UNIVERSITÀ DEGLI STUDI DI PADOVA

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria delle Telecomunicazioni

Tesi di Laurea

ANALYSIS AND DESIGN OF MASSIVELY PARALLEL CHANNEL ESTIMATION ALGORITHMS ON GRAPHIC CARDS

Relatore:
Prof. LORENZO VANGELISTA

Correlatore:
Dr. FRANCESCO ROSSETTO

Laureando:
DAVIDE DA LIO

ANNO ACCADEMICO 2010-2011

to Vije

Ringraziamenti

Prima di tutto e tutti voglio ringraziare mia madre, mio padre e mia sorella Elisa. Grazie per il costante supporto che mi avete dato e per tutti i sacrifici che avete dovuto fare per rendere tutto questo possibile. Sarò sempre in debito con voi. Un grazie speciale va agli zii Paolo e Cornelia e alla famiglia Klaric sapete bene che per me siete come una seconda famiglia.

Un ringraziamento particolare va al Prof. Lorenzo Vangelista per avermi dato l'opportunità di svolgere questo periodo di lavoro in un importante centro di ricerca, ma soprattutto per l'immensa disponibilità, l'entusiamo contagioso e il non aver mai dubitato di me in nessuna occasione.

I miei più sinceri ringraziamenti vanno al Dr. Francesco Rossetto per avermi assistito ed essere stato un vero punto di riferimento durante tutto il periodo di Tesi, nonchè un esempio di grande umanità e serietà.

Un sincero ringraziamento va a Giuliano, Tom, Balazs, Tomaso, Fran e tutti i ricercatori del gruppo di Digital Networks per avermi dapprima accolto con grande entusiasmo...e successivamente sopportato durante tutti questi mesi.

E come potrei dimenticarmi dei membri dello Student Office? Parlo del Re Carlos, della Regina Tudor, la Casalinga Stefan, e il Re uscente Firat. Convinto di aver trovato dei veri amici, vi ringrazio di tutto il divertimento e il tempo passato assieme. E a Giulio, per aver condiviso questa eperienza con me, compresi i momenti difficili, va un ringraziamento speciale e un grosso in bocca al lupo per il futuro.

Come non ringraziare poi tutti gli amici dell'Università di Padova, in modo particolare gli assidui frequentatori dell'aula Ge, i miei grandi amici Carlu e Salmi, il fenomenale trio Baccarizzo, Stefano Fiè, Stefano Busato, Francescon, Pippo, Giulio Ministeri, Fedo, Diego, e Michele. Per tutte le ore trascorse assieme negli ultimi due anni supportandoci a vicenda, grazie ragazzi.

Ed infine un grazie di cuore e un grosso bacio alla mia ragazza Verica, per ascoltarmi ed incoraggiarmi sempre, per essere la mia forza.

Grazie infinite a tutti voi.

Aprile, 2011
Davide Da Lio

Acknowledgments

First of all, I wish to thank my mother, my father and my sister Elisa. Thank you for the continued support you have given me, and for all the sacrifices you had to do to make this possible. I will always be indebted to you. My gratitude goes to my uncles Paolo and Cornelia and to Klaric family, you know you are like a second family for me.

Special thanks go to Prof. Lorenzo Vangelista for giving me the opportunity to carry out this work period in an important research center, but especially for the availability, the contagious enthusiasm and to have never doubted about me.

Many thanks are due to Dr. Francesco Rossetto for assisting me and being a true point of reference during the entire period of the Thesis as well as a great example of humanity and seriousness.

My sincere thanks go to Giuliano, Tom, Balazs, Tomaso, Fran, and all researchers of Digital Networks group, at first to have received me with great enthusiasm...and then to have endured me during all these months.

And how could I forget the members of the Student Office? The King Carlos, the Queen Tudor, the Housewife Stefan, and the outgoing King Firat. I am convinced to have found true friends. Thank you for all the fun and all the time spent together. I would like to thank and wish good luck to Giulio for having shared this experience with me, including the difficult moments.

