mechanism in a form of a factor graph, which is simply accomplished by transforming the respective Bayesian network into a factor graph. The task is to classify example data points shown in Fig. 3.4 into two classes (the data points are virtually separated by the green line)[1].

In a Naïve Bayes classifier setting, it is assumed that each particular feature F_i is independent from the other features $F_{j\neq i}$, given the class variable C:

$$p(C \mid F_1, \cdots, F_n) = \frac{1}{Z} p(C) \cdot p(F_1, \cdots, P_n \mid C)$$

$$p(C \mid F_1, \cdots, F_n) = \frac{1}{Z} p(C) \cdot p(F_1 \mid C) \cdot p(F_2 \mid C, F_1) \cdots p(F_n \mid C, F_1, \cdots, F_{n-1}) \tag{3.2}$$

By assuming conditional independence among features, it means that:

$$p(F_1 \mid C, F_2, \cdots, F_n) = p(F_1 \mid C) \cdot p(F_1 \mid C, F_2, \cdots, F_n) = p(F_1 \mid C)$$

[1]For this example, the data was taken from the homework page of Computational Intelligence class at Technische Universität München, where we gave tutorials in winter semester 2014 on how to use supervised learning techniques using artificial neural networks. The source of the data can be found at http://ci.nst.ei.tum.de/ci_ws2014/homework/hw2/hw2.html

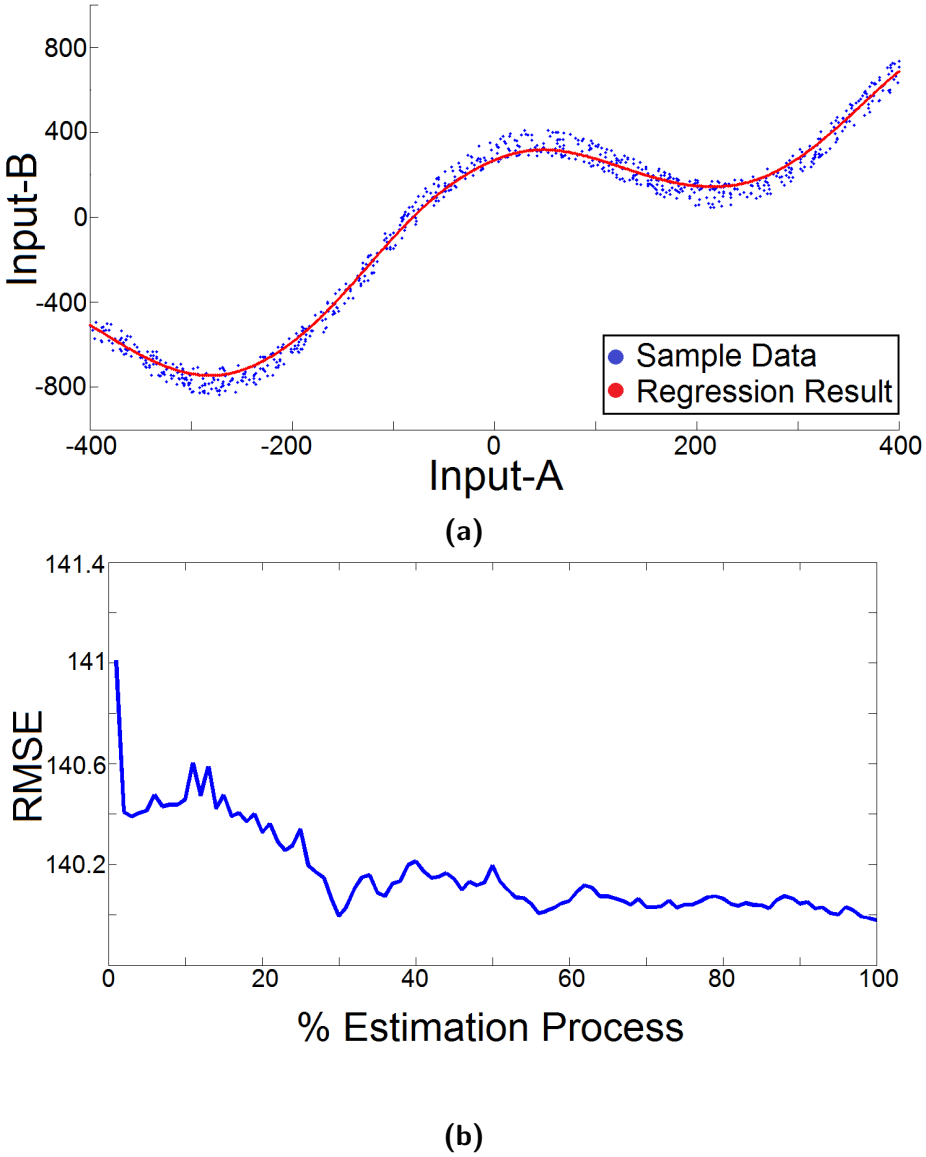


Fig. 3.3: (a) Regression result from data with noise. (b) The RMSE plot during parameter estimation.

So that the expression (3.2) can be written as:

$$p(C \mid F_1, \cdots, F_n) = \frac{1}{Z} \prod_{1}^{n} p(F_n \mid C) \tag{3.3}$$

which results in a network shown in Fig. 3.5a. It can be seen that the structure in Fig. 3.5a is an extension of the structure in Fig. 3.2a. It is important to note that the normalization factor Z in (3.3) needs to be computed when estimating the factor parameters during the MLE process. Although Z mainly gives strong influences when involving more scope variables in a factor node, we have already seen its biasing effect even when the network is relatively small (see Fig. 2.14).

The classification is done by using the same inference procedure in the belief propagation

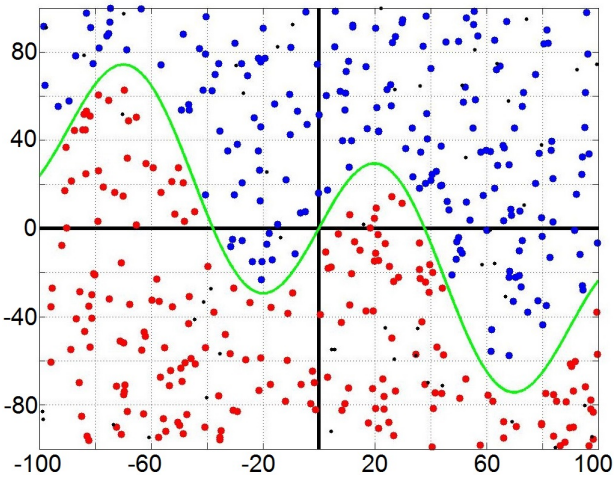


Fig. 3.4: An example of a classification task that requires a non-linear classifier. The green curve is the underlying function which determines the decision boundary for the two-class data. The red and blue dots are data-points of the respective classes.

mechanism. The result shown in Fig. 3.5b was produced using 20 states for each node in Fig. 3.5a. The left end of the curve seems to be shifted upwards and the right end of the curve seems to be shifted downwards. This shifting was resulted from the shallow probability distribution of factors F_1 and F_2 in Fig. 3.5a, which is the side effect of the low probability at the left most and right most part of the Gaussian distribution used in the population coding representation.

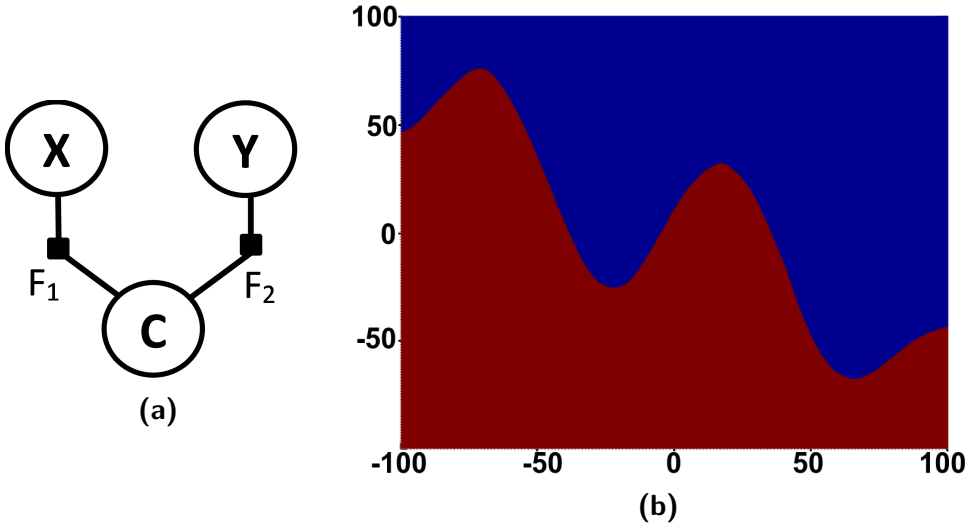


Fig. 3.5: (a) A factor graph for implementing a Naïve Bayes classifier in a two-class classification problem. (b) The result of classification performed by the model in (a) for the dataset depicted in Fig. 3.4. The coloured regions represent the "red" and "blue" class respectively. In this example, the model perfectly splits the two classes using a non-linear decision boundary.

We argue that our naïve approach for classification in a supervised scenario can be extended into an unsupervised version in a clustering task since this extension is also a

standard practice in machine learning. Furthermore, a significant progress in clustering applications has been achieved by algorithms that use a framework for pairwise clustering. This framework is based on the equivalence between the calculation of the typical cut and the inference in an undirected graphical model [148]. Surprisingly, this approach demonstrates that the loopy belief propagation (LBP) and the generalized belief propagation (GBP) can give excellent results on challenging clustering problems with complex datasets.

3.1.3 Factor Graph for Sensor Fusion

In the previous two sections, we give example applications of acyclic factor graphs. In this section, we introduce a toy example of a cyclic factor graph in a context of information fusion. One typical example of this information fusion is the task of sensory data fusion. The goal of sensor fusion, especially in robotics, is to combine measurements from a set of different sensors to improve the quality of the perception about the state of the world. Different sensors have diverse physical characteristics; even data from the same type of sensors could be quite varied due to their internal characteristics depending on their configuration. The idea is quite simple: by combining complementary or redundant information from multiple sensors, more robust estimation can be achieved than using a single sensor. This actually reflects how our brain works on perception; it fuses several modalities from sensory organs to obtain a single interpretation of the human internal states and/or the environment states. Our goal is that, by bringing the sensor fusion task into a factor graph representation, we can merge it with our other robotic subsystems that are also implemented in factor graphs (described in section 3.3).

There are some efforts to incorporate the sensor fusion task into a factor graph framework [149][150][151]. However, their methods rely on the variable elimination mechanism for factor graphs. In this section, we introduce the use of the loopy belief propagation (LBP) mechanism for sensor fusion.

Let's assume that our robot is equipped with two sensors: a gyroscope and a compass. We want to use those sensors to give information about the orientation of the robot. Conceptually, the compass should give the absolute orientation of the robot while the gyroscope data need to be integrated to get the orientation value. For simplicity, let's assume that the robot performs internal integration for the gyroscope data in order to get the robot's pose reading. Now let's assume that the robot is placed in a room with an overhead camera tracking system that give the "ground-truth" data, which is also useful for calibrating the robot's sensors. In this simple scenario, let's call the gyroscope data as sensor-A, the compass data as sensor-B and the direct measurement from the camera tracker as sensor-C. The factor graph network corresponding for fusing these three sensors is shown in Fig. 3.6.

For a test case, we created a simulation data comprising the three sensor values. First we generated the variable-C data randomly and used it as the basis for generating the other sensory data; hence, it is the "ground-truth" data. The data for sensor-A, which will be contained in variable-A, was generated by adding a noise to the data of variable-C. This illustrates a noisy measurement of the gyroscope sensors. Next we generated the data for sensor-B, which will be contained in variable-B, by averaging the data from sensor-A and

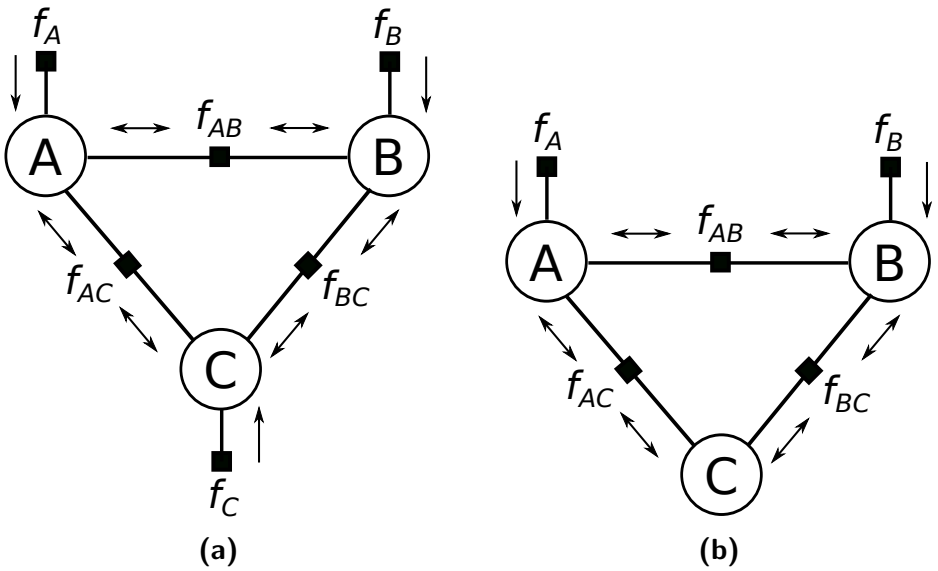


Fig. 3.6: (a) The acyclic factor graph for sensor fusion. In this network, all variable nodes are connected to their respective input nodes. The arrows indicate the messages flow during the training. Bidirectional arrows indicate that the corresponding messages flow in both direction at the same time although programatically they are implemented as two separated messages. (b) During the inference, the input for sensor-C is "unattached" and the product of messages flowing towards node C produces the "belief" about the value of variable C.

sensor-C. This illustrates a stochastic relation between sensor-A and sensor-C. The profile of these sensory data is shown in Fig. 3.7.

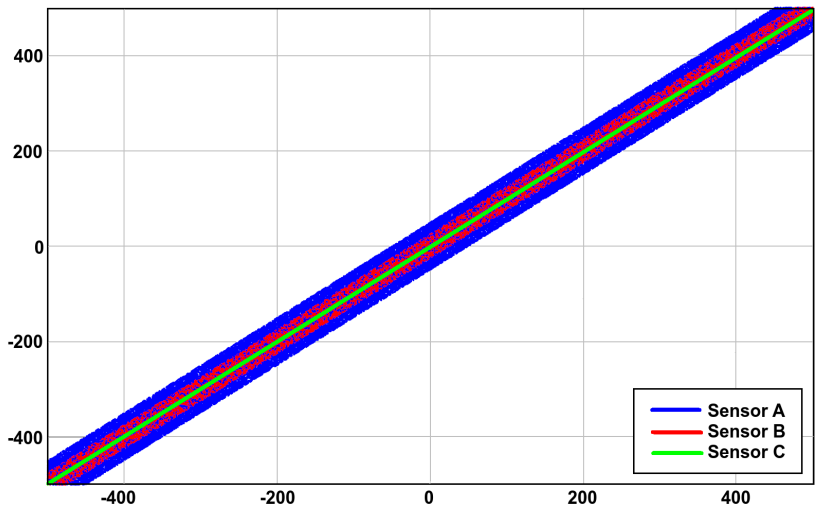


Fig. 3.7: The profile of sensory data used for training the network shown in Fig. 3.6a.

Similar to the case with acyclic factor graphs, during the training phase the factor graph learns the joint probability distribution as the internal function of its factor nodes. For the network shown in Fig. 3.6a, the LBP is used as a means of inference. Since the network

does not contain any hidden nodes, then the MLE described in section 2.2.3 on page 24 can be used for learning network's parameters. To train a factor node in the network, we break down the network (i.e. cut the loop) so that the factor will directly represent the joint probability distribution between the corresponding variable nodes. There is an alternative method to train the network without cutting the loop. However, such a method cannot be used directly with our belief propagation algorithm. In this thesis, we focus on the belief propagation using the standard message-passing algorithm. Hence, since the network has a loop, we have to modify algorithm 1 in a way such that the message-passing is kept running until it reaches a convergence condition, which can be detected by applying the KL-divergence (see equation (2.19)) to the old- and new messages of an edge with the same direction. Once the network reaches the convergence, the internal factor of a factor node can be updated and the process will continue until all training data points are fed to the network. In this LBP implementation, we update the factor node one by one while keeping the other factor nodes for being altered. Otherwise, the resulting factor values are not guaranteed to converge in the next update step. There is another approach to learn multiple factor nodes using EM algorithm described in [152] but this method is beyond the scope of this thesis.

Once the parameter learning has been completed, the network is ready to be used for inferences. In our sensor fusion setting, the inference procedure estimates the correct belief about the sensory reading by fusing the data from the available and connected sensors. To put it differently, this is basically a reversal process to estimate the "ground-truth". The plot shown in Fig. 3.7 depicts a noisy sensor profile (i.e. both sensors are noisy but contains an underlying generative function). By running the LBP and marginalize the messages running towards node C, we can get the robot's belief about its orientation. To test our approach, we generated a dataset containing two sensor values (sensor-A and sensor-B) in a sinusoidal shape. The result is shown in Fig. 3.8. It shows that the estimated robot's orientation has some degree of confidence level (depicted as the variance along the estimated result).

In this toy example, we demonstrate that our factor graph framework can run the LBP and yield the convergence result on a factor graph with a single loop. This result supports the claim that a factor graph with a single loop has a good chance to converge [93]. With this satisfying result, we believe that it can be extended into a more complex task such as estimating robot heading during movement that includes more sensory data.

3.2 Factor Graph for Dynamic Processes

So far we have discussed and shown examples of factor graphs in a "static" inference process. Similar to the other graphical models such as Bayesian networks, factor graphs can also be used to model a dynamic process of a system [153]. However, factor graphs do not have their own mechanism for handling such a dynamic process but they can represent the dynamic process using the underlying mechanism that originating from Markov chains (including the dynamic Bayesian network or DBN). It turns out that many algorithms can be re-interpreted using a DBN as its formalism (see section 2.1 on page 13). It is a common practice to transform a DBN into a factor graph representation and then perform

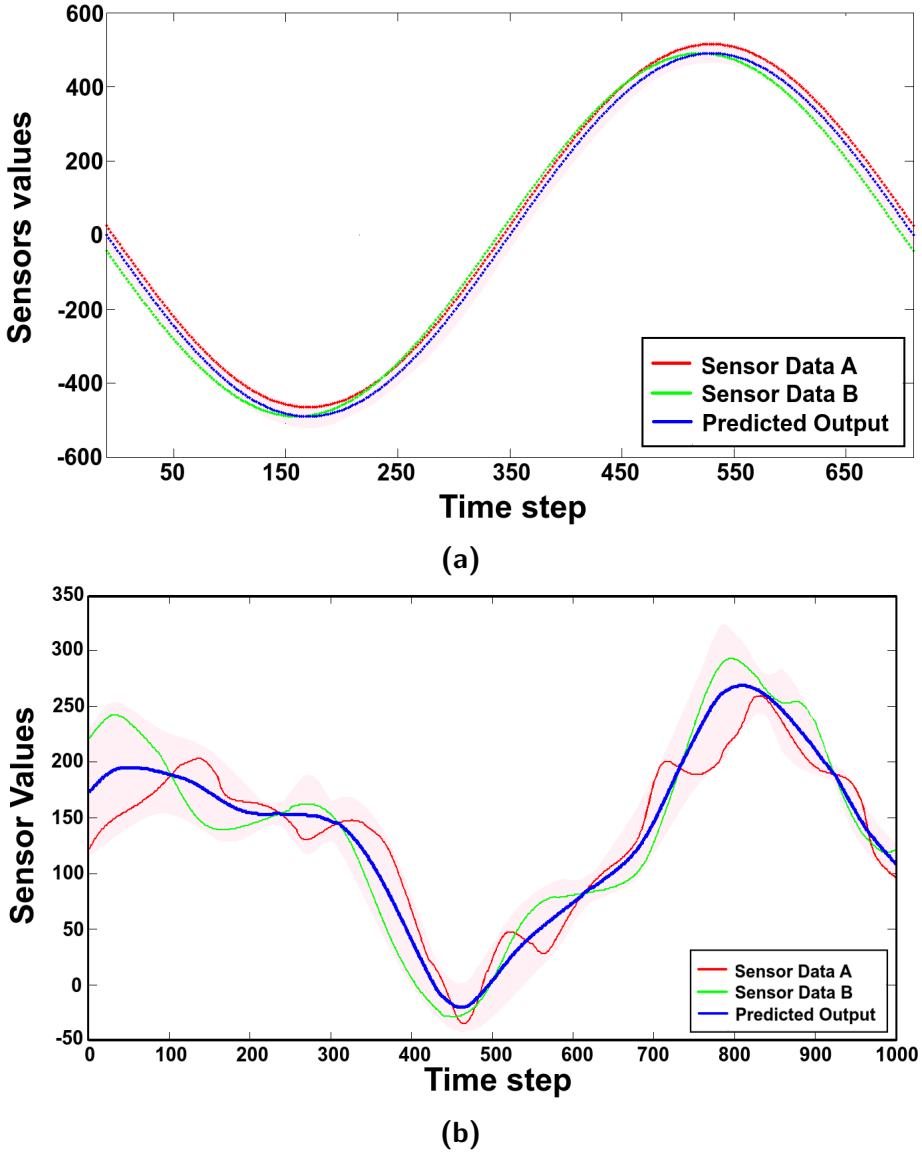


Fig. 3.8: The inference result for estimating the robot orientation given the sensory data from two different sensors. The blue curve is the estimated robot's orientation. (a) Sensor-A has a small offset from Sensor-B. (b) More noises are added to randomly generated values for sensor-A and sensor-B. As a consequence, the estimated result has bigger variance.

the inference on the resulting factor graph. This mechanism (using a factor graph to represent a DBN) is also used in this thesis.

To think about the dynamic process of a system, we can start from the standard representation of a dynamic system itself. In a discrete form, it can be expressed in difference equations as:

$$\begin{aligned} x_{k+1} &= \mathbf{A}x_k + \mathbf{B}u_k \\ y_k &= \mathbf{C}x_k + \mathbf{D}u_k \end{aligned} \tag{3.4}$$

The value of x is then calculated by summing points from initial/starting point $k = 0$ up to some value $0 < k \leq K$, and also considering the value at x_0. Here, the system is "unrolled" several time steps and the value K will determine "how-far" the system will run, which is commonly referred to as the horizon. Using the unrolling mechanism, equation (3.4) can be represented as a Bayesian network resulting in a dynamic version shown in Fig. 3.9.

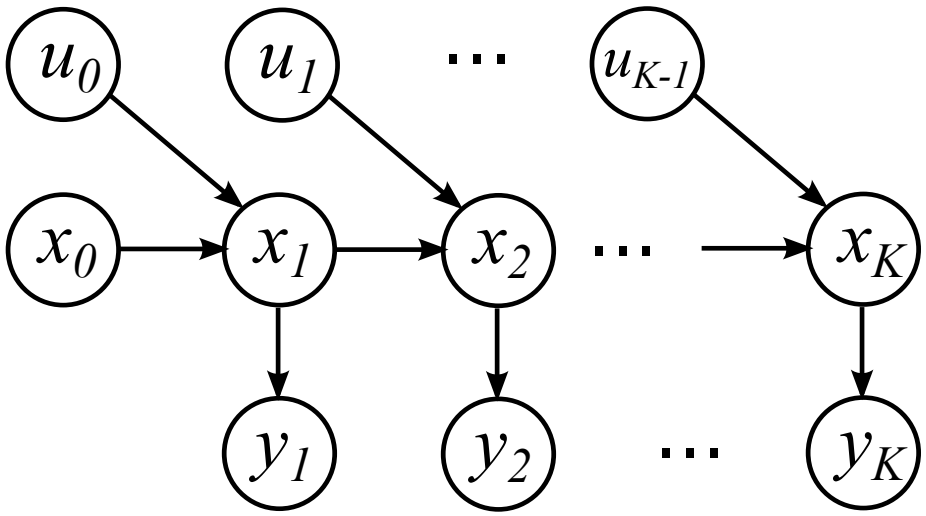


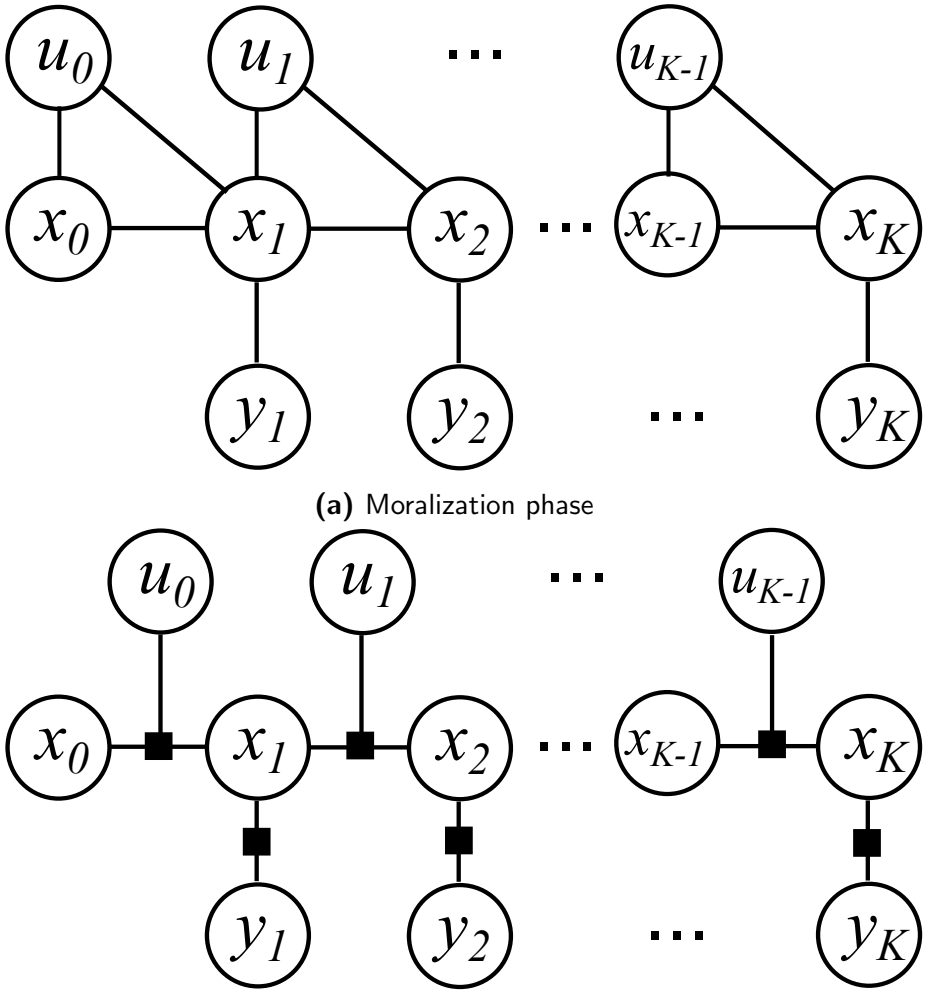
Fig. 3.9: A dynamic Bayesian network representation for a dynamic system expressed in equations (3.4)

.

As with a standard Bayesian network, transforming a DBN into a factor graph requires moralization steps which transform the directed graph into an undirected graph. After the moralization steps, the resulting undirected graph can be transformed into a factor graph. As described in section 2.2.1 on page 15, the factor nodes can be added per link basis or per maximal-clique basis. In general, it is preferable to transform a DBN into a dynamic factor graph in per maximal-clique basis so that loops can be avoided. The resulting dynamic factor graph for the DBN shown in Fig. 3.9 is shown in Fig. 3.10.

From this "default" dynamic Bayesian network, one can create many other networks with different characteristics such as hidden Markov models (HMM), auto associative (AR) models, Kalman filter models, state space models (SSM), etc. In probabilistic robotics, the DBN (and its factor graph version) can be used to characterize the evolution of controls, states, and measurements of the robot. This will lead to the popular Bayesian filter algorithm commonly used in a robotic SLAM (synchronous localization and mapping). We can relate the Bayesian filter algorithm with the sum-product algorithm as follows.

In probabilistic robotics, there are two main probability functions. The first is the state transition probability, expressed as $p(x_k \mid x_{k-1}, u_k)$, which specifies how the robot's internal and environmental states evolve over time as a function of robot control u_k. The second is the measurement probability, expressed as $p(y_k \mid x_k)$, which specifies the probabilistic law according to which measurement y should be observed when the robot is in the state x_k. This measurement probability is useful not only for modelling the sensor measurement but also for the noise which might presents during the measurement. The state transition



(a) Moralization phase

(b) Dynamic factor graph

Fig. 3.10: (a) The undirected graph as the result of moralization process on the DBN in Fig. 3.9. (b) Assuming the factor nodes are added per maximal-clique basis, the resulting dynamic factor graph does not have cycles which makes it easier to implement the sum-product algorithm on it. However, each factor node should represent a conditional probability distribution since the dynamic factor graph originates from a DBN.

probability and the measurement probability together describe the dynamical stochastic system of the robot (and its environment).

In the Bayesian filter setting, the state of the robot cannot be measured directly (hence, the DBN is called a HMM). However, the robot must maintain its own "beliefs" about itself and its environment. Those beliefs are represented as conditional probability distributions, and the belief distributions are posterior probabilities over state variables conditioned on the available data. The belief over a state variable x at step k conditioned on all past measurements $y_{1:k}$ and all past controls $u_{1:k}$ is expressed as:

$$bel(x_k) = p(x_k \mid y_{1:k}, u_{1:k}) \tag{3.5}$$

In the DBN setting, we assume that the states are complete; i.e. the knowledge of past states, measurements, or control inputs does not carry additional information that are relevant with the determination of the current state. It means that we can remove the current measurement y_t from equation (3.5) which yields:

$$meas(y_k) = p(y_k \mid x_k, y_{1:k-1}, u_{1:k}) = p(y_k \mid x_k) \tag{3.6}$$

Hence, equation (3.5) can be re-written as:

$$\overline{bel}(x_k) = p(x_k \mid y_{1:k}, u_{1:k}) = \eta p(y_k \mid x_k) p(x_k \mid y_{1:k-1}, u_{1:k}) \tag{3.7}$$

Equation (3.7) has a recursive form where the term $p(x_k \mid y_{1:k-1}, u_{1:k})$ is actually the prior belief similar to equation (3.5) before incorporating the new measurement y_k. The recursive form of the belief distribution is now become:

$$bel(x_k) = \eta \cdot meas(y_k) \cdot \overline{bel}(x_k) \tag{3.8}$$

where η is the normalizer constant which follows the probabilistic law enforcing that the maximum value is 1.

To implement the Bayes filter in a factor graph, we should look into the mechanism on how the posterior belief is computed from messages which are propagated towards the corresponding variable node. Fig. 3.11 shows a simple example where a DBN is unrolled three times. To compute the posterior belief on x_2, all nodes prior to x_2 will propagate messages which will arrive exactly before x_2 as the message $\overline{bel}_{x2}$. The measurement sensor y_2 will also generate a message and it arrives as $meas_{y2}$. The posterior of x_2, according to the sum-product algorithm, is computed as the product of $\overline{bel}_{x2}$ and $meas_{y2}$ which result resembles equation (3.8).

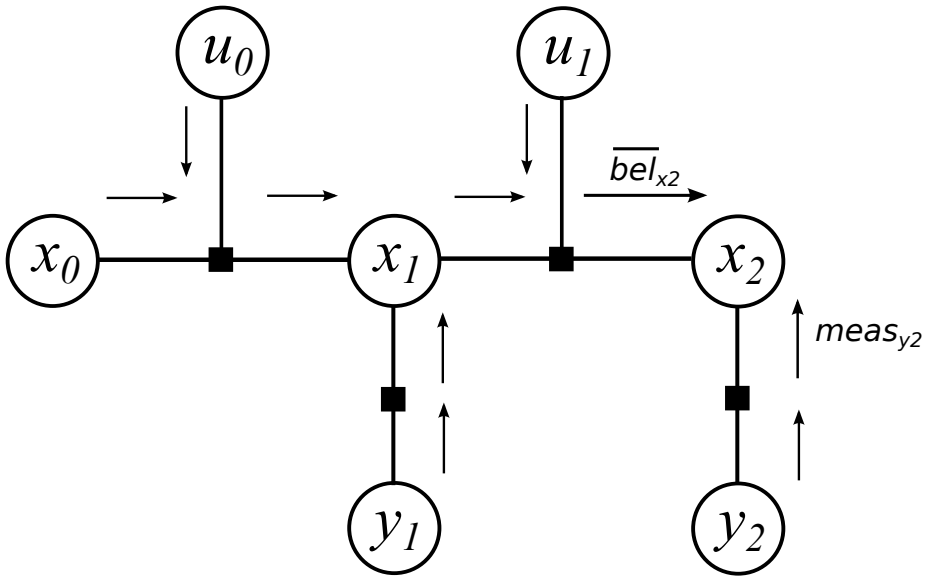


Fig. 3.11: An example of a factor graph that is used for explaining the Bayes filter action in a sum-product messages propagation.

The above Bayes filter mechanism is one of common queries that can be performed in a

dynamic factor graph. In this mechanism, the messages are propagated forward until they reach x_k. The other action that can be performed in the same graph is the smoothing. In this case, the massages are propagated backward (i.e. towards x_0). The combination of filtering and smoothing is similar to the forward-backward algorithm commonly used in HMMs. In HMMs, the state variables are discrete and usually do not have control nodes. In this case, the DBN shown in Fig. 3.9 is modified into a new structure shown in Fig. 3.12. In section 3.3.2, we give an example of how to represent a HMM along with its filtering and smoothing processes using our factor graph framework for modelling a robot manipulator.

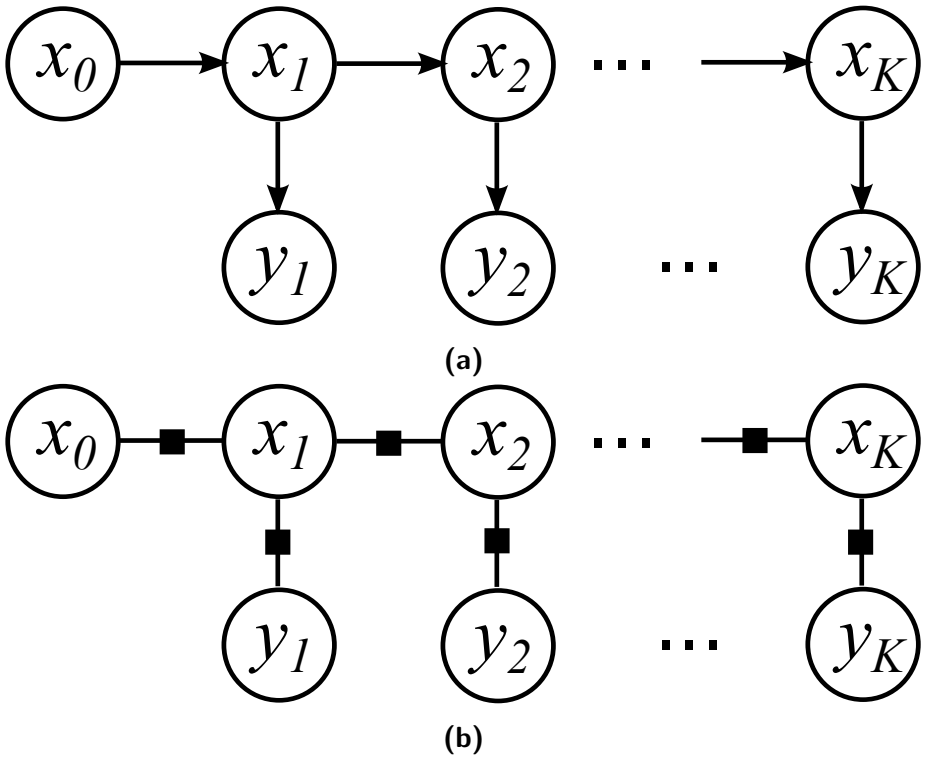


Fig. 3.12: (a) The hidden Markov model created as a special case of the DBN in Fig. 3.9. (b) The factor graph version of (a).

3.3 Application in Robotics

Many recent researches in robotics shift the focus from traditionally-specific industrial tasks to investigations of new types of robots with alternative ways of controlling them. As a result, robotics becomes one of the most prominent interdisciplinary researches around the globe. In the spirit of cognitive intelligence, we strive to find and apply a new and generic method that can enhance our mechanically-traditional robots so that they can help us to understand how intelligence is formed. Even though this sounds like a long term goal, we can start from a simple robotic system. In this section, we describe how we can use factor graphs as means for implementing such a method, and demonstrate that our proposed method is a generic one that can be extended into more complex robotic setup for future work in order to achieve such a long term goal.

We focus on two types of robotic systems: a mobile robot platform and a robotic arm (manipulator) platform. Regarding robot control for motion, we limit ourselves on the kinematic aspect so that we can focus on the evaluation of our factor graph framework in order to harness it later for more challenging and sophisticated tasks. We regard the dynamic aspect of robot control as our future work which is based on our envisioning method described in section 3.2. This is valid because once all relevant positions, velocities, and accelerations have been calculated using the kinematic models, methods from the field of dynamics can be applied on them to study the effect of forces upon robot movements. Not all aspects of kinematics are included in this thesis, but only the aspects of handling redundancy (different possibilities of performing the same movement) and singularity avoidance will be covered.

We follow the idea that a probabilistic graphical model can be an excellent tool for processing information on different levels of abstraction: from low-level sensory input to reasoning about high-level goal and action based on some cognitive architecture [18]. In this paradigm, lies the central problem of modern robotics which requires coherence principle of perception, decision-making, planning and control. In this section, we show that we can harness factor graph's flexibility with regard to the ordering of variables that plays an important role when dealing with the integration of many different models. As an example of how these different abstraction levels will take forms in factor graphs, a hybrid robotic system is used. A mobile manipulator is an example of such a hybrid robotic system whose mechanical structure exhibits the requirements of a hierarchical system example in the first place. Using this complex model, we can apply task constraints that allow robot movements to be controlled in a goal directed manner. Fig. 3.13 shows our mobile manipulator that can be used to demonstrate the flexibility of factor graphs for processing information hierarchically in a certain task constrained scenario.

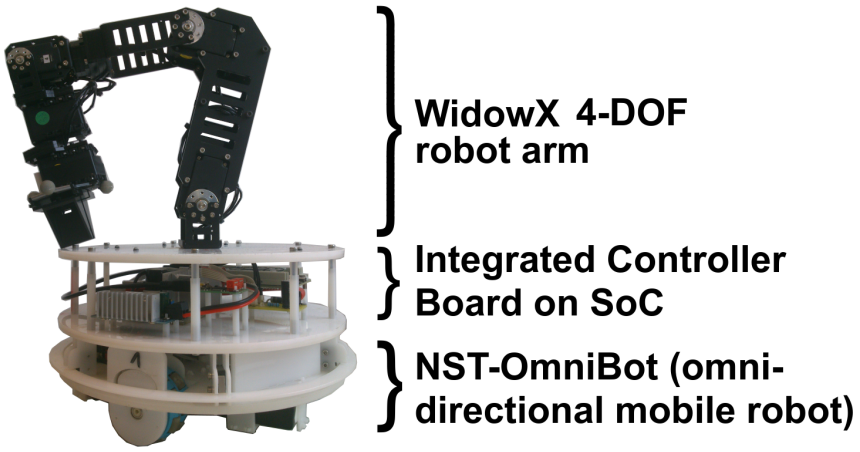


Fig. 3.13: The mobile manipulator developed in this thesis. It is composed of two subsystems: a 4-DOFs robotic arm and a mobile platform. The integrated controller board in the middle is where we implement our factor graph based controller in an SoC (System-on-Chip).

One typical application scenario for this robot is the pick-and-place task. It is true that, even in this simple scenario, the robot will face many technical challenges, such as how

to recognize the object, how to recognize the goal point landscape, etc. To simplify the scenario so that the reader can grasp our proposed method without much distraction, we assume that the robot has already known the object and how to grasp it. Hence, in this thesis, we focus on how to model the robot motion given the object already at hand.

3.3.1 Kinematic Model of a Mobile Robot

The first part of the hybrid robotic system shown in Fig. 3.13 is the mobile platform. It is a three wheel omnidirectional mobile robot (see Fig. 3.14). To perform the robot motion, we have to control the velocity of each wheel. For such a mobile robot, it is important to know its own kinematics before it can perform any high level tasks such as mapping the environment given a set of observations and to localize itself. There are two cases that we explore in our experimental setup: single motor control and three motors control.

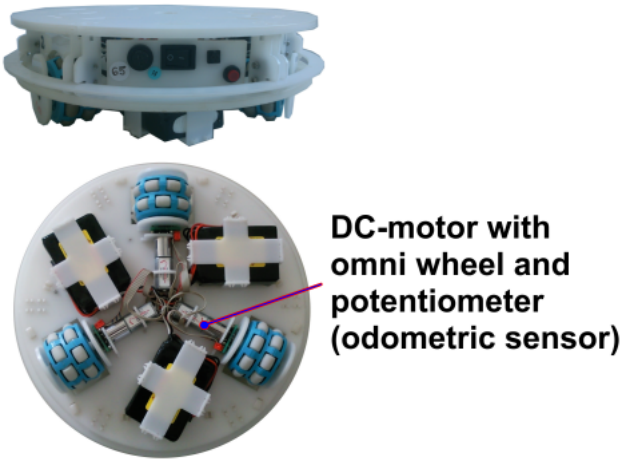


Fig. 3.14: The NST-Omnibot: a three-wheel omnidirectional mobile robot developed at the research group "Neuroscientific System Theory" (NST) in Technische Universität München. It has three independent DC-motors with a potentiometer attached on each motor which serves as an odometric sensor.

Single Input Single Output Motor Control

In this scenario, the motor control signal will be generated if a desired velocity of the motor is given. This is the scenario where we model the mapping function between wheel's sensory input and its motor output of a mobile robot shown in Fig. 3.14. In this scenario, due to the physical constraint of the robot, each motor will be independent from the others. For a single wheel robot control, the model is developed as shown in Fig. 3.15. In the model, node M represents the motor signal that generates motor rotation, and node V represents the sensory reading of motor velocity. For forward kinematics, the task of inference is to update the belief of node V given an input at node M as evidence. Since node V is only connected to the factor node f_{VM}, the output can be obtained from the message $\mu_{f_{VM}\to V}(V)$. For inverse kinematics, the task of inference is to update the belief of node M given an input at node V as evidence; i.e. by computing the message $\mu_{f_{VM}\to M}(M)$. Hence,

in this inverse kinematics, the network is supposed to generate proper motor commands given desired velocities of the wheel.

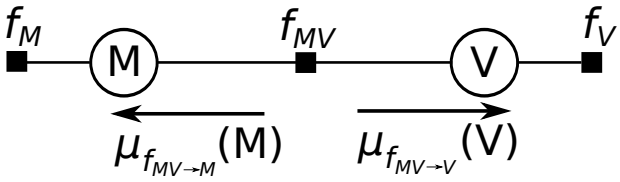


Fig. 3.15: The network for modeling kinematics of a single wheel motor control of the robot shown in Fig. 3.14.

The model shown in Fig. 3.15 basically computes the inference in a form of Bayesian network for modeling a DC-motor system. The implicit causality in that network obviously comes from the fact that if we provide the DC motor with a PWM-motor signal (indicated by variable M) then the motor will rotate at a certain speed which can be observed through the velocity sensor (indicated by variable V). For the forward kinematic task, its inference can be computed by a marginalization after computing the conditional probability:

$$p(V \mid M = m) = \frac{p(V, M = m)}{\sum_v p(V = v, M = m)}$$

This formula can be computed in a factor graph by sending a message from node V to node M. To treat variable V as the observed variable, we simply put one factor node f_V which takes V as its argument.

To compute the factor values of f_{VM}, the network is instantiated by sending N-samples random motor signals to the robot, which also results in N-samples sensory data from the robot. From these data samples, the factor value of f_{VM} is computed as the joint probability mass function of the variables V and M contained in the data. Fig. 3.16a shows the result of the inference for the network shown in Fig. 3.15, using only 40 states to represent a value in the range [-2500, 2500]. The results are very good and smooth even in the presence of noise in the training dataset. Fig. 3.16b shows the joint probability mass function corresponding to the factor values of the factor node f_{VM} of the model shown in Fig. 3.15.

In the context of system control, we are also interested in the performance of the system for the inverse kinematics case. In the inverse kinematics case, the system is supposed to calculate a proper motor command given a desired motor velocity (i.e. by calculating $p(M \mid V)$). Using the model shown in Fig. 3.15 and also using the same dataset for retraining the network (i.e. re-building the factor values of factor node f_{VM} to reflect the correct direction from V to M), we performed the inference again and the result is shown in Fig. 3.17.