Thanks to all my friends of the University of Padua, especially the regulars of the Ge classroom, my great friends Carlu and Salmi, the phenomenal trio Baccarizzo, Stefano Fiè, Stefano Busato, Francescon, Pippo, Giulio Ministeri, Fedo, Diego, and Michele. For all the hours spent together in the last two years supporting each other, thank you guys.

And finally heartfelt thanks and a big kiss go to my girlfriend Verica for always listening to me, encouraging me and for being my strength.

Many thanks to all of you.

April, 2011
Davide Da Lio

Sommario

La necessità di un'accurata stima di canale nei multiuser detector coerenti è ben nota. Essi infatti si basano sull'assunzione che i segnali ricevuti vengano perfettamente stimati, e questo in realtà non viene mai completamente ottenuto. Inoltre i trasmettitori e i ricevitori utilizzati praticamente sono afflitti da alcune non idealità come il rumore di fase, rendendo il problema della stima di canale ancora più impegnativo.

Un'altra questione ben nota è la considerevole complessità computazionale delle tecniche multi-utente. Per questa ragione in questo progetto è stata data particolare attenzione ad architetture per ricevitori altamente parallelizzate e la possibilità di parallelizzare gli algoritmi di stima di canale. Le schede grafiche CUDA prodotte da Nvidia bene si adattano ad affrontare problemi che possono essere espressi in calcolo parallelo. Quest'ultimo compito è molto impegnativo ed ambizioso, dal momento che l'utilizzo di tali schede per il progetto di ricevitori è ancora nella sua fase embrionale.

In questa tesi viene descritto il lavoro svolto durante uno stage presso l'Agenzia Spaziale Tedesca (DLR) dove la realizzazione di un ricevitore multiutente è attualmente studiata. Gli obiettivi finali fissati per il lavoro erano i seguenti: messa a punto e miglioramento dell'attuale algoritmo di stima di canale; esplorazione della metodologia basata su factor graph in modo da migliorare la qualità della stima e sviluppare algoritmi adatti ad essere parallelizzati; implementazione in parallelo degli algoritmi su scheda grafica CUDA.

Tutti gli obiettivi sono stati raggiunti. Due diversi miglioramenti per lo stimatore di fase attualmente implementato vengono proposti. Entrambi si basano sulla stessa approssimazione del modello di fase di Wiener-Levy e richiedono la stessa conoscenza statistica al ricevitore.

Adottando l'approcio a factor graph, presentiamo due algoritmi già esistenti in letteratura per la stima di fase in una nuova versione parallela e mostriamo come, allo stesso tempo, migliorano la qualità della stima e risultano idonei per essere implementati in parallelo sulla scheda.

I miglioramenti delle prestazioni in termini di Errore Quadratico Medio ottenuti per tutti gli algoritmi sono convalidati attraverso una serie di campagne simulative svolte per diversi scenari, molti dei quali caratterizzati da forte rumore di fase e bassi rapporti segnale-rumore.

Infine presentiamo gli algoritmi paralleli per la stima di fase che operano su scheda grafica CUDA e mostriamo come, in alcuni casi, siamo in presenza di una massiccia parallelizzazione nel quale viene raggiunto un incremento di velocità superiore alle 200 volte, rispetto alla versione seriale dello stesso algoritmo.

I risultati ottenuti rappresentano un punto di partenza per la realizzaione di un Ricevitore Parallelo Iterativo da inserire nel multiuser detector esistente e completamente eseguito su scheda grafica CUDA.

Abstract

The necessity of accurate channel estimation for coherent multiuser detectors is well known. Indeed they are based on the assumption that signals are perfectly estimated, and this is never completely achieved in practice. Furthermore, practical transmitters and receivers are affected by many non-idealities like strong phase noise, and thus the task of channel estimation is all the more challenging.

Another notorious issue is the high computational complexity of multiuser techniques. This project has devoted significant attention for massively parallel receiver architectures and the possibility to parallelize channel estimation algorithms. Nvidia CUDA graphic cards are especially well-suited to address problems that can be expressed as data parallel computations. This task is very challenging and ambitious, since the usage of such cards for receiver design is still at its infant stage.

This thesis describes the work carried out at German Aerospace Center (DLR) where a real-world multiuser detector is studied. The desired goals were the following: fine tuning of the already existing channel estimation algorithm; exploration of the factor graph approach in order to improve the estimation quality and to develop algorithms suitable to be parallelized; parallel implementation of the algorithms on CUDA graphic card.

All these points have been covered. Two different improvements for the already implemented phase estimator are proposed. Both are based on the same approximation of the Wiener-Levy phase model and assume the same knowledge at the receiver.

By adopting the factor graph approach, we present two existing algorithms for the phase estimation in a new parallel fashion and we show that, at the same time, they improve the estimation quality, and they are suitable to be parallelized on the board.

The performance improvement for all estimators proposed in terms of Mean Square Error are validated through several simulation campaigns carried out in different scenarios, most of them characterized by strong phase noise and low signal-to-noise ratios.

Finally we present several parallel phase estimation algorithms working on CUDA graphic card and we show that, in some cases, we are in presence of a massive parallelization in which is achieved a speedup more than 200 times compared to the serial implementation.

The results obtained represent a starting point for the implementation of a Parallel Iterative Receiver to be inserted in the existing multiuser detector and completely executed on CUDA graphic card.

Contents

Sommario **ix**

Abstract **xi**

1 Introduction **1**
1.1 The real scenario . . . 1
1.2 The already existing system . . . 2
1.3 The demonstration . . . 4
1.4 Outline of the thesis . . . 5

2 Channel model and Phase noise **7**
2.1 Passband systems . . . 7
2.2 Constant phase model . . . 11
2.3 Phase noise . . . 11
2.4 Wiener-Levy phase model . . . 14

3 Factor Graphs **17**
3.1 Factor Graphs . . . 17
3.2 The Sum Product Algorithm . . . 18
3.3 Factor Graphs for Statistical Inference . . . 21

4 Maximum Likelihood estimators **23**
4.1 System Model description . . . 23
4.2 The ML estimator . . . 24
4.3 The ML estimator improved . . . 30
4.4 The ML estimator with optimal window . . . 34
4.5 Performance evaluation and comparison . . . 38
4.6 Conclusions . . . 47

5 Factor Graphs estimators **49**
5.1 System Model description . . . 49
5.2 Factor Graph of the System . . . 50
5.3 A first simple model . . . 52
5.4 Factor graph of a Markovian channel . . . 54

5.5 Quantized Sum Product Algorithm . . . 58
5.5.1 Markov Chain update rule . . . 59
5.5.2 Quantized SPA for phase estimation . . . 61
5.6 Steepest Ascent algorithm . . . 62
5.6.1 Gradient algorithms . . . 63
5.6.2 A parallel implementation . . . 63
5.6.3 Steepest Ascent algorithm for phase estimation . . . 66
5.7 Performance comparison . . . 67
5.8 Conclusions . . . 71

6 An introduction to CUDA 73
6.1 GPUs as Massively Parallel processors . . . 73
6.2 Programming Model, basic concepts . . . 76

7 Parallel implementation on Graphic Card 81
7.1 Programming Guidelines and the Device . . . 81
7.2 Parallel implementation of the algorithms . . . 84
7.2.1 Parallel Maximum Likelihood estimators . . . 84
7.2.2 Parallel Quantized SPA . . . 88
7.2.3 Parallel Steepest Ascent algorithm . . . 90
7.3 Performance comparison . . . 91
7.4 Conclusions . . . 96

8 Conclusions and future work 97

Bibliography 101

Chapter 1

Introduction

This thesis represents a detailed description of the work carried out during an internship at German Aerospace Center (DLR) located near Munich, Germany. The purpose of this preliminary chapter is to explain the real scenario and to introduce the already existing system. After that we will motivate the necessity of an improvement in the already implemented channel estimation algorithm and the particular challenge of the parallel implementation on graphic cards. Finally the outline of the work and of the thesis will be given.