As it can be seen in Fig. 3.17, the mapping $V \to M$ shows a linear trend with two small irregularities. The first anomaly lies in the center of the curve which reflects the intrinsic behaviour of a Bayesian network in dealing with over-fitting [16]. It shows that the network will smoothen the output against an abrupt change in the input. It also means that the network is able to minimize discontinuity in a function by taking the average of the values around the discontinuity point. The second anomaly lies at the both ends of

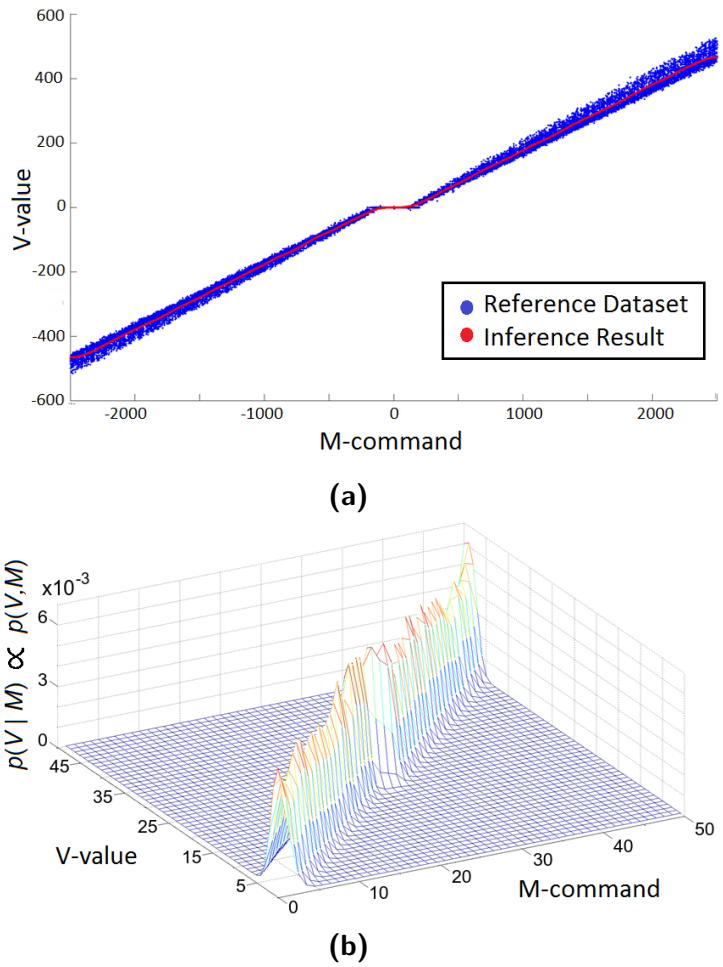


Fig. 3.16: (a) The forward inference result for the network shown in Fig. 3.15. (b) The joint probability mass function for factor node f_{VM}.

the plot. This is due to the low probability coverage on those regions as a result of using a Gaussian distribution, which is a non-uniform distribution.

Multiple Inputs Single Output (MISO) Motor Control

In this scenario, we developed a model that covers the whole body motion of the robot. The purpose of implementing a factor graph in this setting is to convey the full-body kinematic control of the mobile robot. The overall robot motion with respect to the world coordinate system is achieved by controlling the velocity of each wheel in a correct manner. This motion control in the world coordinate system is necessary because we want to give information about the current position of the mobile platform that influences the decision whether the hybrid system shown in Fig. 3.13 has to activate the robotic arm or not. In conventional control system theory, this is achieved by transforming wheels velocity into the robot velocity in the world coordinate system by using the following formula:

$$\mathbf{v}^{\mathbf{w}} = G \cdot \mathbf{v}^{\mathbf{r}} \tag{3.9}$$

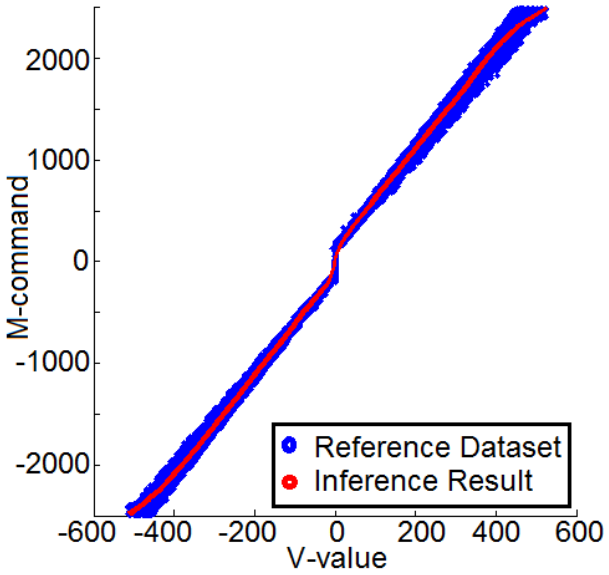


Fig. 3.17: The result of the inverse kinematics model. Here it shows the mapping $V \rightarrow M$.

$$\text{where } \mathbf{v^r} = \begin{bmatrix} \frac{\sqrt{3}}{3} & \frac{-\sqrt{3}}{3} & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{-2}{3} \\ \frac{1}{3L} & \frac{1}{3L} & \frac{1}{3L} \end{bmatrix} \dot{\mathbf{q}} \tag{3.10}$$

$$\text{and } G = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.11}$$

In formula (3.9) to (3.11), $\mathbf{v^w}$ is the robot velocity in the world coordinate system, $\mathbf{v^r}$ is the robot velocity in the robot-self coordinate system, $\dot{\mathbf{q}}$ is the vector of wheels velocities, and G is the coordinate transformation matrix which takes robot pose α as its argument.

Computing kinematics using the above formulas has at least two drawbacks. First, it relies heavily on deterministic sensor values which in reality will be easily disturbed by noises; hence, it is only good for simulations. Second, during the real implementation, the robot motion will be affected by some physical uncertainty such as frictions between the wheel and the floor which introduce a drift due to wheel slip. In other words, the exact relation between the three wheels' velocity and the Cartesian robot motion is unknown.

Our graphical model, on the other hand, offers a comprehensive way not only to overcome such limitations but also makes it more adaptive to the environmental changes. If we look into the formula (3.10), we can easily understand that each wheel contributes independently to the global robot motion. We capture this insight into our robot model. Our graphical model approach can be trained to map the Cartesian robot motion into the three wheels velocity of the robot. The robot shown in Fig. 3.14 can receive driving commands such as move forward/backward, move sideways, and rotate. Hence, we build a model that captures this kinematics relation as shown in Fig. 3.18.

Nodes $M_1 \cdots M_3$ shown in Fig. 3.18 represent wheel velocities, and nodes $\dot{\text{X}}$, $\dot{\text{Y}}$, $\dot{\text{R}}$ represent robot velocity in X and Y direction in the Cartesian coordinate system as well as

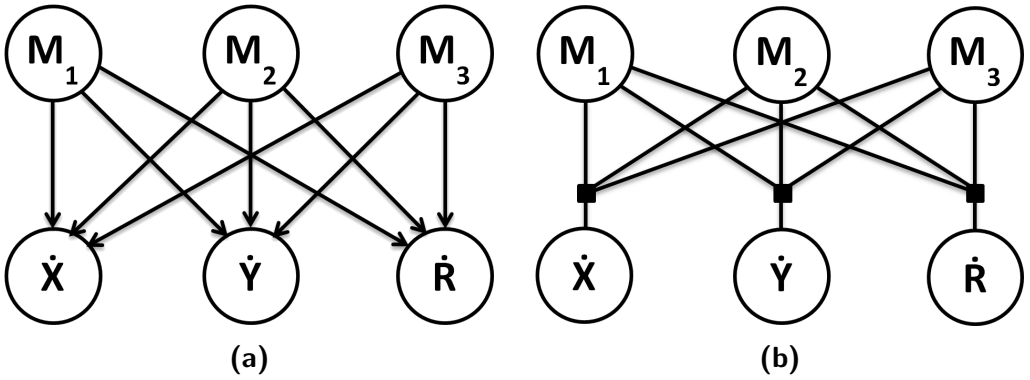


Fig. 3.18: (a) A Bayesian network model for the kinematics of the mobile robot. (b) The factor graph version of the model in (a).

its rotational velocity. Basically, the model represents a Multiple Inputs Multiple Outputs (MIMO) system. We can decouple the network into three separate MISO systems if we consider the flow of messages only in one direction, i.e. by assuming independence between scope variables of the factor. This mechanism will produce similar models for both forward and inverse kinematics since the structure basically represents a fully constrained model. Each factor of factor nodes in the models is computed as the joint probability mass function of its argument variables. The only difference between the forward- and the reverse kinematics model is the scope of the factor nodes. We denote this difference as f_M and f_{XM} as shown in Fig. 3.19. Thus, it needs to be retrained in order to get the correct parameters for each network before running the inference on them.

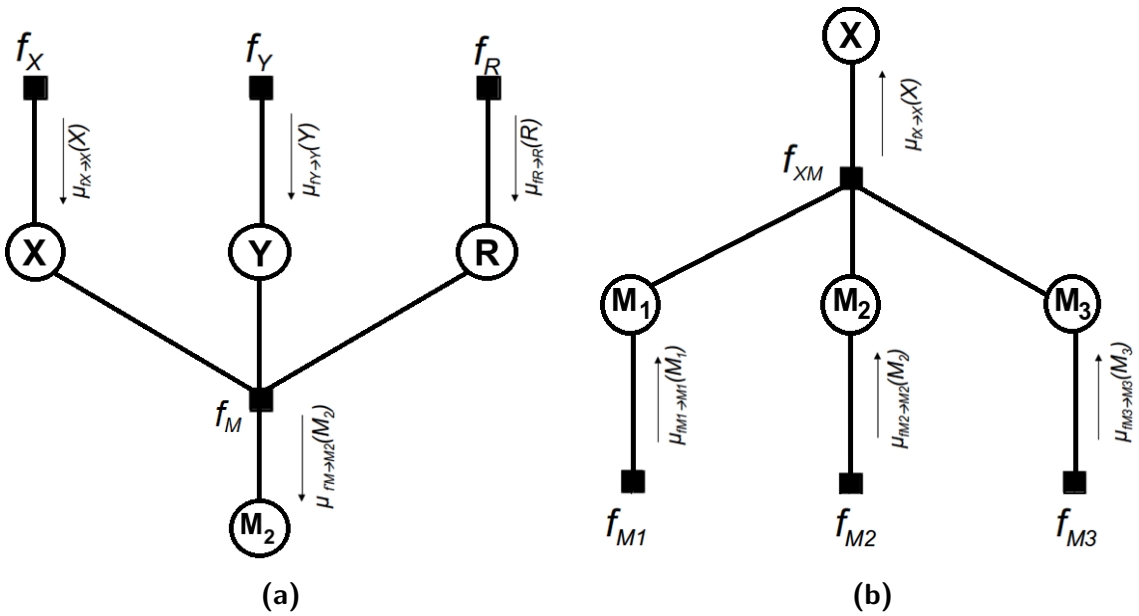


Fig. 3.19: Decoupling the network by assuming independencies given the observed variables: (a) for forward kinematics, (b) for inverse kinematics.

After designing the model structure, the next step is to determine the parameters of the model by learning them from data. To generate data for our model, we follow the

idea of motor babbling [154][155][156][157][158]. Motor babbling is a process of repeatedly performing a random motor command for a short duration in which the robot can autonomously develop a model that relates its internal states with its environment. In neuroscience, researchers show that during the early life of infants, the brain tries to learn the relationship between the real body motion and the intended motion developed in the motor cortex by generating brain signal targeting muscular system [159]. Although the motion is seemed to be random, it conveys the information about the effect of motor signal propagated into muscles and affected by the interaction with the environment. The result of this learning will influence later development of complex motion in human. We follow this idea by repeatedly generating a random motor command for each wheel of the robot and inspect its effect (i.e. robot motion).

It is interesting to study robot behaviours based on our factor graph model in a simulation environment because we will have full control on noises that artificially generated and introduced to the model. Hence, this simulation model might reveal important information such as how robust the model operates in the presence of noise. We are also interested to see the effect of the Gaussian distribution in our population codes for discretizing continuous values. The challenge of using Gaussian distributions for tuning curves in a population code is how to properly define the variance values of such distributions. The selection of the variance values is application dependent and it might produce different results for the same given task. For this purpose, we generated simulation data based on robot kinematics model expressed in equation (3.10). When we generated the dataset, we also introduced some levels of white noise in order to evaluate the robustness of our model in the presence of noise. Fig. 3.20a shows the generated random robot commands for the robot movement in X-direction (depicted as red-coloured plot) as well as the estimated motor commands by the factor graph model shown in Fig. 3.19b. It shows that the predicted commands are quite close to the originally-generated commands. By plotting the data in a correlation graph, we can see that there is a slight variance along the ideally-diagonal line as shown in Fig. 3.20b. However, the linearity of the curve is maintained within the valid input range. In general, similar results are obtained for the robot movement in Y-direction as well as for the rotational movement.

Fig. 3.21 shows the performance result of the inference against the presence of noise in terms of RMSE between the reference dataset and the inference result. It can be seen from the graph that the inference performance (both for forward- and inverse kinematics) degrades with the increasing of noise levels. The decreasing rate in the performance of the inverse kinematics model is a bit higher than that of the forward kinematics model. With the presence of noise up to 15%, the performance is still good and acceptable. With additional noises more than 15%, the performance becomes deteriorated. This information tells us that, in a real implementation later on, the model is capable of handling a Gaussian noise with zero mean and variance 1.38 [2]. Beyond that, our model might not be able to produce correct control signals for the robot.

The results demonstrated in Fig. 3.20b and Fig. 3.21 use the data from our simulator program. To evaluate our model further, we tested it in a real scenario. We performed

[2]This value is computed by subtracting the maximum Gaussian function of the data and the noise that should differ by 15%

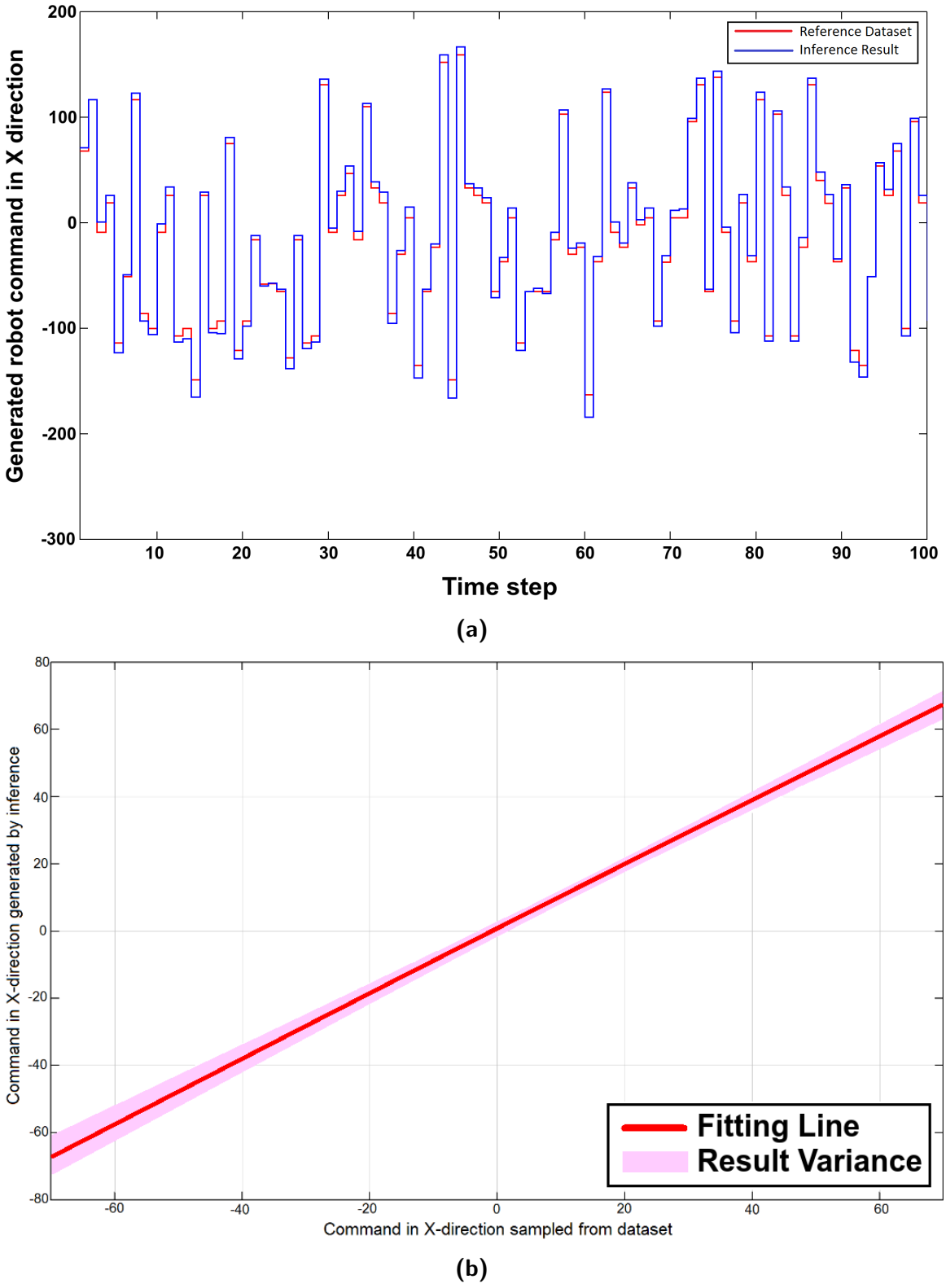


Fig. 3.20: Plot of generated motor commands given the desired robot velocities in the inverse kinematics case: (a) plotted in time sequence, (b) presented as a correlation plot.

an experiment by using a real robot. In this experiment, we used a camera tracking

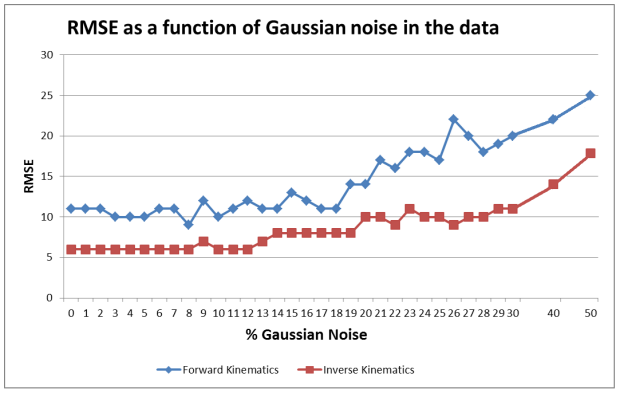


Fig. 3.21: The performance of the kinematics inference against the noisy data.

system to localize robot position in a planar space and to calculate its moving speed. The camera tracking system provides information about the position of the robot in the world coordinate system. We need to transform this absolute position value of the robot into the robot velocity value. Fig. 3.22 shows how the robot generates the data and records the trajectory.

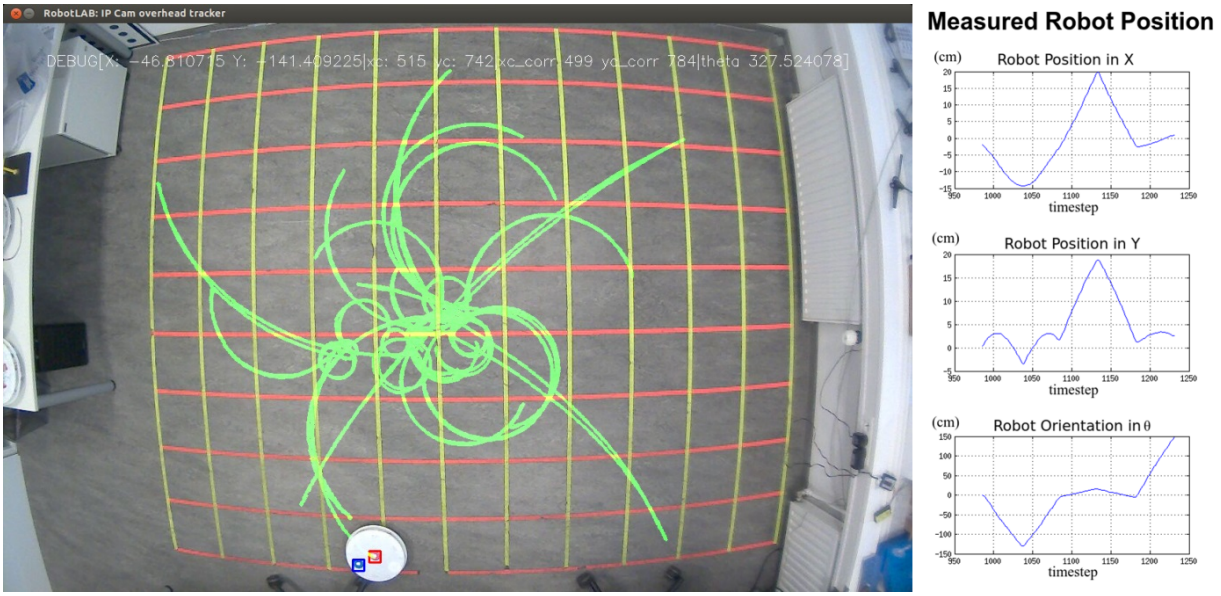


Fig. 3.22: The robot velocity data was generated using a motor babbling scenario. The camera tracking system records the robot trajectory which will be translated into the robot velocity data.

Before performing the transformation, the data were filtered to reduce the noise in the camera recording data. Filtering was done on the data that conveys information about the pose (position and orientation) of the robot. From our previous experiment using simulation data, we know that it is better for our model to have streams of input data with noise's variance (by assuming it is a Gaussian noise) less than 1.38%. After filtering, the Cartesian velocities as well as the rotational velocity of the robot were calculated. In our experiment, second order low-pass Butterworth filter works better that the Elliptic or FIR filter, although it runs slower and introduces a longer delay. However, since the

command is changed every two seconds during the experiment, it gives us enough sampling points where Butterworth filter's delay can be neglected. Fig. 3.23 shows the filtering action of the data from the camera tracking system.

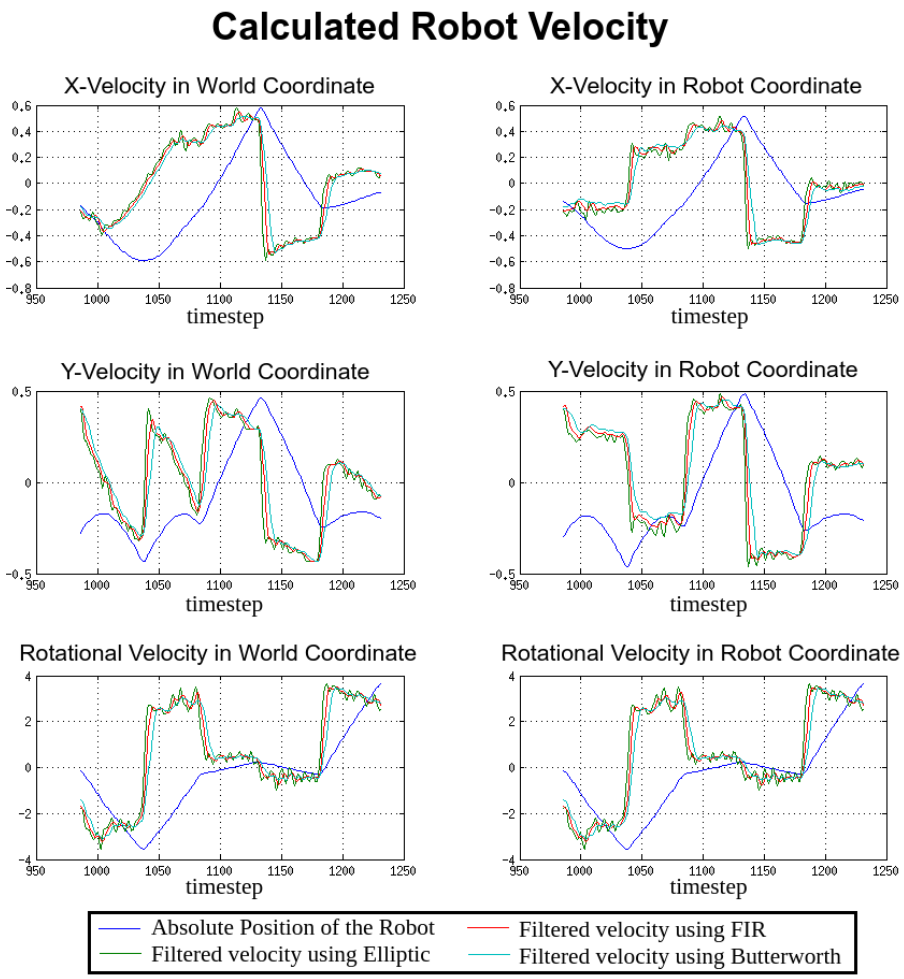


Fig. 3.23: Preparing the data before feeding them into the network. The raw data from the camera tracking system is very noisy. In order to reduce the noise effect so that its variance fall below 1.38, the data went through the filtering process.

The coordinate transformation matrix expressed in (3.11) is used to get the robot velocity in the robot-self coordinate system. Within the robot-self coordinate system, the velocity of the robot can be mapped properly into wheels velocities. During one iteration[3], we consider only 50% of the data in the middle of the period since both ends of the data-stream within that period contain a transitional fluctuation between successive iterations that should be neglected. Fig. 3.24 shows this process.

After collecting data and preprocessing them, we fed the data into the network shown in Fig. 3.18 in order to perform the inverse kinematics inference (i.e. to compute the robot's command given the desired robot velocity ($\dot{X}$, $\dot{Y}$, $\dot{R}$)). The result is depicted in Fig. 3.20.

[3]One iteration in this context means a period where the robot receives a valid command from the data acquisition program and produces a steady motion accordingly. In the experiment shown in Fig. 3.22, one iteration lasted for two seconds

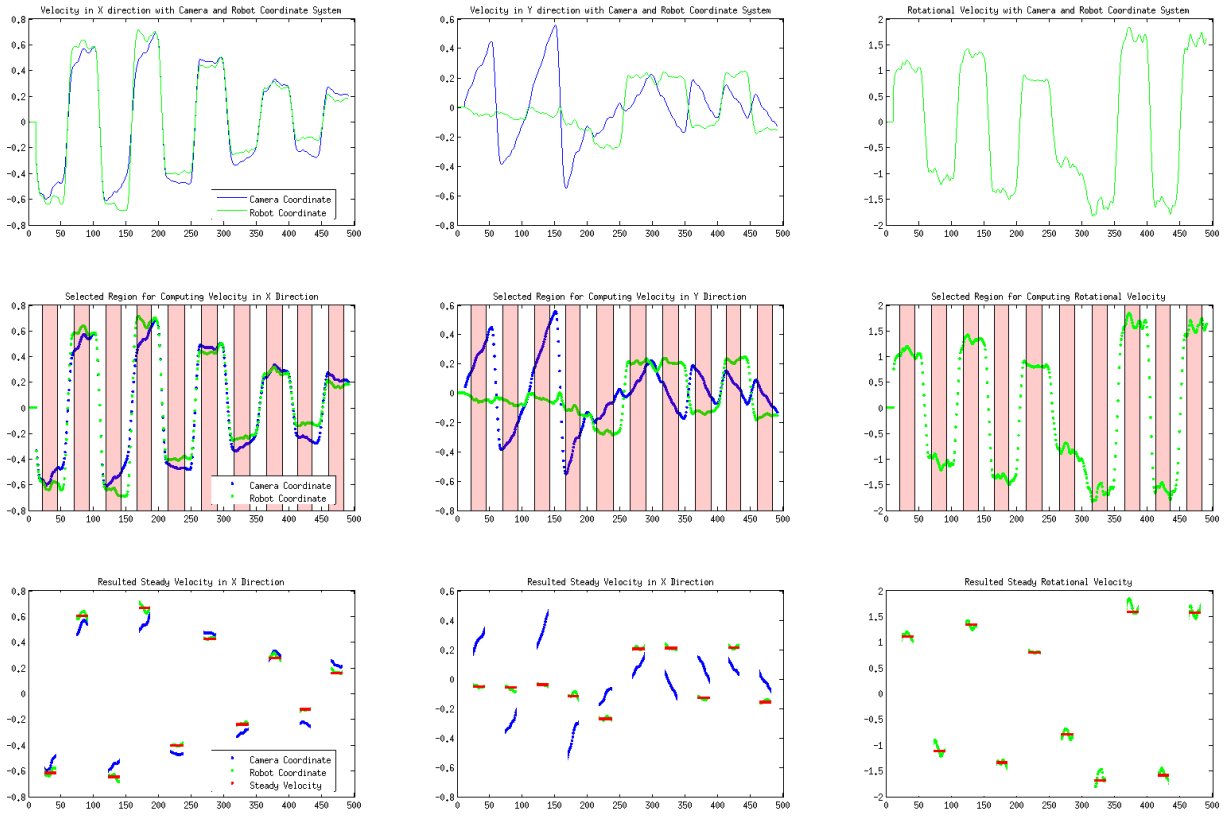
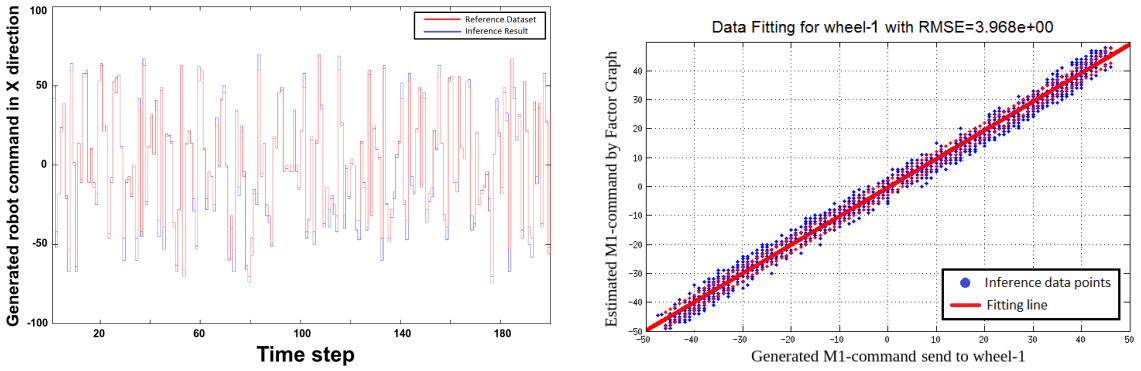


Fig. 3.24: The camera tracking system provides information about the robot's pose in the world coordinate system. In order to work with our model, the data need to be transformed into the robot-self coordinate system.

It shows a similar result, as expected, with the simulation version shown in Fig. 3.20. However, the variance is bigger due to various uncertainties in the experiment.



(a) **(b)**

Fig. 3.25: Plot of generated motor commands given the desired robot velocities in the inverse kinematic case: (a) plotted in time sequence, (b) presented in a correlation plot.

We are also interested to investigate the influence of the number of states (i.e. the variable's cardinality) to the performance of the system. Thus, we re-run the inference with several cardinalities, and the result is shown in Fig. 3.26. It can be seen from the

graph that the number of states (i.e. the number of neurons in the context of population coding) used by variables in the network influences the accuracy of the system. The graph also gives a hint on what variance value should be used in this application. Independent of this variance selection, it is interesting to note that the system does not require a big number of states to get a good result. It is sufficient to use 15 bins as the number of states in variables of the network shown in Fig. 3.18b.

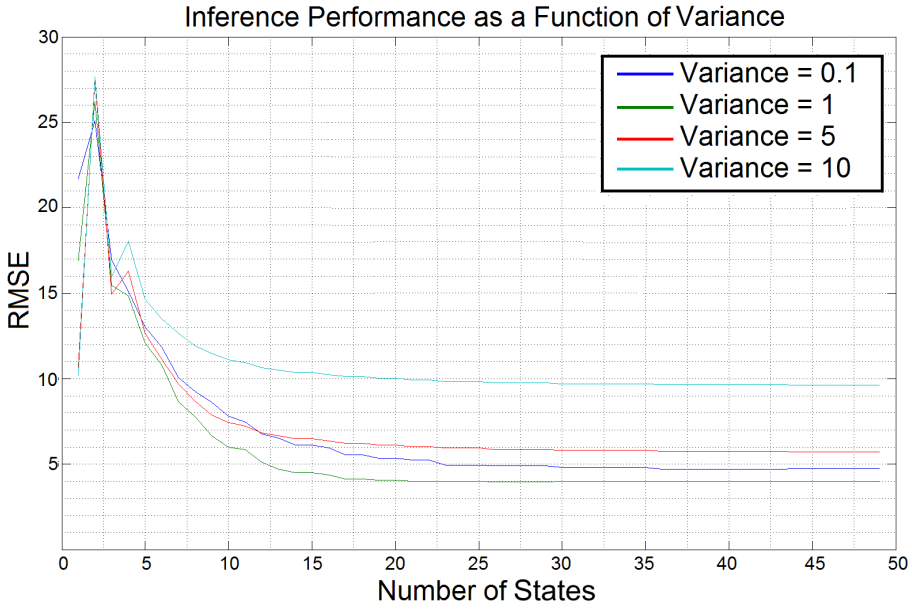


Fig. 3.26: The performance evaluation of the kinematic model for the mobile robot as a function of the number of states for each variable in the model. The number of states in a variable reflects the number of neurons in population coding to represent the variable's value. The Gaussian function is used to represent the variable's probability distribution. The graph shows that our model requires at least 15 states to produce good result.

With this example application, we present one important remark: the proposed method can be used to learn any mapping function without deriving its exact mathematical formulas. This heuristic approach gives us benefits especially when working with a high order dynamic system or in a dynamic environment. It is also preferable due to the fact that heuristics is used extensively in the manipulation and creation of cognitive maps, one of most fundamental properties of human intelligence. Cognitive maps are internal representations of our physical environment, particularly associated with spatial relationships. These internal representations of our environment are important and work as guidance to move in and to interact with external environment. Our kinematic model can be used in a more complex scenario that extends the basic functionality of the mobile platform, such as in the SLAM scenario. Nonetheless, this complex scenario is beyond the scope of this thesis since it requires more sensing components. We leave this extension as an opportunity for our future work.

3.3.2 Kinematic Model of a Manipulator

The second part of the hybrid robotic system shown in Fig. 3.13 is the robotic arm. It is a four degree of freedoms (DOFs) manipulator. In this thesis we use only its three DOFs since the last link does not determine the final position of the actuator (i.e. the robot gripper).

When developing a model of a robotic arm, the starting question is how the robotic arm will be configured or used. The standard task of a robotic arm is to manipulate the pose and the orientation of an object through its actuator. This is also valid for our example scenario with the mobile platform: we want the robot to move and orientate the actuator after the mobile platform reaches the correct position. Since each servo in the robotic arm has its own PID controller, we only need to develop the kinematic model of the robotic arm. Here we develop the forward kinematic model of the robotic arm using the similar model shown in Fig. 3.18 and modify the variables accordingly. The resulting model is depicted in Fig. 3.27. This forward kinematic model basically performs a mapping function from the joint space to the task space, and it considers only the ML problem instead of the MAP problem (i.e. there is no direct link between joint variables (θs) which implies that the model does not involve the interlink-dynamic between robot's joints, see Fig. 3.30a for comparison) [160].

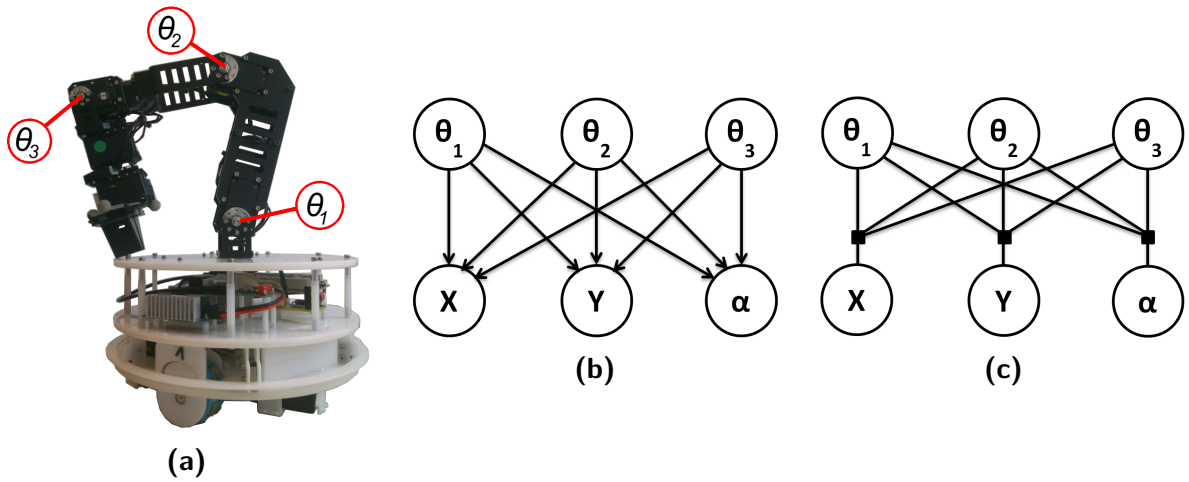


Fig. 3.27: (a) The robotic arm on top of the mobile robot along with its joint's labels. (b) A Bayesian network model for the kinematics of the robotic arm. Here θ_1, θ_2, θ_3 are the joints of the robot; X,Y is the Cartesian position of the actuator relative to the robotic arm base, and α is the actuator pose. (c) The factor graph version of the model shown in (b).

As in any standard forward kinematic model, we usually do not have problems with redundancy and singularity. However, the situation is completely different when we deal with the inverse kinematics. It is well-known that solving an inverse kinematic problem is very difficult. Sometimes the solution does not even exist. For our robot shown in Fig. 3.27a, there will be two solutions for the inverse kinematics given the input value X, Y and α: "elbow-up" and "elbow-down" configurations. One practical solution for our robot is to constrain the value of θ_2 to be always in the "elbow-down" position. This is

the most efficient solution for the robot because it is intended to be used in a pick-and-place scenario where the arm will always be in a curved-down position. This is also true because the second orientation of the gripper will be determined solely by the pose of the mobile base (see Fig. 3.28). With this constraint, we can use the same network shown in Fig. 3.27c to perform the inverse kinematics. Fig. 3.29 shows the result of the fully-constrained inverse kinematics of the robotic arm (it shows only the value of θ_1, where the value of the other θ are similar). It is perfectly linear with a small variance along the straight regression line due to the varying number of states used in the discretization of the variables.

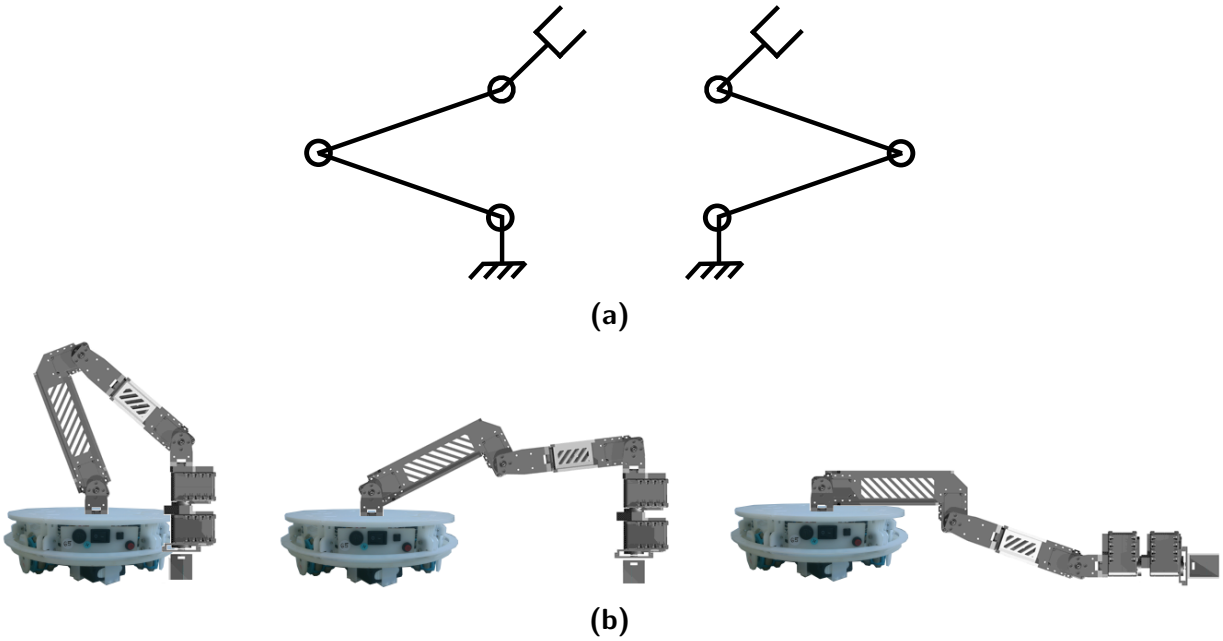


Fig. 3.28: (a) With the robot having 3 DOFs, it produces two different configurations either "elbow-down" (left) or "elbow-up" (right) for exactly the same actuator's pose. (b) The mobile robot platform helps the attached robotic arm to be easily configured in the "elbow-down" configuration. In this circumstance, the most efficient pose in a pick-and-place scenario can be obtained .

Sometimes the general solution other than the fully-constrained configuration is preferred. One common way to solve the inverse kinematic problem in robotics usually involves the computation of inverse Jacobian and its derivative, analytically or iteratively [161][162]. For a redundant robotic arm, however, the Jacobian matrix is no longer a square matrix; hence, it cannot be directly inverted and will require the computation of pseudo Jacobian. Unfortunately, the pseudo inverse solution is computationally expensive in most cases, and is subject to numerical instabilities around the singular configuration. In addition, multiple solutions will always exist for a redundant robot. Although those multiple solutions for a manipulator's inverse kinematic problem can be computed quite fast in an analytically-closed form using current computer technology, usually the number of DOFs is limited up to the sixth order [163] [4]. The iterative approach, on the other

[4]IKfast is an example of such an algorithm which is implemented in ROS (Robot Operating System) within the packet OpenRAVE, see http://wiki.ros.org/openrave

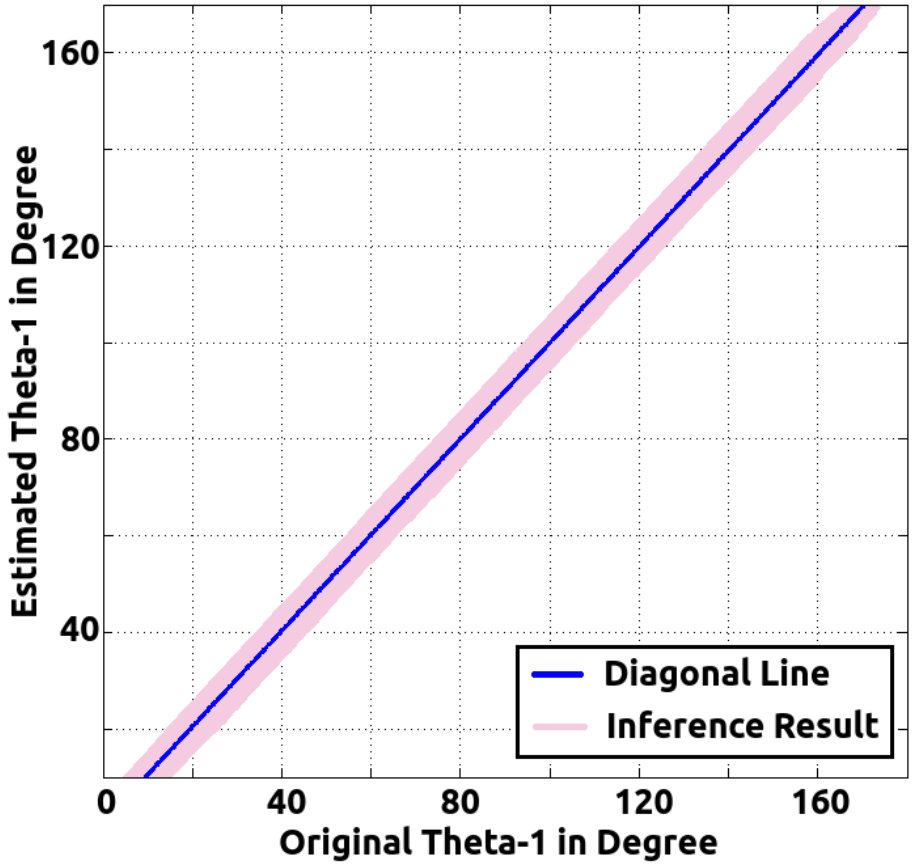


Fig. 3.29: The fully constrained inverse kinematics result.

hand, can be used to find a limited solution in any degree of freedom [164] [5]. Beyond this standard method, some other heuristic-based methods such as neural networks, fuzzy systems, and genetic algorithms also exist [165][166][167][168][169][170][171][172].

As an extension to our model shown in Fig. 3.27, we propose the second approach for solving the inverse kinematic problem using a dynamic factor graph. At some extent, this second approach can be considered as a heuristic approach similar to the iterative method, and it works as follows.

The basic idea of the iterative method is to approach the final state from a starting state by continuously examining the optimality expressed by the reduced cost function during each step. This action-sequence idea is not new and has already been used in robot kinematic modeling that tries to exploit the geometrical property of the kinematic chain. The implementation of this kinematic chain itself comes in several flavours. One example of these schemes is called CCD (Cyclic Coordinate Descent) [173], which is an iterative algorithm quite popular in many computer graphic and games industries. The CCD method iteratively minimizes errors by evaluating one joint variable at a time, starting from the end effector inward towards the manipulator base until a convergence point is obtained. The solution offered by the CCD method depends on the initial posture, and can only provides a feasible posture if manipulator constraints for restricting motions are

[5] KDL is a package in ROS which provides an iterative-based approach for solving the manipulator inverse kinematic problem, see http://wiki.ros.org/kdl

incorporated. The similar approach implemented in a Bayesian network setting is proposed by Sturm et.al. [174]. In their approach, Sturm models the connectivity of rigid parts that constitute the object, including the articulation model of the individual link. Another biomimetic approach for solving an inverse kinematic problem that is inspired by a human upper limb is proposed by Artemiadis et.al [175]. However, their method is not a complete probabilistic graphical modeling treatment since they use the Bayesian network only for describing the dependencies among the human joint angles, and then use an objective function in a standard Jacobian inverse kinematics to finally solve the problem.

In this thesis, we also follow the idea of mimetic approach which is based on our interpretation of how a human arm is usually moved during the task of reaching and placing an object. When a human moves his hand to grasp an object, first he makes some initial posture estimations of his hand, and later adjusts his forearm and upper arm until the object is reached. With this insight, we develop a graphical model for the robot kinematics using a Markov chain model shown in Fig. 3.30. In this model, we simplify the representation of the variable Θ into the corresponding θ values since it is basically a Markov chain network unrolled three times.

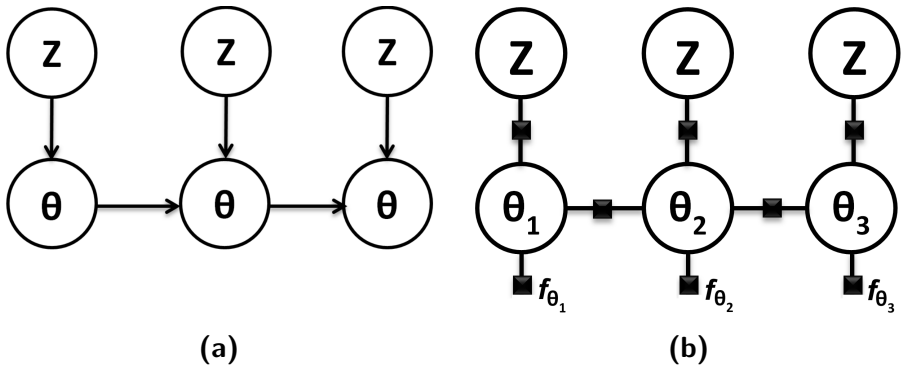


Fig. 3.30: (a) A Markov chain network for modeling inverse kinematics. Here variable Z represents the pose of the robot actuator (i.e. the gripper). The pose variable Z can be decomposed into the actuator Cartesian position X,Y and the actuator orientation α. (b) The factor graph version where each factor represents the conditional distribution associated with an edge in the model (a). The factors f_{θ_1}, f_{θ_2}, and f_{θ_3} are the estimated actuator's joints.

The network shown in Fig. 3.30 works as follows. It iterates between forward and backward phases until some convergence values are obtained. The forward phase is started by sending the desired actuator's pose (Z value) and the current actuator's joints (f_{θ_1} value) to the variable θ_1. It then produces a message which contains the prior belief about θ_1 given the desired pose to the variable θ_2. The variable θ_2 then produces a message to θ_3 incorporating all prior beliefs about θ_1 and θ_2 given the desired actuator's pose. The backward phase begins by generating a message from variable θ_3 which will be propagated and modulated towards θ_2 and θ_1. The posterior beliefs about θ_1, θ_2, and θ_3 are obtained by multiplying the incoming messages to the corresponding variable. These will become the new estimated joint values which will re-enter the network through f_{θ_1}, f_{θ_2}, and f_{θ_3}. The network iterates until all joint values converge into steady values.

To test the performance of the network, we created a simple task in which the robot follows a rectangular trajectory. We have tested the network by using both simulation data and the real robotic arm. The rectangular trajectory occupies a planar space within the range [10, 30] cm on both directions (X and Y), sampled every 0.5 cm with the actuator orientation kept constant at 0°. In the simulation environment, those sampled points were fed into the inverse kinematic network, and the computed f_{θ_1}, f_{θ_2}, as well as f_{θ_3} are fed into the forward kinematic network shown in Fig. 3.27c. Using the real robot, the same process was repeated for calculating the inverse kinematics, but the resulted f_{θ_1}, f_{θ_2}, and f_{θ_3} were sent directly to the robot. Fig. 3.31 shows the results of this experiment.

In summary, we present a generic model which can be used for both forward and inverse kinematic computation. The physical constraint of our robot makes it possible to produce a satisfactory result because it can be modelled in a fully-constrained configuration; hence, reducing redundancy and providing a single valid solution. Conceptually, the other solution can also be computed but it will not give any benefit to our robot. We also present the second model which is based on a mimetic approach. We have applied the second model on the robot to follow a simple trajectory. The result shows that it is quite reliable in a simulation environment. In the next section, we explain the combination of both the factor graph models of the manipulator and the mobile platform into a hybrid robot system.

3.3.3 Model-based Learning for Mobile Manipulator

Programming the robot to carry certain task in a factory or in a daily life service is tedious work. Conventional method will usually involve precise mathematical formulation of the task followed by some fine-tuning parameters of the preprogrammed robot system. Even in a simple robot system, this routine will be cumbersome in a frequently changing tasks setting. This motivates many researchers in AI and robotic systems to find a more compliant/relaxing way to handle this situation. The idea is simple: teach the robot to generate motions based on human experience. It is a straightforward observation that humans can easily change from task to task without too much effort, revealing the fact that humans have a long history of adapting the model-based learning features. These model-based features are also believed by scientists, leading to the conclusion that intelligent mammals rely on their own internal models in order to generate their actions. While conventional robotics relies on manually generated models that are based on human insights into physics, it is also predicted that future autonomous, cognitive robots need to be able to automatically generate models that are based on information perceived by the robot [8].

New findings in cognitive neuroscience suggest that the human brain maintains the integral model of the human internal states along with its external states of the environment in a distinctive manner. It is believed that the brain produces the optimal policy (to generate proper actions from the current states) based on its internal model and use the model-based learning paradigm to maintain that model in a robust manner. One example model is based on the cerebellum structure which distinguishes the forward from the inverse model [176]. There are many evidences which support the idea that there exist two distinct systems for producing those optimal policies: model-based and model-free system. Unfortunately, there is little evidence showing how the brain actually determines which of these systems control behaviour at one moment in time [177][178]. In a model-based

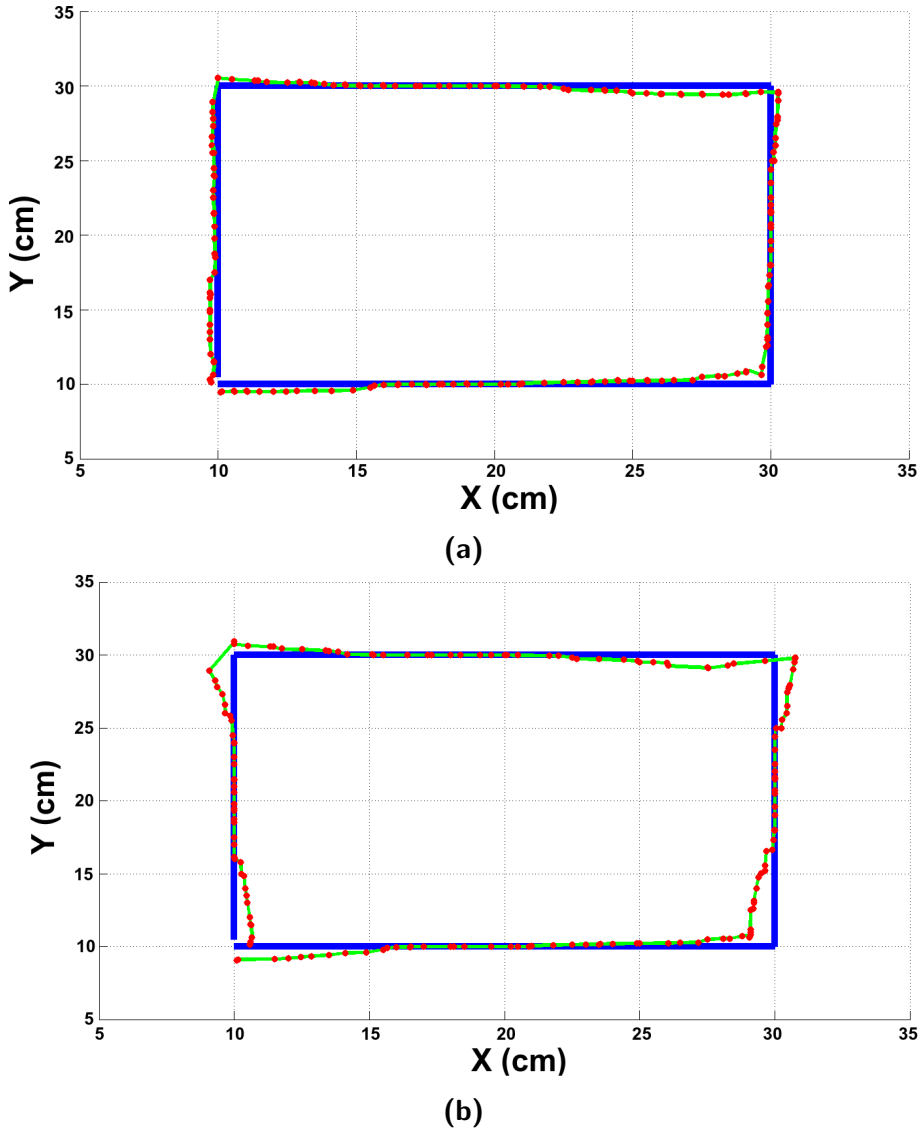


Fig. 3.31: (a) The inference result of kinematic models in a simulation environment. The outputs of the inverse kinematics network shown in Fig. 3.30b are sent to the forward kinematics network shown in Fig. 3.27c. The blue line is the trajectory for the robot to follow and the red dots which are connected by the green line are the resulting trajectory. (b) The experiment was conducted using the real robot. Although it produces a visible trajectory, it is not a perfect rectangular one. However, this impreciseness is merely due to technical problems and noises in the robot hardware for data communication.

approach, the agent can predict the consequences of actions before they are taken. It is similar with generating virtual experience and then performing the search through a problem space to find the most efficient solution. This model-based approach, which requires a predefined model before the model can learn the optimal policy deliberately, integrates both learning on the basis of past experiences and planning future actions [179]. On the other hand, the model-free approach does not require a specific predefined model, and the agent attempts to learn the model in a reflexive training scenario; hence, it is more

complex and challenging since the internal model of the agent as well as the environment model are unknown.

In this thesis, we are interested to explore the model-based approach because it offers several major advantages, including the opportunity to create highly tailored models for specific scenarios, as well as rapid prototyping and comparison of many alternative models. As an example test case, we developed a robotic application in a placing-an-object scenario. In this scenario, the robot moves an object from one point to another point in space where the distance may be far, so that the robot's arm is unable to put the object directly at the destination point. For this scenario, we developed a mobile manipulator - a hybrid robotic system, where a robotic arm is installed on top of a mobile robot, as shown in Fig. 3.13.

This hybrid system is challenging because the robot should maintain its stability while moving (e.g. by aligning the arm so that the overall center-of-mass will be close to the center of the whole body of the robot). In our experiment, the robot operation does not start by picking up an object because it requires complicated scenarios (e.g. object shape/pose recognition), see for example [180]. Instead, the object is placed into the gripper of the robotic arm. We defined a session as a single placing-an-object task. In the beginning of the session, the original coordinate of the object's center and the goal position is given. If required, the robot should move to the point where the arm is able to reach the object. When the robot is in the right position, the robot lifts the object and move the arm, so that its overall center-of-mass is in a stable position. The robot then approaches the destination position. When it is close enough, the arm is moved (i.e. stretched), so that the object can be placed at the goal position. To reduce the complexity, so that we can focus on our factor graph evaluation, we do not deal with the synchronous localization and mapping scenario. Our proposed model is shown in Fig. 3.32.

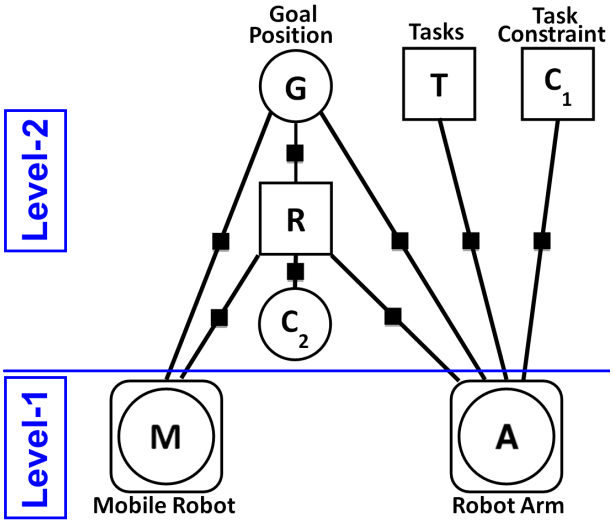


Fig. 3.32: The graphical model of our mobile manipulator shown in Fig. 3.13.

Level-1: Motion Primitives of Robotics Platform

Shown in Fig. 3.32, we define two levels of the model that create a hierarchical solution: the lower part (Level-1) corresponds to the low level robot motions and the upper part

(Level-2) corresponds to the strategical motion specific to a given task. In this sub-section, we explain the idea of motion-primitives generation by the model in Level-1.

Level-1 is responsible for the low level learning of individual motions, where individual actions could be taught separately instead of all at once. These individual motions are basically motion primitives which are kinematically feasible to form the basis movement performed by the robot platform. The motion primitives of the mobile robot have been learned using the method described in section 3.3.1. Hence, in this sub-section, we explain only the remaining model for the motion primitives of the robotic arm.

Different methods for learning the motion primitives have been developed by many other researchers using algorithms such as Gaussian Mixture Models and Regression (GMM/GMR), Hidden Markov Model (HMM), Support Vector Machine (SVM), Locally Weighted Projection Regression (LWPR), Dynamic Movement Primitives (DMP), etc. [181][182][183][184][185]. In this thesis, we turn our attention to the paradigm of programming-by-demonstration (PbD) or commonly known as the imitation learning. In this paradigm, the robot is expected to develop and improve its skill after some demonstrations provided by the human teacher. The robot then creates the skill model using some learning algorithm that exploits the statistical regularities across multiple observations [186][187]. Fig. 3.33 shows the basic principle of learning new skill using the demonstration paradigm.

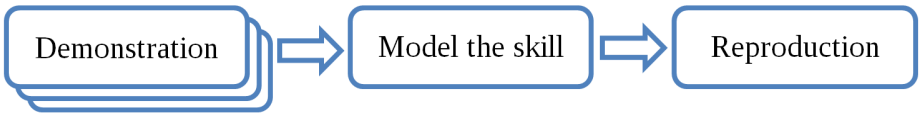
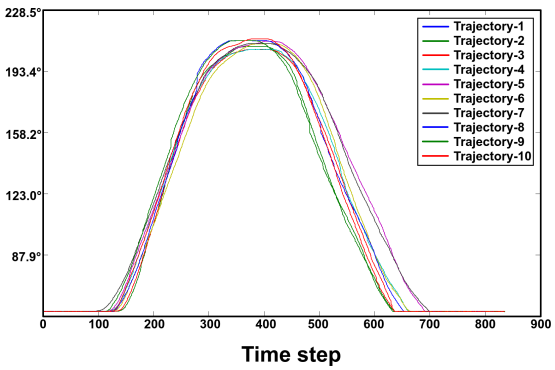


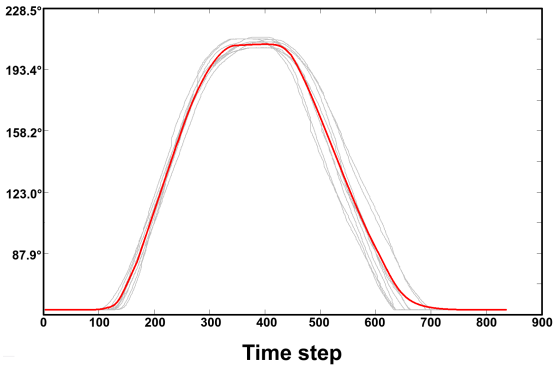
Fig. 3.33: The conceptual principle of programming by demonstration.

The most intriguing challenge in this paradigm is how the model generalizes the skill so that it can also be applied in different contexts [188][181]. Skill learning itself can be developed either at a symbolic level or a trajectory level. Here we are interested in the trajectory-based approach for two reasons. First, many of our factor graph models originate from Bayesian networks which do not permit any cyclical hierarchy. Second, we are still in the early stage of research in which we do not want to be distracted with complicated high-level skill problems. Hence, we stay focused on the trajectory level and evaluate the performance of our approach as much as possible in order to ensure the continuation of our work. Although there is no consensus on which method performs the best when dealing with the generalization problems at the trajectory level, most practical PbDs are usually performed using either statistical modeling (such as GMM and HMM) or dynamic-system-based modeling (such as DMP). In this thesis, we use a regression technique similar to the Gaussian Mixture Regression (GMR) [189] but we create the model entirely in a factor graph and use its inference mechanism to learn the trajectory.

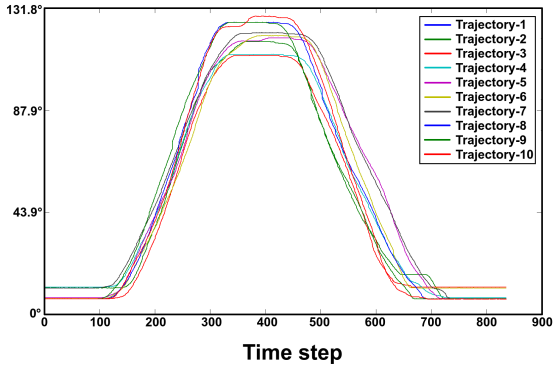
In section 3.3.2 we describe a model that calculates the kinematics of the robotic arm. To use the PbD paradigm for generating the skill, we need to provide several demonstrations from which the trajectory of the movement can be learned. Each demonstration trajectory is fed into the regression network and the parameter of the network will be updated accordingly. Fig. 3.34 shows the trajectory result from the regression network over several demonstrations for each joint of the robotic arm.



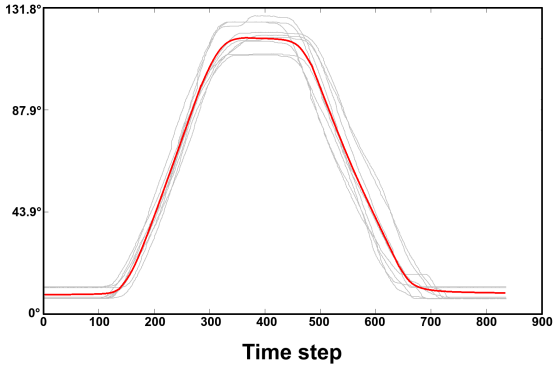
(a) θ_1 demonstrations.



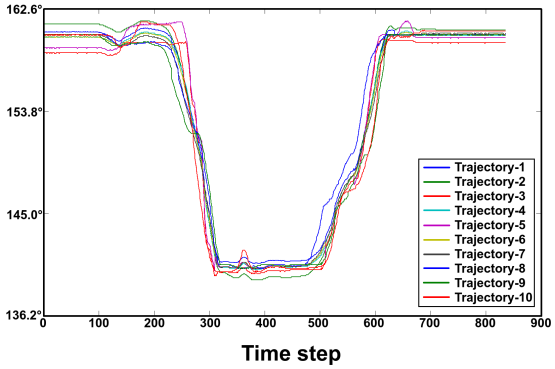
(b) Estimated θ_1 trajectory.



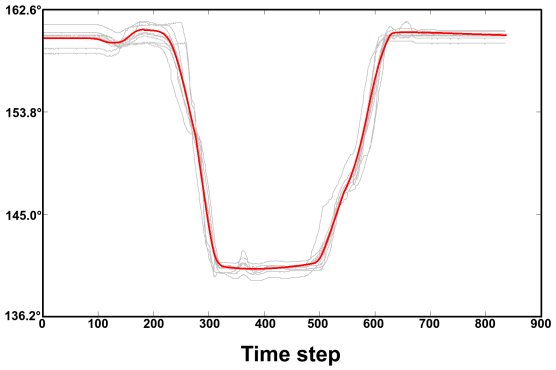
(c) θ_2 demonstrations.



(d) Estimated θ_2 trajectory.



(e) θ_3 demonstrations.



(f) Estimated θ_3 trajectory.

Fig. 3.34: Learning robot trajectory from several demonstrations for each joint of the robotic arm (θ_1, θ_2, and θ_3 correspond to the joints of the robotic arm shown in Fig. 3.27a).

Fig. 3.34 shows only a snapshot of the teaching of the robot where we hold the robotic arm and then extend and retract the arm to create a trajectory. In general, any sequential actions can be performed and learned by using this PbD paradigm. As an example for a complex trajectory, we "guided" the robotic arm to pick up an object from one position and then place it on another position. This scenario is depicted in Fig. 3.35. After several demonstrations, we perform the regression using the same mechanism shown in Fig. 3.34. The estimated trajectory is then sent to the robot's kinematic controller. The snapshots of this run after learning the trajectory is depicted in Fig. 3.36.

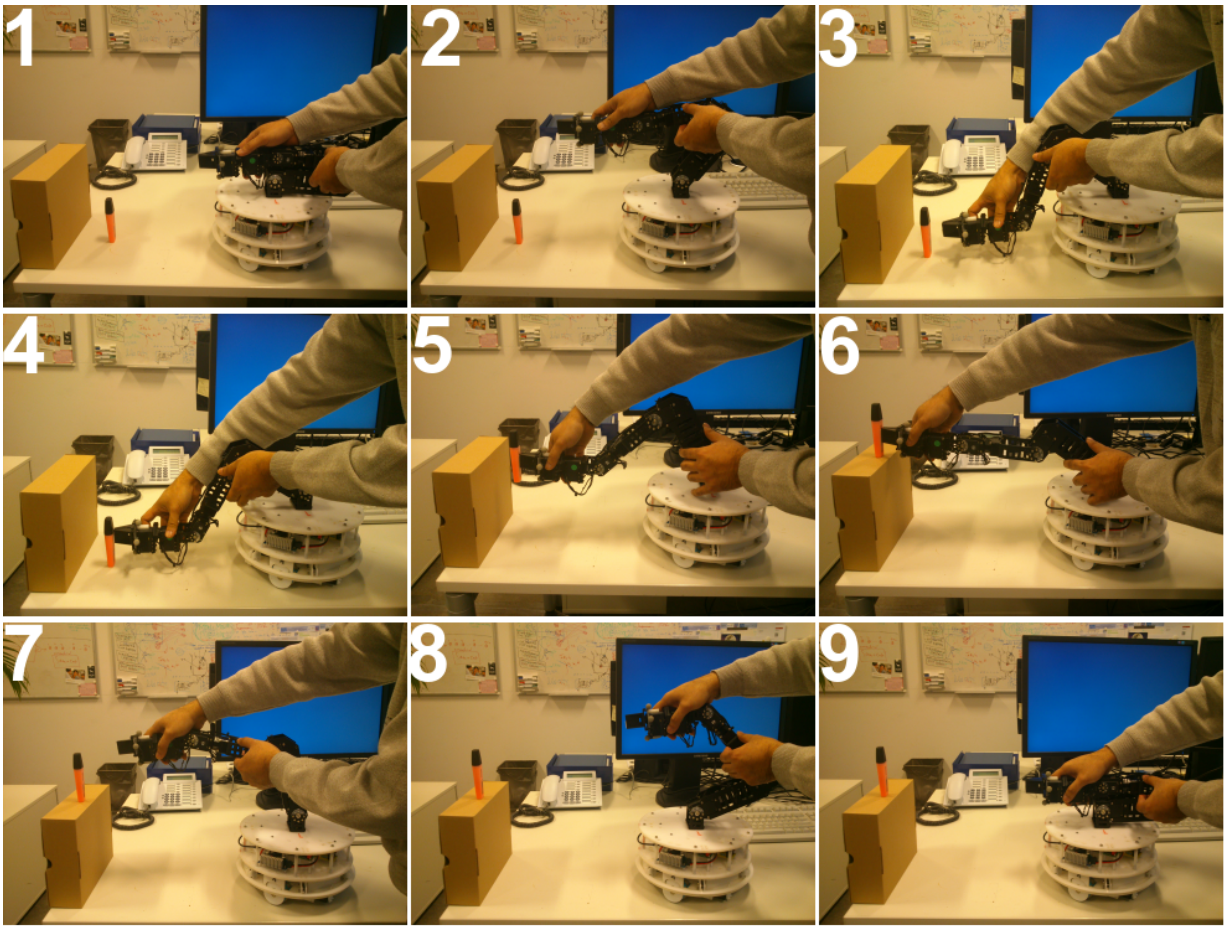


Fig. 3.35: Guiding the robotic arm to follow a trajectory.

Once the robot learn the skill (i.e. the trajectory), it can use the standard scaling procedure to get different effects such as the different start and goal positions. However, this scaling mechanism is a static procedure and will not take into account the transitional dynamic between two successive learned skills[190]. A better way to accommodate the dynamic behaviour of skills is by using the dynamic imitation paradigm [185][191]; however, we do not cover this advance technique in this thesis and we leave it as an opportunity for our future work.

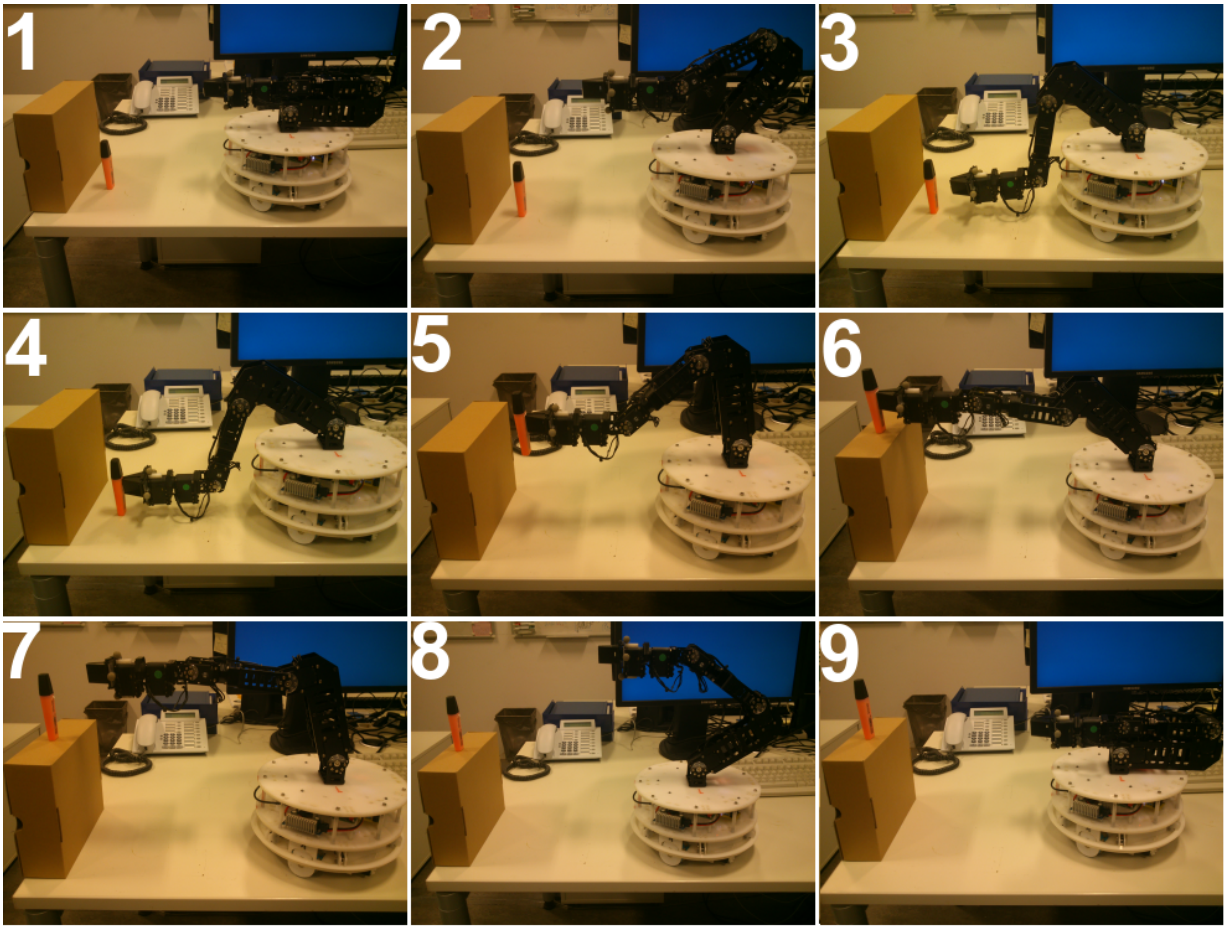


Fig. 3.36: Robotic arm executes the trajectory it learned before.

Level-2: Task Constrained Model

The upper level of the model shown in Fig. 3.32 is a model of a task constrained control of our mobile manipulator. We present here only the conceptual model due to the complexities inherited by the model: it contains a hybrid factor graph (a factor graph that has both discrete and continuous variable nodes) with multi-loop regions. Although we can discretize the continuous variables, they will produce a mixture distribution[6] which needs different treatments in a factor graph with many loops. We regard this challenge as an opportunity for our future work.

In this model, we define several discrete variables (shown in boxes) as well as real-valued variables (shown in circles). In our notation, a task $\mathbf{T} = \{\text{extend}, \neg\text{extend}\}$ refers to a "basic task" that the robotic arm should do; that is, either extending or retracting its arm. The position of the end-effector will be determined by the variable $\mathbf{G}$, which indicates the target/goal position, and the current position of the mobile robot $\mathbf{M}$. During the movement,

[6] In a Bayesian network, when a continuous-valued child node has discrete parent nodes, the structure will stipulate a unique probability distribution such as the mixture Gaussian distribution. However, since a Bayesian network does not have any loop, this mixture distribution will not have any problem when circulated in a belief propagation setting. However, a mixture distribution will behave differently in a network with many loops; in this case, people prefer to use approximate inference approaches, which are not covered in this thesis, in order to perform probabilistic queries.

the arm's orientation will be determined by the task constraint $C_1 = \{\text{flat}, \neg\text{flat}\}$. The flat pose means that the arm should keep its joints configuration to be 0 in total, so that the end-effector will always be in a horizontal position. The non-flat pose means that the arm can use all possible joints configurations to reach the target, allowing it to have the longest reaching posture. The reachability indicator $\mathbf{R} = \{\text{reachable}, \neg\text{reachable}\}$ determines whether the mobile robot should move (further toward or away to/from the object), or stay at the current position. It also determines whether the mobile robot should perform the rotation in order to align itself in-line with the object (positioned straight directly toward the object), or stay still. Hence, the value of this variable will be depending on the position of the object along with its goal position, the current pose of the mobile robot, and the current pose of the robotic arm. In addition, we need to inform the variable R about the physical constraint (e.g. the diameter) of the mobile robot via the variable C_2.

At this point, we can see that there exist several limitations in the model, such as the missing active controllers for the grasp stability of the robotic arm as well as for the synchronous motion stability of the mobile robot. For readers who are interested in solving these stability problems, we refer to the book written by Schröder [17] which gives thorough explanation on how to develop a robust controller. Nevertheless, our model is proposed to give an intuitive example of how to use a factor graph for such a complex scenario.

3.4 Discussion

In this chapter, we present many examples of applications where we can use factor graphs for reasoning. In section 3.1, we show that our factor graph framework is well suited for solving core problems in the machine learning domain. Using a Bayesian network as the underlying mechanism, we show that a factor graph network can perform a probabilistic regression resulting in estimation consistent with the Bayesian treatment for curve fitting. We then extend the task into the classification domain and again show that our factor graph framework is convenient to be used for solving classification problems with high accuracy. Although we present only a two-class classification task as an example, we argue that we can easily create a larger Bayesian classifier network since it just a matter of adding more variables without introducing any loop. Regarding the presence of a loop in a factor graph, we show that our belief propagation implementation for factor graph is consistent with the general consensus stating that the convergence is very likely to be achieved in a finite time if the network has only a single loop. We demonstrate this capability of our factor graph framework in a sensor fusion scenario with artificial data. We realize that there is still a room for improvement in future work, for example by introducing the residual belief update instead of standard asynchronous update in our loopy factor graph, which theoretically produces a better approximation.

We also introduce the usage of a dynamic factor graph by emulating a DBN in a factor graph. To our knowledge, the factor graph itself does not have capability for dealing with dynamic behaviour of the system. Hence, we just "borrow" the idea of unrolling a static network several time steps (called a horizon) and use the powerful inference mechanism of the factor graph to do some queries on the network (filtering, smoothing, etc.). Later on, we demonstrate the usage of such a dynamic factor graph in a real robotic application.

Finally, we use our factor graph framework for applications in a robotic domain. We mainly develop the model for solving the kinematics problem and argue that the robot dynamics is beyond the scope of this thesis. There are two different robot platforms that become the subject of our factor graph: the NST omnidirectional robot and the WidowX robotic arm. We began the exploration on the mobile robot kinematics control and found that we can create an N-to-N mapping network using factor graph for transforming the expected robot velocity to/from each wheel velocity. We have tested the network in a scenario where a top-head camera tracking system is used for acquiring the robot position in real-world coordinate system. The inverse kinematics model then predicts the expected wheels velocity given the robot velocity. Although the result is linear as expected, we still see some variation due to many uncertainties in the experiment. However, we argue that this variation is a common problem in many real applications because if we look at the simulation version, the inference result of the network is almost perfect.

The next robot platform that we used in our experiment was a robotic arm. Unfortunately, the generic mapping network that we used for modeling the omnidirectional mobile robot could not be used directly for the robotic arm. Similar with the model for the mobile robot, we did not have any problem with the forward kinematics case. As an alternative for the inverse kinematics model, we developed a new model using a dynamic factor graph based on a mimetic approach by which we interpret how human arm is usually moved during reaching and placing tasks. Although this is not a new idea, our method is the first to implement such an idea using a dynamic factor graph. We have tested our approach and we concluded that the accuracy of the result can be improved by increasing the resolution of the network parameters without unrolling the network further beyond the number of robot's joints.

We proceeded the experiment by introducing a scenario where we combine both the robotic arm and the mobile robot into a hybrid mobile manipulator. In this scenario, we explored the model-based learning domain to create a hierarchical robot model and control. We used our previous model for controlling the mobile robot, but for the robotic arm we used the programming by demonstration (PbD) paradigm as the learning strategy. Our method is based on the regression approach to learn a new skill for the robotic arm. The PbD implementation itself was successful. However, we had a challenging situation that requires much more examination and further exploration, which we could not accomplish it at that time, when we combined all networks into one unified factor graph model. This is because in our proposed unified model, there exist many loops that could not produce convergence results when a mixture distribution message circulated around in the network. Hence, we take the liberty of solving this problem in future work. Nevertheless, we are satisfied with the performance result of our implementation of this model-based learning paradigm.

In summary, our PC-based factor graph framework is successfully developed and applied in many application domains. In this thesis, we explore the inference scheme in the direction of belief propagation via message-passing. There exist other exact inference scheme such as variable elimination algorithm, which seems to work quite well in some applications. However, we argue that the belief propagation method is more suitable to be implemented natively in dedicated distributed-computing-capable hardware rather than

the variable elimination approach. This is because the message-passing (i.e. the sum-product) algorithm can perform the variable elimination seamlessly during its iteration. Hence, we do not need external processes just for reordering the variables like the variable elimination algorithm normally does during its inference. With this argument and also supported by the successful application results of our PC-based factor graph, we continue our research work in the next phase: embedded factor graph on dedicated hardware.

4 Factor Graph in SpiNNaker

As we have described in chapter 1, there are good reasons why we should go into the hardware level for achieving a flexible and efficient factor graph framework which supports our long term goal in developing machines with higher level cognitive capabilities. We cannot always rely on standard PCs for applying factor graphs, especially when working with real-world applications such as in robotics, which demand intriguing requirements such as low-power and real-time performance. Hence, we look into the hardware level in order to accomplish such requirements.

Our PC-based factor graph framework described in section 2.4 is our starting point for understanding the core computations required to implement effective factor graphs along with their belief propagation algorithm and messages encoding using the population coding principle. Our journey on hardware implementation of factor graphs starts by selecting the SpiNNaker platform as our initial hardware implementation. There are two main reasons for choosing the SpiNNaker platform:

1. It is a massively-parallel hardware structure supported by a unique software platform that is already well defined and quite mature. Hence, we do not need to worry too much about many technical difficulties before implementing an embedded factor graph. Designing an embedded factor graph from scratch requires thorough explorations on two different platforms: dedicated hardware and an embedded operating system. With the help from the existing operating system platform of SpiNNaker, we can focus on many low-level aspects such as the computing core's internal structure, interlink/internetwork between computing cores, memory bottlenecks, input/output interfaces, etc. Hence, lighten our efforts to understand the basic building blocks of efficient embedded factor graphs.

2. We will gain some insights and a comparative benchmark for the SoC version of our embedded factor graph. Since SpiNNaker is originally intended to be used for developing applications based on spiking neural network approach, we might get some unintended drawbacks when using it for implementing factor graphs.

With these motivations, we re-implement our original PC-based factor graph framework and optimize it for SpiNNaker. The optimization is performed after we analyze its performance in some application scenarios. This chapter describes this process starting by describing our implementation method and followed by our optimization strategy. We call this first embedded factor graph implementation in this thesis as SpiNNaker-FG.

4.1 Introduction to SpiNNaker

The SpiNNaker (Spiking Neural Network Architecture) system is a distributed computing system designed to simulate spiking neural networks [192][193][194]. It was designed by

the Advanced Processor Technologies Research Group (APT) at the University of Manchester in United Kingdom within the EPSRC (Engineering and Physical Sciences Research Council, UK) funded project in collaboration with the University of Southampton, ARM Ltd., and Silistix Ltd. It is initially targeted towards three main application areas[1]:

- Neuroscience: The largest SpiNNaker machine will be capable of simulating millions of neurons in a spiking neural network model with complex structure and internal dynamics.
- Robotics: It offers a low-power embedded system with standard interfacing mechanisms used in a typical robotics scenario. Hence, SpiNNaker might be a good platform for researchers in robotics, especially for mobile robots.
- Computer Science: SpiNNaker offers opportunities to explore new principles of massively parallel computation which cannot easily be performed by traditional supercomputers that rely on deterministic, repeatable communications and reliable computation.

The SpiNNaker system is composed of many SpiNNaker chips in hexagonal topology, which allows simulation of thousands of artificial neurons per chip in real time. The chip is designed as a Globally Asynchronous Locally Synchronous (GALS) system. Each chip is a multi-core system, consisting of 18 ARM968-based cores and also several internetworking elements and supporting modules. Every chip is also equipped with 1Gbit DDR SDRAM for applications that exploit and share these local resources for computation. Fig. 4.1 shows the internal structure of a SpiNNaker chip [2].

Our original factor graph framework can be broken down into three main components:

- Factor and variable nodes, where the computations of factor product and marginalization take place. Since we use discrete factor graphs, this component consumes the most memory resources.
- Messages encoder and decoder, where we implement population coding for discretizing continuous variables that take real value inputs.
- Scheduler, where we implement the scheduling mechanism for updating messages during inference processes in a belief propagation scheme.

The main task in the design of our embedded factor graph is how to efficiently map these three components into available on-chip resources as well as the available SpiNNaker board for running an application.

Looking further into the SpiNNaker chip's internal structure, we focus on how to utilize the cores inside the chip to implement distributed computations of a factor graph network. The chip contains interconnected microcontrollers (ARM968) with a specific routing mechanism inspired by neurobiology. It uses a packet-switched network to emulate the very high connectivity of biological neurons. The packets are source-routed;

[1] According to the project description advertised in http://apt.cs.manchester.ac.uk/projects/SpiNNaker/

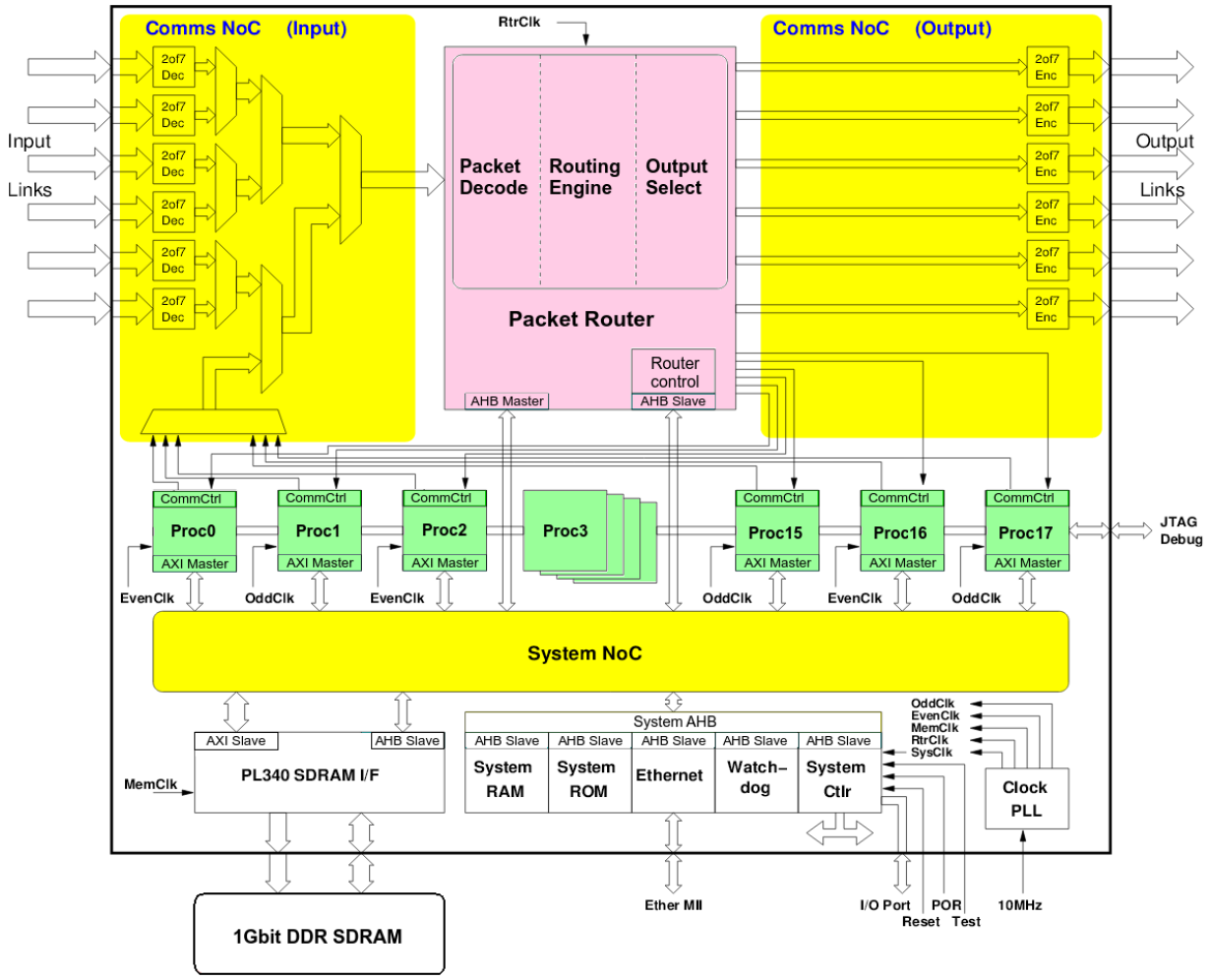


Fig. 4.1: The internal architecture of a SpiNNaker chip (adapted from [2]). The coloured blocks/regions represent our interest in exploring the SpiNNaker chip infrastructure which can help us harnessing it for our embedded factor graph implementation. The green blocks are the ARM-CPU cores where algorithmic computations take place. The pink block is the router block responsible for routing and distributing SpiNNaker packets. The yellow blocks are hardware level interface components which perform the real messages transmission via SpiNNaker links. By looking into the detail of those three blocks, we gain some insights of how should we implement our factor graph as a distributed system.

i.e., they only carry information about the spike issuer, and the network infrastructure is responsible for delivering them to their destinations. The heart of the communication infrastructure is the bespoke multicast router that is able to replicate packets and, where necessary, to implement the multicast function associated with sending the same packet to several different destinations. This routing mechanism involves several modules inside the chip in which a special look-up table is maintained by a Packet Router. The key feature of this chip lies on this aspect: by properly configuring the Packet Router, the developer can create an efficient massively distributed computing system [195]. In addition, a SpiNNaker chip has six bidirectional, inter-chip links that allows creation of networks

with various topologies. Inter-chip communication uses self-timed channels, which are significantly more power efficient than synchronous links of similar bandwidth, although they are costly in wires.

Relating the hardware infrastructure with the corresponding SpiNNaker software platform, we found that SpiNNaker has an intrinsic constraint which can be exploited in order to implement a message-passing algorithm efficiently. For our purpose, we use only some of the SpiNNaker communication protocols which are available for an application program through the SpiNNaker API (application programming interface). For developing our SpiNNaker-FG, two of them are used extensively: the neural event multicast protocol (MC) and the SpiNNaker datagram protocol (SDP). The MC packets are used for transferring "messages" between nodes in a factor graph network and SDP packets are used for communication between the SpiNNaker system and the host PC.