1.1 The real scenario

The German Aerospace Center (DLR) is involved in the development of a real-world *multiuser detector* (MUD). Many users are transmitting at the same time and in the same frequency through a satellite channel to a common receiver, which is imagined to be a ground station (Fig. 1.1).

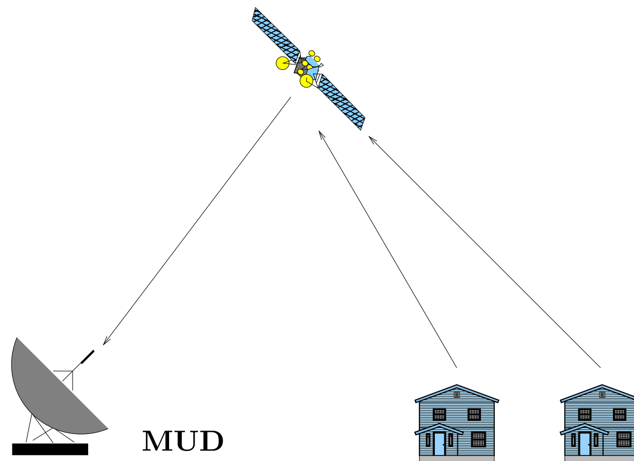

Figure 1.1: The scenario.

The multiuser detector studied at DLR is based on coherent modulation and therefore requires precise channel estimation at the receiver. One of the main performance limitations for this architecture is the error propagation due to imperfect channel estimation. The entire concept of coherent multiuser detection is based on the premise that the received signal can be reliably estimated. While communication systems are by definition designed to allow the transmitted signal to be recovered, reconstructing the user signals in multiuser detectors requires an accurate description of:

- what was transmitted;
- what the channel did to that transmission.

Inaccurate channel estimation is a problem for coherent MUD since it is based on the assumption that signals are perfectly estimated, and this is never completely achieved in practice. This error propagation causes later users to have unacceptably bad performance, causing a major fairness problem as well as on overall degradation in bit error rate and system capacity. The importance of channel estimation under these conditions can not be downplayed and algorithms that enable to achieve accurate channel estimation are of paramount importance. Different methods have been proposed for addressing this problem. One of the most appealing technique is the one that concern *Iterative Channel Estimation*, as we will see soon, this is the architecture choice for the already existing multiuser detector.

It is also important to remark that the transmitters and the receiver will be affected by many non-idealities like strong *phase noise*, and thus the task of channel estimation is all the more challenging. Indeed during the early years of communication this impairment was of secondary concern. Since communication systems were used to operate far from the performance bound, i.e., the channel capacity, the main goal was to develop efficient error-correcting techniques in order to close the gap between the performance of practical communication systems and channel capacity. Recent iterative decoding techniques lead the communication systems to operate close to the ultimate performance limits, so issues such as synchronization have now become one of the major limiting factors in the design of communication systems. In this thesis, we are focused on *phase estimation*, i.e., the alignment of the phase of the local oscillator at the receiver to the phase of the incoming carrier.

1.2 The already existing system

A simulator of the system introduced already exists. Such system is composed by U users that transmit simultaneously and in the same frequency to the common receiver. All involved nodes are equipped with a single antenna and spreading is not used.