4.2 Mapping Factor Graph on SpiNNaker System

Deploying a program in dedicated hardware, especially the one with intrinsic parallelism, requires different treatments and explicit considerations [196]. The same challenge is also valid for our SpiNNaker-FG. Here we develop our framework with the following criteria.

- **Scalability** The framework should be able to work with a variable number of chips, allowing us to resize the networks conveniently.
- **Flexibility** The framework should be flexible enough to be reconfigured for many general purpose applications without too much modification in the framework.
- **Cross-boundary** The framework should be able to connect the separated elements of a factor graph seamlessly.

SpiNNaker chip has many different resources that can be utilized in order to implement a distributed computation platform for a factor graph. There are four elements that need to be specified and allocated in advance:

- the SpiNNaker cores which handle the nodes (variable or factor nodes)
- the routing mechanism which transfers messages from one node to other nodes
- the memory layout for vectors such as messages and local functions
- the conversion mechanism between real-valued data to/from discrete probabilistic representations

In [197], the authors describe their parallelism approach based on checker-boarding partition (CBP) method for implementing a multi-layer perceptron in a standard feed-forward neural network (FFNN) structure. They map the FFNN's weight matrix onto SpiNNaker chips and use the multicast packet transmission to distribute the computation. However, this mechanism cannot be used directly for implementing a factor graph in SpiNNaker for two reasons:

1. In an FFNN, the number of connections between layers is fixed; hence, the propagation of computations can be predicted. That is why the weight matrix in [197] can be partitioned easily. While in a belief propagation method for factor graphs, the number of connections between nodes varies. Furthermore, the propagation of computations in a factor graph can be asynchronous, which is contrast to an FFNN where the computation in a layer is always synchronous.

2. Ordinary factor graphs have two types of nodes which generate values (or "messages") differently, while FFNNs have only a single type of node (i.e. the neuron) and all of those nodes in an FFNN perform the same computation principle. Hence, even though the factor of a factor node may be stored in a matrix similar to the CBP method, each factor node must handle its own matrix and operates asynchronously.

Based on these observations, we implement different strategies for mapping a factor graph onto a SpiNNaker system. Our PC-based factor graph framework gives us insight that a factor graph has two important aspects which require a distributed computation. The first is population coding for representing the state distribution of input/output values, and the second is the sum-product computation of the message-passing algorithm. At the moment, we implement all network mapping and packets management manually by hand; i.e., by defining the routing key entries in the SpiNNaker routing table manually based on a given factor graph network. In the future, we want to enhance our SpiNNaker-FG framework so that it can offers much better functionality similar to "PACMAN" [198], which is the main SpiNNaker program for managing spiking neural networks, in the sense that it can provide mapping and configuration functionalities automatically for any factor graph based application.

4.2.1 Neurons Population Mapping

Although the SpiNNaker system is originally designed to emulate spiking neurons, we do not use this emulation mechanism since we are not interested in neuron-by-neuron spike generation. Instead, our SpiNNaker-FG will only use SpiNNaker abundant resources to implement population coding principles over a fully connected homogeneous neurons population. Fig. 4.2 shows how the population coding with a Gaussian response is mapped into SpiNNaker cores.

As our test platform, we used the SpiNN-3 board which contains four SpiNNaker chips (see Fig. 4.3a). We use the population coding principle to discretize input/output values as described in chapter 2. For this discretization, chip-0,0 (see Fig. 4.3b, box colored in green) was used. The rest of the chips were used for distributing nodes as well as for the sum-product computing engine (see Fig. 4.3b, boxes colored in pink).

The mapping shown in Fig. 4.2 uses 15 cores and those cores are controlled by core-1 which also operates as an I/O port for sending and receiving data to/from external devices (e.g. a robot or the host PC) using SDP. Core-0 and core-17 are used by the SpiNNaker kernel for monitoring purposes; both cores cannot be used by any application program except the SpiNNaker kernel. In our implementation, there are two separate source codes for those working cores. We create a "master" code for code-1 and several copies of

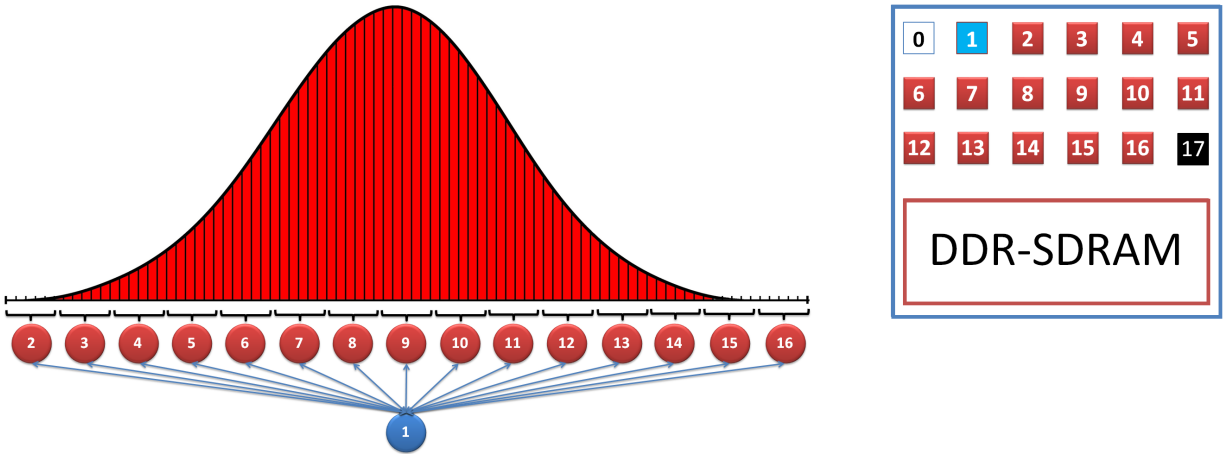
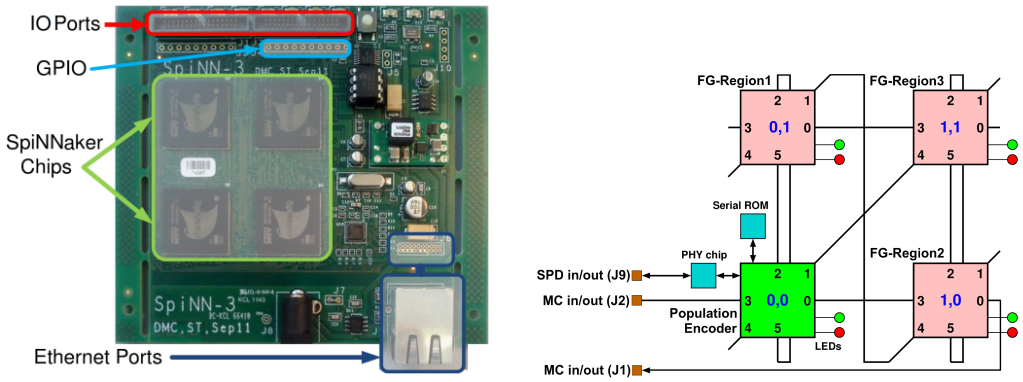


Fig. 4.2: Mapping neurons population into SpiNNaker cores in one chip (note: the white and the black cores are reserved for SpiNNaker kernel).



(a) SpiNN-3 board **(b)** SpiNN-3 chips layout

Fig. 4.3: The SpiNN-3 board and its chips layout. Chip-0,0 is chosen for managing population codes since it has a direct Ethernet connection to external systems. (Figure (b) is adapted from [3])

"worker" code for core-2 until core-16. After compiling the master and worker codes, we send the binaries to the system-RAM and instruct those cores to run the binary code on the system-RAM of the SpiNNaker chip-0,0. Each core in the "worker" group executes the same code to perform the discretization as well as to compute the probabilistic expectation based on the predefined discretization parameters which are hard-coded in the worker's source code.

The network's belief message will be split into several "chunks" corresponding to the number of available cores for discretization. Since the available cores for the computation are only 15, the number of states used to represent a population code will be the multiple of 15. In chapter 2 and 3, we explain that in most applications, 50 states for each variable are more than enough to produce high resolution (i.e. smooth) results. Hence, we limit the number of "chunk" of data per core to be 4 so that the maximum number of states can be as many as 60.

For our factor graph, we use the SpiNNaker's MC packet with 72 bit length since we

need to use the "payload" part of the packet to carry the probability value of each state. The routing-key part of the MC packet carries information such as the variable index and the chunk index of the propagated message. The diagram shown in Fig. 4.4 illustrates how we create the transmitting protocol for each chunk of the split message.

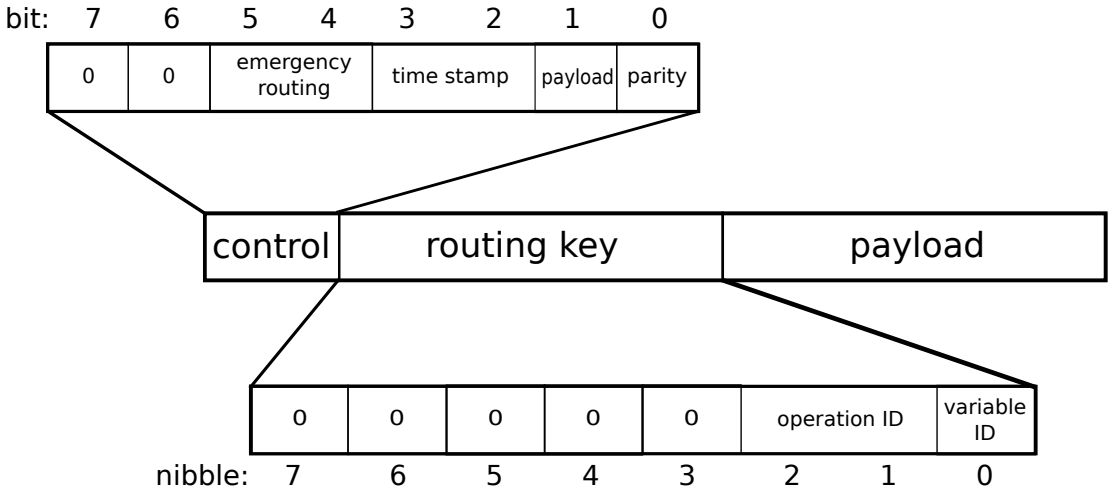


Fig. 4.4: Message transmission protocol using MC packet. The "operation-ID" part in the routing-key block determines what kind of operation is expected to be executed in the destination core. For example, if the sender is core-1, then the operation that can be performed by "worker" cores are setting-up parameters, Gaussian partitioning, expectation calculation, and transmitting the data chunk to the predefined destination factor node (see section 4.2.2). On the other hand, if the senders are the "worker" cores, then the operation that can be performed by the core-1 are acknowledging the worker's presence and accumulating the expectation values. The "variable-ID" part in the routing-key block determines to which variable does the message belong to. The maximum number of variables that belongs to a factor node is 15 and each variable is assigned with a unique ID.

When data comes from an external device, core-1 will distribute the data to the worker cores (which are not part of the monitoring cores of the SpiNNaker kernel) as an MC packet. Those working cores will start immediately the partitioning process to discretize the data and store the result internally. Later on (or when requested), they will transfer the discretized values to the other chips through the links 0, 1 and 2 of chip-0,0 (see Fig. 4.3b). Those values will be used by the other chips for a factor graph inference. On the other hand, when core-1 receives a notification from the other chips (in a form of variable ID), it will inform the worker cores to start computing the expectation value using the mechanism described in chapter 2. The overall process that happens in chip-0,0 is summarized in table 4.1, which shows how we manage the routing table for the router in chip-0,0.

4.2.2 FG-Nodes Mapping

In our SpiNNaker-FG, every core in the chip can be assigned as either factor node or variable node. We also define a Region as a subset of a factor graph that can be mapped

Tab. 4.1: Routing key definition for massage transmission

ENTRY	KEY	MASK	ROUTE	Notes
Distributing real value to worker cores				
0x00000000	0x00000000	0xFFFFFFFF	0x007FFF00	CPU-1 send a value for varID-0 to be discretized
0x00000001	0x00000001	0xFFFFFFFF	0x007FFF00	CPU-1 send a value for varID-1 to be discretized
0x0000000E	0x0000000A	0xFFFFFFFF	0x007FFF00	CPU-1 send a value for varID-15 to be discretized
Computing the expected values				
0x00000010	0x00000010	0xFFFFFFFF	0x007FFF00	CPU-1 send a request for an expected value on varID-0
0x0000001E	0x0000001E	0xFFFFFFFF	0x007FFF00	CPU-1 send a request for expected value on varID-14
0x00000020	0x0000001E	0xFFFFFFFF	0x00000080	workers send the chunk to CPU-1 for varID-0 and use payload as chunk value
0x00000021	0x00000021	0xFFFFFFFF	0x00000080	workers send the chunk to CPU-1 for varID-1
0x0000002E	0x0000002E	0xFFFFFFFF	0x00000080	workers send the chunk to CPU-1 for varID-14
Requesting for pmf, eg. to be sent to host or other chip				
0x00000030	0x00000030	0xFFFFFFFF	0x007FFF00	CPU-1 send PMF request for VarID-0 to workers
0x0000003E	0x0000003E	0xFFFFFFFF	0x007FFF00	CPU-1 send PMF request for VarID-14 to workers
0x00000100	0x00000100	0xFFFFFFFF	0x00000080	workers reply the PMF for varID-0 at seq-0
0x00000110	0x00000110	0xFFFFFFFF	0x00000080	workers reply the PMF for varID-0 at seq-1
0x000001E0	0x000001E0	0xFFFFFFFF	0x00000080	workers reply the PMF for varID-0 at seq-14th
0x00000101	0x00000101	0xFFFFFFFF	0x00000080	workers reply the PMF for varID-1 at seq-0
0x000001EE	0x000001EE	0xFFFFFFFF	0x00000080	workers reply the PMF for varID-1 at seq-14th
Sending parameters to workers				
0x00000040	0x00000040	0xFFFFFFFF	0x007FFF00	CPU-1 send #States
0x00000041	0x00000041	0xFFFFFFFF	0x007FFF00	CPU-1 send gVar
0x00000042	0x00000042	0xFFFFFFFF	0x007FFF00	CPU-1 send vRange
0x00000043	0x00000043	0xFFFFFFFF	0x007FFF00	CPU-1 send extRatio
0x00000044	0x00000044	0xFFFFFFFF	0x007FFF00	ALL_STATES_BASE_ADDR
0x0000004E	0x0000004E	0xFFFFFFFF	0x007FFF00	init command to workers
0x0000004F	0x0000004F	0xFFFFFFFF	0x007FFF00	workers send ping to CPU-1

efficiently into one SpiNNaker chip. This Region might contain one or more factor nodes together with its associated variable nodes as many as possible. An example for this Region splitting is shown in Fig. 4.5. The constraint of this design is that all associated variable nodes should reside in the same chip with its associated factor node as much as possible. The reasons is that we want to minimize the traffic overhead of "messages" in the Region and also for the purpose of load balancing between cores.

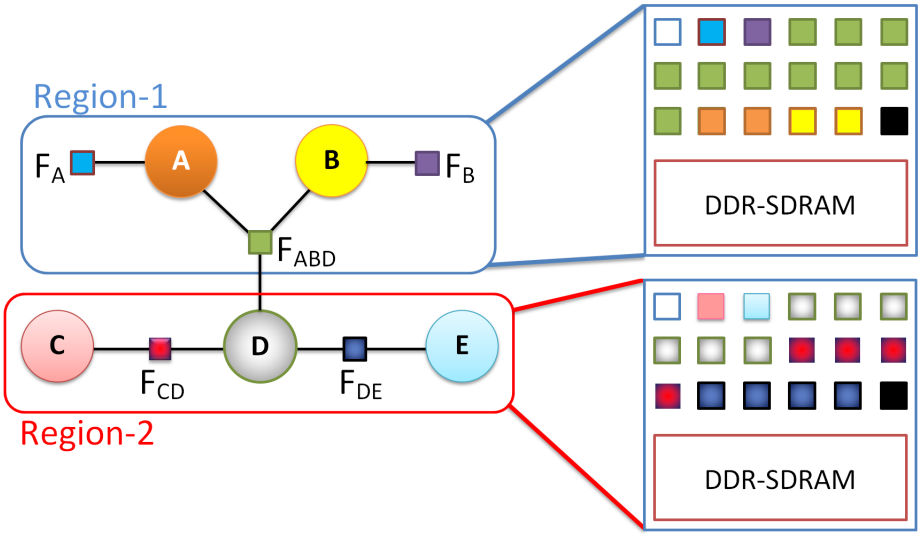


Fig. 4.5: Mapping regions into SpiNNaker chips. The color illustrates node-to-core mapping.

When implementing a variable node or a factor node in a group of SpiNNaker cores, one core of this group will be assigned as the "master", which is responsible for supervising the other cores in the group. This is important especially for a factor node because the valid communication with the SpiNNaker monitor processor must be made during the course of the program in order to get the correct access to the SDRAM. Each node will store its processing result in the SDRAM and all factor nodes will store their local function also in the SDRAM. Accessing the SDRAM for storing and retrieving data is done through the DMA callback mechanism provided by the SpiNNaker API.

In Region-1 shown in Fig. 4.5, the factor nodes F_A and F_B each occupy only one core since these factor nodes are essentially inputs for node A and node B in case node A and node B are observed. These factor nodes also do not have a vector value but only '1' as its local function. However, the factor node F_{ABD} occupies 10 cores since it is the only factor node in the region which has intense computation processes of (2.4) and (2.5) due to its link to the three nodes. Also, factor nodes F_A and F_B could be assigned with the communication task with chip-0,0 (see Fig. 4.3b), i.e. to receive the input as well as to send the message to chip-0,0. The only computation that might be performed by node A and node B is the marginalization; hence, we assign each node with only 2 cores. If the application does not require marginalization in node A and node B, they can be assigned with only one core for each node. The local function of F_{ABD} will be stored in the internal SDRAM.

In Region-2, node C and node D also occupy two cores each since they might still need to compute its marginal (however, if they don't, then it can be reduced to only one core

for each node and assign the remaining cores to the more "busy" nodes). The node D, factor node F_{CD} and factor node F_{DE}, each occupy four cores since they compute message products intensively. Same as in Region-1, the local function of F_{CD} and F_{DE} will be stored in the internal SDRAM of the chip. In the case of node D, it can be accessed in the following way. If Region-1 is placed on chip-0,1 and Region-2 is placed on chip-1,0 (see Fig. 4.3b), then the routing table for the output of factor node F_{ABD} towards node D can be assigned with link-1 of the chip-0,1 and, correspondingly, the routing table for the input of node D from factor node F_{ABD} must be assigned with link-4 of the chip-1,0.

4.2.3 Mapping and Routing Factor Graph in SpiNNaker

Factor graphs with exact inference using belief propagation operate two types of messages: variable-to-factor message and factor-to-variable message. In our SpiNNaker-FG, those messages will be encoded and sent using the multicast (MC) mechanism as the payload of the corresponding MC packet. Specific to factor nodes: the $f(X)$ in expression (2.5) is the local function of the corresponding factor node. It usually takes a form of a vector in discrete factor graphs. We have to provide this vector function before the factor graph executes its inference and this vector function is normally learned off-line. Once we have these internal factors for each factor nodes in the factor graph, we send these factors via SDP from the host to the SpiNNaker system. For this purpose, we have to specify exactly where the corresponding factor nodes are located within the SpiNNaker system with respect to the chip layout shown in Fig. 4.3b.

Regarding this routing mechanism, each node maintains its routing table, and registers it only once (due to the SpiNNaker's constraints). The node also has its own input-output matrix that reflects its neighborhood and determines which node has sent the message or has a pending message. This is important since the node computes the outgoing message only when all neighboring nodes have sent their message. In this thesis, we create a routing table similar to the one shown in table 4.1 manually. The content of the routing table depends on how many cores as well as in which cores are the nodes of the factor graph assigned to (see section 4.2.2).

Fig. 4.6 shows an example of how the belief propagation's messages are transmitted within the SpiNNaker system. The messages are propagated within the Region-1 shown in Fig. 4.5.

Assuming that the factor graph framework has already been downloaded onto the SpiNNaker system, the message-passing shown in Fig. 4.6 runs as follows:

a The PC sends input values to the SpiNNaker system via SDP which go to the core-1 in chip-0,0. The master program in core-1 will decode the SDP packet and extract input values from it.

b During the invocation of the factor graph framework for the first time, the master program (in core-1) sends program's parameters to all worker cores. Afterwards, each worker knows in which partitioning region it is supposed to work. Core-1 will send the input values to all worker cores as MC packets. When the worker cores has finished their job, they will send a notification back to core-1 which can be used later

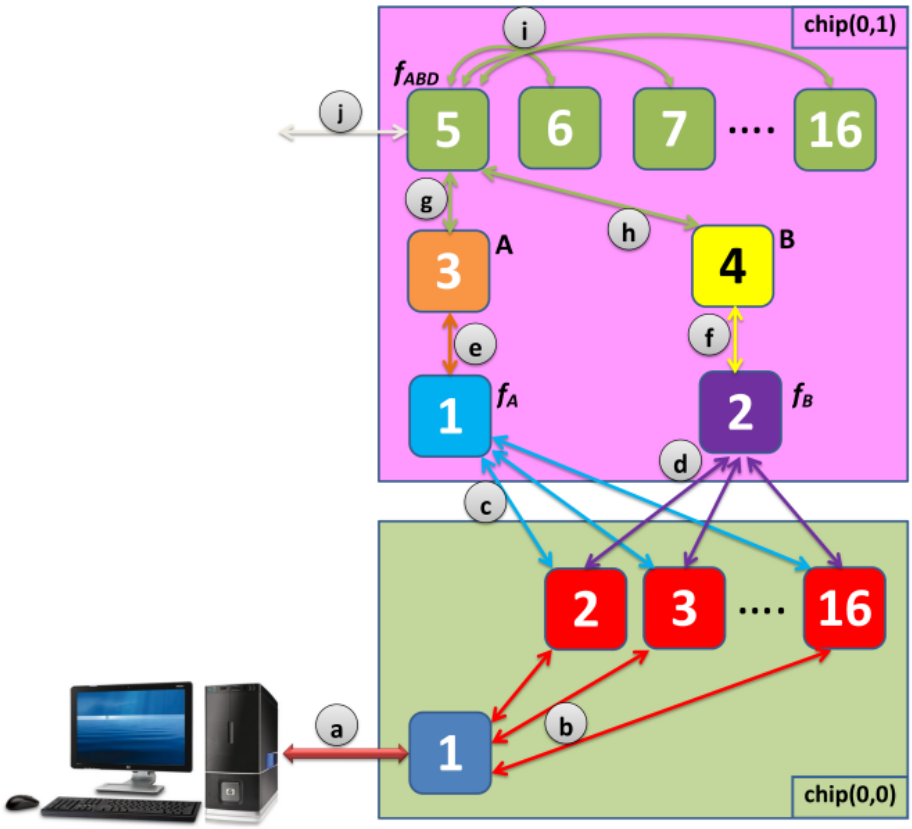


Fig. 4.6: An example of message routing in the Region-1 for the factor graph shown in Fig. 4.5. The coloured-and-numbered squares match the corresponding cores shown in Fig. 4.5

.

on by the core-1 to notify back the host PC that the input values have been fed to the factor graph network.

c The first input value is encoded using population coding and the resulting vector that contains the probabilistic message is sent to core-1 in chip-0,1 which serves as the factor node f_A.

d The second input value is encoded and the resulting vector is sent to core-2 in chip-0,1 which serves as the factor node f_B.

e The factor node f_A will pass the message to the variable node A in core-3 of chip-0,1. Since the variable node A is connected only to the factor node f_{ABD}, it will not perform any factor product computation. Hence, we can assign only one core for the factor node f_A.

f Similar to the situation in (e). Here, the factor node f_B only passes the message to variable node B.

g The variable node A passes the message to the factor node f_{ABD}. The message, which is carried on several MC packets, will be sent only to core-5 which serves as the "master" controller for the factor node f_{ABD}.

h Similar to the situation in (g). Here, the factor node f_B delivers the message to the factor node f_{ABD} and lets the core-5 (the "master" controller) to distribute the message and to manage the sum-product algorithm which runs on the worker cores (core-6 through core-16).

i This is where the paralellized sum-product algorithm takes place. The master core of the factor node f_{ABD} receives several MC packets from core-3 and core-4 as the representation of the variable node A and B respectively. It then distributes the packets to the worker cores and starts the sum-product algorithm sequence. The sum-product algorithm has two sequences: factor product operation and marginalization. In the first step, each worker core runs the factor product algorithm in parallel and use a DMA mechanism to retrieve the "internal function" of the factor node f_{ABD} from internal SDRAM. Once they have finished performing the factor product, they will inform core-5 (as the master) so that core-5 can synchronize all worker cores before running the second step (i.e. the marginalization step). When all worker cores have reported their finishing job on the factor product algorithm, core-5 will send a command to all worker cores to perform once again an ensemble of running the summary algorithm (i.e. marginalization phase). Once they have finished the marginalization, the core-5 will collect the chunks from each worker core.

j The core-5 sends the collected chunks from its worker sequentially to the variable node D, which resides in chip-1,0 (see Fig. 4.3b and Fig. 4.5), as a new message encoded in an MC packet.

The message-passing process then continues in Region-2 until the final messages have been delivered to the destination nodes. At the moment, this process cannot be generalized and solely depends on the factor graph structure that needs to be implemented in a SpiNNaker system. In this thesis, this mapping and routing mechanism is still hard-coded and we take the liberty of continuing it in our future work.

4.3 Performance Evaluation and Optimization Strategy

To evaluate the performance of our factor graph framework on SpiNNaker, we run two different test scenarios. In the first scenario, we created a factor graph representing a consistency test described previously in section 2.3.2 on page 39. The purpose of this test scenario is to evaluate how effective is the usage of multi-core processors within a SpiNNaker chip to carry the parallelism strategy for our factor graph framework. The second test scenario, which uses the same example network described in section 3.3.1, demonstrates the applicability of our factor graph framework in a SpiNNaker system for real robotics scenarios.

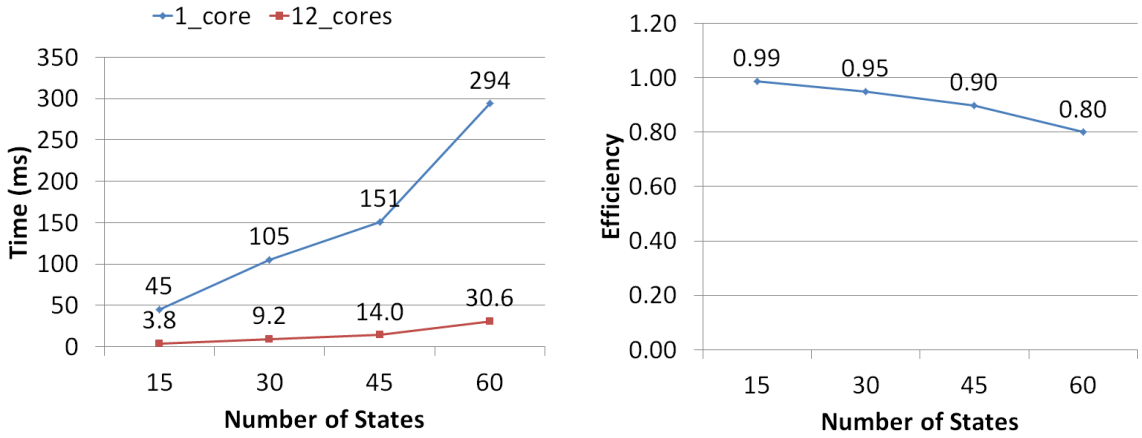
Multi-core Parallelism Evaluation

In the test scenario for evaluating the multi-core parallelism strategy, first we implemented a network consisting three factor nodes and two variable nodes shown in Fig. 2.13 completely in one core of a SpiNNaker chip and performed the inference therein. Next, we

implemented the same network but distributed the factor node f_{AB} into 12 cores, while factor nodes f_A and f_B as well as variable nodes A and B occupy one core each. We run the inference on those two scenarios and recorded the elapsed time for a single run. The result of this first test scenario is shown in table 4.2 which is also depicted as plots in Fig. 4.7a.

Tab. 4.2: Summary of evaluation on multi-core parallelism in a SpiNNaker chip

	Running Time (in ms)	
#States	Single-core	12-cores
15	45	3.8
30	105	9.2
45	151	14.0
60	294	30.6



(a) Performance Graph **(b)** Efficiency Graph

Fig. 4.7: Plotted result of the consistency test for evaluating the multi-core performance in a SpiNNaker chip that runs a message-passing algorithm.

Obviously the 12-cores scenario runs faster than the other scenario, but we are interested to see the effect of high cardinality on the MC packet's delay. We define the efficiency as the ratio between the speed up (from using many cores compared to the single core usage) and the number of cores used in the network:

$$efficiency = \frac{speedup}{\#cores} = \frac{\frac{t_1}{t_{12}}}{12} = \frac{t_1}{t_{12} * 12} \tag{4.1}$$

where t_1 is the running time on a single core (shown in the second column of table 4.2) and t_{12} is the running time on 12 cores (shown in the third column of table 4.2).

Using the data in table 4.2, we plot the efficiency values calculated using equation (4.1) as shown in Fig. 4.7b. This plot shows that even though we have gained the speed up for using more cores for a factor node, the increasing speed did not scale linearly as we increased the cardinality of variables in the network. This is due to the excessive packets transmission and coordination between cores when each core is assigned with a task to

handle a specific part of a vectorized data (see Fig. 4.2). We argue that this efficiency will degrade even further if the cores used for a factor node reside in different chips. With this situation, there will be a trade-off between the coverage area and the number of nodes that can be included in a region (as illustrated in Fig. 4.5).

One option to optimize this design is by considering the single-scope factor node (along with its variable node) as a special case, since this interconnected factor-variable node basically does not perform any computation, but just passes the value to other "inner" factor nodes in the network. We call this special configuration as an IO-node. We can then merge the similar IO-nodes in the same region to be handled by only single core. Using the same scenario described above, we can combine factor nodes f_A and f_B as well as variable nodes A and B in a single core and then enlarge the factor node f_{AB} so that it uses 15 cores. Again, we performed the inference on this new setting and measured its efficiency. The result is shown in Fig. 4.8.

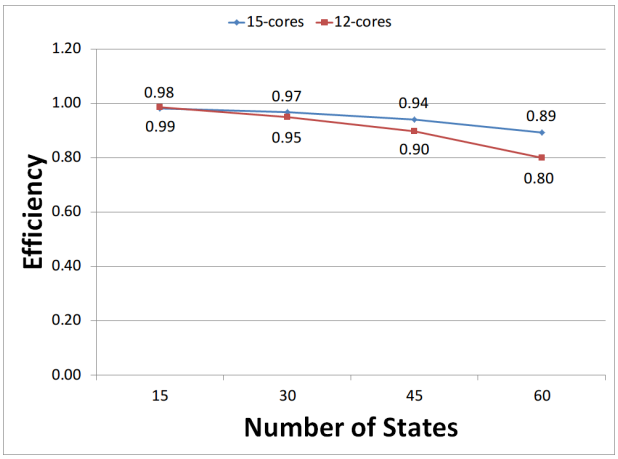


Fig. 4.8: Maximizing SpiNNaker cores usage. By exploiting the fact that factor nodes f_A and f_B as well as node A and B in Fig. 2.13 do not carry any sum-product computation, we can assign only a single core to handle those nodes and let the other "free" cores to be used by the factor node f_{AB}. With this modification, factor node f_{AB} now has 15 cores which increase the computation speed and at the same time also reduces the number of MC packets circulating in the chip. As the result, the efficiency of the chip usage is also increased.

From the efficiency plots shown in Fig. 4.8, we can conclude the following trade-off. If we want to gain the highest efficiency while using high cardinality to achieve the highest accuracy (e.g. 60 states for each variable) then we should merge all IO-nodes in the same region into one core. This extra work will require modification on both the routing table and the source code of the factor graph framework for a SpiNNaker system. In contrast, if we use a moderate cardinality (e.g. 30 states for each variable), or even lower, then we can use the original design without any extra modification on the routing table and the source code. In future work, we should include this circumstance as one option that can be automatically detected and solved by the automatic mapper program.

Robotics Application Evaluation

In the second test scenario, we use the scenario described in chapter 3 (see section 3.3.1 on page 61). In this scenario, the task is to compute the correct robot commands given the desired translational and rotational velocities. The model has been trained using data from a camera tracking system which provides the absolute pose of the robot (see Fig. 3.22). This scenario shall be viewed as a proof of concept which demonstrates a small sub-set of the features from our factor graph framework on SpiNNaker system.

The robot (see Fig. 3.14) has three wheels and the complete factor graph model of the robot will involve at least 12 nodes. The model is broken down into three similar networks and the kinematics model for each wheel is shown in Fig. 3.19. This also gives us benefits such that it makes easier to fit the model into three regions. The models are then implemented in chip-0,1 for wheel-1, in chip-1,0 for wheel-2, and chip-1,1 for wheel-3 (see Fig. 4.3b).

For training, the model is fed with the transformed data from the camera tracking system which provides the robot velocities in the robot coordinate system. After the training has been completed, the vector value of factor node f_{XYR} is sent to the SpiNNaker system via SDP mechanism. To evaluate the performance of our embedded factor graph, we sent the desired velocities of the robot (represented as factor nodes f_X, f_Y and f_R in Fig. 3.19) and observed the computed motor commands by the model (represented as variable node M_2, which reflects the motor command for the second wheel of the robot). We measured the time needed to complete one such an inference to see how effective the proposed parallelism strategy is. The result is shown in table 4.3. It shows two different measurements by which we can use later on to help us predicting the total latency of the system in a real robotic application[2].

Tab. 4.3: Execution time for a single run of inference

	Running Time (in ms)	
#States	On-board	On-PC
15	12.6	114.9
30	17.4	120.1
45	23.2	127.2
60	30.5	143.1

Although it is obvious that the number of node's states linearly influences the execution time, it is interesting to note that for the highest number of the states in the scenario, the system just needs 30.5ms to complete one full inference computation. Using 60 states, actually the system computes 26.14 MFLOPS (million floating point operations per second) for one complete cycle (from discretization until final message decoding); a very fast

[2]The second column of the table 4.3 represents the measured execution time within the SpiNNaker system exactly after it receives the input data from the host-PC and before it sends the inference result to the host-PC. The third column of the table represents the total measured time starting from the sending of input data to the SpiNNaker until it receives back the output result from the SpiNNaker. From these two different measurements, we can observe another useful information such as how fast actually the SpiNNaker distributes and processes the SDP packets since the data from/to the host-PC is encapsulated in SDP packets.

computation, especially when regarding the core speed which is only about 200MHz and without any dedicated floating point unit (i.e. it relies on software emulation for floating point operations). As a comparison, our previous work, which uses a standard PC with processor Intel i5 3.30GHz and memory 16GB DDR3 running at 1.3GHz, takes 5ms to complete one full inference computation.

Regarding the optimization option for this particular evaluation scenario, we might turn into the same strategy as before. Basically, the network shown in Fig. 3.19(a) has four IO-nodes. We could then allocate a single core for these nodes, allowing the factor node f_M to use up the rest of the cores in the chip. Increasing the number of cores that will be used by the busiest node as well as compacting all the IO-nodes in a single core will increase the efficiency of the chip's usage. However, this is not a final solution and we opt to find a more generic optimization strategy which is not only deal with the single chip constraint, but also capable of handling a larger network stretching out to the other chips on a big SpiNNaker board. We describe one possible approach in the next sub-section.

4.4 Discussion

SpiNNaker is the first embedded platform for our factor graph. In addition to the initial reasons why we start implementing embedded factor graphs, our SpiNNaker-FG offers two important advantages:

1. The SpiNNaker version consumes much lower energy than the PC implementation. Thorough analysis in [199] reveals that it can go less than 1 watt per SpiNNaker chip even in big dynamical networks.

2. If we increase the problem size such that the network shown in Fig. 3.19 is replicated three times using the remaining chips in the SpiNNaker board, for this particular example, the execution time in table 4.3 remains the same; while in a PC, it needs three times longer.

These advantages show that our SpiNNaker-FG framework has very promising features for future real robotics application.

In section 4.2.2, we explore one possible configuration of a SpiNNaker chip as a Region. We fit the Region with arbitrary nodes and split the CPU cores accordingly. Another possible configuration which can increase the computational efficiency is introducing a generic Region, which only contains a smaller number of nodes (e.g. three variable nodes and one factor node). This is similar with the idea of binary DAG used in [196]. For example, the network shown in Fig. 3.19 can be decomposed into the network shown in Fig. 4.9a. Although it will introduce hidden nodes which need to be learned beforehand and also requires additional chips, the sum-product algorithm will run faster due to a smaller number of items to be processed by the algorithm. This is preferable for future implementations using bigger SpiNNaker systems (see Fig. 4.9b).

Since the embedded factor graph in SpiNNaker is constrained in terms of hardware-link resources and relies heavily on the routing mechanism of router component within the SpiNNaker chip, it is necessary to ensure that the construction of routing table for

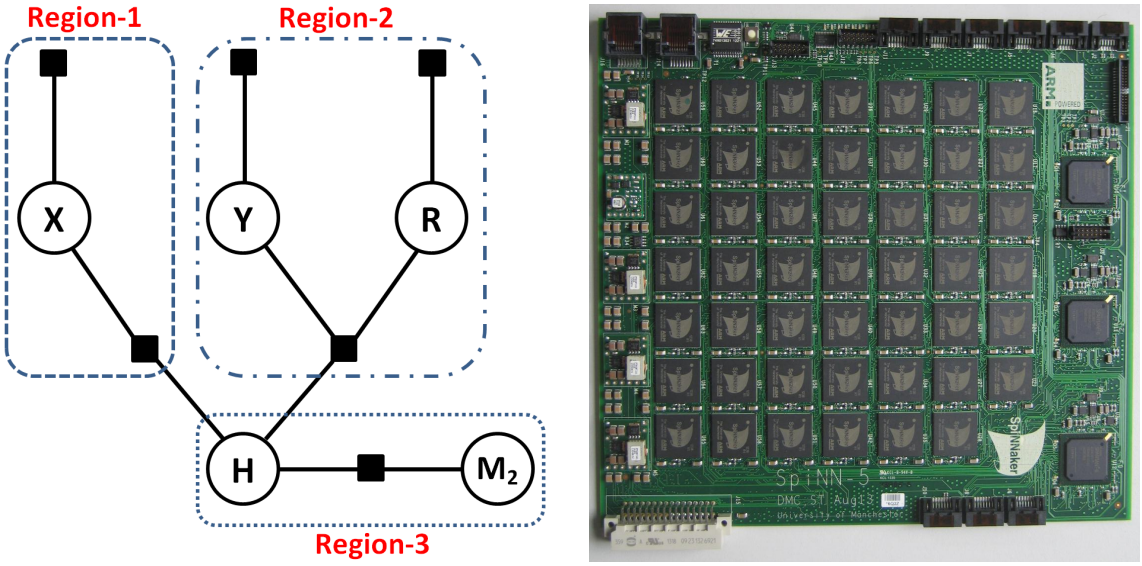


Fig. 4.9: Further optimization strategy and a larger SpiNNaker system. (a) Decomposing the network shown in Fig. 3.19 into three Regions for better computational efficiency. (b) Increasing the number of region in (a) requires more chips which can be solved by using the new SpiNNaker board with 48 chips.

MC packets does not spend up all available entries. Therefore, it might be efficient to use FFGs (Forney-style factor graph) instead of ordinary factor graphs to be implemented in a SpiNNaker system. This is because in FFGs, the interconnection between variable and factor nodes are simplified. Using an FFG, a variable node is only connected to two factor nodes at most. As the consequence, the variable nodes in FFGs do not perform any calculation but just pass the message from one factor node to the other factor node. Hence, a variable node in FFGs will not consume any processing core of SpiNNaker chip and we can fully utilize the cores for sum-product algorithm processing. However, as described in section A.1, the FFG implementation requires an additional type of factor node where the operations of this new node is quite different to the standard sum-product algorithm. Not only does an FFG have additional node types, it also imposes a new challenge of implementing the population code in it. It is straightforward to use population code for ordinary factor nodes but we do not have any proof yet that the same mechanism can be used directly on those special nodes. Hence, implementing an FFG in a SpiNNaker system will open a new challenge and a new perspective on exploiting routing mechanism as well. We leave these challenges for future work in optimizing factor graph in a SpiNNaker system.

5 Factor Graphs in System-On-Chip

This chapter describes our second contribution with the embedded factor graph. This second experiment is focused on the development of embedded factor graph in a system-on-chip (SoC) hardware. An SoC is an integrated circuit (IC) that integrates all components of a microprocessor or other electronic system into a single chip. It may contain digital, analog, mixed-signal, and often radio-frequency functions packaged together on a single chip substrate. It is usually equipped with a programmable logic unit such as FPGA (Field Programmable Gate Array). And just like normal FPGAs, the SoC is basically an "empty" device but with some additional predefined specific configuration with respect to its embedded microprocessor. A typical application is in the area of embedded systems.

Working with SoC entails two perspectives. On one side, SoC offers extensive system level integration and flexibility; on the other side, this device imposes a new challenge of integrating both concurrent and sequential programming paradigm. Due to this different paradigm, we cannot just re-implement our PC-based factor graph framework into the SoC. It is especially because we need to implement the distributed computing mechanism in a very low hardware level down to the signal and bus topology that must be designed and implemented within the chip itself. Hence, this chapter describes our design strategy on implementing efficient embedded factor graph in an SoC.

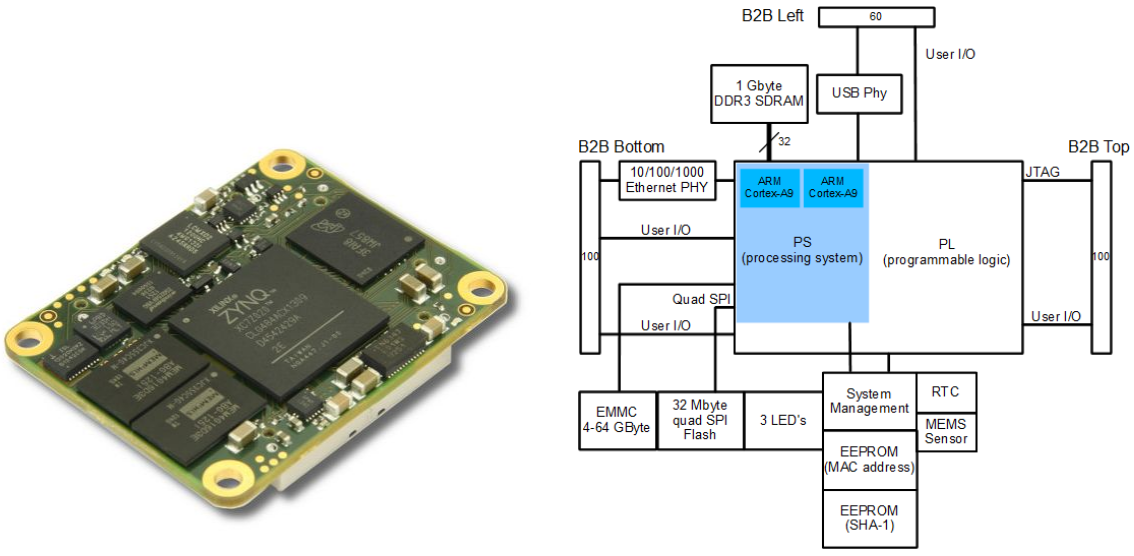
In this thesis, we implement two different methods for embedded factor graph. The first method regards the FPGA only as a supporting device for computation enhancement of the processor inside the SoC. In this method, the FPGA is utilized as an accelerator for parallelizing the sum-product computation. This is similar to the idea of using GPU for general purpose programming to speed-up some computations. In the second method, we implement the whole factor graph framework in the FPGA. This method is inspired by our first embedded factor implementation in a SpiNNaker system (see chapter 4). In this method, we strive to optimize the embedded factor graph by exploiting all available resources in the FPGA.

5.1 Introduction to Xilinx Zynq-7000

FPGAs have been long considered as the most practical devices in contrast to standard ASICs technology for achieving faster execution time by exploiting the intrinsic parallelism that they provide while preserving the low power consumption constraint of the system design paradigm. However, working with FPGAs as a standalone system has been found too impractical in many embedded system applications. Integration of FPGAs with one or many ordinary microcontollers in separated modules has become a standard solution in embedded designs. Now, many FPGA vendors turn to the new paradigm of integrating both FPGA and microcontroller into a single chip popularly known as System-on-Chip (SoC). One of the most recent SoC family produced by Xilinx Inc. is the Zynq-7000 which com-

bines the software programmability of a processor with the hardware programmability of an FPGA. With this combination, the flexible programmable logic of the Zynq-7000 devices enables optimization and differentiation by allowing the designer to add peripherals and accelerators for the dedicated processing system in the SoC. According to the company's publication, the main selling values of this new SoC family are its system performance, flexibility, and scalability while providing benefits in terms of power reduction [200]. By using this SoC, a truly embedded system which integrates both high-level-software-based control and real-time-hardware-based processing can be achieved. Furthermore, this SoC has an ARM-based processing system so that we can spare a lot of logic gates for more factor graph computations within the FPGA section of the SoC.

In this thesis, we use a tailored module TE0720 produced by Trenz-Electronics (see Fig. 5.1). TE0720 is an industrial-grade SoC module which adds a gigabit Ethernet physical layer transceiver, 8 Gbit DDR3 SDRAM with 32-bit width, 32 Mbyte flash memory for configuration and operation, and powerful switch-mode power supplies for all on-board voltages. It provides a large number of configurable I/Os useful for interfacing with many embedded peripherals. The module's form factor 5x4 cm^2 makes it the good choice for embedded factor graph ready for real robotics applications. To provide a convenient way for the Xilinx SoC Z-7000 to control all of those components on the board, module TE0720 is equipped with a so-called System Management Controller (SC). This SC is a CPLD (Complex Programmable Logic Device) type X02-1200 produced by Lattice Semiconductor and is responsible for power sequencing, reset generation and Zynq initial configuration (mode pin strapping). Regarding booting mechanism, this module supports two types of boot modes: QSPI (flash memory) boot mode and SD card boot mode. In the factory default setting, the module will boot from its QSPI.



(a) Physical appearance **(b)** Block diagram

Fig. 5.1: The modul TE0720 (GigaZee) from Trenz Electronic GmbH carries a Xilinx Z-7020 and several additional components required for building a complete embedded system such as a gigabit Ethernet transceiver (physical layer), 8 Gbit (1 Gbyte) DDR3 SDRAM and 32 Mbyte SPI Flash. (Source: www.trenz-electronic.de)

The module TE0720 is basically a "daughter" card. In order to use it in real applications, the module must be plugged into a carrier board. There are several carrier boards available for this module. In this thesis, we use the carrier board TE0703 which is also manufactured by Trenz Electronic. This carrier module TE0703 provides connectivity to external devices through standard interfaces such as USB2.0 and ethernet RJ45. It also has many terminal pins which are directly connected to the FPGA of Z-7000. The USB connection of TE0703 has many purposes; one of them is for configuring the FPGA of Z-7000 through the JTAG mechanism.

5.1.1 Internal Architecture

The SoC from Zynq-7000 family is composed of two tightly coupled sub-systems: PS (processing system, i.e. the microprocessor core) and PL (programmable logic, i.e. the FPGA component). The PS sub-system consists of equivalently two ARM Cortex-9 processors, and the PL sub-system is equivalent with an FPGA Artix-7 from Xilinx. The internal structure of this SoC is shown in Fig. 5.2. Table 5.1 summarizes the most relevant features of Xilinx Z-7020 for this thesis.

Tab. 5.1: The important features of Xilinx Z-7020

Processor Core	Dual ARM® Cortex™-A9 MPCore™
L1 Cache / L2 Cache	512 KB / 256 KB
Memory Interfaces	DDR3, DDR3L, DDR2, LPDDR2, 2x Quad-SPI, NAND, NOR
Peripherals	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO
Logic Cells	85K Logic Cells
BlockRAM (Mb)	560 KB
DSP Slices	220

The Zynq-7000 processing system can operate independently from the programmable logic and it boots on reset like any other processor-based device. In addition, the processor acts as the "system master" and controls the configuration of the programmable logic, enabling full or partial reconfiguration of the programmable logic during operation. In this thesis, we use Z-7020 from the Zynq-7000 family as our embedded system device.

5.1.2 Software Development

The TE0720 module is supported by the free Xilinx Vivado WebPACK tool which requires registration with a valid Xilinx-ID for receiving the activation key. Xilinx Vivado is a new software framework developed by Xilinx and is especially targeted for embedded processor hardware designs using the new SoC Zynq-based technology. Within this software framework, a software package called Vivado-HLS, which provides a high level synthesis for FPGA-based designs, is included. Using Vivado-HLS, the FPGA can be programmed using C, C++ or SystemC.

Developing an SOC-based embedded system will always involve at least two different programming platforms: the software platform for the central processing unit and the software platform for the programmable logic (FPGA) unit. In the past, engineers did

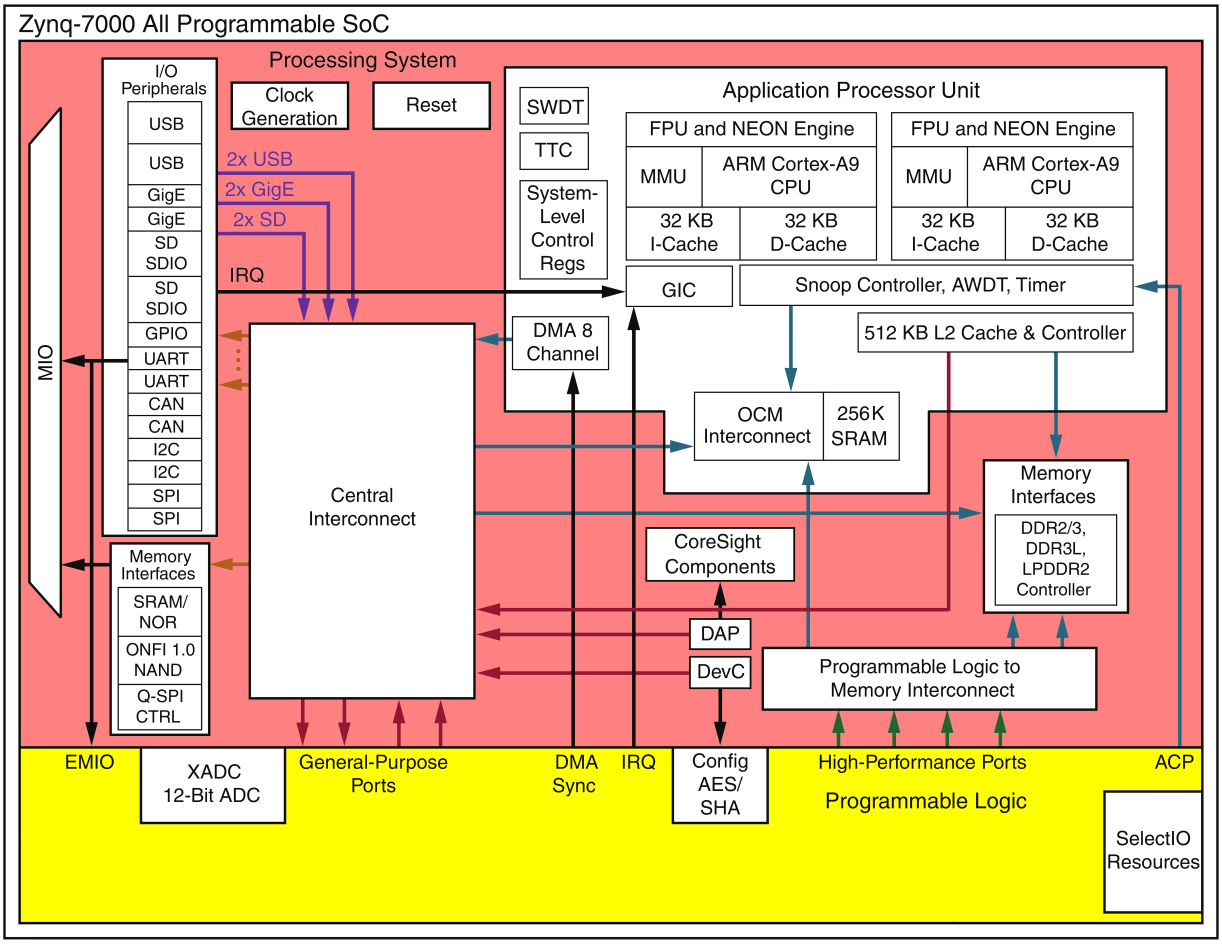


Fig. 5.2: The internal architecture of Zynq-7000 SoC family (adapted from [4]). The yellow area represents the programmable logic part (i.e. the FPGA) of Zynq-7000, where we implement most of our factor graph core modules for the method-2 described in this chapter. The pink area represents the processing system part (i.e. the microprocessor) inside the Zynq-7000, where we implement the protocol of communication with external devices (e.g. the host PC and/or the robot) as well as the main factor graph framework using method-1.

not have many options for developing the program on the programmable logic side and usually rely on the programming flow of the Hardware Description Language (HDL) such as VHDL or Verilog. However, since 1999, several hardware vendors such as ARM Ltd., CoWare, Synopsys, and CynApps teamed up to develop a new framework which utilizes already existing C-programming. This new framework has capabilities of emulating and synthesizing concurrent process commonly used in an event-driven hardware simulation system.

This new framework, which is called SystemC, is a set of C++ classes which extends the communication mechanism in a simulated real-time environment, using signals of all the data types offered by C++. SystemC extends the conventional register-transfer-level (RTL) abstraction into the new transaction-level modeling (TLM). In this respect, SystemC is not only able to mimic the low level hardware description languages (VHDL or Verilog), but also capable for synthesizing the model in the system level. Later on, Xilinx

adopted the SystemC into its own development environment and combined it with another already existing HDL framework into the new platform called Vivado-HLS.

In this thesis, all of the main cores of our embedded factor graph were developed using Vivado-HLS. Only the glue-logic components and small-size but frequently used non-behaviour-based logic blocks are written purely in VHDL in order to reduce total latency and achieve high area optimization. Further optimization approaches will be described in the following sub-sections. Beside this Vivado-HLS, Vivado Suite also has SDK (Software Development Kit) framework which is very useful to create an embedded application. We use Vivado SDK for developing our factor graph-based controller which runs under embedded Linux called Petalinux. The diagram in Fig. 5.3[1] shows the work-flow of our SoC-based embedded factor graph design in this thesis.

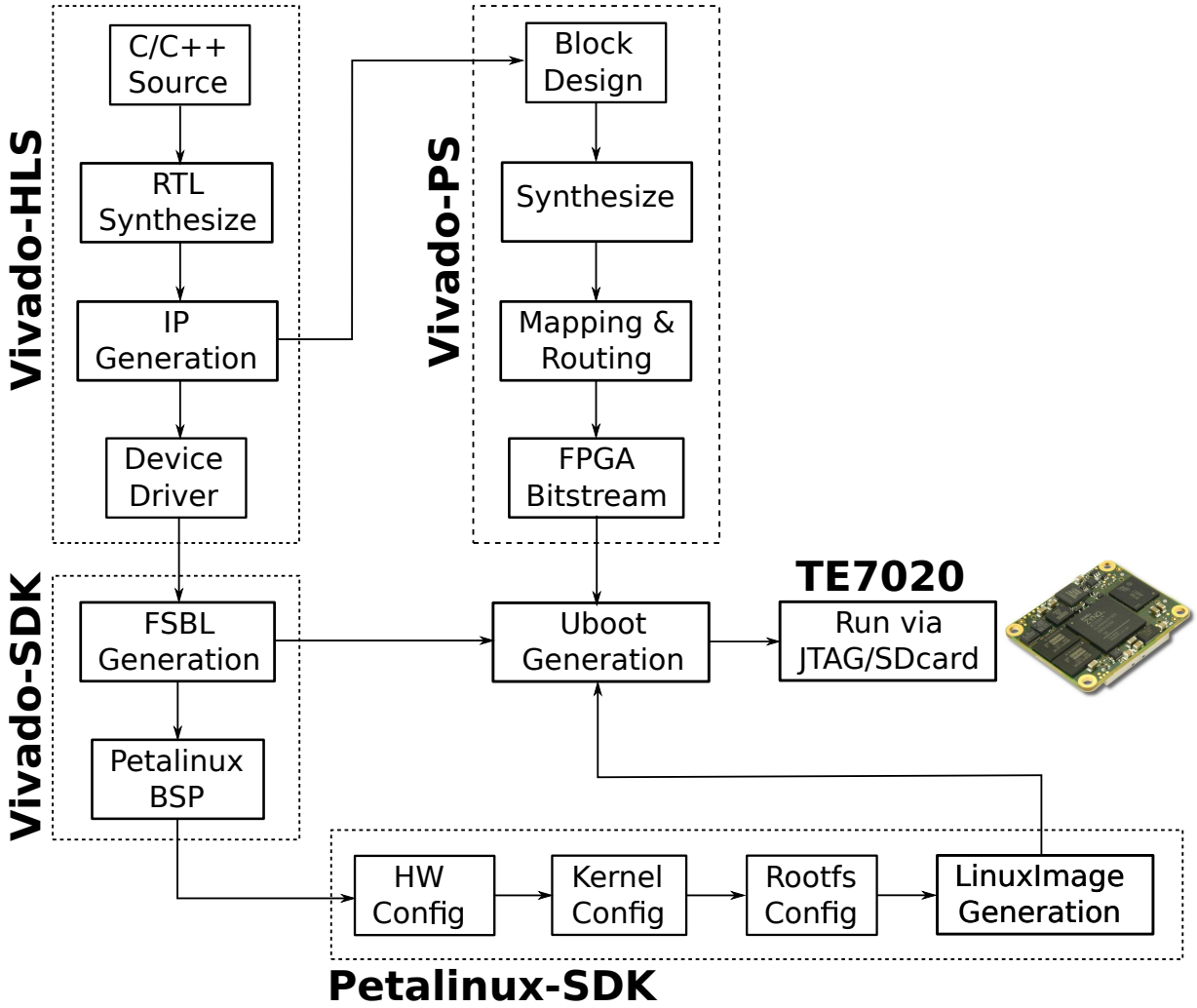


Fig. 5.3: The overall design flow for creating embedded system applications based on SoC. This includes several development tools with different customizations.

From our previous implementation of factor graphs using a SpiNNaker system, the kernel will emulate the floating-point operation since SpiNNaker chip does not have capability

[1]In this diagram, the abbreviations mean: BSP=Base System Package, FSBL=First Stage Boot Loader, RTL=Register Transfer Level, IP=Intellectual Property

for performing floating-point operation natively. This in turn slows down the overall computation in our SpiNNaker implementation. One common solution of this problem is using fixed-point arithmetic, but it will introduce another difficulty in probabilistic computation: it will reduce resolution. It is very difficult to compromise between large real-value number (higher than 100.0) and very low probability value (less than 0.000001) using fixed-point arithmetic. Hence, an extra effort must be provided when we implement our factor graph's algorithm in SpiNNaker. That is why we kept using floating-point (even only emulation) when we implement our factor graph framework in SpiNNaker.

The same reason is also valid for our factor graph implementation in the FPGA. However, even with abundance logic resources in Zynq-7000, we still need to be cautious when implementing floating-point operation. This is because floating-point numbers, inherently, have these two main artefacts: accumulation of rounding error and improper handling of subnormals [201]. Also, even though every FPGA synthesizer tries to perform optimization by default, it is important not to assume that those synthesizers always make optimization that seem obvious and trivial to a programmer. Our experience when dealing with floating-point arithmetic for implementing our proposed method, teaches us that the operation involving floating-point literals might not be optimized during synthesis. Thus, we have to inspect carefully the synthesis report produced by the FPGA synthesizer and look for the mismatches and artefacts. Another option for our future work that can be used to increase the efficiency, in particular the area optimization and in addition to the default optimization strategy offered by Xilinx platform, is by incorporating the optimization approach offered by Physical Synthesis Toolkit (PST) introduced by Tomasz Czajkowski [202]. His approach, which is based on functionally linear decomposition technique and dynamic power reduction, can help the designer to gain area optimization up to 25%. Still, a proper adaptation effort is needed since the PST technique runs on a different platform other than Xilinx development environment.

One important benefit of using floating-point arithmetic in our factor graph is that floating-point arithmetic can represent real numbers in a much wider dynamic range, which allows the data to be used through long sequences of calculations such as in a belief propagation computation. We argue that floating-point is the best choice for our factor graph implementation in the FPGA, even though it consumes many FPGA logic resources, because we can see the trend of FPGA becoming denser and relatively cheaper. Hence, for the future of our work, this issue will not become a problem.

5.1.3 Technical Considerations

In the next sections, we describe our methods to optimally implement factor graphs in a Zynq-7000 platform. Before we explain in great detail, there are some important considerations regarding the hardware constraints that we need to take care of prior to the implementation work.

Optimization Trade-off

The optimization trade-off is the most daunting issue for all embedded system engineers since there is no rule or theory of how the optimization with respect to the area and speed

will be best achieved. Not only it depends on the synthesizer quality (which will turn into the "price" issue), but it is also very application-dependent. Furthermore, in an FPGA-based system, not all FPGA chips share the same internal supporting components (such as distributed RAM, DSP blocks, etc.); hence, the re-implementation from one chip to another chip (from different family) will always produce different result.

Regarding the trade-off between area and speed optimization, in this thesis we put preference on the speed over the area optimization. The argument is that the speed optimization is a more generic issue than the area optimization for at least two reasons:

1. In general, the basic motivation of using dedicated hardware is to speed up the computation process. We often assume that the chip vendor will create/produce chips with necessary elemental units

2. Many of the optimization options provided by the synthesizer vendor are more related to the speed issue, where the system designer can play with.

With this principal consideration in mind, we develop our optimal implementation strategy. The area optimization is left to the synthesizer[2] platform with an assumption that the chip price and/or availability are not an issue.

Regarding the area optimization with respect to the storage/memory allocation on internal resources of the FPGA, the main trade-off usually lies on the choice between using the basic logic gates (in the form of Look-up Tables (LUTs)) or using block RAMs (BRAMs). Although it is possible to use external memory, we prefer to avoid this method since external memory access is an expensive task in terms of FPGA resources. Controlling external memory needs explicit routing strategies in order to match the interfacing protocols and timing constrains required by the memory hardware [4][203][204][205][206][207]. Hence, we strive to optimize our design by only instantiating memory elements either on LUTs or BRAMs. Table 5.1 shows that Z-7020 has limited BRAMs and it should only be used when the latency is not the main issue since the location of BRAM units within the chip is sparse. In contrast, the LUTs will provide the fastest response (i.e. lower latency) since LUT-based memory can be allocated right beside (very close to) the computing cores. However, LUTs are the elemental logic units necessary for implementing the core elements of a factor graph. It turns out that many parts in our algorithm require accesses to memory units in a form of an array. Array is a basic construct to express a memory access in Xilinx Vivado-HLS. The optimization strategy for arrays includes reshaping and partitioning. By optimizing an array (either reshaping or partitioning), the data transmission bottleneck can be avoided. Fortunately, Xilinx Vivado-HLS provides a convenient way to handle this array optimization which helps us to inspect and analyse the resource usage/consumption for later optimization.

Regarding the speed optimization, our approach is mainly based on the idea of exploiting the "unbounded" parallelism paradigm in the sense that we can parallelize any task, in any degree, in resourceful dedicated hardware such as FPGAs. Although this sounds to be a strong assumption, it proves to be very useful in many situations [208][209][210].

[2]The term "synthesizer" is a common jargon in FPGA-based designs which, at some extent, has an equivalence to the term compiler in generic computer programming

Basically, there are two types of parallelism: data parallelism and function parallelism. In an FPGA-based design using high level synthesis (HLS) provided by Xilinx, the focus is more to the function parallelism which usually takes a form of statically-scheduled instruction-level parallelism in loops and/or dynamic task-level parallelism between loops. In this scenario, Xilinx offers two important optimization scenarios in its Vivado-HLS development framework: unrolled and pipelined. The unrolling mechanism is one of the key concepts in implementing parallelisms. The unrolling mechanism for performing a task parallelism is shown in Fig. 5.4. The figure illustrates the main difference between a pipeline process and the standard sequential flow of instruction. The unrolling mechanism definitely will increase the resource consumption; hence, the occupation area in which the FPGA resources will be used by unrolling mechanisms will be larger. Further improved performance can theoretically be achieved by using pipeline mechanisms on the same loop. Fig. 5.5 shows the combination of unrolled and pipelined mechanisms.

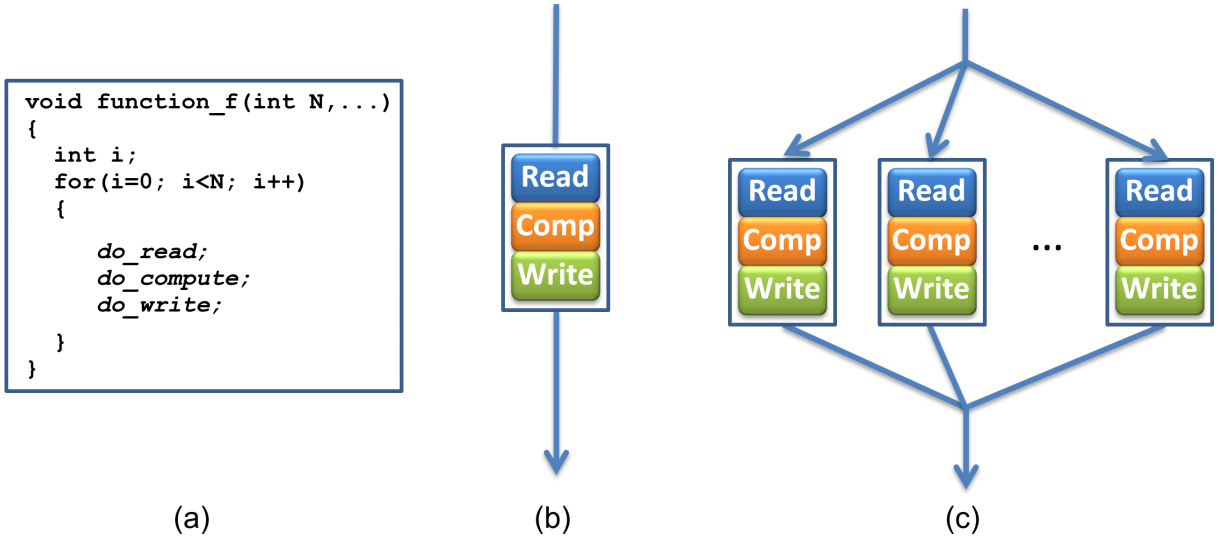


Fig. 5.4: The unrolling mechanism to implement parallelism in hardware. (a) An example of a simple function which has a loop. Within the loop, Vivado-HLS will transform and split the code into generic tasks which involve three different basic operations: read data from the memory, perform arithmetic/logic computation, and write the result to the memory. (b) In a sequential process such as the standard program execution in a single core CPU, the three operations will be executed consecutively one after the others. (c) By unrolling, the three operations will be "copied" into N different instances and they are executed together. Sometimes, this unrolling is also referred to as naïve parallelism. The number of N instances that can be synthesized depends on the the FPGA resources.

In our method described in the following section, we utilize both optimization scenarios. Using these scenarios, there will be a trade-off during each loop's iteration on which we have to make a balance between the hardware state and the hardware resources. Undoubtedly, these two scenarios require more resources in exchange to the increased speed. We use two metrics to measure the efficiency of our optimization approach: clock latency (which indicates the success of our speed-based optimization) and resource consumption (which measures how well our area-based optimization has been carried on by the synthesizer).

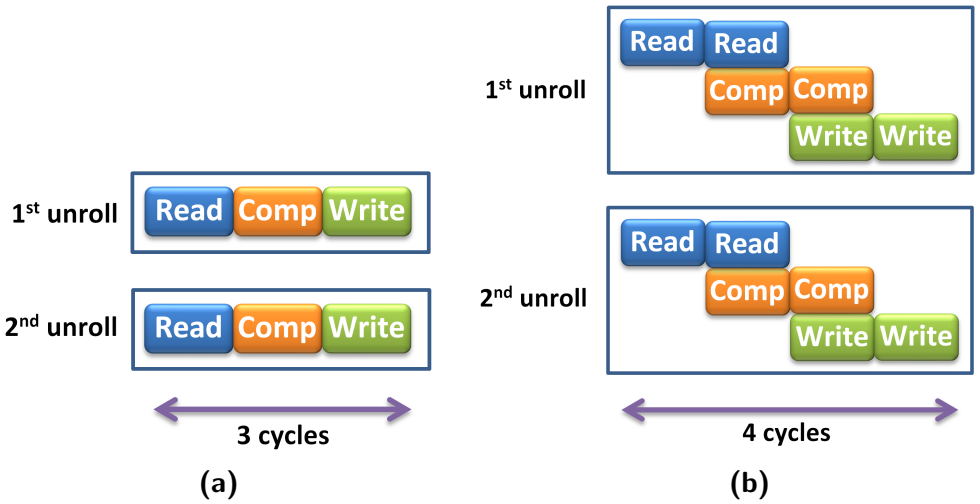


Fig. 5.5: The combination of unrolling and pipelining. (a) The unrolling only mechanism where N identical instances of the process shown in Fig. 5.4 are executed at the same time, requiring only three basic cycles to complete N processes (here N = 2). (b) The unrolling is combined with a pipeline mechanism. Although the latency is increased a bit, the overall result is twice as much as the result produced by (a) with the same number of N instances. Even much higher speed can be achieved by several pipeline stages (in this figure, the number of pipeline stage is two).

Based on these two measures, we define a cost function to measure the overall optimization result.

Supporting Data Type

In our PC-based factor graph framework, we use double-precision floating-point values so that we can get the best or smoothest result for the inference using belief propagation. Unfortunately, this double-precision is very expensive in terms of hardware resource usage in an FPGA. Hence, in this thesis, we use single-precision floating-point values. Although basically we can use any number of bits, the Xilinx synthesizer restricts the use of such an approach and only optimizes a design that uses 32-bit representation. As a result, we cannot perform any further optimization in this case.

In the past, computation using floating-point numbers in FPGAs is very difficult. It was also worsen by the fact that the FPGA vendors did not provide the library for free; hence, many designers relied on the open source version of such a library, which in turn resulting in inefficient implementations (because efficiency and optimality are very closely related to the synthesizer exploitation on hardware resources, in which only the vendor knows how to do it properly). Fortunately, in these days, the FPGA vendors are more generous and they provide the library usually for free, even in C/C++ syntax (not only in VHDL version). This thesis uses this opportunity to implement the factor graph computation conveniently. As an alternative to floating-point, we can also use the fixed-point representation. However, we found that the fixed-point arithmetic, despite the fact that it is much faster and lower hardware consumption, produces coarse results which might be less useful in real robotics applications.

Based on this evaluation, we prefer to use floating-point for the base of our numerical computation. Another reason which motivates this decision is that the fixed-point arithmetic is only supported by C++ in Vivado-HLS. If we use fixed-point arithmetic in C++ while most of our embedded factor graph codes are written in C, then we will have a mix framework which is hard to maintain. In contrast, the floating-point data can be used either in C or C++. To maintain the consistency of our code, so that it will ease future development for the extension of our current embedded factor graph, we stick to the C programming.

5.2 Method-1: FPGA as Accelerator

Hardware accelerators are becoming increasingly commonplace in delivering high performance computing solutions at a fraction of the cost of conventional supercomputers and standalone CPU clusters, despite the additional programming effort required to utilize them [211]. In many applications, FPGAs are used only for providing acceleration to the other existing processing units (e.g. microprocessors). Since the computation in such conventional processing units happens sequentially, the presence of such accelerators with true parallelism will speed up the overall computation. Even with modest configuration and optimization, the speed-up gain can be higher than its implementation counterpart using multi-core systems and clusters [212]. In this thesis, this paradigm is applied as the first approach and its performance result will be compared against the second approach which will be described in the next section. Fig. 5.6 shows the block diagram of this first approach.

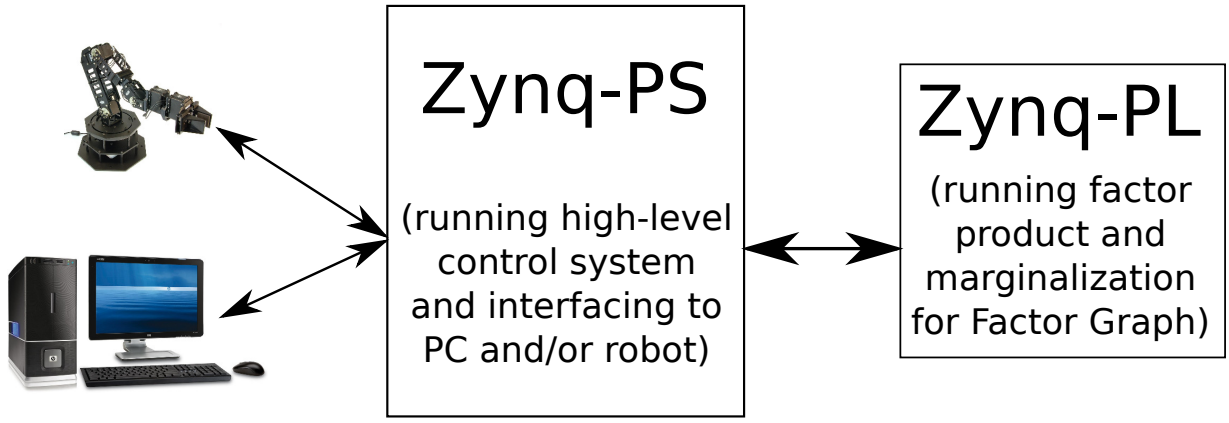


Fig. 5.6: Using the FPGA part of SoC as an accelerator for a factor graph.

Our method that uses an SoC as an accelerator for a factor graph works as follows. The main factor graph framework is implemented in C and will be compiled and run on the PS part of the SoC. This factor graph framework is tailored to match a dedicated application, for example as the controller of a robot shown in Fig. 5.6. In this scenario, the PS part of the SoC will be responsible for running the factor graph and for communicating with external devices such as the host PC (for further data analysis) or the robot (which will be controlled by a factor graph). During the inference process using the belief propagation

mechanism, the PS will send the message values to the PL part of SoC. In this PL part, those messages will be processed by the sum-product algorithm. Hence, the PL only performs the parallel version of the factor product and marginalization. Once the parallel computation in PL has been completed, the results will be sent back to the PS and will be delivered to the external devices or propagated to different nodes within the respective factor graph running in the PS. During the PL execution, the PS might be in the idle state or performing some other tasks; it depends on the application.

In the PS part, the main factor graph framework consists of several modules similar to the PC-based version of the framework (see section 2.4 on page 42), including the population codes encoder/decoder. All nodes that have a scheduler for the message-passing are also implemented in this framework. In the PL part, the factor product and marginalization, which perform a lot of loopy computations, are implemented in a parallel fashion by unrolling the block of code which has a loop. Theoretically, further optimization can be achieved by utilizing a pipeline mechanism. To communicate the factor graph messages between PS and PL, the AXI protocol is used. AXI (Advanced eXtensible Interface (AXI)) is the third generation of AMBA (Advanced Microcontroller Bus Architecture) which is an open-standard developed by ARM for on-chip interconnect specification that manages the connection of functional blocks in SoC designs. The AXI is the only recommended bus communication protocol by Xilinx Vivado and is used extensively in this thesis. The factor graph messages from PS will be sent to PL (and vice versa) in the form of an array. To facilitate computation on an array, the external memory part (either using the distributed BRAM or LUTs) must be included in the design. In most parts of our implementation, BRAM is used instead of LUTs due to the high cost of LUTs (although BRAM is a bit slower than LUTs). The trade-off between the number of factor nodes and the maximum cardinality for each variable should match the capacity of BRAM listed in table 5.1. Fortunately, Xilinx Vivado provides a convenient IP for controlling BRAMs. Fig. 5.7 shows the block design of the factor graph accelerator.

The accelerator design shown in Fig. 5.7 facilitates the interrupt mechanism so that when the factor product or the marginalization has been done, PL will send an interrupt signal which will be caught by the accelerator driver in the Petalinux running on the PS. The linux kernel then notifies the factor graph program that the sum-product acceleration has been completed and the factor graph's message can be fetched from PL. Fig. 5.7 also shows that the SumProduct module has a special input called factor_PORTA which facilitates direct access to the ROM containing an internal function of a factor node. This internal factor's function must be supplied in the beginning of the program also using the device driver mechanism generated by the Vivado-HLS. The number of the functions can vary (depending on the factor graph network being instantiated in the PS) and can be determined by the memory address allocated for the SumProduct module. The size of each factor's function depends on the number of scope variables and the cardinality of each variable.

We measure the efficiency of our method using the standard metric commonly used in FPGA-based designs; i.e. the clock latency for measuring the speed optimization result and the resource consumption in percentage for measuring the area optimization result. It has been described in section 5.1.3 that the optimization on one aspect (e.g. the area

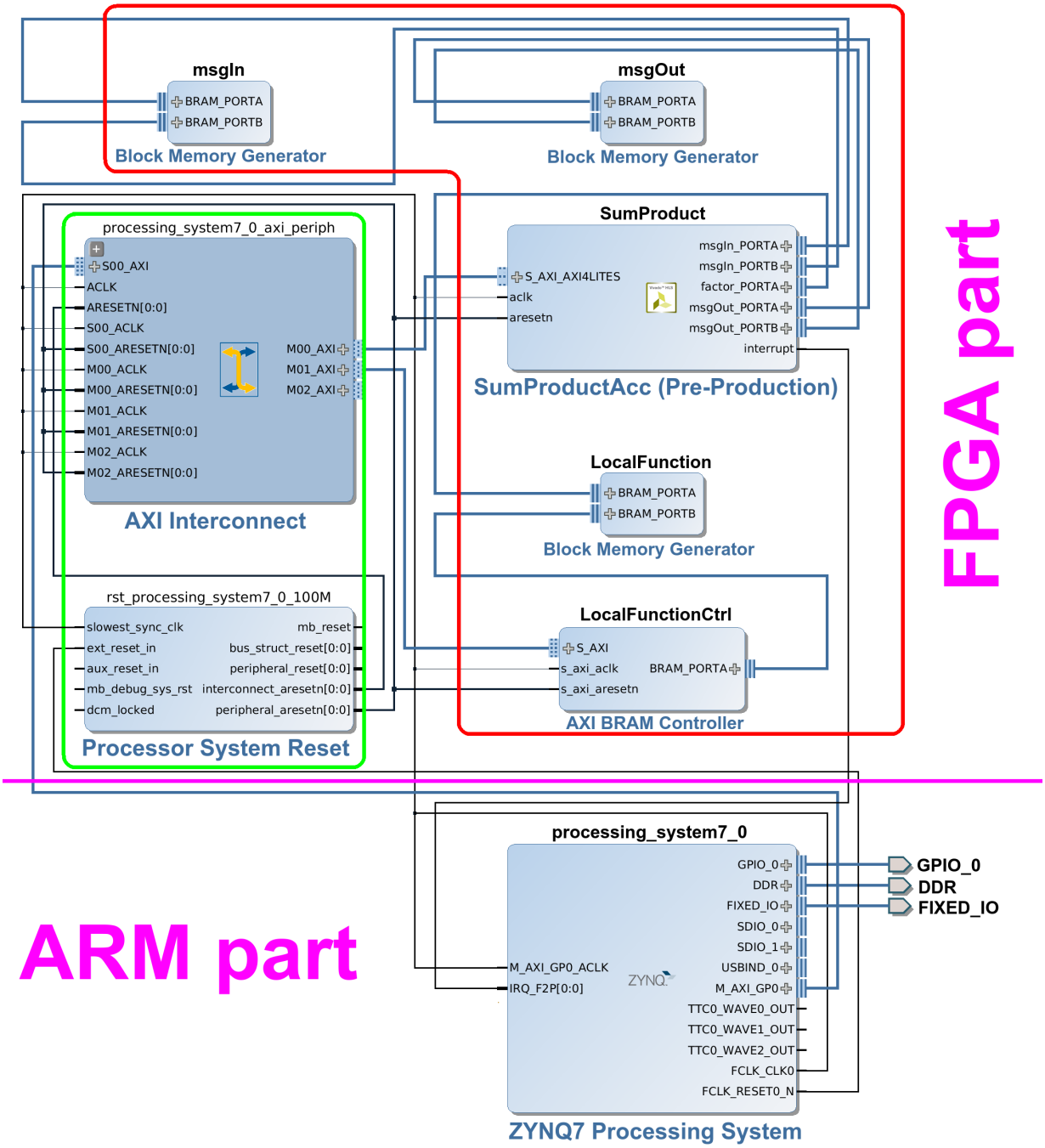


Fig. 5.7: Block design of the factor graphs engine in an SoC with FPGA as an accelerator. All modules in the upper region are implemented in the FPGA while the lower region represents the ARM processor of the Zynq-7000. The modules in the red block are the main elements of the accelerator while the modules in the green block are supporting elements that connect the accelerator to the ARM processor.

optimization) will affect the other aspect due to the limited resources. In this thesis, we are interested to explore these options balancing to find the best solution we can get for our factor graph. For this experiment, we use the networks shown in Fig. 5.8. All of variable nodes in the networks are observed; i.e. each node will have its corresponding factor input,

for example A will be connected to f_A, B will be connected to f_B, etc. Applying the unrolling and pipeline mechanism requires that the loop is a perfect or semi-perfect loop. A perfect loop means that the loop bound is constant while a semi-perfect loop might have variable bound but needs to apply an exit check protocol. Table 5.2 and table 5.5 show the comparison of our framework implementation with and without additional optimization.

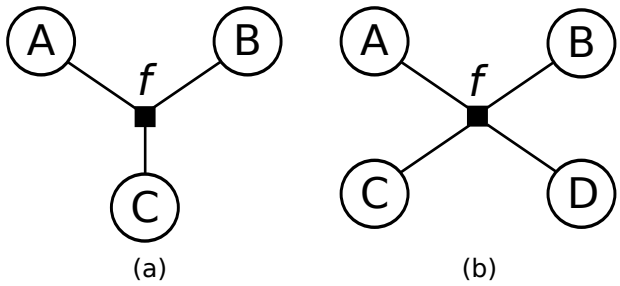


Fig. 5.8: Networks for test cases.

Tab. 5.2: Latency comparison between unrolling with pipeline vs unrolling without pipeline.

	Without Pipeline				With Pipeline			
	3-Variables		4-Variables		3-Variables		4-Variables	
Number of States	min	max	min	max	min	max	min	max
10	688	76129	9462	442182	675	34400	9279	160052
20	2574	284244	37848	1768728	2530	120978	36984	628728

Tab. 5.3: FPGA resources consumption (in %) in the pipelined and non-pipelined design of the network shown in Fig. 5.8(b).

	Without Pipeline			With Pipeline		
Number of States	BRAM	FF	LUT	BRAM	FF	LUT
10	48	9	27	48	19	51
20	96	17	52	96	39	83

Tab. 5.4: Comparison of FPGA resource consumption in the network with three and four nodes (in %). Both networks are fully optimized in term of speed (i.e. using both unrolling and pipeline mechanisms).

	3-Variables			4-Variables		
Number of States	BRAM	FF	LUT	BRAM	FF	LUT
10	16	8	20	48	19	51
20	32	14	42	96	39	83

In table 5.2, the value in min and max columns reflect the minimum and maximum clock latency that is required to move from one state to the next state in the FSM (finite state machine) implementation of the algorithm[3]. From these values, we can estimate how long it will take for the algorithm to run. These values are only estimations based on the given clock frequency in the Vivado-HSL and not the real clock frequency of the hardware. For example, in our design we usually specify the clock frequency to be 100 MHz, so that the value 688 means it takes 6.88 μs to complete the execution. In real hardware implementation, where the frequency clock is 667 MHz, the latency 688 is estimated to be completed in 1.03 μs. Likewise, for the maximum latency of 1768728, it will take roughly 17.687 ms in 100 Mhz clock systems (which will run effectively 2.6 ms in real hardware). In that table, we can see that the minimum values do not differ much for both the optimized and the unoptimized design, revealing the fact that there are some parts of the code that cannot be further optimized. Usually these values are related with the inter-block data exchange in the code. The maximum values, on the other hand, show a big difference between the optimized and the unoptimized design. Dividing the maximum value obtained from the unoptimized design by the value from the optimized design shows an average speed-up ratio of 2.53. We also observe that the clock latency is heavily affected by the number of states used to encode a factor graph's message, which increases exponentially.

Table 5.3 shows the optimization efficiency of the design with respect to the number of FPGA resources consumed by the design. BRAM is the distributed memory units mainly used for instantiating arrays in our design. The FF (flip-flop) and LUT (look-up-table) are the main constituents of the configurable logic block in an FPGA. It can be seen in the table that if we do not use the pipeline mechanism, we can create two independent networks with 10 states for each variable. However, if we implement the pipeline mechanism, there is no more space available to create another network even if we use only 10 states for each variable. With the pipeline mechanism, almost all the resources are consumed by the network with 20 states variables. The remaining question is then, how big is a network that can be synthesized when we apply the pipeline mechanism? The answer is shown in table 5.4. With only three variables connected to a factor node, we can instantiate up to two independent networks, either using 10 states or 20 states for the variable's cardinality. It means, we can implement the network shown in Fig. 2.6(b) on page 27 conveniently because adding one more hidden variable will not consume too many resources (since the hidden node H in that network only passes the message without performing any computation). Such a network is very common to be found in many applications. We have implemented it using 10 states for its variable's cardinality and observe that the maximum latency is reached at point 214056 (which corresponds to the predicted execution time about 0.32 ms in real hardware). The resource consumption is also increased up to 65% for LUTs and 28% for the FFs. The internal routing and mapping of FPGA resources for this network is shown in Fig. 5.9.

For the final test case with this method, we perform a complete inference test using the network shown in Fig. 5.8(a) and with artificial data generated similarly to the data we

[3]Every code which is written using C/C++ in Vivado-HLS will be transformed into the corresponding RTL format in which the behavioural process will be synthesized into FSM.

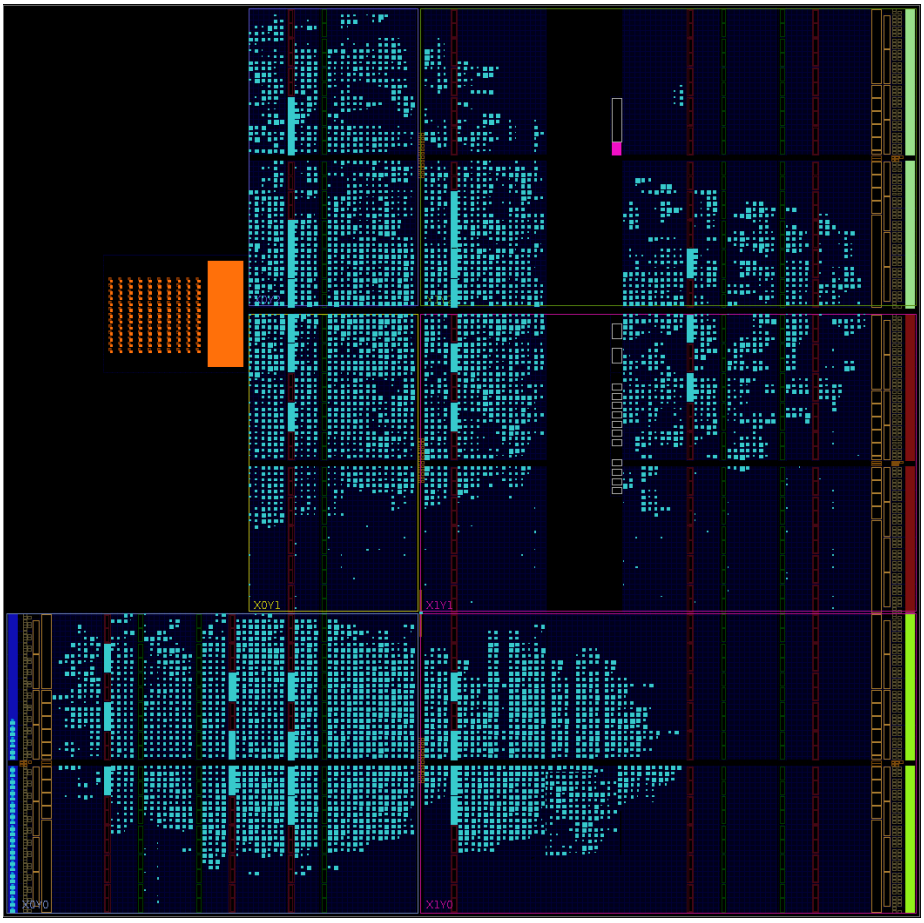


Fig. 5.9: Inside the chip: the factor graph accelerator program will be translated and mapped into FPGA resources (BRAM, DSP, FF and LUT) and scattered all over the chip to match the routing policy of the synthesizer.

used for sensor fusion application (see section 3.1.3 on page 52). We create a dataset with three variables and use the first variable A as the basis (ground-truth) for creating data for the other variables (the other variables are just the noisy version of the data in variable A). The inference task in this case is computing the marginal probability of variable A given input data for variable B, and C. For this test case, we run the MLE algorithm on our PC-based factor graph to learn the parameters of the factor node f and send the results to the embedded factor graph running on the SoC. First we create the network and run the belief propagation on it using only one processor of SoC[4]. We feed the dataset to the network, collect the inference result and send it to the host PC. Next we modify the program to use the accelerator, re-create the network and re-run the belief propagation again using the same dataset as before. The resulting data from the inference is sent to the host PC.

The combined result of these two runs is shown in Fig. 5.10. As expected, the accelerator can speed-up the factor graph computation with a ratio almost reaching 8-times when the variable's cardinality is 25. This proves that the optimization strategy in our module

[4]Since Z-7020 has two microprocessors in it, we configure the linux kernel to use only one core and disable the other core.

is implemented successfully. Unfortunately we could not test the four-variables network shown in Fig. 5.8(b) with the number of states higher than 25 using our current hardware, but we argue that our accelerator can be extended further given a denser FPGA part of the SoC (for example, the moderate SoC chip in the Zyng-7000 family, that is Z-7035, has a capacity as much as four times of our Z-7020). This is contrast with the factor graph without an accelerator which runs only on the microprocessor of the SoC. Without the accelerator, we can use any number of variable's cardinality, but at the cost of slow performance.