For the specific system model adopted the transmission channel is a *frequency flat Additive White Gaussian Noise* (AWGN) channel which adds complex zero mean circularly symmetric White Gaussian Noise (WGN) with variance per dimension σ_ω^2. Its impulse response takes the following form:

$$h(t) = |h(t)|e^{j\theta(t)}$$

The channel is not assumed to be slowly time varying, this means that it may change from symbol to symbol. This variability of the channel coefficient is induced at least by the phase noise due to the oscillator instabilities. The purpose of Chapter 2 is to describe the channel models adopted in this thesis, however we anticipate that such type of channel shows up, for example, with mobile users due to Doppler effect, or when phase noise is not negligible. The impact of phase noise is particularly relevant for applications at high carrier frequency (say Ku or Ka band, typical in modern satellite environments), where high stability oscillators can be very expensive; furthermore cheap, consumer-grade terminals are significantly affected by such problem.

At the receiver side the multiuser detector performs a matched filtering and sampling, obtaining a sequence of received samples y_k, k denotes the discrete temporal index.

$$y_k = \sum_{u=1}^{U} h_k^u x_k^u + \omega_k \tag{1.1}$$

h_k^u is the frequency flat complex channel at time k for the user u, x_k^u is the modulated symbol transmitted at time k by user u, and ω_k is the complex WGN.

After that it performs a *Multistage MUD* [5] in which each stage is provided with *Iterative Channel Estimation and Decoding blocks* [6]. In order to fight the channel variability, an iterative channel estimation/channel decoding algorithm is adopted. The outputs of the block are the bit and channel estimates.

Since this thesis is focused on the phase estimation we report that, in the existing system, it is performed iteratively by adopting the following update rule, j denotes the iteration index:

$$\hat{\theta}_k^{j+1} = \arg\left\{ \sum_{i=k-W}^{k+W} \hat{x}_i^{*j} y_i \right\} \tag{1.2}$$

As we can see, since the channel is time variant, the channel estimation is performed over a sliding window of size $N = 2W + 1$ samples around the desired time index k. The window parameter W depends on the coherence time of the channel [1].

Eq. (1.2) corresponds to the *Maximum Likelihood* (ML) estimator for the phase process and its performance in terms of estimation quality depends on the window size N. For this reason the first goal of the proposed work was to study and implement advanced phase estimation methods for the existing system.

1.3 The demonstration

The multiuser detector will be in the end demonstrated by means of a demo. The scenario will be relatively simple: two users will transmit simultaneously and in the same frequency over a satellite channel. The channel is emulated by a *Channel Emulator* and the common receiver is implemented on a *cluster server*. The senders will employ conventional, non-spreaded modulation.

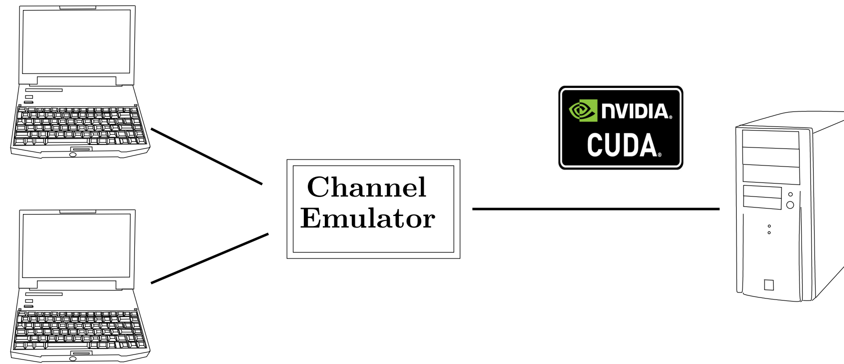

Figure 1.2: The demo.

Fig. 1.2 shows the system setup and also reveals an important issue. Indeed the cluster that will implement the multiuser detector is provided with *Graphic Processor Units* (GPU) explicitly developed for scientific programming. These boards are developed by Nvidia and their particular architecture is called *CUDA* [16], [17], [18].

In recent years, driven by the insatiable multimedia market demand for realtime, high-definition and 3D graphics, the GPU has evolved into highly parallel, multithreaded, *many-core* processor with high computational power and bandwidth. The reason behind this is the difference in the floating-point capability between the canonical CPU and the GPU. The latter is specialized for computer-intensive, highly parallel computation for graphic rendering and therefore it is designed such that more chip area is devoted to data processing rather then data caching and flow control.