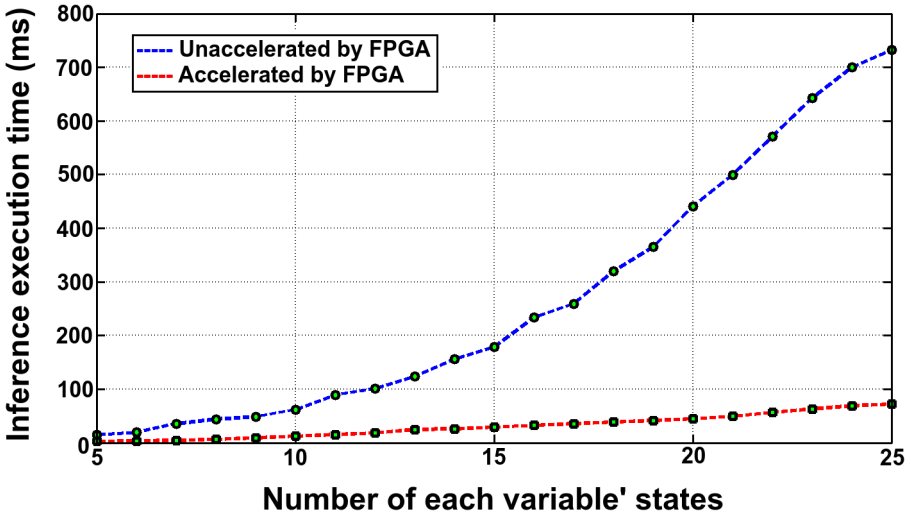


Fig. 5.10: Performance comparison of inference execution between accelerated- and not-accelerated mode by FPGA.

In summary, our accelerator module works well and arguably can be extended into a more powerful module in a denser chip. Currently we cannot handle a big network with high precision but the acceleration result scales up with the increasing size of the network's parameters. Our current hardware for this accelerator mode produces optimal results when factor nodes only have three scope variables. Adding more connected variables to the corresponding factor node requires a modification on the module because we have to allocate more memory space in LUTs instead of the BRAMs. Also, our current implementation of the accelerator still needs bridging access via the microprocessor to the external RAM where the factor parameters are stored. This in turn will slow down the performance. A better solution will be using a direct memory access (DMA) to the external memory. This is an interesting idea which needs to be explored further. However, the DMA access from the PL requires the use of special IP for handling this mechanism. To our knowledge, this IP for using DMA via AXI bus will consume a considerable amount of FPGA resources (up to 10%) which is impractical for our current hardware (see [213] for pros and cons of using DMA in a design). Considering this trade-off, we decide to rule out this idea in our current implementation.

5.3 Method-2: Factor Graph Framework on FPGA

Using the FPGA as an accelerator is proven to be useful for speeding up the belief propagation computation in a factor graph as shown in Fig. 5.10. However, method-1 has two limitations:

1. It is still heavily affected by the sequential process of the ARM-processor for encoding and decoding messages as well as delivering message(s) from node to node. Thus, it cannot work in a truly parallel fashion in a factor graph network with more than one factor node since the program running on ARM-processor will determine the scheduling of the messages propagation (i.e. which node will firstly send the message to the other nodes).

2. It is not easy to extend the IP core of the accelerator for a larger network whose nodes reside in another chip.

In order to make a generic implementation of factor graphs in FPGAs (or SoCs), we extend our first method and develop the second method to implement a whole factor graph framework in the FPGA. In this method, we want to maximally exploit the abundance of FPGA's logic fabrics and its routing channels. Inside the FPGA chip, there are numerous signal lines that interconnect the logic blocks within the chip. These lines can be arranged and managed to create buses which later can be customized for our embedded factor graph. Fortunately, the design software makes this interconnect routing task hidden to the user unless specified otherwise, thus significantly reduces design complexities.

From our experience with the SpiNNaker system, it becomes clear to us that the routing management is the key to successfully implement an efficient embedded factor graph (see the section 4.4 on page 102). In the SpiNNaker system, our embedded factor graph is constrained by the router component of the SpiNNaker chips which limits the flexibility of the network construction. We propose a solution for this problem by simplifying the network such that it contains only triple connections at maximum for each node. However, we did not implement such a solution yet with the SpiNNaker system but we use such an idea for developing our second factor graph framework on SoC.

In addition to the similar goals we set to our SpiNNaker version (see section 4.2 on page 90), we also target the second method towards the following criteria.

1. **Continuity** The framework should be able to be used in a simple transitional step from its original PC-based framework. This means, when we have finished simulating a factor graph network on a PC (using our PC-based framework), the network should be able to be translated into the SoC-based embedded version seamlessly. A small modification might be required but it should not burden the application developers themselves.

2. **Modularity** Having an environment which offers a high degree of flexibility, the modules in the SoC-based embedded factor graph should be flexible enough to be reconfigured or remapped into a new, possibly denser, SoC chip.

3. **Platform-friendly** The modules should be flexible enough to be re-synthesized on the new version of the development environment, both for the FPGA's bitstream generation and the Linux kernel reconfiguration. As we have described in section 5.1.2, this co-development paradigm requires a careful design of both hardware and software for the resulting embedded system to work efficiently.

With these goals in mind, we develop our new full embedded factor graph on SoC. This new framework is composed of several modules with different microarchitectures.

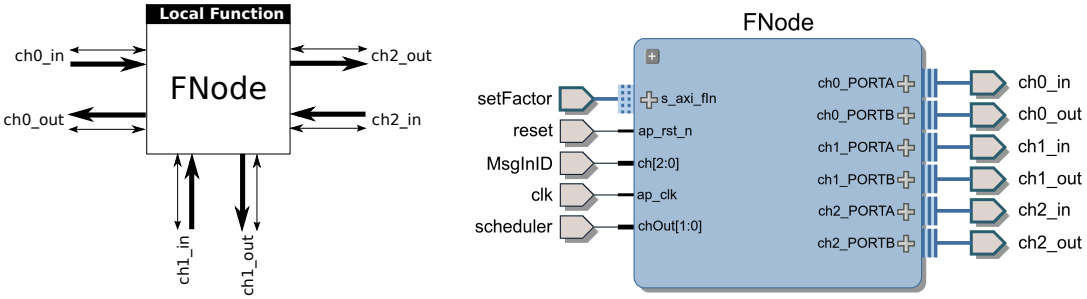
Similar to our previous attempt on creating embedded factor graph in a SpiNNaker system, we identify the following core modules necessary to build a complete discrete factor graph based on population coding: factor and variable nodes, message encoder/decoder, and scheduler. In this thesis, we regard the factor/variable node and message encoder/decoder as the most generic parts of the system where we can use them almost in any scenario, while the scheduler is a flexible module which can be tailored according to the application scenario. Hence, we leave the scheduler part as a "template" that should be modified according to the application. For some applications where the network has loops, then the scheduler should be modified to meet the preferred scheduling strategy (e.g. synchronous, asynchronous, residual, etc.). At the moment, we only provide the default scheduling for tree-like network as asynchronous.

5.3.1 Factor and Variable Node Controller

Basically there is nothing new in the factor and variable node implementation in this second method. We replicate the factor and variable node classes in our PC-based factor graph framework into the FPGA version. We then optimize the module with the following default setting: unrolled into N instances (where N is the variable's cardinality) and pipelined with one initiation interval (i.e. fully pipelined without resource sharing of operators).

The basic algorithm for the factor node controller and the variable node controller is the same because both employ the same message-passing algorithm in equation (2.4) and (2.5). However, they differ only in the internal factor product operation which only happens for the factor node. We first create the factor node controller and then reduce its operation (i.e. removing the internal factor part) to create the variable node controller.

Fig. 5.11a shows the functional symbol of a factor node controller (marked with the name FNode). It represents three channels bidirectional module that can be connected up to three variable nodes. Fig. 5.11b shows the resulting IP-block representation already included in the Xilinx Vivado IP repository after its successful synthesize. The internal factor's function can be supplemented into the controller using a simple function call which is already encapsulated in the common AXI interface together with the main function of the module. The internal structure of this FNode module is shown in Fig. 5.12.



(a) Functional Symbol **(b)** IP Symbol

Fig. 5.11: FNode symbol representation.

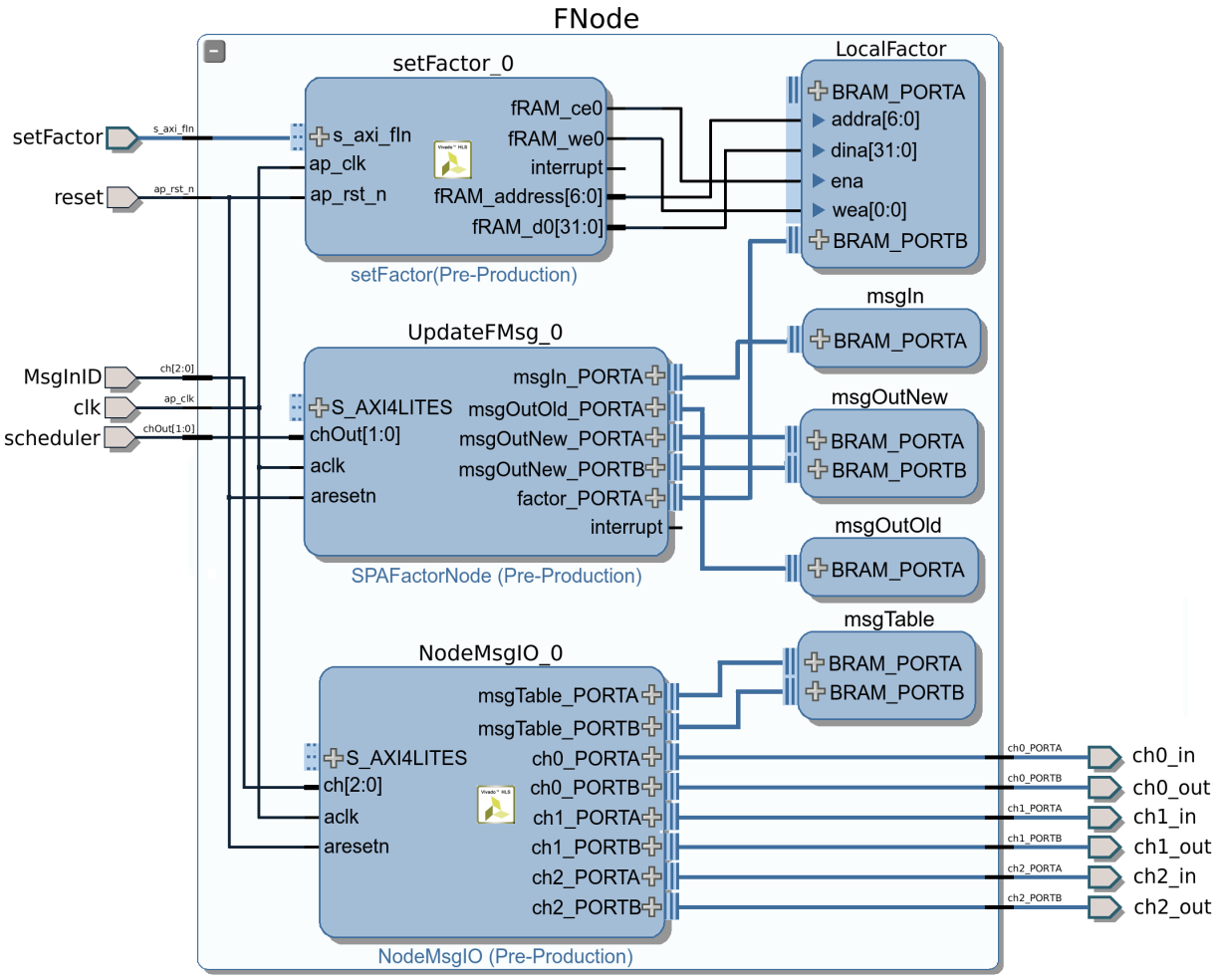
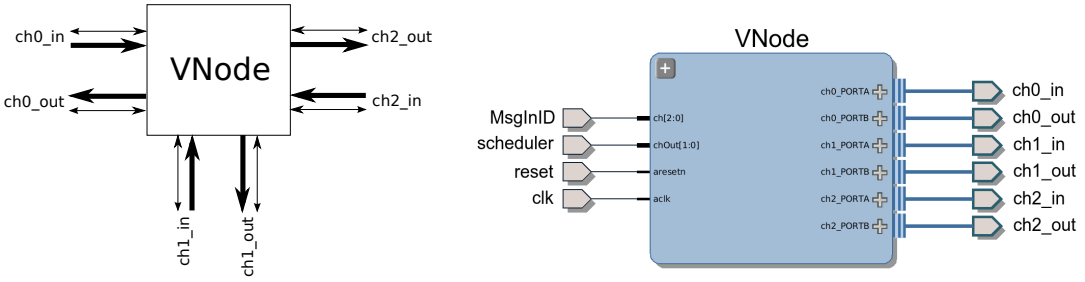


Fig. 5.12: Internal block diagram of module FNode shown in Fig. 5.11b.

Within the FNode module, there are four sub-modules: setFactor, UpdateFMsg, NodeMsgIO, and LocalFactor. Beside this four sub-modules, the FNode module require several instances of BRAM (or can be replaced with distributed memory using LUTs) to facilitate data array processing. The sub-module setFactor and LocalFactor work together and are responsible for managing the internal factor's function of the corresponding factor node. The value of this internal factor's function can be set by using standard function

call which is already available in the device driver to make it convenient when using the embedded Linux the PS part of the SoC. The sub-module NodeMsgIO is responsible for receiving and delivering messages from/to external variable nodes. It contains three bidirectional channels and has a small table which tracks the messages' exchange during the transaction. Whenever a new message arrives, this table will be updated and the sub-module UpdateFMsg will be notified when incoming messages are complete and ready to be used for computing the output message. The sub-module UpdateFMsg is the core module where we implement the sum-product algorithm. This sub-module also has an internal memory to save old messages which can be used for example during the training using MLE or EM algorithm in message-passing scenario. This sub-module also has scheduler input signals that can be used to control the scheduling of the message-passing exclusively on this FNode instance. By default, we implement the asynchronous scheduling which is the standard scheduling for an acyclic factor graph.

From the module FNode, we derive the VNode module which is responsible for handling a variable node in a factor graph. In principle, the VNode module works similar to the FNode module but without the functionality for setting up an internal function. The functional symbol and the IP symbol of this VNode module are shown in Fig. 5.13 while its internal structure is shown in Fig. 5.14.



(a) Functional Symbol **(b)** IP Symbol

Fig. 5.13: VNode symbol representation.

There is one more input signal which is present in both FNode and VNode modules and responsible for giving an ID to a message. This input signal is called MsgIn ID and is preserved for future improvement of our embedded factor graph. Using this signal, the node can identify if this message is the same message circulating in the network or this is a new message. This mechanism is useful for solving the cycling error in a cyclic factor graph. We borrow this idea from the SpiNNaker system which is called Errant Packet Trap mechanism [2]. However, we do not fully explored this mechanism yet and we simply set its counter to be 0, meaning that the incoming message is always assumed to be a new one. As a consequence, if our embedded factor graph is going to be used for implementing a cyclic network, then the software running in the PS must check manually if there is a cycling error in the network.

5.3.2 Message Encoder and Decoder

We implement a special module responsible for encoding and decoding messages called NodeIO. This module is a direct implementation of our population coding algorithm pre-

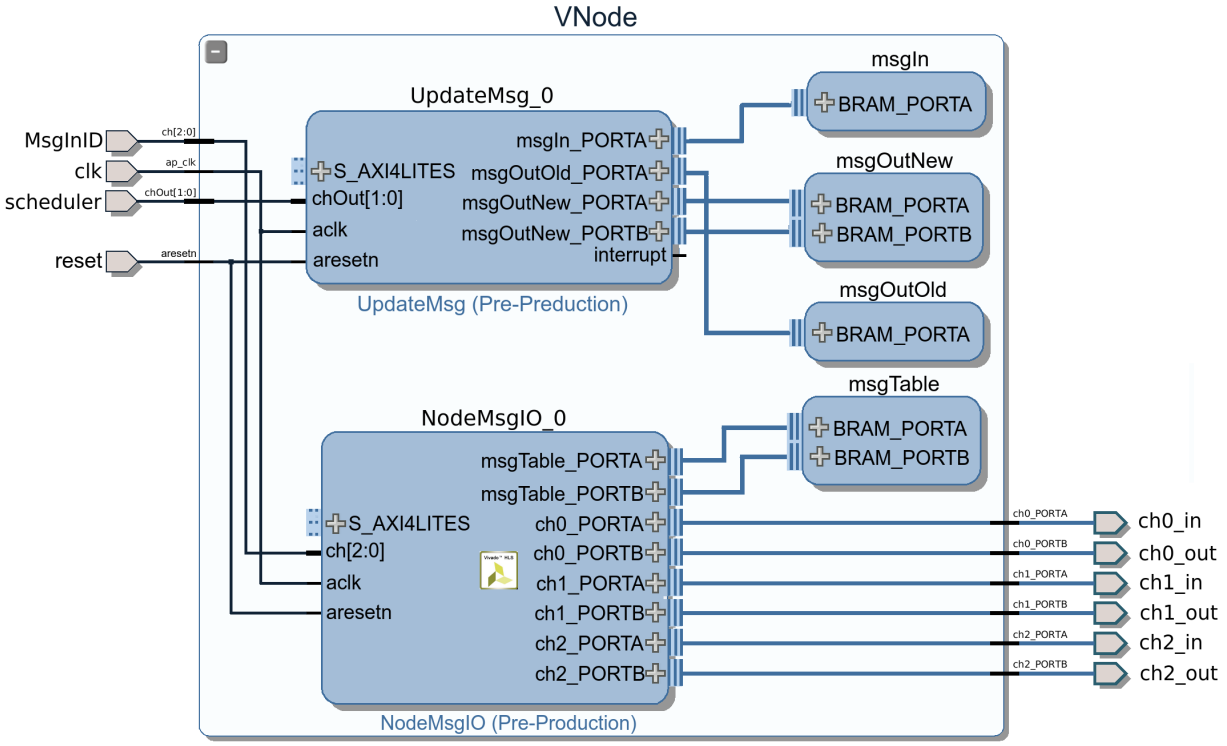
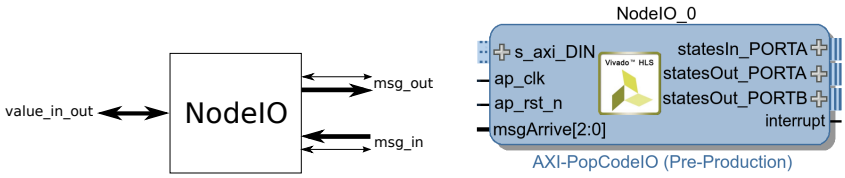


Fig. 5.14: Internal block diagram of module VNode shown in Fig. 5.13b.

sented in chapter 2. Regarding the structural complexity of the design, this module is simpler than the FNode or VNode modules and contains only a single bidirectional channel. The functional symbol and the IP symbol of this NodeIO module are shown in Fig. 5.15.



(a) Functional Symbol **(b)** IP Symbol

Fig. 5.15: Node-IO symbol representation.

Basically this module works as follows. This module should only be connected to a VNode module because it represents the input factor of the corresponding variable node. Once this module is connected to a VNode (a variable node), we can send a value to the respective variable node via kernel's function getStates(). This module then encodes the value into a probabilistic vector and sends it to the variable node via its output channel. Afterwards, that vector will be propagated in the network as a message. At some point in time, this NodeIO module might receive a message from its variable node. When a new message arrives, this module will start decoding the message and when it's done, it will raise an interrupt informing the hardware driver that a new value is ready to be picked up. The program in the PS then able to retrieve this value via calling the getRealVal() function of the kernel.

In summary, the sending of a value to this NodeIO module will trigger the belief propagation automatically. This message-passing circulation will stop in a definite time if the the network is an acyclic one. Reading a message arriving at this NodeIO corresponds to the reading of the marginal value of the corresponding variable node. There is no guarantee, however, that the message circulation will converge (or not) in a cyclic network. To stop the circulation, the software needs to tell all nodes in the network by sending a value -1 to the scheduler of each node.

5.3.3 Putting Them All Together

Here we give an example of how to use our embedded factor graph for constructing a network. An example network, shown in Fig. 5.16(a), has five variable nodes, two factor nodes and four input nodes. Assume that we have learned the factor's parameters using the PC-based factor graph framework, we can send those factor's parameters to the factor f_{ABH} and f_{CDH} by calling setFactor() through the kernel's driver prior to running the belief propagation on the network. We translated the network in Fig. 5.16(a) into its symbolic diagram depicted in Fig. 5.16(b).

As it can be seen in Fig. 5.16, building the factor graph network using our framework is straightforward: given a structural description of a network, we can translate it directly to the symbolic representation. The next step is to draw such diagram in the Xilinx Vivado Diagram Editor and fetch all corresponding IP blocks from the repository. The complete diagram for this network is attached in Appendix-C.

5.3.4 Evaluation

Similar to our first method, here we want to evaluate the modules used in this second approach in terms of their clock's latency characteristic as well as their consumption on the FPGA resources. Since the optimization using pipeline mechanism works only with perfect or semi-perfect loops, we have to specify manually the cardinality of the variables. This number will be hard-coded in the source code before synthesizing it. For the evaluation, we specify only five states as the default variable's cardinal value. This cardinal value is the highest value we can achieve in our hardware due to limited resources of the Zynq Z-7020. As before, we compare the overall estimated performance on two scenarios: fully optimized and unoptimized designs. In a fully optimized design, both unrolling and pipeline are maximized. Table 5.5 summarizes the characteristics of those modules in a fully optimized design.

The clock latency parameters in table 5.5 represent only the maximum values; i.e. these are the longest delay we will get during execution. For the NodeIO module, the encoder section in general has a higher profile than the decoder section. This is not surprising because in the encoder part, the sampling of each tuning curve will take more time and resources due to the use of exp() function to compute the Gaussian distribution. Comparing to the other module (FNode and VNode), the clock latency is much lower, revealing the fact that the encoding and decoding of a message will not create a bottleneck for the entire system. For the FNode and VNode modules, their latency do not differ much since basically those modules originate from the same source. With the average clock's latency

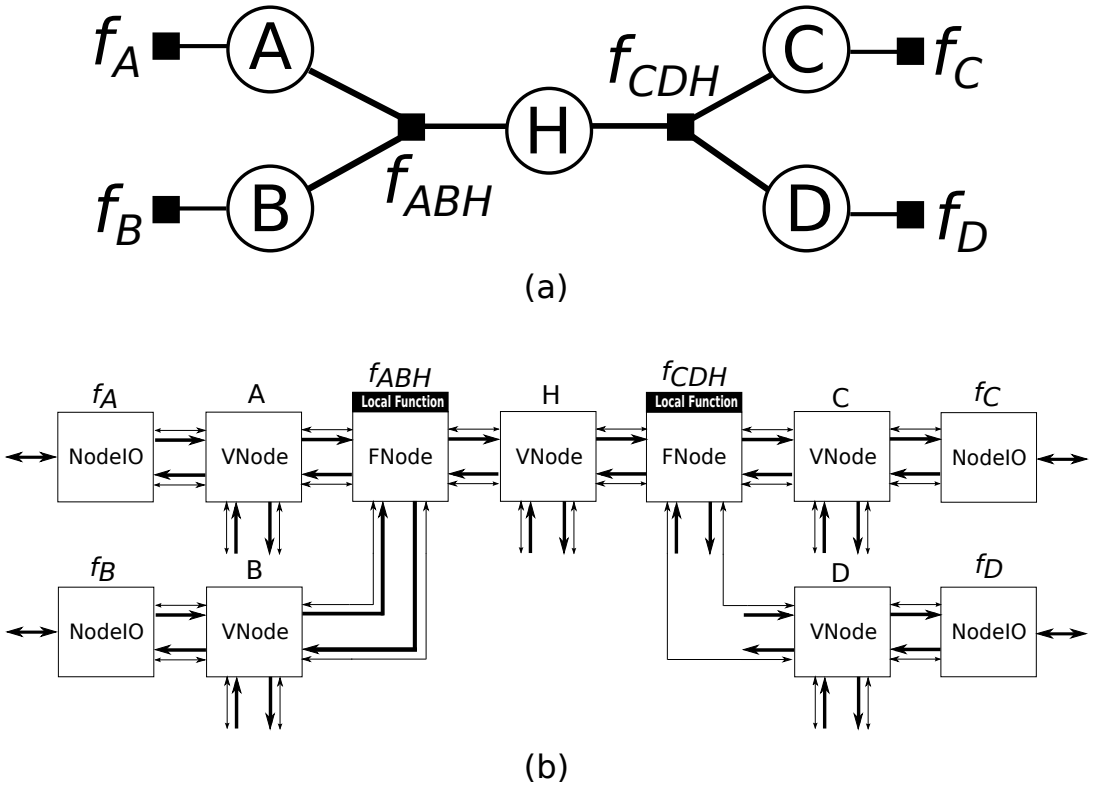


Fig. 5.16: An example of how to construct a factor graph network using core modules of our factor graph framework. (a) The original network consists of two factor nodes, five variable nodes, and four IOs. (b) The network in (a) is re-constructed using core elements of our factor graph framework and later can be implemented in the FPGA.

Tab. 5.5: Summary of latency characteristic and FPGA's resources consumption of the main modules in our embedded factor graph.

	NodeIO		FNode			VNode		
	Encoder	Decoder	MsgIO	Factor Product	Sum	MsgIO	Factor Product	Sum
Latency	7780	1829	242	163030	11747	242	163542	10609
BRAM (%)	5	5	2	14	12	2	12	10
DSP48E (%)	7	3	0	2	2	0	2	2
FF (%)	4	3	1	17	10	1	14	10
LUT (%)	12	3	1	9	6	1	7	4

of 160529, it will spend about 0.24 ms for the node to compute an output message when we run the system with frequency clock 667 MHz.

Comparing the FPGA resources (BRAMs, DSPs, FFs, and LUTs) usage, still FNode and VNode modules show similar characteristics. In total, the NodeIO module will consume 10% BRAMs, 7% FFs, and 15% LUTs. For the network shown in Fig. 5.16, the total NodeIO modules will consume 40% BRAMs, 21% FFs, and 60% LUTs. The total FNode modules will consume 56% BRAMs, 56% FFs, and 32% LUTs. The total VNode modules will consume 120% BRAMs, 20% FFs, and 60% LUTs. With this excessive resource con-

sumption, unfortunately we could not synthesize the network with our current hardware although we were still able to simulate the network using a denser SoC chip such as Z-7035 or Z-7045. The unoptimized design, however, requires lower resource consumption. Table 5.6 shows the unoptimized design also using five states for variable's cardinality.

Tab. 5.6: Summary of latency characteristic and FPGA resources consumption for implementing the network shown in Fig. 5.16 using only five states for each variable's cardinality. The modules were synthesize without further optimization. The report was generated for Zynq Z-7020.

	NodeIO		FNode			VNode		
	Encoder	Decoder	MsgIO	Factor Product	Sum	MsgIO	Factor Product	Sum
Latency	58350	12986	1912	1092301	81054	1912	1112086	74263
BRAM (%)	3	3	1	5	5	1	5	4
DSP48E (%)	2	1	0	1	1	0	1	1
FF (%)	1	1	0	4	2	0	3	2
LUT (%)	3	1	1	2	1	1	1	1

Using the simple module design (without optimization), the total resources for NodeIO modules are: 24% BRAMs, 8% FFs, and 16% LUTs. The total resources for FNode modules are: 22% BRAMs, 12% FFs, and 8% LUTs. The total resources for VNode modules are: 50% BRAMs, 25% FFs, and 15%. Combined for implementing the network in Fig. 5.16 using Zynq Z-7020, it requires 96% BRAMs, 45% FFs, and 39% LUTs. It can fit into our small density SoC but with very long delay about 3 ms for a node just to generate the output message before propagate it out the other nodes.

In summary, the trade-off between area and speed optimization is really a daunting process. From the above analysis, we observe that this second method is a resource-hungry approach. Even with low variable's cardinality and without optimization, it almost consumes the entire FPGA main resources. Hence, we propose the solutions for this problem as follows. First, the NodeIO module should be modified such that one instance of this module can be used by all nodes in the network. A gating mechanism for determining to which node the encoded message will be delivered could be used to incorporate this approach. Of course, a new delay effect will be introduced to the system but we can ignore it in a big network since the total delay will be dominated by delays in the message-passing computation. Second, there are nodes without intense computation in its microarchitecture, especially the second order variable nodes. For these nodes, it is beneficial if we do not create a complete working node by deriving it from the FNode module. Rather, we create a simple message forwarding module. With this approach, we will have different modules for variable nodes; the choice of using it will depend on the degree of the corresponding variable. Third, the optimization strategy can be set to a moderate level to make a balance between the area and the speed. Finally, using several FPGA/SoC chips in one board (similar to the SpiNNaker board shown in Fig. 4.3a on page 92) will give much more flexibility.

5.4 Discussion

In this chapter, we present two different strategies for using an FPGA/SoC as the platform for an embedded factor graph. In the first approach, we propose to use the FPGA part in the SoC as the accelerator for the factor graph framework running on the PS part of the SoC. In the second approach, we propose to build a full factor graph framework along with its belief propagation scheme. These two approaches have their own advantages and limitations. For the first method, the main advantage is that, once the module has been synthesized, it can be used in almost every factor graph running on the PS without modifying and re-synthesizing the module. Hence, it is advantageous for fast prototyping of a factor graph application. For the second method, it offers more flexibility and will run the belief propagation faster compared to the first method, especially in a large network. Unfortunately, the second method requires a lot of FPGA resources in its implementation.

In section 5.3 we describe our implementation strategy for factor graphs in the FPGA side of the Xilinx Zynq SoC. A typical question that will arise regarding the routing mechanism is that, why don't we use the network on chip (NoC) approach so that the network can be expanded to a larger system? Actually, we do implement such a concept but with very limited capabilities. In order to know where the signals which contain message data should be delivered, we have to "inject" an ID of that message. Since we develop the system on a node-basis, this simple routing mechanism is implemented directly on the node. With this approach, we can save a lot of FPGA resources, since implementing a complete and generic NoC is a resource consuming task for FPGA-based designs [214][215]. We also agree that a reconfigurable NoC should be used when the granularity of the network is the main issue, which is not the case with our current factor graph network. We also have learned during SpiNNaker exploration that there are many aspects regarding the chip's semiconductor itself that need to be taken into consideration to avoid an excessive buses access, for example in the case of non-blocking concurrent access to the shared resources [216][217].

A small drawback of our "simple" NoC approach is that our embedded factor graph on the FPGA still requires extra efforts on the side of the factor graph network designer. However, if we can create a user interface that automatically assigns/allocates the FPGA resources for this NoC mechanism, then we do not have this problem any more. We consider this flexible user interface as a potential improvement for our future system. The work of Czajkowski (see [202]) can be used as the starting point for such further improvement.

In our models, we do not use external memory for storing messages, rather we simply maximize the utilization of the existing BRAMs and LUTs. The main reason for this choice is that accessing the external memory independently requires more resources, which in turn will not leave enough resources for the factor graph modules themselves. Controlling external memory needs explicit routing strategies in order to match the interfacing protocols and timing constrains required by the memory hardware. If we rely on the PS for intermediating between the FPGA and the external memory chips, then we have to implement the DMA protocol. Unfortunately, this DMA mechanism (using AXI-DMA IP in the Vivado library) is not a good choice for our current hardware because it consumes a considerable amount of FPGA resources.

6 Evaluation and Outlook

6.1 General Evaluation

6.1.1 On the Applicability of our PC-based Factor Graph Framework

In the initial stage of our research, we focus on developing our own PC-based factor graph framework in order to get a thorough and deep understanding of how the optimal factor graph along with its inference process can be developed. Our PC-based factor graph is developed using C++ and uses the Qt framework for implementing a message-passing mechanism using SIGNAL-SLOT mechanism. From several examples implemented using our PC-based factor graph framework, we have very good confidence to apply the framework for broader range of applications. More importantly, it shows very good potential to be successfully implemented in a dedicated hardware. We also show the improved performance in terms of computation speed when we implement our factor graph in heterogeneous parallel platforms such as a multi-cores computer and a GPU. Although this is not our main goal, it can be a good starting point for researchers who want to exploit heterogeneous parallel platforms for running more complex and challenging factor graphs.

In our PC-based implementation, there are two important aspects that we have explored in this thesis. The first is the learning mechanism in a discrete factor graph. We show that by using the same inference mechanism, the parameters of the network can be efficiently learned. If we provide complete data to the network (i.e. without any missing/hidden variables), then the belief propagation algorithm can be used conveniently to learn the network's parameters by estimating the maximum likelihood of the factors in the factor graph. However, when the data contains missing/hidden information, we have to use an iterative process to estimate the parameters. Motivated by the work of Gösdorf [73], we explored and improved the existing expectation maximization (EM) algorithm and we found out that we can still use the same belief propagation algorithm in an iterative way to produce an approximation to the likelihood functions of the factors. Further exploration on the EM algorithm, leads us to the development of its variant. In this extension, we combine sum-product and max-product algorithms for estimating the parameters. Both sum-product and max-product algorithms can use the same principle of message-passing. Although it is natural to use this approach as an extension to the standard EM algorithm, we found that this approach is a bit trickier than the standard EM algorithm. Regarding our goal of implementing the belief propagation algorithm in a dedicated hardware, we argue that this "combination" approach is not suitable for hardware with very limited memory resources. The reason is, the max-product algorithm that we want to use as the maximization step of the EM requires keeping all records of the previously found maximizer state before the final state will be selected as the most probable state that produces the maximum probability. Although it is easy to be implemented in a PC, we decided not to

implement this combination approach in our current hardware. It does not mean that this approach should always be discarded, because if we have higher capacity hardware, this approach will be implementable and give better approximation result then the standard EM algorithm. Hence, we consider this challenging method for our future work on the embedded factor graph.

The second is the generality of population codes in a discrete factor graph with the message-passing as its inference algorithm. In our method, we use a homogeneous neurons population to represent states of the discretized value. After proper SOM training, the neurons in the population then specialize themselves to correctly represent regions of the variable's domain. There are many methods instead of our population coding that can be used for discretization. We found several techniques which have similarities to ours in term of approximating the continuous value using a "specialization" approach. For example, the method developed by Achan et.al. (see [218]), uses non-uniform expansion of hypercubes which tries to localize interesting features (such as sharp peaks) in the marginal belief distributions. However, their method, which also employs a greedy heuristic approach to select a partitioning with low entropy, seems to be computationally expensive and give good accuracy only on specific problems. Another method proposed by Noorshams et.al. (see [101]) tries to approximate the continuous value in two steps of approximation: a deterministic approximation that involves projecting messages onto the span of r basis functions, and a stochastic approximation that involves approximating integrals by Monte Carlo estimates. Their deterministic approximation is quite similar with the population coding approach in that the continuous value is projected onto a span of basis functions. Recall that our population coding approach basically also projects the value onto several radial basis functions represented as neurons. However, their method is focused exclusively on models with pairwise interactions, i.e. binary factor graph. Our population coding, on the other hand, can be used in any degree without necessary modification. Compared with standard methods for discretization in statistical theory, our population coding approach combined with the SOM technique for learning the neurons specialization shows some similarities with the MDL-based approach, in particular the one described in [51]. In principle, both methods use the information content as the basis for dynamic partitioning.

6.1.2 The Mapping Strategy on the SpiNNaker System

The first hardware platform in which we implemented our factor graph framework is the SpiNNaker system. The idea is to use all available chips on the board as much as possible for performing fine-grained parallel computation. This parallelism paradigm requires that all cores should work as an ensemble by distributing workloads among them equally. The challenge is that the SpiNNaker system uses asynchronous communication protocol whereas our PC-based factor graph framework was designed in a synchronous fashion. Due to this fundamental difference, we split the task and transform it into a mapping problem. As a result, we have two different mapping strategies: population codes mapping and factor graph's nodes mapping. This in turn requires different routing management. To communicate the messages between these two mapping strategies, we exploit the MC packet transmission protocol. The result of the implementation shows us that there are several issues regarding the message distribution in our embedded factor graph.

First, since we use the multicast (MC) packet routing protocol for transmitting an array of a discretized message, we have to predefine the one-to-one routing path in advance. This is because in the MC packet protocol, we can only supply the destination core encapsulated in a masked pattern. We exploit the routing key part of the MC packet to carry information such as the state ID of the corresponding message dedicated to a certain variable ID and the operation that we want to execute in the destination core. The state's value itself is carried on the payload part of the MC packet. These MC packets distribution might be triggered by the encoder part (which resides in chip 0,0) or by any nodes in different chips. The problem is that we have to create the routing table consisting entries of all possible source-destination pairs. In this thesis, this routing table entry description is done manually since we are not yet targeting a generic embedded factor graph on the SpiNNaker system. This generic embedded factor graph will require in-depth exploration in order to produce an automatic partitioning and routing manager similar to the PACMAN of the current SpiNNaker application framework for spiking neural networks. Although we have managed to create an example factor graph on the SpiNNaker, in general this is an exhaustive process and prone to error. Due to this circumstance, we might need to adapt the already existing PACMAN framework in the future to incorporate the configuration steps of our discrete factor graph in addition to its standard configuration menu for spiking neural network applications.

Second, our partitioning scheme for the routing table leaves the question of the maximum capacity the implemented factor graph can support. This capacity corresponds to the maximum number of connections a factor node can have and also the maximum number of states the factor graph's message can have. For example, if we define that the variable's cardinality is 60 and create a factor graph which has a factor node with four variables as well as assigning 15 cores for that factor node, then we will end up in a 256 entries in the routing table. However, if we only have 10 cores available for that factor node, then we need to assign 1296 entries in the routing table, which is not possible. Hence it is a tricky implementation strategy. To solve this problem, we might turn our attention to the FFG discussed in section 4.4 on page 102.

Third, we extensively use the MC packet routing protocol to transmit a factor graph's message and we only use SDP protocol for receiving in or sending out the data from/to the host PC. It might be useful if we also use the SDP protocol for transmitting factor graph's messages within the board itself. Our MC packet routing table design seems reliable enough in a small SpiNNaker system (such as the 4-chips board shown in Fig. 4.3a on page 92). However, we did not test its efficiency in terms of its actual throughput on a bigger system yet (such as the 48-chips board shown in Fig. 4.9b on page 103). According to the SpiNNaker datasheet [2], the SDP mechanism is fast enough for the communication within the chips direct interconnection. It might be useful, for example, to use the SDP mechanism to send a large chunk of discretized message to a distance chip rather than to send the chunk piece-by-piece using the MC packet routing protocol. Hence, for the improvement of our factor graph on SpiNNaker in the future, we might consider this new mechanism in addition to the idea of enhancing the PACMAN framework for our factor graph.

To summarize, we have not created an automatic packet routing management for factor

graphs on a SpiNNaker system yet, but we have laid the foundation of a generic factor graph framework on a SpiNNaker system. From our experiment, we have gained some insights to further improve our embedded factor graph. The same insights have inspired us to develop the second embedded factor graph using a SoC.

6.1.3 The Factor Graph on a Chip

The second hardware platform for our embedded factor graph is the Xilinx SoC Zynq-7000. We propose two types of implementation: an accelerator vs fully embedded factor graph. In the accelerator mode, the FPGA inside the SoC is responsible only for performing the paralleled version of the factor product and summation in the belief propagation algorithm. While in the full mode, the FPGA is used for implement all aspects of our discrete factor graph. Much of the in depth evaluations have been given in section 5.4 on page 129. Here we raise another important issue: the operating system where our embedded factor graph will run.

One might observe that the second method does not produce a ready-to-go design with which a non-hardware developer can use without touching too much the design entry in HDL domain. This is an unfortunate circumstance, because FPGA-based system design will always require resynthesizing and eventually regenerating the bitstream of the design. It does not mean that once we generate the bitstream, the hardware-side development is done. We still need to make sure that the higher level abstraction programmer (i.e. application programmer) can use our hardware. Actually, the bitstream generation triggers the next step in embedded system design: developing the driver which is accessible through a simple API (application programming interface) [219][220]. This is a very challenging topic and we do not cover all aspects of it except the most important issue related to the kernel configuration.

Fortunately, the Xilinx Vivado-HLS provides a convenient way to produce the hardware's driver for every module designed using the Vivado-HLS. However, it appears to us during the development of our embedded factor graph that the driver produced by Vivado-HLS seems to be intended for a standalone system[1]. An empty template for the driver usage in a Linux-based operating system is provided by Xilinx but needs to be adapted and completely modified according to the targeted Linux system such as Petalinux [221]. Xilinx argues that this choice should be left to the user since there is no consensus on what is the best way to invoke the driver in an embedded Linux kernel. Some suggests that the driver should be automatically loaded when the Linux kernel start but others argue that the driver should only be loaded whenever the users want to. These two different paradigms are related to this question: will the driver run in kernel space or in user space? [222]

In this thesis, we use Petalinux as our embedded Linux platform and we have compiled the driver differently. For our first method, we prepare the system to run in kernel space;

[1] In Xilinx terminology, a standalone system means a running program on a microprocessor without invoking a special supervisory program known as kernel. The standalone system is usually contrasted to the system which uses an operating system. Both the standalone system or the operating system can run either on the existing ARM core of the SoC or any soft-processor that can be created on top of the FPGA logic fabric such as Microblaze or Picoblaze.

hence, the driver will be loaded automatically when the Petalinux starts. The reason for this decision is that the accelerator driver is small enough to be loaded during kernel start and there are not so many functions that should be handled in the kernel side. In our experiment we provide two hardware instances for the accelerator module. The first instance is intended to be used by a kernel which runs in user space using UIO mechanism. The second instance is intended to be used in the kernel space. The advantage of running the driver in user space is that the user's factor graph program will be completely independent and not affected by the operating system update. This is because running the driver in kernel space means we have to recompile the entire kernel when a new modification has been made to the hardware that affects its driver. This might be a preferable choice for most future users of our module. But for a long-time run in a robust manner, we argue that it should not run in user space due to security issues related to the user access permission in Linux environment [223][224].

However, we cannot use the same approach for our second method because it involves many different modules' drivers and unfortunately it is very difficult to compact them into one block of driver to simplify the loading and accessing. We therefore prefer to use the full embedded version as a loadable module. We have prepared the system so that the user who wants to use our second method can conveniently load the drivers using the appropriate Linux command such as **modprobe()** for the driver loading and **rmmod()** for unloading it. Unfortunately, the Linux driver for our embedded factor graph is not perfect yet and might contain some bugs that need to be solved by thoroughly examining the entire source code of the driver.

Finally, we also agree with the conclusion made by Juan Carlos et.al in [225] stating that in general FPGA-based design for real robotics application is difficult due to two main reasons: difficulty in changing platform's functionality (very often requires specialized person) and tools system dependency. However, we also believe that our embedded factor graph framework on SoC has a prospective future, because we see an increasing trend of bringing the FPGA design into the heterogeneous computing platform (such as SystemC, OpenCL, etc.) and also an increasing effort to bring the embedded Linux kernel into the mainstream. This in turn will make the future optimization and further development for broader application easier. Furthermore, the price per chip for FPGA technology also shows decreasing tendency, which makes the FPGA solution a very good choice in the future.

6.2 Another Possible Platform for Embedded Factor Graphs

In this thesis, we explore two different dedicated hardware platforms for implementing factor graphs. There is another interesting option to be explored in our future work for improving our second embedded factor graph implementation strategy (i.e. fully embedded factor graph on a chip). Adapteva Inc. produces a scalable multi-core chip called Epiphany which has 16 or 64 processors inside the chip that share a common 32 bit memory space [226] (see Fig. 6.1a for its internal architecture). The core units inside the Epiphany

chip, aside from those 32-bit floating point RISC processors, are the symmetric routers that are distributed throughout the chip and connect the processors in a 2D grid. This router topology is simpler than the SpiNNaker router, and according to Adapteva, it can be used to link up 4095 processors by interlinking many Epiphany chips (see the example configuration in Fig. 6.1b). Unfortunately, the Epiphany chip was not ready for full production when we started this research.

Adapteva has developed a prototype board which consists of an Epiphany chip combined with a SoC also from the Zyng-7000 family (see Fig. 6.1c). In this thesis, we have implemented a factor graph framework in a TE0720 board (which also has a SoC from the Zynq-7000 family on the board) and we should have no problem to port our factor graph framework on the Adapteva board. We can then enhance the factor graph module by utilizing the Epiphany chip for gaining more fine-grained parallelism. This might be an interesting option for embedding factor graphs on hardware.

6.3 Beyond Limited Hardware Implementations

From chapter 2 to chapter 5, we have demonstrated the "evolution" of our factor graph framework from the standard PC-based implementation to the full embedded version. The initial goal of this research, which motivated us to develop the embedded factor graph, is to create building blocks of a modular machine that can be embedded in a larger system to produce more powerful, flexible and efficient intelligence machine with cognitive capabilities. This is not an ambiguous goal if we look back to the first era of AI when people started to dream of having such a machine and we regard this as a progressive vision. In this thesis, we demonstrate that such a machine can be built using a probabilistic graphical model framework on top of a reconfigurable dedicated hardware.

We use the population coding principle in our discrete factor graph, a topic that originates from computational neuroscience. In line with this theme, we also learn from the field of neuroscience that there is an effort to use graph theory to describe the fundamental aspect of the information processing of the brain in terms of interconnection between groups (population) of neurons. Going back to our motivation in this thesis of having reciprocal benefits between engineering and neuroscience, here we envision our embedded factor graph to be extended and implemented in a more futuristic and biologically plausible application such as brain graph for human connectome.

In his paper, Bullmore describes that brain graphs provide a simple way with high degrees of generalizability and interpretability to model the human brain connectome, using graph theory to abstractly define a nervous system as a set of nodes and interconnecting edges [140]. In such a model, nodes represent collections of similar neurons (in terms of their intrinsic characteristic), while edges represent structural - and hence functional - connections between those nodes. A connectome is a complete point-to-point spatial connectivity of neural pathways in the brain which represents a comprehensive map of neural connections in the brain [227]. This "wiring diagram" of the brain can be used for building a computational model of whole-brain dynamics. Computational neuroscientists build these brain graph models using topological and geometrical approaches that demonstrate both structural and functional organization of the human brain [140][228]. They strive

to find the best model that consistently demonstrates key topological properties such as small-worldness, modularity, and heterogeneous degree distributions. Today, many neuroscientists use brain graphs as a way of modeling the human brain connectome, by abstractly defining the nervous system as a set of computational nodes and interconnecting edges.

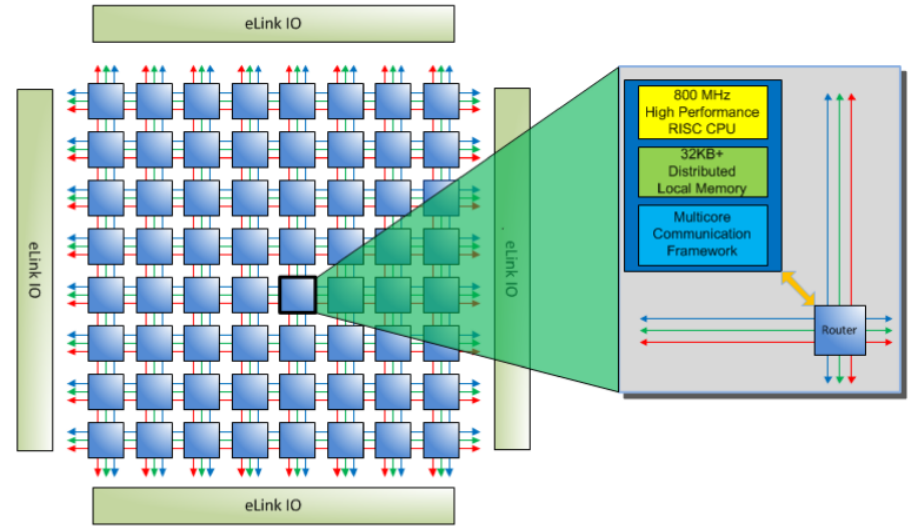
While the brain graph related project focuses on developing the "perfect" model without concerning too much the real impact on the intelligence emerged from the model, many scientists argue that such models should also be consistent with the embodiment theory in cognitive neuroscience. At least, such models should be able to be used for describing the predictive capability of the human brain. This transformation from just drawing the connectome's wire diagram to a predictive modelling has been made possible by the extensive use of machine learning [229]. Some literature have demonstrated the applicability of the brain graphs to cognitive neuroscience using machine learning approaches [230][231][232]. We believe that this is the starting point where we can transform the brain graph into a probabilistic graphical model. Eventually, this will turn into inverse inference problem which can be used to synthesize intelligent behavior in a fashion similar to neural computation in the brain.

We have demonstrated in chapter 3 that our factor graph framework is well suited for applications in the domain of machine learning and robotics. We argue that we can make a significant contribution in the future, where machine learning ideas meet technical systems. Rather than exploring the underlying neural substrate, here we propose to use the inference mechanism on factor graphs to give better understanding on the relation between brain graph's topological features and emerged cognitive intelligence. It is believed that this cognitive intelligence can naturally be explained if human cognitive representations are understood to be structured like graphical models [233]. As we have described in chapter 1, this structured representation underlies the mechanism of the perception-action cycle of human beings which eventually becomes the motivation to bring "life" into robotics systems. Transforming brain graphs into factor graphs can help the machine learning community to understand the technical constraints which make those brain graph models more applicable in technical systems.

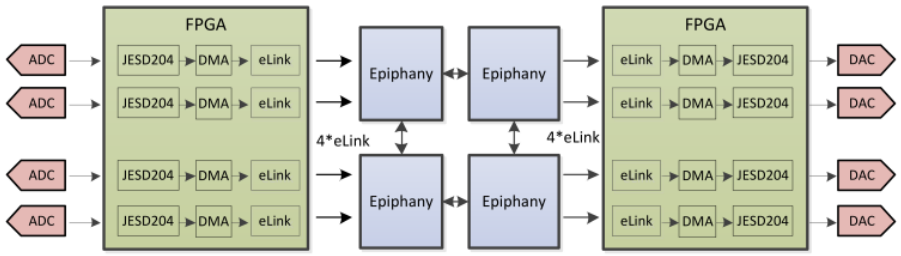
In line with this idea, we are also interested to make a link between brain graph theory and robotics application through factor graphs. One visionary framework that is currently being conceptualized is the work of Saxena et.al. known as "Robo Brain" [234]. In principle, their goal has already been conceived by many AI revolutionists decades ago but their idea to bring the cloud computing down to the everyday robotic systems has attracted a lot of attention not only from the media, but also robotic scientists and engineers. In addition to their idea of developing a large-scale computational system that learns from publicly available Internet resources, computer simulations, and real-life robot trials, they are also looking into the foundation of graphical models that can be used to create a comprehensive knowledge-based system. However, their high-level abstraction framework needs to be interlinked with the real robotics platform. In this circumstance, we see the big potential of our factor graph to be embedded into such a framework. From chapter 3 to 5, we have demonstrated that our probabilistic graphical models are also capable of handling such tasks in a low hardware level.

In summary, we saw promising potentials for our proposed factor graph framework in

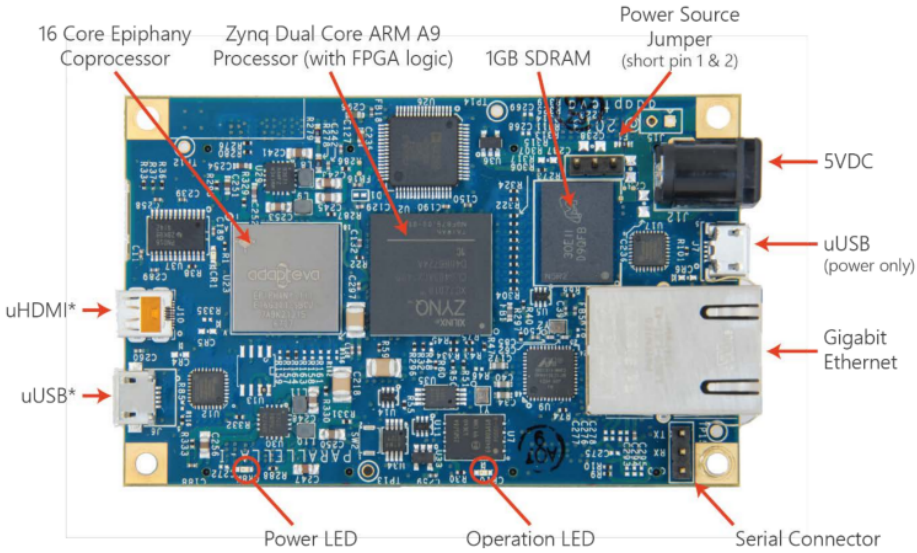
the future. Starting by exploring the basic principle of factor graph's applications, we have demonstrated that we can extend our framework to solve more challenging tasks. We believe that our factor graph framework will find its place in broader applications in the future since we have laid the core of a progressive framework. Fig. 6.2 shows our interest of working beyond the limited hardware implementation.



(a)

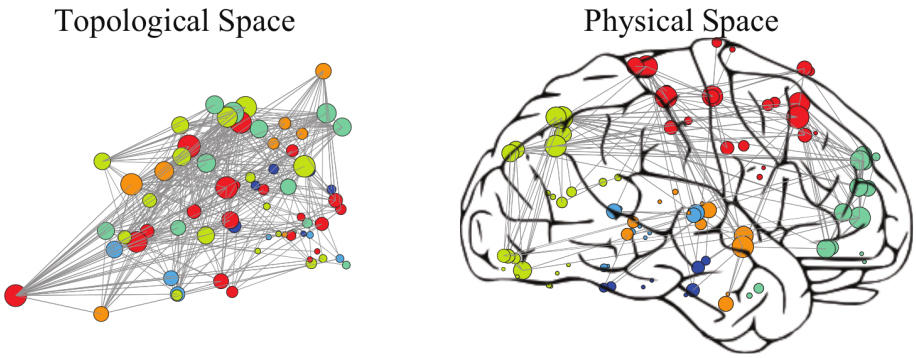


(b)

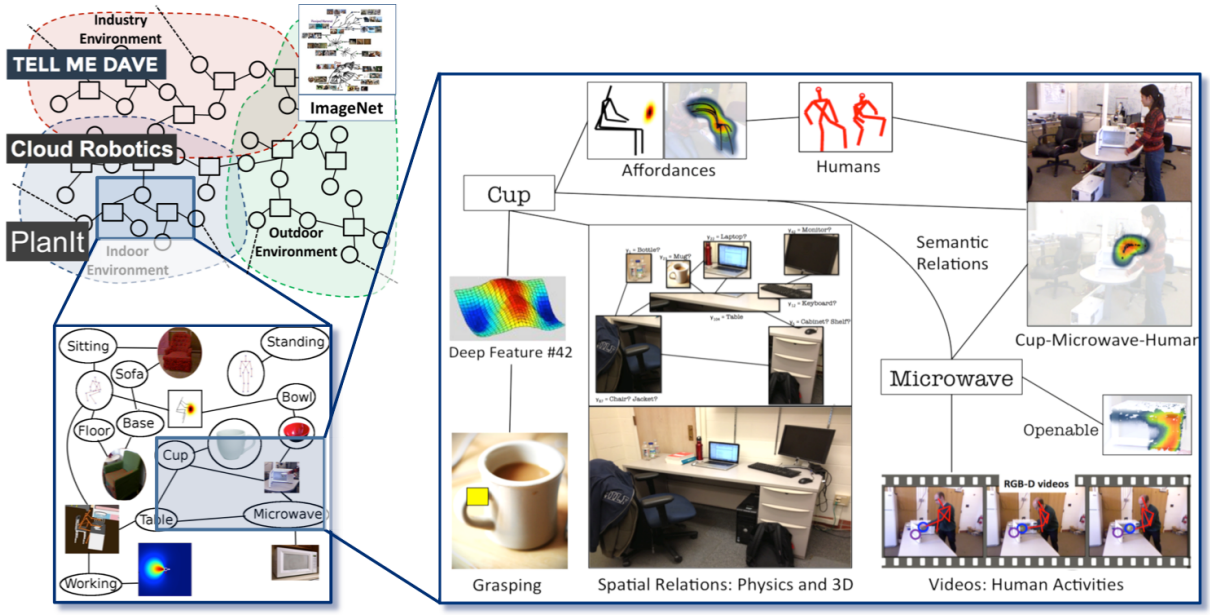


(c)

Fig. 6.1: The Parallela board and its Epiphany chip. (a) The internal architecture of an Epiphany chip. It is mainly composed of 16 or 64 RISC processors interconnected by symmetric router units. (b) An example of using Epiphany chips for building a massive embedded system in which up to 4095 processors can be linked. (c) The current Parallela board which uses an Epiphany 16-core and a Zynq Z-7020. Except for the presence of the Epiphany chip, this Parallela board shares many similarities to the board TE0720 that we use in this thesis. Figures (a) and (b) are adapted from [226] while (c) can be found at adapteva website http://www.adapteva.com/



(a) Topological and geometrical properties of a functional brain graph. Left: In the topological space, the distance between nodes is larger if those nodes are separated by a longer path length and smaller if they are separated by a shorter path length. The size of nodes indicates nodal degree whereas the color of nodes indicates its lobar identity. Right: Brain graph plotted in the physical space where the distance between nodes is the Euclidean distance between regional centroids in the anatomical space of the real brain. The size of nodes indicates nodal degree whereas the color of nodes indicates its lobar identity.



(b) Robo Brain: Large-Scale Knowledge Engine for Robots. In this conceptual idea, the authors intend to utilize an internet-scale system to generate an intelligence system which integrates multiple data modalities including symbols, natural language, haptic senses, robot trajectories, visual features, etc. However, in our vision, such a complex machine can be broken down into many modular parts where each part can be constructed from our embedded factor graph.

Fig. 6.2: Brain graph related experiments and robotic applications that motivate us envisioning future applications of our embedded factor graph. The top figure is adapted from [140]. The bottom figure is adapted from [235] although the main idea originally developed by Robo Brain team [234].

7 Summary

This thesis is about an investigation on a probabilistic graphical model known as a factor graph. We explore the applicability of a factor graph especially in the domain of artificial intelligence for robotics as well as the strategy to implement it efficiently in dedicated hardware. The motivation of exploring this factor graph along with its hardware implementation is because we belief that a factor graph can be used to mimic the information processing mechanism in the brain. In general, the brain processes information in a way such that its functional units exchange information between them to produce the overall performance while maintaining the consistency of brain states throughout the process. We imitate this mechanism in the belief propagation fashion which can be implemented efficiently in a factor graph. The complexity of the complete brain operation is undoubtedly beyond the grasp of our current technology, but we can approximate a small part of it and imitate its behaviour in a computer that can be used to reinforce non-biological technical systems. Once we understand the main principle of that part, we might be able to construct a larger system by combining it with the other parts. This thesis strives to find an optimal way to abstract principles from the brain-style information processing mechanism and implements it as adaptive as possible using various platforms.

We began by exploring the core principle of belief propagation in a factor graph. In chapter 2, we describe the mechanism of constructing the model as well as the identification of the network's parameters via learning procedures using the inference process. We started by implementing the standard maximum likelihood estimation (MLE) in a message-passing scenario and then extended the principle to the complex estimation scenario using the expectation maximization (EM) paradigm. We show that our program works well in accommodating the challenge of informational missing in the data. We also describe our method for working with discrete factor graphs by using a population code. The population coding is a principle that originates from neuroscience in which we can encode and decode a real-valued number into/from a set of probabilistic values. With this approach we demonstrate that we can have a compact representation of a "message" as well as handling the uncertainty that commonly arises in a non-ideal system. We also demonstrate that we can use another concept from neuroscience, which is the self-organizing-map (SOM), to enhance the population codes so that they can discover and represent the characteristic of the data adaptively. All of these aspects are implemented in a PC-based program.

In chapter 3, we give several examples ranging from the simple but standard task in the machine learning domain to the more complex and challenging task in robotics application. We show that our PC-based factor graph framework can be used to solve problems in regression and classification tasks. We then demonstrate that we can extend the static model into a dynamic one using the mechanism similarly to the dynamic Bayesian network, i.e. by unrolling the model several times so that the dynamic behaviour of the system can be captured by the model. We complete the chapter by giving examples in the robotics

domain. We focus on the kinematics model of two different robot systems: a mobile robot and a manipulator. We show that our PC-based factor graph works well with these two robot systems and exemplify the usage of the dynamic factor graph. We give the final example of a complex robotics system in the context of model-based learning for a mobile manipulator. In this example, a hybrid robot that is built by combining two robots that differ in the model and the control is presented. Although not all aspects of this challenging robot system are covered, we show in principle that our factor graph framework is very well suitable for solving problem in the state-of-the-art of robotics domain: imitation learning. We close chapter 3 with confidence that our PC-based factor graph framework produces very good results and can be implemented in dedicated hardware with similar quality but with higher performance in terms of power efficiency and higher degree of flexibility. This is the crucial aspect which plays an important role for a real technical system.

We started exploring the hardware implementation of our factor graph using a SpiNNaker system. In chapter 4 we describe that originally the SpiNNaker system is intended to be used for a specific spike-based neural network application, but we show that we can exploit the SpiNNaker's resources so that it can be used for implementing the belief propagation algorithm for factor graphs. To use the SpiNNaker system efficiently for factor graphs, we use two different strategies for mapping the factor graph components into the SpiNNaker's resources. The first is related with the mapping of the population code algorithm, and the second is related with the mapping of factor graph nodes. In both strategies, we show that the most important aspect of our embedded factor graph lies in the management of the routing protocol of the SpiNNaker system. The belief propagation then relies on how the MC packets are exploited to carry the messages. To exemplify the applicability of our factor graph on the SpiNNaker system as well as the proof of concept, we present a network for robotics application which is proven to be well executable with our PC-based factor graph. The result shows that even though the factor graph on the SpiNNaker system runs slower than its PC-based version, it actually performs marvellously when we regard the characteristics of the SpiNNaker hardware. First, each core runs at the clock speed 200 Mhz, which is far below our PC which runs at the clock speed 3300 MHz. Second, it consumes extremely low power with only about 4 Watt on the Spinn3 board, compared to 700 Watt power consumption of our PC. These are very interesting results which motivate more applications using our SpiNNaker-based factor graph. To close our exploration on the SpiNNaker system, we propose to improve the performance of our SpiNNaker-based factor graph in the future by using the Forney-style factor graph (FFG), which simplifies the routing mechanism of the belief propagation on the factor graph. With the insights from this successful implementation, we continued our exploration on the embedded factor graph using the second hardware: a SoC (System-on-Chip).

In this final hardware exploration, we use a SoC from the family of Xilinx Zynq-7000, especially the Z-7020. This SoC contains two independent parts: an ARM microprocessor and an FPGA. We started exploring this device under the common assumption that the FPGA in the SoC is very useful to speed up processes which run on the microprocessor part of the SoC. In this paradigm, we use the FPGA only as the accelerator for the factor graph framework which runs on the ARM processor. As the accelerator, the FPGA is responsible for transforming the sequential nature of the sum-product algorithm into

the parallel fashion. The result shows that the accelerator can speed up the computation almost up to eight times compared to the plain run of the factor graph without acceleration. This is a quite impressive result, although we still see some limitations on this approach. Therefore, we propose the second method in which we maximally use the resources of the FPGA to produce a fully embedded factor graph. Using this second approach, the entire factor graph can run on the FPGA part of the SoC. To measure the effectiveness of our approach, we use two metrics: clock latency and resource consumption. From the implementation of this second approach, we gain some insights about the nature of the trade-off between speed-and-area optimization for our factor graph. These will be useful to achieve much more efficiency when we reimplement our SoC-based factor graph in the future, probably using a denser and richer resources SoC.

With these results, we are confident that we have already built an important fundamental framework for a powerful embedded factor graph that opens many possibilities for further exploration and applications. Finally, we envision the future applications of our embedded factor graph in the domain of cognitive intelligence, especially in the direction of a massively distributed computing engine. Our vision on this aspect has been described thoroughly in chapter 6. In summary, the embedded factor graphs, both using the SpiNNaker system and a SoC, are the main contribution of this thesis.

A Appendix-A

A.1 Beyond the Standard Factor Graph

In this thesis, we use standard factor graph notations which are originally introduced by Kschischang et.al in [31]. However, there exist other notations introduced by Forney in [236] which are popularly called Forney-style factor graphs (FFGs) and are used commonly within the communication and signal processing community [237]. Forney introduced notations that different from standard factor graph notations in order to exploit many duality phenomena found in coding theory and signal processing (such as in the Fourier transform and its duality). As a result, his FFG is more compatible with standard block diagrams and is more suited for hierarchical modeling.

In an FFG, the presence of two different node types in ordinary factor graphs is simplified by considering the factor nodes as the only active nodes in the network (i.e. all nodes correspond to factor nodes, whereas variables are represented by edges connecting two nodes). As a consequence, each variable may only be connected to at most two factor nodes. This may seem as a strong restriction in representational power compared to ordinary factor graphs, however, this restriction can be overcome by introducing a specific type of factor nodes called "equality constraint factor" $f_{=}(x_a) := \prod_{i=1}^{n} \delta(x_{aj} - x_{ai})$ where $f_{=}(x_a) = 1$ if and only if $x_{aj} = x_{ai}$. With this constraint, if all variables x_{ai} (i.e. $f_{=}(x_a) = 1$) take on the same value then the factorization of $p(\mathbf{X})$ in equation (2.3) remains unaffected. However, for all other combinations of variable values (i.e. $f_{=}(x_a) = 0$), $p(\mathbf{X}) = 0$. This way, the factor $f_{=}(x_a)$ prevents any variable from assuming different values than the others and thus the variable is 'cloned' in all directions.

An FFG can be obtained from an ordinary factor graph by replacing variable nodes through edges if these variable nodes are connected to at most two factor nodes, or by equality constraint factors $f_{=}(x_a)$ if these variable nodes are connected to more than two factor nodes. For an example, the ordinary factor graph in Fig. 4.5 can be converted into an FFG as shown in Fig. A.1b.

In some cases, an FFG is simpler than an ordinary factor graph with respect to the sum-product message update rule. This is true especially when we work with Gaussian factor graphs. Many factor graph models are built using Gaussian distributions to exploit the nice property of the Gaussian function: if the operands are Gaussian then the result is also Gaussian. It also means, if the input messages to a linear function node are Gaussian, then the output message from that node is also Gaussian. Therefore, the whole belief propagation process can be described by means of the parameters of the messages, e.g. mean and covariance matrix. Sascha Korl derived several other special factor nodes for this Gaussian factor graph [63].

Back to Fig. A.1, let's assume that C is the measurement result of D using noisy sensor X_C and E is the measurement result of D using noisy sensor X_E. Also by assuming that

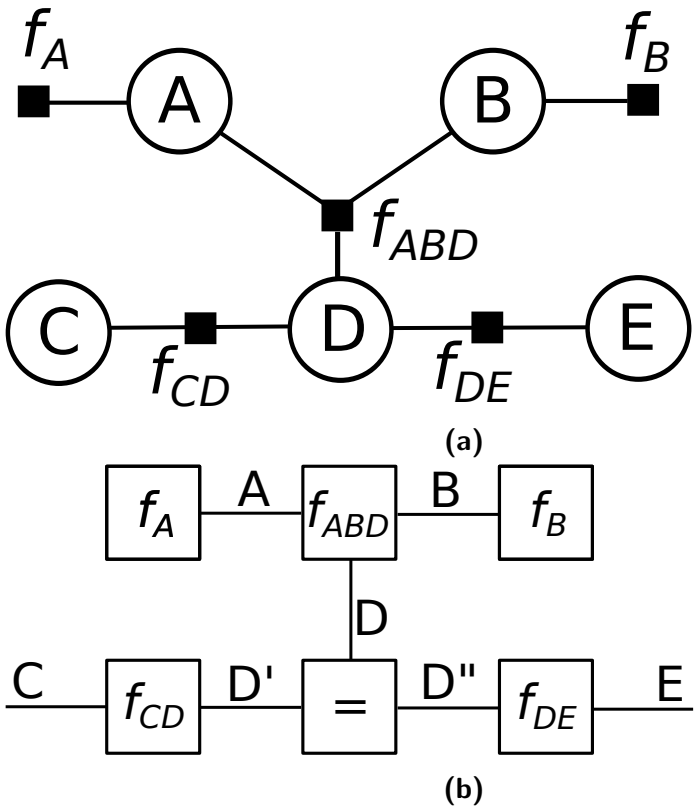


Fig. A.1: An example of converting an ordinary factor graph to an FFG. (a) An ordinary factor graph similar to the network shown in Fig. 4.5. (b) The resulting FFG from (a) with the equality constraint function $f_{=}(D, D', D'') \triangleq \delta(x - x')\delta(x - x'')$. The variable D is 'cloned' since it is connected to three factor nodes.

the noise induced by both sensors are Gaussian, we can represent the factor graph shown in Fig. A.1b into a more detailed version:

The addition symbol '+' in Fig. A.2 represents an addition operation. The functional operation of this symbol in the Gaussian factor graph is represented in Fig. A.3 where m_X is the mean of X, V_X is the variance of X, W_X is the weighted matrix for X in which $V_X = W_X^{-1}$, and ξ_X is the weighted mean of X. More of additional symbols can be found in [63].

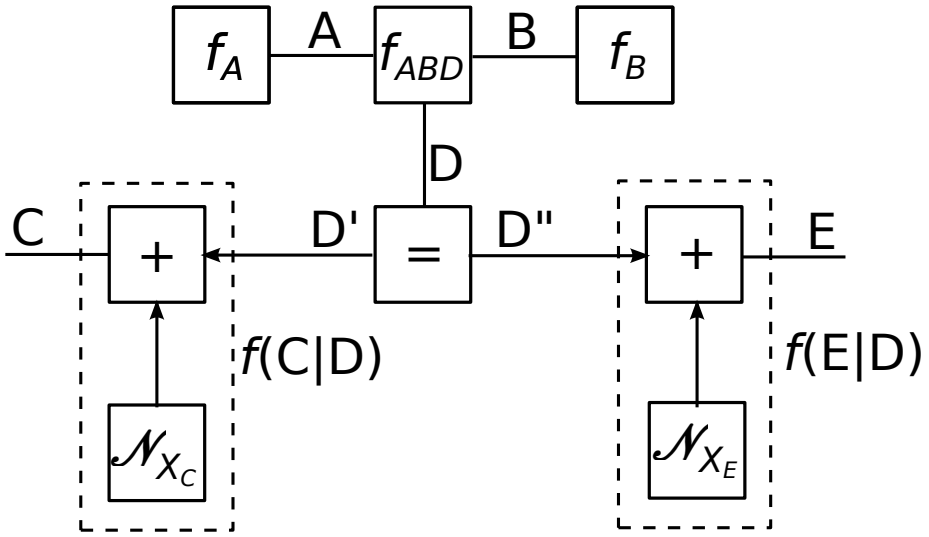


Fig. A.2: Detailed version of Fig. A.1b where we assume that the factor f_{CD} represents the conditional distribution of measurement C for hidden variable D where the sensor has Gaussian noise characteristics. The symbol '$+$' in this block represents the function $f_+(C, \mathcal{N}_{X_C}, D') = \delta(D' + \mathcal{N}_{X_C} - C)$. Likewise, the factor f_{DE} represents the conditional distribution of measurement E for hidden variable D where the sensor has Gaussian noise characteristics. The symbol '$+$' in this factor represents the function $f_+(E, \mathcal{N}_{X_E}, D'') = \delta(D'' + \mathcal{N}_{X_E} - E)$.

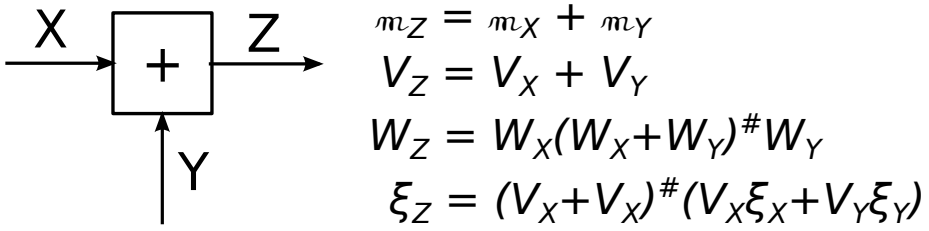


Fig. A.3: An addition symbol represents an addition operation which is very common being used in Gaussian FFGs. The # represents the Moore-Penrose pseudoinverse of the corresponding matrix.

B Appendix-B

B.1 Discrete Factor Graph with Population Coding

In this thesis, we have developed a PC-based factor graph framework which uses population codes for discretizing message's values. The motivation for developing this framework is not to produce another new one that competes with so many factor graph frameworks already exist online, rather to give us insight on how an efficient factor graph can be developed and implemented in dedicated hardware. Nevertheless, we open our PC-based factor graph framework for anyone who wants to learn the basic principles of discrete factor graphs. Our framework was developed in C++ and it requires a special library called Qt, at least with version 4.8. To use our framework, one can recompile it on his own machine or use our pre-compiled library, assuming that it is targeted to a 32-bit Linux machine. In this appendix, we give a simple example of how to use our factor graph framework.

Assume that we want to run an inference on a factor graph with three variables as shown in Fig. B.1.

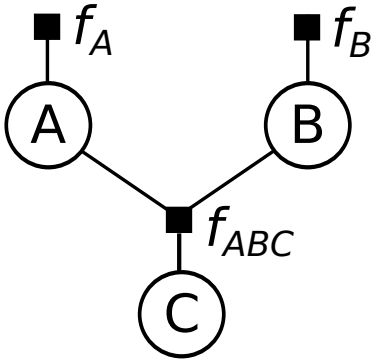


Fig. B.1: Example three variables factor graph.

The inference task is to compute the value of the variable C given the inputs on the variable A and B. The following is the code for implementing such task.

```
/************************* The header file ****************************/
/* It contains the class description for the network                  */
/* In this example we assume that OpenMP is going to be used          */
/**********************************************************************/
#ifndef C3VARMAPPING_H
#define C3VARMAPPING_H

#include <QObject>
#include <QVector>
#include <stdio.h>
#include <factorgraph.h>   //invoke the factor graph library

#define THREAD_NUM  4

struct parameter
{
    QString _pdf;
    QString _nS;
    QString _vR;
    QString _eR;
    QString _gV;
    bool _verbose;
    QString _dataset;
    QString _testset;
    QString _result;
};

class c3VarMapping : public QObject
{
    Q_OBJECT
public:
    explicit       c3VarMapping(parameter p, QObject *parent = 0);
    void           run();
private:
    /* Factor Graph nodes */
    CNode          A, B, C, fA, fB, fC, fABC;

    /* Messages in training flow */
    CMessage       ufA_to_A, ufB_to_B, ufC_to_C;
    CMessage       uA_to_fABC, uB_to_fABC, uC_to_fABC;
    /* Messages during inference flow */
    CMessage       ufABC_to_C;

    /* Other class members */
    QString        JPDfname;    /* for reading/storing factor in a file */
    parameter      params;
    void init      ();
    bool trainJPD  ();
    bool infer     ();
    void info      (char *txt){
                          if(params._verbose) printf("%s", txt);
                          fflush(stdout);}
};

/* Helper functions for visualization only */
int getPercentage(int i, int total);
int getNumLines(char * fname);

#endif // C3VARMAPPING_H
```

```
/************************* The C++ source file ************************/
/* It contains the implemented class for the network                  */
/**********************************************************************/

#include "c3varmapping.h"
#include <QFile>
#include <cmath>
#include <QDebug>
#include <QCoreApplication>
#include <omp.h>

c3VarMapping::c3VarMapping(parameter p, QObject *parent) :
    QObject(parent)
{
    params = p;
    init();
}

void c3VarMapping::init()
{
    /*********************** create factor graphs ************************/

    /* First, create nodes */
    A.init(FG::Variable, 1, p._nS.toInt());
    fA.init(FG::Factor, -1, p._nS.toInt());

    B.init(FG::Variable, 2, p._nS.toInt());
    fB.init(FG::Factor, -2, p._nS.toInt());

    C.init(FG::Variable, 3, p._nS.toInt());
    fC.init(FG::Factor, -3, p._nS.toInt());

    fABC.init(FG::Factor, -5);

    /* Second, assign neighborhood */
    fA.addNeighbor(A.getID()); A.addNeighbor(fA.getID());
    A.addNeighbor(fABC.getID());

    fB.addNeighbor(B.getID()); B.addNeighbor(fB.getID());
    B.addNeighbor(fABC.getID());

    fC.addNeighbor(C.getID()); C.addNeighbor(fC.getID());
    C.addNeighbor(fABC.getID());

    fABC.addNeighbor(A.getID()); fABC.addNeighbor(B.getID());
    fABC.addNeighbor(C.getID());

    /* Third, initialize factor nodes */
    QVector<int> s,c; //for scope and cardinal
    s.clear(); c.clear(); s << A.getID(); c << A.getCard(); fA.initFactor(s, c);
    s.clear(); c.clear(); s << B.getID(); c << B.getCard(); fB.initFactor(s, c);
    s.clear(); c.clear(); s << C.getID(); c << C.getCard(); fC.initFactor(s, c);
    s.clear(); c.clear();
    s << A.getID() << B.getID() << C.getID();
    c << A.getCard() << B.getCard() << C.getCard();
    fABC.initFactor(s, c);

    /* Fourth, setup the messages for training */
    ufA_to_A.setID(fA.getID()); ufA_to_A.setName("Message from fA to A");
```

```
    ufA_to_A.setScope(A.getID()); ufA_to_A.setCard(A.getCard());
    ufA_to_A.setSourceID(fA.getID());

    ufB_to_B.setID(fB.getID()); ufB_to_B.setName("Message from fB to B");
    ufB_to_B.setScope(B.getID()); ufB_to_B.setCard(B.getCard());
    ufB_to_B.setSourceID(fB.getID());

    ufC_to_C.setID(fC.getID()); ufC_to_C.setName("Message from fC to C");
    ufC_to_C.setScope(C.getID()); ufC_to_C.setCard(C.getCard());
    ufC_to_C.setSourceID(fC.getID());

    uA_to_fABC.setID(A.getID()); uA_to_fABC.setName("Message from A to fABC");
    uA_to_fABC.setScope(A.getID()); uA_to_fABC.setCard(A.getCard());
    uA_to_fABC.setSourceID(A.getID());

    uB_to_fABC.setID(B.getID()); uB_to_fABC.setName("Message from B to fABC");
    uB_to_fABC.setScope(B.getID()); uB_to_fABC.setCard(B.getCard());
    uB_to_fABC.setSourceID(B.getID());

    uC_to_fABC.setID(C.getID()); uC_to_fABC.setName("Message from C to fABC");
    uC_to_fABC.setScope(C.getID()); uC_to_fABC.setCard(C.getCard());
    uC_to_fABC.setSourceID(C.getID());

    /* Finally, setup the messages used in the inference flow */
    ufABC_to_C.setID(fABC.getID());
    ufABC_to_C.setName("Message from fABC to C");
    ufABC_to_C.setScope(C.getID()); ufABC_to_C.setCard(C.getCard());
    ufABC_to_C.setSourceID(fABC.getID());
}

bool c3VarMapping::trainJPD()
{
    /* First, determine the type of probability distribution.
       By default, we assign Gaussian. The other possible values are single,
       binary, beta, and triangle. */
    FG::pdfType type;
    if(params._pdf==”single”) _type = FG::Single;
        else if(params._pdf==”binary”) _type = FG::Binary;
        else _type = FG::Gaussian;

    /* Determine the file for storing the resulted factor in the training */
    JPDfname = params._result;
    if(_type==FG::Gaussian) JPDfname.append(".gjpd");
        else if (_type==FG::Single) JPDfname.append(“.sjpd”);
        else JPDfname.append(".bjpd");

    QFile raw(params._dataset), jpdABC(JPDfname);
    if(!raw.open(QIODevice::ReadOnly | QIODevice::Text)) {
        info("Cannot open the dataset!");
        return false;
    }

    /* 2nd step: build the JPD */
    CFactor jpd[THREAD_NUM], F = fABC.getFactor();
    QVector<double> tmpjpd; //temporary jpd, contains all jpd during iteration

    /* Since we will be using OpenMP, then we decompose the jpd into several
       instances. The following is just to set jpd[i] to NULL */
    F = fABC.getFactor();
```

```
    QVector<double> nulljpd(getElementProd(F.getCard()),0.0);
    for(int t=0; t<THREAD_NUM; t++){
        jpd[t].setScope(F.getScope());
        jpd[t].setCard(F.getCard());
        jpd[t].setJPD(nulljpd);
    }

    /* read the training dataset */
    info("Building JPD...\n");
    qint64 nLines = getNumLines(params._dataset.toLocal8Bit().data());
    int i = 0, ci, oi = -1, j;
    QVector < QVector<int> > ds;
    QString line;
    QStringList strList;

    ds.resize(nLines);
    for(j=0; j<nLines; j++) ds[j].resize(3);
    j=0;
    while(!raw.atEnd()){
        ci = getPercentage(i, nLines); i++;
        if(ci!=oi){
            oi = ci;
            info(QString("\rReading dataset%1\%").arg(ci).toLocal8Bit().data());
        }
        line = raw.readLine();
        strList = line.split(',');
        if(strList.size()==3){
            for(int k=0; k<3; k++) ds[j][k] = strList.at(k).toInt();
            j++;
        }
    }
    raw.close();

    oi = -1; ci = 0; int cntr = 0; nLines = ds.size();
    //QVector<int> jpdScope = jpd.getScope();

    omp_set_num_threads(THREAD_NUM);

#pragma omp parallel for
        for(i=0; i<ds.size(); i++){
            CPMF pmf(_type, params._nS.toInt(), params._eR.toInt(),
                     params._gV.toInt());
            ci = getPercentage(cntr, nLines);
#pragma omp atomic
            cntr++;
            info(QString("Processing %1\%: line-%2 out of %3 by thread-%4\n")
                .arg(ci).arg(i).arg(nLines).arg(omp_get_thread_num())
                .toLocal8Bit().data());

            bool success;
            CFactor product;
            QVector<double> tmpjpd; //temporary jpd, local within the for-loop

            /* process fA and A */
            pmf.setVarValue(ds.at(i).at(0));
            fA.setFactor(pmf.getVarStates());
            ufA_to_A.updateFactor(fA.computeMessage());
            A.putMessage(ufA_to_A);
            uA_to_fABC.updateFactor(A.computeMessage(fABC.getID(), &success));
```

```
        if(!success)
            qFatal("Something wrong with message from A node to fABC!");

        /* process fB and B */
        pmf.setVarValue(ds.at(i).at(1));
        fB.setFactor(pmf.getVarStates());
        ufB_to_B.updateFactor(fB.computeMessage());
        B.putMessage(ufB_to_B);
        uB_to_fABC.updateFactor(B.computeMessage(fABC.getID(), &success));
        if(!success)
            qFatal("Something wrong with message from B node to fABC!");

        /* process fC and C */
        pmf.setVarValue(ds.at(i).at(2));
        fC.setFactor(pmf.getVarStates());
        ufC_to_C.updateFactor(fC.computeMessage());
        C.putMessage(ufC_to_C);
        uC_to_fABC.updateFactor(C.computeMessage(fABC.getID(), &success));
        if(!success)
            qFatal("Something wrong with message from C node to fABC!");

        /* Now proceed with MLE */
        /* IMPORTANT: The Order DOES Matter! */
        product = uA_to_fABC;
        //at this point, variable product only contains A

        product.prod(uB_to_fABC, false);
        //now product will contains A and B

        product.prod(uC_to_fABC, false);
        //finally, product should contains A, B and C

        tmpjpd = addVectorElements(jpd[omp_get_thread_num()]
                    .getJPD(), product.getJPD());
        jpd[omp_get_thread_num()].setJPD(tmpjpd);
    }

/* now combine the jpd */
CFactor jpdAll;
jpdAll.setScope(F.getScope());
jpdAll.setCard(F.getCard());
tmpjpd = jpd[0].getJPD();
for(int j=1; j<THREAD_NUM; j++)
    tmpjpd = addVectorElements(jpd[j].getJPD(), tmpjpd);
jpdAll.setJPD(tmpjpd);

/* 3rd step: normalize jpd */
info("Normalize JPD and writing to a file");
tmpjpd = normalizeStates(jpdAll.getJPD());
jpdAll.setJPD(tmpjpd);

/* and then write to a file */
if(!jpdABC.open(QIODevice::WriteOnly | QIODevice::Text)) return false;
QTextStream jpdOut(&jpdABC);
for(int i=0; i<tmpjpd.count(); i++)
    jpdOut << QString("%1\n").arg(tmpjpd.at(i));
jpdABC.close();
fABC.setFactor(tmpjpd);
return true;
```

```
}


bool c3VarMapping::infer()
{
    bool result = true;
    FG::pdfType type;
    if(params._pdf==”single”) _type = FG::Single;
        else if(params._pdf==”binary”) _type = FG::Binary;
        else _type = FG::Gaussian;

    QString fResName = JPDfname;
    if(_type==FG::Gaussian) fResName.append(".gres");
        else if(_type==FG::Single) fResName.append(“.sres”);
        else fResName.append(".bres");
    QFile fRes(fResName);
    QFile fTest(params._testset);
    QTextStream fOut(&fRes);
    QString line;

    info("Performing inference...\n");

    if(!fRes.open(QIODevice::WriteOnly | QIODevice::Text)) {
        info("Cannot create a file for storing the inference result!");
        return false;
    }

    if(!fTest.open(QIODevice::ReadOnly | QIODevice::Text)) {
        info("Cannot open the validation test file!");
        fRes.close(); return false;
    }

    CPMF pmf(_type, params._nS.toInt(), params._vR.toInt(), params._eR.toInt(),
         params._gV.toInt());
    bool success;
    int res, i = 0, ci, oi = -1;
    qint64 nLines = getNumLines(params._testset.toLocal8Bit().data());
    QStringList l;
    while(!fTest.atEnd()){
        ci = getPercentage(i, nLines); i++;
        if(ci!=oi){
            oi = ci;
            info(QString("\rReading testset %1\%").arg(ci)
                .toLocal8Bit().data());
        }
        line = fTest.readLine();
        l = line.split(',');
        if(l.size()==3) {

            /* give A to fABC */
            pmf.setVarValue(l.at(0).toInt());
            fA.setFactor(pmf.getVarStates());
            ufA_to_A.updateFactor(fA.computeMessage());
            A.putMessage(ufA_to_A);
            uA_to_fABC.updateFactor(A.computeMessage(fABC.getID(), &success));
            if(!success){
                info("Something wrong with message from A to fABC!");
                result = false; break;
            }
```

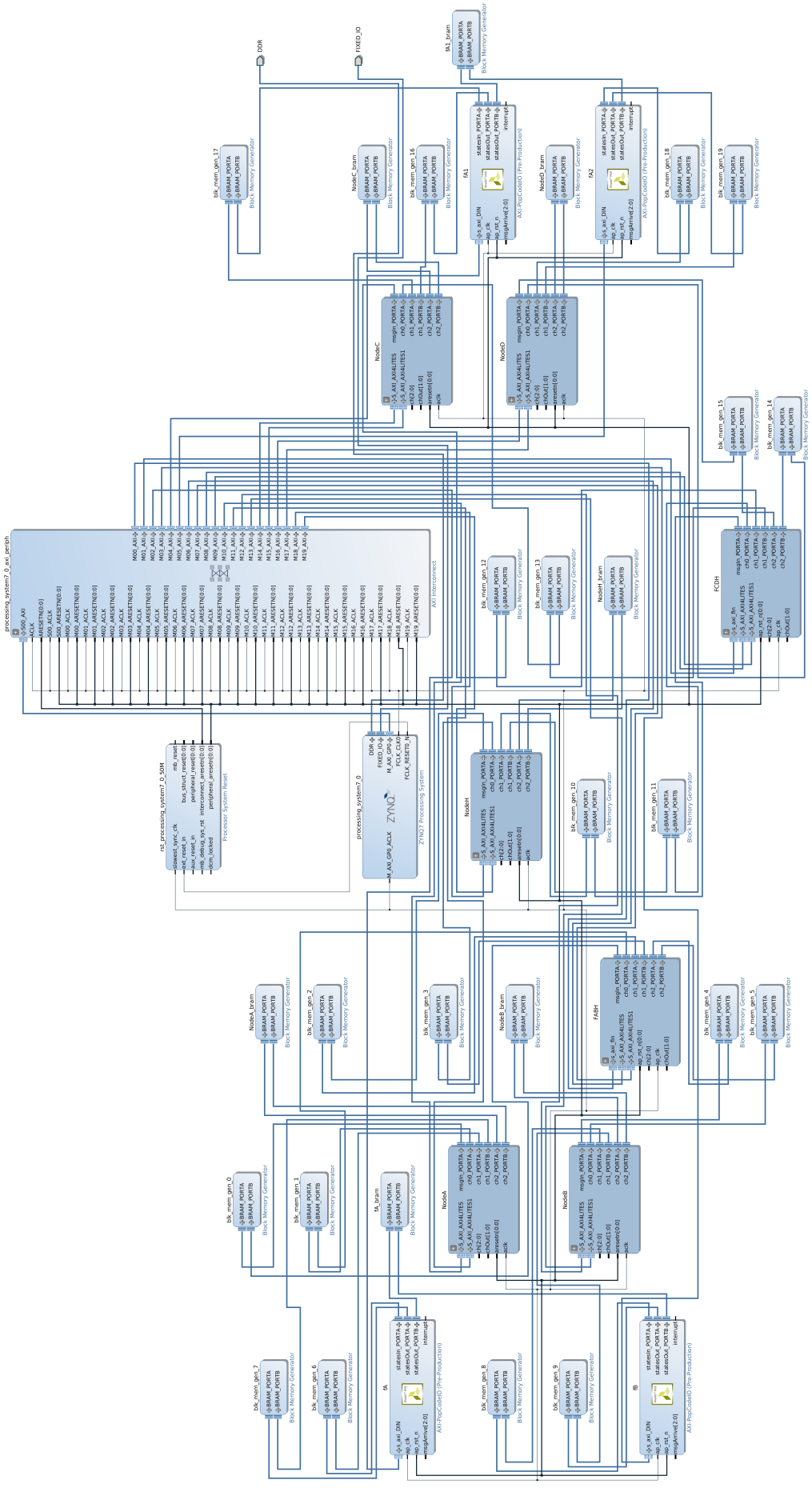
```
            fABC.putMessage(uA_to_fABC);

            /* give B to fABC */
            pmf.setVarValue(l.at(1).toInt());
            fB.setFactor(pmf.getVarStates());
            ufB_to_B.updateFactor(fB.computeMessage());
            B.putMessage(ufB_to_B);
            uB_to_fABC.updateFactor(B.computeMessage(fABC.getID(), &success));
            if(!success){
                info("Something wrong with message from B to fABC!");
                result = false; break;
            }
            fABC.putMessage(uB_to_fABC);

            /* finally, get the value out for C */
            ufABC_to_C.updateFactor(fABC.computeMessage(C.getID(), &success));
            if(!success){
                info("Something wrong with message from fABC to node C!");
                result = false; break;
            }

            pmf.setVarStates(normalizeStates(ufABC_to_C.getJPD()));
            res = pmf.getiVarValue();
            fOut << QString("%1\n").arg(res);
        }
    }
    fTest.close(); fRes.close();
    if(result)
        info("Performing inference done!");
    else
        info("Performing inference fail!");
    return result;
}

void c3VarMapping::run()
{
    if(trainJPD()) infer();
}

/********************** HELPER FUNCTIONS ***************************/
/* Tips for getNumLines: wc -l use_ps_grep_awk_kill.tips | awk '{print $1}'
 * we use awk because the output from “wc -l” also includes the filename
 */
int getNumLines(char *fname)
{
    int result = -1;
    char cmd[256];
    sprintf(cmd, "wc -l %s | awk '{print $1}'",fname);
    FILE *in;
    char buff[10];
    if(!(in = popen(cmd, "r"))){
        return -1;
    }
    fgets(buff, sizeof(buff), in);
    pclose(in);
    result = atoi(buff);
    return result;
}
```

```
int getPercentage(int i, int total)
{
    double result = (double)i*100.0/(double)total;
    return (int)round(result);
}
```

C Appendix-C

C.1 Embedded Factor Graph on SoC

This is a snapshot of the implementation of the network shown in Fig. 5.16 (page 5.16) using the SoC Z-7020.



Bibliography

[1] K. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, August 2012.

[2] The University of Manchester, *SpiNNaker: Universal Spiking Nerual Network Architecture*, datasheet version 2.02 ed., January 2011.

[3] S. Temple, *AppNote 1 - SpiNN-3 Development Board*. SpiNNaker Group, School of Computer Science, University of Manchester, November 2011.

[4] Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual*. Xilinx, Inc., February 2014.

[5] A. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, pp. 433–460, 1950.

[6] N. Liberman and Y. Trope, "The psychology of transcending the here and now," *Science*, vol. 322, pp. 1201–1205, 2008.

[7] S. Schaal and N. Schweighofer, "Computational motor control in humans and robots," *Current Opinion in Neurobiology*, vol. 15, no. 6, pp. 675–682, 2005.

[8] S. Schaal, "Is imitation learning the route to humanoid robots?," *Trends in Cognitive Sciences*, vol. 3, pp. 233–242, 1999.

[9] R. Frackowiak and H. Markram, "The future of human cerebral cartography: a novel approach," *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, vol. 370, no. 1668, 2015.

[10] W. Zhao, G. Agnus, V. Derycke, A. Filoramo, J.-P. Bourgoin, and C. Gamrat, "Nanotube devices based crossbar architecture: Toward neuromorphic computing," *Nanotechnology*, vol. 21, p. 175202, 2010.

[11] M.Soltiz, D. Kudithipudi, C. Merkel, G. Rose, and R. Pino, "Memristor-based neural logic blocks for nonlinearly separable functions," *IEEE Transactions on Computers*, vol. 62, pp. 1597–1606, August 2013.

[12] D. Monroe, "Neuromorphic computing gets ready for the (really) big time," *Communications of the ACM*, vol. 57, pp. 13–15, 2014.

[13] S. Russell and P. Norvid, *Artificial Intelligence: A Modern Approach, 3rd Ed.* New Jersey: Prentice Hall, 2010.

[14] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. Cambridge, Massachusetts: The MIT Press, 2009.

[15] M. Wainwright and M. Jordan, "Graphical models, exponential families, and variational inference," *Foundations and Trends in Machine Learning*, vol. 1, no. 1—2, pp. 1–305, 2008.

[16] C. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[17] D. Schröder, *Intelligent Observer and Control Design for Nonlinear Systems*. Springer, 2000.

[18] M. Toussaint and C. Goerick, "A bayesian view on motor control and planning," in *From Motor Learning to Interaction Learning in Robots*, Berlin: Springer, 2010.

[19] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. Massachusetts: The MIT Press, 2005.

[20] J. Pineau, M. Montemerlo, M. Pollack, N. Roy, and S. Thrun, "Towards robotic assistants in nursing homes: Challenges and results," *Robotics and Autonomous Systems*, vol. 42, no. 3, pp. 271–281, 2003.

[21] M. Toussaint, N. Plath, and N. Jetchev, "Integrated motor control, planning, grasping and high-level reasoning in a blocks world using probabilistic reasoning," in *IEEE International Conference on Robotics and Automation (ICRA) 2010*, (Anchorage, Alaska), 2010.

[22] R. Deventer, J. Denzler, and H. Niemann, "Control of dynamic systems using bayesian networks," in *International Conference on Computational Intelligence for Modelling, Control and Automation (CIMCA 2003)*, (Wien, Austria), 2003.

[23] A. Hommersom and P. J. Lucas, "Using bayesian networks in an industrial setting: Making printing systems adaptive," in *19th European Conference on Artificial Intelligence (ECAI2010)*, (Lisbon, Portugal), 2010.

[24] M. Jordan, *Learning in Graphical Models*. Kluwer Academic Pub., 1998.

[25] N. Vlassis and M. Toussaint, "Model-free reinforcement learning as mixture learning," in *International Conference on Machine Learning*, (Montreal, Canada), 2009.

[26] M. Kaess, V. Ila, R. Roberts, and F. Dellaert, "The bayes tree: An algorithmic foundation for probabilistic robot mapping," in *International Workshop on the Algorithmic Foundations of Robotics*, (Singapore), December 2010.

[27] I. Cox and J. Leonard, "Modeling a dynamic environment using a bayesian multiple hypothesis approach," *Artificial Intelligence*, vol. 66, no. 0, pp. 311–344, 1994.

[28] J. Leonard and H. Durrant-whyte, "Simultaneous map building and localization for an autonomous mobile robot," in *IEEE/RSJ International Workshop on Intelligent Robots and Systems'91 (IROS'91)*, (Osaka, Japan), 1991.

[29] M. Montemerlo and S. Thrun, "Simultaneous localization and mapping with unknown data association using fastslam," in *IEEE Int. Conf. Robotics and Automation*, (Taipei, Taiwan.), September 2003.

[30] F. Dellaert and M. Kaess, "Square root sam: Simultaneous localization and mapping via square root information smoothing," *International Journal of Robotics Research*, vol. 25, December 2006.

[31] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions On Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[32] H.-A. Loeliger, "An introduction to factor graphs," *Signal Processing Magazine, IEEE*, vol. 21, pp. 28–41, Jan 2004.

[33] Y. Weiss, C. Yanover, and T. Meltzer, "Map estimation, linear programming and belief propagation with convex free energies," in *The 23rd Conference on Uncertainty in Artificial Intelligence (UAI2007)*, (Vancouver, Canada), pp. 416–425, July 2007.

[34] F. Dellaert and M. Kaess, "Square root sam: Simultaneous location and mapping via square root information smoothing," *International lJournal of Robotics Research (IJRR)*, vol. 25, no. 12, pp. 1181–1213, 2006. Special issue on RSS 2006.

[35] M. Kaess, A. Ranganathan, and F. Dellaert, "isam: Incremental smoothing and mapping," *IEEE Transactions on Robotics (TRO)*, vol. 24, pp. 1365–1378, Septmber 2008.

[36] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. Leonard, and F. Dellaert, "isam2: Incremental smoothing and mapping using the bayes tree," *International Journal of Robotics Research (IJRR)*, vol. 31, pp. 216–235, February 2012.

[37] D. Rosen, M. Kaess, and J. Leonard, "An incremental trust-region method for robust online sparse least-squares estimation," in *IEEE International Conference on Robotics and Automation (ICRA) 2012*, (St. Paul, Minnesota, USA), pp. 1262–1269, May 2012.

[38] F. Dellaert, J. Carlson, V. Ila, K. Ni, and C. Thorpe, "Subgraph-preconditioned conjugate gradients for large scale slam," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2010)*, (Taipei, Taiwan), pp. 2566–2571, October 2010.

[39] L. Carlone, Z. Kira, C. Beall, V. Indelman, and F. Dellaert, "Eliminating conditionally independent sets in factor graphs: A unifying perspective based on smart factors," in *IEEE International Conference on Robotics and Automation (ICRA) 2014)*, (Hongkong, China), May 2014.

[40] T. Meltzer, A. Globerson, and Y. Weiss, "Convergent message passing algorithms - a unifying view," in *The 25th Conference on Uncertainty in Artificial Intelligence (UAI2009)*, (Montreal, Canada), pp. 393–401, June 2009.

[41] H. Guo and W. Hsu, "A survey of algorithms for real-time bayesian network inference," in *The joint AAAI-02/KDD-02/UAI-02 Workshop on Real-Time Decision Support and Diagnosis Systems*, (Edmonton, Canada), 2002.

[42] T. Minka, *A family of algorithms for approximate Bayesian inference*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 2001.

[43] J. Yedidia, W. Freeman, and Y. Weiss, "Constructing free-energy approximations and generalized belief propagation algorithms," *IEEE Transactions on Information Theory*, vol. 51, no. 7, pp. 2282–2312, 2005.

[44] T. Minka, "Expectation propagation for approximate bayesian inference," in *17th Conference in Uncertainty in Artificial Intelligence*, (Washington, USA), 2001.

[45] M. Beal, *Variational Algorithms for Approximate Bayesian Inference*. PhD thesis, Gatsby Computational Neuroscience Unit, University College London, 2003.

[46] B. J. Frey and N. Jojic, "A comparison of algorithms for inference and learning in probabilistic graphical models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 9, pp. 1–25, 2005.

[47] W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing mlp-bp on fpgas: a study," *IEEE Transaction on Neural Networks*, vol. 18, pp. 240–252, January 2007.

[48] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, pp. 81–98, January 1989.

[49] J. Dougherty, R. Kovani, and M. Sahami, "Supervised and unsupervised discretization of continuous features," in *The 12th International Conference on Machine Learning*, pp. 194–202, San Francisco: Morgan Kaufmann, 1995.

[50] S. Garcia, J. Luengo, J. Sáez, V. López, and F. Herrera, "A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, pp. 734–750, April 2013.

[51] E. Clarke and B. Barton, "Entropy and mdl discretization of continuous variables for bayesian belief networks," *International Journal of Intelligent Systems*, vol. 15, pp. 61–92, 2000.

[52] S. Kotsiantis and D. Kanellopoulos, "Discretization techniques: A recent survey," *GESTS International Transactions on Computer Science and Engineering*, vol. 32, no. 1, pp. 47–58, 2006.

[53] U. Fayyad and K. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," in *the International Joint Conference on Uncertainty in AI (IJCAI'93)*, (Chambery, France), pp. 1022–1027, 1993.

[54] A. Mathis, A. Herz, and M. Stemmler, "Resolution of nested neuronal representations can be exponential in the number of neurons," *Physical Review Letters*, vol. 109, p. 018103, Jul 2012.

[55] H. Guo and W. Hsu, "A survey of algorithms for real-time bayesian network inference," in *The joint AAAI-02/KDD-02/UAI-02 workshop on Real-Time Decision Support and Diagnosis Systems*, (Edmonton, Alberta, Canada), 2002.

[56] V. Namasivayam, A. Pathak, and V. Prasanna, "Scalable parallel implementation of bayesian network to junction tree conversion for exact inference," in *The 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'06)*, (Ouro Preto, Minas Gerais, Brasil), October 2006.

[57] V. Sudhakar and C. Murthy, "Efficient mapping of backpropagation algorithm onto a network of workstations," *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 28, pp. 841–849, 1998.

[58] M. Charalambous, P. Trancoso, and A. Stamatakis, "Initial experiences porting a bioinformatics application to a graphics processor," in *10th Panhellenic Conference on Informatics (PCI2005)*, (Volas, Greece), November 2005.

[59] D. Ayres, A. Darling, D. Zwickl, P. Beerli, M. Holder, P. Lewis, J. Huelsenbeck, F. Ronquist, D. Swofford, M. Cummings, A. Rambaut, and M. Suchard, "Beagle: An application programming interface and high-performance computing library for statistical phylogenetics," *Systematic Biology*, vol. 61, pp. 170–173, 2012.

[60] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. Owens, "Efficient computation of sum-products on gpus through software-managed cache," in *Proceedings of the 22nd annual international conference on Supercomputing (ICS'08)*, (New York, NY, USA), pp. 309–318, ACM, 2008.

[61] N. Piatkowski, "Parallel algorithms for gpu accelerated probabilistic inference," in *NIPS 2011: workshop on parallel and large-scale machine learning*, (Sierra Nevada, Spain), December 2011.

[62] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on gpus," in *the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'13)*, (Shenzhen, China), pp. 147–156, February 2013.

[63] S. Korl, *A Factor Graph Approach to Signal Modelling, System Identification and Filtering*. PhD thesis, Swiss Federal Institute of Technology, Zürich, 2005.

[64] E. Maneva, *Belief Propagation Algorithms for Constraint Satisfaction Problems*. PhD thesis, Graduate Division of the University of California, Berkeley, 2006.

[65] L. Kroc, *Probabilistic Techniques for Constraint Satisfaction Problems*. PhD thesis, Faculty of the Graduate School of Cornell University, August 2009.

[66] R. Stengel, "Toward intelligent flight control," *IEEE Transaction on System, Man, and Cybernetics*, vol. 23, no. 6, pp. 1699–1725, 1993.

[67] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011.

[68] A. Omondi, J. Rajapakse, and M. Bajger, *FPGA Implementations of Neural Networks*, ch. FPGA Neurocomputers. Springer, 2006.

[69] D. Uliana, K. Kepa, and P. Athanas, "Fpga-based hpc application design for non-experts," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 261–264, June 2013.

[70] V. Manshinghka, *Natively Probabilistic Computation*. PhD thesis, Department of Brain & Cognitive Sciences, Massachusetts Institute of Technology, June 2009.

[71] F. Palmieri, "Learning non-linear functions with factor graphs," *IEEE Transactions on Signal Processing*, vol. 61, no. 17, pp. 4360–4371, 2013.

[72] A. Steimer, *Neurally Inspired Models of Belief-Propagation in Arbitrary Graphical Models*. PhD thesis, ETH Zürich, Switzerland, 2012.

[73] D. Göhlsdorf, *Motor Control with Graphical Models*. PhD thesis, ETH Zürich, Switzerland, 2012.

[74] S. Roweis and Z. Ghahramani, "A unifying review of linear gaussian models," *Neural Computation*, vol. 11, no. 2, pp. 305–345, 1999.

[75] J. Diard, P. Bessiere, and E. Mazer, "A survey of probabilistic models using the bayesian programming methodology as a unifying framework," in *International Conference on Computational Intelligence, Robotics and Autonomous Systems (IEEE-CIRAS)*, (Singapore), 2003.

[76] D. Barber, *Bayesian Reasoning and Machine Learning*. Cambridge University Press, March 2012.

[77] G. Winkler, *Image Analysis, Random Fields and Markov Chain Monte Carlo Methods*. Springer-Verlag, 2003.

[78] 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, pp. 1068–1080, June 2008.

[79] S. Li, *Markov Random Field Modeling in Image Analysis*. Advances in Computer Vision and Pattern Recognition, Springer-Verlag London Ltd., 2009.

[80] A. Blake and P. Kohli, *Markov Random Fields for Vision and Image Processing*. The MIT Press, July 2011.

[81] J. Domke, A. Karapurkar, and Y. Aloimonos, "Who killed the directed model?," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2008)*, (Anchorage, Alaska, USA), June 2008.

[82] Y. Yeh, K. Breeden, L. Yang, M. Fisher, and P. Hanrahan, "Synthesis of tiled patterns using factor graphs," in *The 40th International Conference and Exhibitation on Computer Graphics and Interactive Techniques*, (Anaheim, California, USA), 2013.

[83] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan kaufmann, 1988.

[84] E. Charniak, "Bayesian networks without tears," *AI Magazine, AAAI*, vol. 12, no. 4, pp. 50–63, 1991.

[85] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian networks without classifiers," *Machine Learning*, vol. 29, pp. 131–163, 1997.

[86] D. Niedermayer, *Innovations in Bayesian Networks: Theory and applications*, ch. An introduction to Bayesian Networks and their contemporary applications, pp. 117–130. Studies in computational intelligence, Berlin: Springer, 2008.

[87] G. Cooper, "The computational complexity of probabilistic inference using bayesian belief networks," *Artificial Intelligence*, vol. 42, no. 2-3, pp. 393–405, 1990.

[88] D. MacKay, "Probable networks and plausible predictions – a review of practical bayesian methods for supervised neural networks," *Networks: Computation in Neural Systems*, vol. 6, pp. 469–505, 1995.

[89] B. Frey, "Extending factor graphs so as to unify directed and undirected graphical models," in *The Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, (Acapulco, Mexico), 2003.

[90] K. Murphy, Y. Weiss, and M. Jordan, "Loopy belief propagation for approximate inference: An empirical study," in *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, UAI'99, (San Francisco, CA, USA), pp. 467–475, Morgan Kaufmann Publishers Inc., 1999.

[91] A. Ihler, W. J. Fischer III, and A. Willsky, "Loopy belief propagation: Convergence and effects of message errors," *The Journal of Machine Learning Research*, vol. 6, pp. 905–936, December 2005.

[92] J. Bolt and L. van der Gaag, "On the convergence error in loopy propagation," in *16th Belgian-Dutch Conference on Artificial Intelligence*, BNAIC2004, (Groningen, The Netherlands), 2004.

[93] J. Mooij and H. Kappen, "Sufficient conditions for convergence of loopy belief propagation," in *the Twenty-First Conference on Uncertainty in Artificial Intelligence*, UAI2005, (Edinburg, Scotland), 2005.

[94] G. Elidan, I. McGraw, and D. Koller, "Residual belief propagation: Informed scheduling for asynchronous message passing," in *Proceedings of the Twenty-second Conference on Uncertainty in AI (UAI)*, (Boston, Massachussetts), July 2006.

[95] P. Sen and L. Getoor, "Empirical comparison of approximate inference algorithms for networked data," in *ICML workshop on Open Problems in Statistical Relational Learning 2006*, SRL2006, (Pittsburgh, Pennsylvania, USA), 2006.

[96] C. Yanover and Y. Weiss, "Finding the m most probable configurations using loopy belief propagation," in *Advances in Neural Information Processing Systems 16*, pp. 289–296, MIT Press, 2004.

[97] P. Abbeel, D. Koller, and A. Y.Ng, "Learning factor graphs in polynomial time and sample complexity," *Journal of Machine learning*, vol. 7, pp. 1743–1788, 2006.

[98] D. Ackley, G. Hinton, and T. Sejnowski, "A learning algorithm for boltzmann machines," *Cognitive Science*, vol. 9, no. 1, pp. 147–169, 1985.

[99] G. Hinton, "A practical guide to training restricted boltzmann machines," tech. rep., Department of Computer Science, University of Toronto, August 2010.

[100] A. Fischer and C. Igel, "Training restricted boltzmann machines: An introduction," *Pattern Recognition*, vol. 47, pp. 25–39, 2014.

[101] N. Noorshams and M. Wainwright, "Belief propagation for continuous state spaces: Stochastic message-passing with quantitative guarantees," *Journal of Machine Learning Research*, vol. 14, pp. 2799–2835, 2013.

[102] C. K. Chow and C. Liu, "Approximating discrete probability distributions with dependence trees," *IEEE Transaction on Information Theory*, vol. 14, no. 3, pp. 462–467, 1968.

[103] R. Neal, "Connectionist learning of belief networks," *Artificial Intelligence*, vol. 56, pp. 71–113, 1992.

[104] W. Gerstner and W. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[105] J. Kim and J. Pearl, "A computational model for combined causal and diagnostic reasoning in inference systems," in *Eighth International Joint Conference on Artificial Intelligence. IJCAI-83*, (Karlsruhe, Germany), 1983.

[106] L. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, pp. 257–286, February 1989.

[107] Z. Ghahramani, "An introduction to hidden markov models and bayesian networks," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 15, no. 1, pp. 9–42, 2001.

[108] Y. Weiss and W. Freeman, "On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs," *IEEE Transactions on Information Theory*, vol. 47, pp. 736–744, February 2001.

[109] Y. Watanabe and K. Fukumizu, "Graph zeta function in the bethe free energy and loopy belief propagation," in *Advances in Neural Information Processing Systems 22* (Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, eds.), pp. 2017–2025, Curran Associates, Inc., 2009.

[110] Y. Weiss and W. Freeman, "Correctness of belief propagation in gaussian graphical models of arbitrary topology," *Neural Computation*, vol. 13, pp. 2173–2200, October 2001.

[111] M. Tappen and W. Freeman, "Comparison of graph cuts with belief propagation for stereo, using identical mrf parameters," in *the Ninth IEEE International Conference on Computer Vision (ICCV)*, (Nice, France), pp. 900–907, October 2003.

[112] M. Tappen, B. Russell, and W. Freeman, "Efficient graphical models for processing images," in *the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 2, pp. 302–309, July 2004.

[113] D. Rosen, M. Kaess, and J. Leonard, "Robust incremental online inference over sparse factor graphs: Beyond the gaussian case," in *IEEE International Conference on Robotics and Automation (ICRA) 2013*, (Karlsruhe, Germany), pp. 1025–1032, May 2013.

[114] C. Rasmussen, "The infinite gaussian mixture model," *Advances in Neural Information Processing Systems*, vol. 12, pp. 554–560, 2000.

[115] I. Myung, "Tutorial on maximum likelihood estimation," *Journal of Mathematical Psychology*, vol. 47, pp. 90–100, 2003.

[116] N. Slonim and Y. Weiss, "Maximum likelihood and the information bottleneck," in *Advances in Neural Information Processing Systems 15* (S. Becker, S. Thrun, and K. Obermayer, eds.), pp. 351–358, MIT Press, 2003.

[117] J. Jang, C.Sun, and E. Mizutani, *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Prentice Hall, 1997.

[118] W. Forst and D. Hoffmann, *Optimization - Theory and Practice*. Springer, 2010.

[119] G. McLachlan and T. Krishnan, *The EM Algorithm and Extensions*. Wiley-Interscience, 2nd ed. ed., 2008.

[120] J. Dauwels, S. Korl, and H.-A. Loeliger, "Expectation maximization as message passing," in *International Symposium on Information Theory 2005 (ISIT 2005)*, pp. 583–586, September 2005.

[121] Y. Zhao, J. Xu, and Y. Gao, "A parallel algorithm for bayesian network parameter learning based on factor graph," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI2013)*, (Washington DC, USA), pp. 506–512, November 2013.

[122] J. Mooij, "libDAI: A free and open source C++ library for discrete approximate inference in graphical models," *Journal of Machine Learning Research*, vol. 11, pp. 2169–2173, August 2010.

[123] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communication of the ACM*, vol. 51, pp. 107–113, January 2008.

[124] Z. Song, "Bayesian max-product expectation maximization algorithm for structured sparse signals reconstruction," Master's thesis, Iowa State University, 2012.

[125] A. Deneve, P. Latham, and A. Pouget, "Efficient computation and cue integration with noisy population codes," *Nature Neuroscience*, vol. 4, no. 8, pp. 826–831, 2001.

[126] S. Wu, S. Amari, and H. Nakahara, "Population coding and decoding in a neural field: a computational study," *Neural Computation*, vol. 14, no. 5, pp. 999–1026, 2002.

[127] R. Ince, R. Senatore, E. Arabzadeh, F. Montani, M. Diamond, and S. Panzeri, "Information-theoretic methods for studying population codes," *Neural Networks*, vol. 23, no. 6, pp. 713–727, 2010.

[128] E. Doi and M. Lewicki, "A simple model of optimal population coding for sensory systems," *PLoS Computational Biology*, vol. 10, pp. 1–14, August 2014.

[129] J. M. Beck, W. Ma, R. Kiani, T. Hanks, A. Churchland, J. Roitman, M. Shadlen, P. Latham, and A. Pouget, "Probabilistic population codes for bayesian decision making," *Journal Neuron*, vol. 60, no. 6, pp. 1142–1152, 2008.

[130] J. Beck, A. Pouget, and K. Heller, "Complex inference in neural circuits with probabilistic population codes and topic models," *Advances in Neural Information Processing Systems*, vol. 25, pp. 3068–3076, 2012.

[131] N. Rougier and J. Vitay, "Emergence of attention within a neural population," *Neural Networks*, vol. 19, no. 5, pp. 573–581, 2006.

[132] E. Todomorov, "Optimality principles in sensorimotor control," *Nature Neuroscience*, vol. 7, no. 9, pp. 907–915, 2004.

[133] H. Snippe, "Parameter extraction from population codes: A critical assessment," *Neural Computation*, vol. 8, no. 3, pp. 511–529, 1996.

[134] S. Deneve, P. Latham, and A. Pouget, "Reading population codes: a neural implementation of ideal observers," *Nature neuroscience*, vol. 2, pp. 740–745, August 1999.

[135] X. Cui and A. Alwan, "Robust speaker adaptation by weighted model averaging based on the minimum description length criterion," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 15, pp. 652–660, February 2007.

[136] Y. Shulin and C. Kuo-Chu, "Comparison of score metrics for bayesian network learning," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 32, pp. 419–428, May 2002.

[137] R. Rohwer and J. van der Rest, "Minimum description length, regularization, and multimodal data," *Neural Computation*, vol. 8, pp. 595–609, April 1996.

[138] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, no. 1, pp. 59–69, 1982.

[139] A. Gorban and A. Zinovyev, "Principal manifolds and graphs in practice: from molecular biology to dynamical systems," *International Journal of Neural Systems*, vol. 20, no. 3, pp. 219–232, 2010.

[140] E. Bullmore and D. Bassett, "Brain graphs: Graphical models of the human brain connectome," *Annu. Rev. Clin. Psychol.*, vol. 7, pp. 113–140, December 2011.

[141] C. Butts, "Revisiting the foundations of network analysis," *Science*, vol. 325, pp. 414–416, July 2009.

[142] M. Matell, E. Shea-Brown, C. Gooch, A. Wilson, and J. Rinzel, "A heterogeneous population code for elapsed time in rat medial agranular cortex," *Behav. Neurosci.*, vol. 125, no. 1, pp. 54–73, 2011.

[143] J. Jun, P. Miller, A. Hernandes, A. Zainos, L. Lemus, C. Brody, and R. Romo, "Heterogenous population coding of a short-term memory and decision task," *The Journal of Neuroscience*, vol. 30, pp. 916–929, January 2010.

[144] M. Shamir and H. Sompolinsky, "Implications of neuronal diversity on population coding," *Neural Computation*, vol. 18, pp. 1951–1986, August 2006.

[145] K. Murphy, "Software packages for graphical models." http://www.cs.ubc.ca/~murphyk/Software/bnsoft.html, 2015.

[146] L. Breiman, "Hinging hyperplanes for regression, classification and function approximation," *IEEE Transaction on Information Theory*, vol. 39, no. 3, pp. 999–1013, 1993.

[147] R. Schalkoff, *Pattern Recognition: Statistical, Structural and Neural Approaches*. Singapore: John Wiley & Sons, Inc., 1993.

[148] N. Shental, A. Zomet, T. Hertz, and Y. Weiss, "Pairwise clustering and graphical models," in *Advances in Neural Information Processing Systems 16* (S. Thrun, L. Saul, and B. Schölkopf, eds.), pp. 185–192, MIT Press, 2004.

[149] S. Lange, N. Sunderhauf, and P. Protzel, "Incremental smoothing vs. filtering for sensor fusion on an indoor uav," in *IEEE International Conference on Robotics and Automation (ICRA) 2013*, (Karlsruhe, Germany), pp. 1773–1778, May 2013.

[150] H. Chiu, X. Zhou, L. Carlone, F. Dellaert, S. Samarasekera, and R. Kumar, "Constrained optimal selection for multi-sensor robot navigation using plug-and-play factor graphs," in *IEEE International Conference on Robotics and Automation (ICRA) 2014)*, (Hongkong, China), May 2014.

[151] V. Indelman, S. Wiliams, M. Kaess, and F. Dellaert, "Information fusion in navigation systems via factor graph based incremental smoothing," *Robotics and Autonomous Systems*, vol. 61, pp. 721–738, August 2013.

[152] M. C. J.T. Rolfe, "Multifactor expectation maximization for factor graphs," in *International Conference on Artificial Neural Networks (ICANN) 2010*, (Thessaloniki, Greece), pp. 267–276, September 2010.

[153] P. Mirowski and Y. LeCun, "Dynamic factor graphs for time series modeling," in *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD 2009*, (Bled, Slovenia), pp. 128–143, September 2009.

[154] R. Saegusa, G. Metta, G. Sandini, and S. Sakka, "Active motor babbling for sensorimotor learning," in *IEEE International Conference on Robotics and Biomimetics 2008 (ROBIO2008)*, pp. 794–799, February 2009.

[155] A. D'Souza, S. Vijayakumar, and S. Schaal, "Learning inverse kinematics," in *International Conference on Intelligence in Robotics and Autonomous Systems (IROS 2001)*, (Hawaii, USA), pp. 298–303, 2001.

[156] M. Lungarella, G. Metta, R. Pfeifer, and G. Sandini, "Developmental robotics: a survey," *Connection Science*, vol. 15, no. 4, pp. 151–190, 2003.

[157] Y. Demiris and A. Dearden, "From motor babbling to hierarchical learning by imitation: a robot developmental pathway," in *the Fifth International Workshop on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems*, (Nara, Japan), pp. 31–37, July 2005.

[158] D. Caligiore, T. Ferrauto, D. Parisi, N. Accornero, M. Capozza, and G. Baldassarre, "Using motor babbling and hebb rules for modeling the development of reaching with obstacles and grasping," in *International Conference on Cognitive Systems 2008 (CogSys2008)*, (Karlsruhe, Germany), 2008.

[159] A. Streri and J. Feron, "The development of haptic abilities in very young infants: From perception to cognition," *Infant Behavior and Development*, vol. 28, no. 3, pp. 290–304, 2005.

[160] K. Konolige and M. Agrawal, "Frameslam: from bundle adjustment to realtime visual mappping," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 1066–1077, 2008.

[161] A. D'Souza, S. Vijayakumar, and S. Schaal, "Learning inverse kinematics," in *IEEE International Conference on Intelligent Robots and Systems (IEEE/RSJ)*, (Maui, Hawaii, USA), pp. 298–303, 2001.

[162] A. Aristidou and J. Lasenby, "Inverse kinematics: a review of existing techniques and introduction of a new fast iterative solver," tech. rep., Cambridge University Engineering Department, September 2009.

[163] R. Diankov, *Automated Construction of Robotic Manipulation Programs*. PhD thesis, The Robotics Institute, Carnegie Mellon University, August 2010.

[164] G. Grudic and P. Lawrence, "Iterative inverse kinematics with manipulator configuration control," *IEEE Transactions on Robotics and Automation*, vol. 9, pp. 476–483, August 1993.

[165] T. Fujiwara, Y. Maeda, and H. Ito, "Learning of inverse-kinematics for robot using high dimensional neural networks," in *SICE Annual Conference 2013 (SICE2013)*, pp. 2743–2748, September 2013.

[166] E. Oyama, A. Agah, K. MacDorman, T. Maeda, and S. Tachi, "A modular neural network architecture for inverse kinematics model learning," *Neurocomputing*, vol. 38–40, pp. 797–805, 2001.

[167] D. Gorinevsky and T. Connolly, "Comparison of some neural network and scattered data approximations: The inverse manipulator kinematics example," *Neural Computation*, vol. 6, pp. 521–542, May 1994.

[168] S.-W. Kim, J. J. Lee, and M. Sugisaka, "Inverse kinematics solution based on fuzzy logic for redundant manipulators," in *IEEE/RSJ International Conference on Intelligent Robots and Systems '93 (IROS'93)*, vol. 2, pp. 904–910, July 1993.

[169] J. Jih-Gau, "Fuzzy neural network approaches for robotic gait synthesis," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 30, pp. 594–601, August 2000.

[170] A. Khwaja, M. Rahman, and M. G. Wagner, *Advances in Robot Kinematics: Analysis and Control*, ch. Inverse Kinematics of Arbitrary Robotic Manipulators Using Genetic Algorithms, pp. 375–382. Springer Netherlands, 1998.

[171] F. Chapelle and P. Bidaud, "A closed form for inverse kinematics approximation of general 6r manipulators using genetic programming," in *IEEE International Conference on Robotics and Automation (ICRA) 2001*, vol. 4, pp. 3364–3369, 2001.

[172] P. Kalra, P. Mahapatra, and D. Aggarwal, "An evolutionary approach for solving the multimodal inverse kinematics problem of industrial robots," *Mechanism and Machine Theory*, vol. 41, pp. 213–1229, October 2006.

[173] T. Li-Chun and C. Chih, "A combined optimization method for solving the inverse kinematics problem of mechanical manipulators," *IEEE Transactions on Robotics and Automation*, vol. 7, pp. 489–499, August 1991.

[174] J. Sturm, C. Stachniss, and W. Burgard, "A probabilistic framework for learning kinematic models of articulated objects," *Journal of Artificial Intelligence Research*, vol. 41, pp. 477–526, August 2011.

[175] P. Artemiadis, P. Katsiaris, and K. Kyriakopoulos, "A biomimetic approach to inverse kinematics for a redundant robot arm," *Autonomous Robots*, vol. 29, no. 3–4, pp. 293–308, 2010.

[176] M. Ito, "Mechanisms of motor learning in the cerebellum," *Brain Research*, vol. 886, pp. 237–245, December 2000.

[177] M. McDannald, Y. Takahashi, N. Lopatina, B. Pietras, J. Jones, and G. Schoenbaum, "Model-based learning and the contribution of the orbitofrontal cortex to the model-free world," *European Journal of Neuroscience*, vol. 35, pp. 991–996, April 2012.

[178] S. Lee, S. Shimojo, and J. O'Doherty, "Neural computations underlying arbitration between model-based and model-free learning," *Neuron*, vol. 81, pp. 687–699, February 2014.

[179] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[180] D. Song, K. Huebner, V. Kyrki, and D. Kragic, "Learning task constraints for robot grasping using graphical models," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2010)*, pp. 1579–1585, Oct 2010.

[181] S. Schaal, *Adaptive Motion of Animals and Machines*, ch. Dynamic Movement Primitives - A Framework for Motor Control in Humans and Humanoid Robotics. Springer Tokyo, 2006.

[182] J. Aleotti and S. Caselli, "Robust trajectory learning and approximation for robot programming by demonstrations," *Robotics and Autonomous Systems*, vol. 54, pp. 409–413, May 2006.

[183] S. Calinon, F. Guenter, and A. Billard, "On learning, representing, and generalizing a task in a humanoid robot," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 37, pp. 286–298, April 2007.

[184] K. Loken, "Imitation-based learning of bipedal walking using locally weighted learning," Master's thesis, The University of British Columbia, 2006.

[185] D. Grimes, R. Chalodhorn, and R. Rao, "Dynamic imitation in a humanoid robot through nonparametric probabilistic inference," in *Proceedings of Robotics: Science and Systems*, (Philadelphia, USA), August 2006.

[186] A. Ijspeert, J. Nakanishi, and S. Schaal, "Movement imitation with nonlinear dynamical systems in humanoid robots," in *IEEE International Conference on Robotics and Automation (ICRA) 2002*, vol. 2, pp. 1398–1403, May 2002.

[187] N. Delson and H. West, "Robot programming by human demonstration: Adaptation and inconsistency in constrained motion," in *IEEE International Conference on Robotics and Automation (ICRA) 1996*, pp. 30–36, 1996.

[188] K. Mülling, J. Kober, O. Krömer, and J. Peters, "Learning to select and generalize striking movements in robot table tennis," *International Journal of Robotics Research*, vol. 32, no. 3, pp. 280–298, 2013.

[189] S. Calinon, *Robot Programming by Demonstration: a Probabilistic Approach*. EPFL Press, 1st edition ed., 2009.

[190] A. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal, "Dynamical movement primitives: learning attractor models for motor behaviors," *Neural Computation*, vol. 25, pp. 328–373, February 2013.

[191] E. Rückert, G. Neumann, M. Toussaint, and W. Maass, "Learned graphical models for probabilistic planning provide a new class of movement primitives," *Frontiers in Computational Neuroscience*, vol. 6, January 2013.

[192] S. Furber, D. Lester, L. Plana, J. Garside, E. Painkras, S. Temple, and A. Brown, "Overview of the spinnaker system architecture," *IEEE Transactions on Computers*, vol. 62, pp. 2454–2467, December 2013.

[193] E. Painkras, L. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. Lester, A. Brown, and S. Furber, "Spinnaker: A 1w 18-core system-on-chip for massively-parallel neural network simulation," *IEEE Journal of Solid-State Circuits*, vol. 48, pp. 1943–1953, August 2013.

[194] M. Khan, D. Lester, L. Plana, A. Rast, X. Jin, E. Painkras, and S. Furber, "Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor," in *International Joint Conference on Neural Networks (IJCNN 2008)*, (Hongkong, China), 2008.

[195] L. Plana, J. Bainbridge, S. Furber, S. Salisbury, Y. Shi, and J. Wu, "An on-chip and inter-chip communications network for the spinnaker massively-parallel neural net simulator," in *The Second ACM/IEEE International Symposium on Networks-on-Chip*, (Newcastle, UK), 2008.

[196] M. Lin, I. Lebedev, and J. Wawrzynek, "High-throughput bayesian computing machine with reconfigurable hardware," in *The 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'10)*, (Monterey, USA), 2012.

[197] X. Jin, M. Luján, M. Khan, L. Plana, A. Rast, S. Welbourne, and S. Furber, "Algorithm for mapping multilayer bp networks onto the spinnaker neuromorphic hardware," in *2010 Ninth International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 9–16, July 2010.

[198] F. Galluppi, S. Davies, A. Rast, T. Sharp, and S. Furber, "A hierarchical configuration system for a massively parallel neural hardware platform," in *The 9th Conference of Computing Frontiers*, (Cagliari, Italy), 2012.

[199] E. Stromatias, F. Galluppi, C. Patterson, and S. Furber, "Power analysis of large-scale, real-time neural networks on spinnaker," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, August 2013.

[200] L. Crockett, R. Elliot, M. Enderwitz, and R. Stewart, *The Zynq Book: Embedded Processing with ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. University of Strathclyde, Glasgow, UK, 2014.

[201] IEEE, *IEEE Standard for Binary Floating-Point Arithmetic*. The Institute of Electrical and Electronics Engineers, Inc., 2008.

[202] T. Czajkowski, *Physical Synthesis Toolkit for Area and Power Optimization on FPGAs*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Canada, 2008.

[203] Xilinx, *Spartan-6 FPGA Memory Controller*. Xilinx, Inc., August 2010.

[204] L. Semiconductor, *Implementing High-Speed DDR3 Memory Controllers in a Mid-Range FPGA*. Lattice Semiconductor, March 2010.

[205] E. Marchi, M. Cervetto, and M. Tenorio, "A ddr3 memory based time interleaving fpga implementation for isdb-t standard," in *Programmable Logic (SPL), 2011 VII Southern Conference on*, pp. 1–5, April 2011.

[206] S. Aqueel and K. Khare, "Design and fpga implementation of ddr3 sdram controller for high performance," *International Journal of Computer Science & Information Technology (IJCSIT)*, vol. 3, no. 4, pp. 101–110, 2011.

[207] N. Margolus, "An fpga architecture for dram-based systolic computations," in *The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 2–11, April 1997.

[208] P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006.

[209] D. Capalija and T. Abdelrahman, "An architecture for exploiting coarse-grain parallelism on fpgas," in *International Conference on Field-Programmable Technology 2009 (FPT 2009)*, pp. 285–291, December 2009.

[210] P. Banerjee, M. Haldar, A. Nayak, V. Kim, D. Bagchi, S. Pal, and N. Tripathi, "A behavioral synthesis tool for exploiting fine grain parallelism in fpgas," in *Distributed Computing* (S. Das and S. Bhattacharya, eds.), vol. 2571 of *Lecture Notes in Computer Science*, pp. 246–256, Springer Berlin Heidelberg, 2002.

[211] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *IEEE Design Test*, vol. 31, pp. 19–30, February 2014.

[212] A. Papadopoulosa, I. Kirmitzogloub, V. Promponasb, and T. Theocharides, "Fpga-based hardware acceleration for local complexity analysis of massive genomic data," *INTEGRATION, the VLSI Journal*, vol. 46, pp. 230–239, June 2013.

[213] Xilinx, *LogiCORE IP AXI DMA v7.1.* Xilinx, Inc., April 2014.

[214] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *IEEE Computer Society Annual Symposium on VLSI 2002*, pp. 105–112, 2002.

[215] S. Sarkar, G. Kulkarni, P. Pande, and A. Kalyaraman, "Network-on-chip hardware accelerators for biological sequence alignment," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 29–41, 2010.

[216] S. Furber, S. Temple, and A. Brown, "On-chip and inter-chip networks for modeling large-scale neural systems," in *IEEE International Symposium on Circuits and Systems 2006 (ISCAS 2006)*, May 2006.

[217] A. Rast, Y. Shufan, M. Khan, and S. Furber, "Virtual synaptic interconnect using an asynchronous network-on-chip," in *IEEE International Joint Conference on Neural Networks 2008 (IJCNN 2008)*, pp. 2727–2734, June 2008.

[218] M. Isard, J. MacCormick, and K. Achan, "Continuously-adaptive discretization for message-passing algorithms," in *Advances in Neural Information Processing Systems 21* (D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, eds.), pp. 737–744, Curran Associates, Inc., 2009.

[219] C. Hollabaugh, *Embedded Linux: Hardware, Software, and Interfacing.* Addison-Wesley Professional, 1st edition ed., March 2002.

[220] Xilinx, *PetaLinux Tools Application Development Guide.* Xilinx, Inc., June 2014.

[221] Xilinx, *PetaLinux Tools Board Bringup Guide.* Xilinx, Inc., June 2014.

[222] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers.* O'Reilly Media Inc., 3rd edition ed., January 2005.

[223] M. Mitchell, J. Oldham, and A. Samuel, *Advanced Linux Programming.* New Riders Publishing, 1st edition ed., January 2001.

[224] Xilinx, *PetaLinux Tools Getting Started Guide.* Xilinx, Inc., June 2014.

[225] J. Eugenio and M. Estrada, "Hardware/software fpga architecture for robotics applications," in *the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'09, (Berlin, Heidelberg), pp. 27–38, Springer-Verlag, 2009.

[226] A. Inc., *E64G401 Epiphany 64-core Microprocessor Datasheet.* Adapteva Inc., March 2011.

[227] A. Toga, K. Clark, P. Thompson, D. Shattuck, and J. V. Horn, "Mapping the human connectome," *Neurosurgery*, vol. 71, no. 1, pp. 1–5, 2012.

[228] M. Cao, J.-H. Wang, Z.-J. Dai, X.-Y. Cao, L.-L. Jiang, F.-M. Fan, X.-W. Song, M.-R. Xia, N. Shu, Q. Dong, M. Milham, F. Castellanos, X.-N. Zuo, and Y. He, "Topological organization of the human brain functional connectome across the lifespan," *Developmental Cognitive Neuroscience*, vol. 7, no. 0, pp. 76–93, 2014.

[229] J. Richiardi, S. Achard, H. Bunke, and D. V. D. Ville, "Machine learning with brain graphs: Predictive modeling approaches for functional imaging in systems neuroscience," *IEEE Signal Processing Magazine*, vol. 30, pp. 58–70, May 2013.

[230] M. Jordan and T. Sejnowski, *Graphical Models: Foundations of Neural Computation (Computational Neuroscience)*. A Bradford Book: The MIT Press, 2001.

[231] W. Shirer, S. Ryali, E. Rykhlevskaia, V. Menon, and M. Greicius, "Decoding subject-driven cognitive states with whole-brain connectivity patterns," *Cerebral Cortex*, vol. 22, pp. 158–165, January 2012.

[232] J. Heinzle, M. Wenzel, and J. Haynes, "Visuomotor functional network topology predicts upcoming tasks," *The Journal of Neuroscience*, vol. 32, pp. 9960–9968, July 2012.

[233] D. Danks, *Unifying the Mind: Cognitive Representations as Graphical Models*. The MIT Press, September 2014.

[234] A. Saxena, A. Jain, O. Sener, A. Jami, D. Misra, and H. Koppula, "Robobrain: Large-scale knowledge engine for robots." http://www.cs.cornell.edu/ asaxena/papers/robobrain2014.pdf, 2014.

[235] W. Magazine, "The plan to build a massive online brain for all the world's robots." http://www.wired.com/2014/08/robobrain/, 2014.

[236] G. F. Jr., "Codes on graphs: normal realizations," *IEEE Transactions on Information Theory*, vol. 47, pp. 520–548, February 2001.

[237] H.-A. Loeliger, J. Dauwels, J. Hu, S. Korl, L. Ping, and F. Kschischang, "The factor graph approach to model-based signal processing," *Proceedings of the IEEE*, vol. 95, pp. 1295–1322, June 2007.