Many algorithms outside the field of image processing and video rendering can be accelerated by data parallel processing. Thus actual GPUs are especially well-suited to address problems that can be expressed as data parallel computations. The reason is that when an arithmetic intensity, i.e., the ratio of arithmetic operations to memory operations, is high, there is a lower requirement for sophisticated flow control. CUDA refers to a general purpose parallel computing architecture with a new parallel programming model and instruction set introduced by Nvidia. Hence another important element that sets this project apart is the attention for massively parallel computer architectures and the possibility to parallelize the algorithms. An important and desired goal is to design such channel estimation methods so as to exploit a massively parallel architecture.

In this regard the *factor graph approach* [11] applied to the phase estimation has been investigated. Indeed a possible change for the architecture described previously is to place an *Iterative Receiver* [14] in the Multistage MUD that performs jointly channel estimation and channel decoding. The topic of phase estimation on factor graph has been widely and exhaustively covered in literature [8] and our purpose is to revisited the already existing factor graph algorithms in a new parallel fashion in order to obtain an efficient parallel implementation on the GPU.

1.4 Outline of the thesis

The outline of the work can be split in three major parts:

1. Fine tuning and performance improvement of the already existing phase estimator.
2. Exploration of the factor graph approach in order to find algorithms that, at the same time, improve the estimation quality and are suitable to be parallelized on GPU.
3. Implementation of such algorithms on the graphic card in order to have such systems that exploit the computational power as efficiently as possible.

Finally this thesis has been structured as follows:

- Chapter 1 gives an introduction of the work and the thesis outline.
- Chapter 2 regards the channel models adopted and the phase noise phenomenon.
- Chapter 3 represents a brief introduction on factor graph.
- Chapter 4 concerns the first goal of the outline above. Validated by experimental results two improvements for the existing estimator will be given.
- Chapter 5, based on the work in [8], reports an investigation on factor graph estimators revisited in order to obtain a good parallel implementation.
- Chapter 6 introduces the CUDA programming model and its basic concepts.
- Chapter 7 reports some details on the parallel implementation of the algorithms on the graphic card.
- Chapter 8 gathers conclusions and future work.

Chapter 2

Channel model and Phase noise

The aim of this chapter is to describe the two channel models considered in this thesis. They are a simple description of a single-carrier passband communication system that takes into account the phase offsets between the received carrier and the local carrier of the receiver, but disregards timing offsets. In particular we will encounter the well known phase noise phenomenon produced by random fluctuations in local oscillators and clocks.

This chapter is organized as follows: in Section 2.1 we review the basic notion of passband systems and in Section 2.2 a first simple stochastic model in which the phase offset is constant will be presented. After that, in Section 2.3 we present through some examples how phase noise degrades the performance of a communication system and in Section 2.4 a more defined stochastic channel model. In this chapter we closely follow the description reported in [8].

2.1 Passband systems

The two major categories of analog waveforms are baseband and passband signals and they lead to different architectures for the transmitter and the receiver. Baseband signals are pulse trains in which the information is encoded in the amplitude of the pulses. They are used in applications such Integrated Services Digital Networks (ISDN), Local Area Networks (LANs) and digital magnetic recording systems. In passband communication systems, a baseband signal is modulated onto a sinusoidal carrier such that the resulting waveform fits into the frequency range available for the transmission [8]. In radio, wireless and satellite communication systems information is transmitted by means of passband signals, for this reason we will focus on such systems. In the following outline we will assume that the noisy medium solely adds white noise $\omega(t)$ to the transmitted signal $s(t)$. The received signal is then given by

$$r(t) = s(t - \tau_C) + \omega(t) \tag{2.1}$$

where τ_C is the delay introduced by the channel.

The basic block diagram of a passband communication system is depicted in Fig. 2.1.

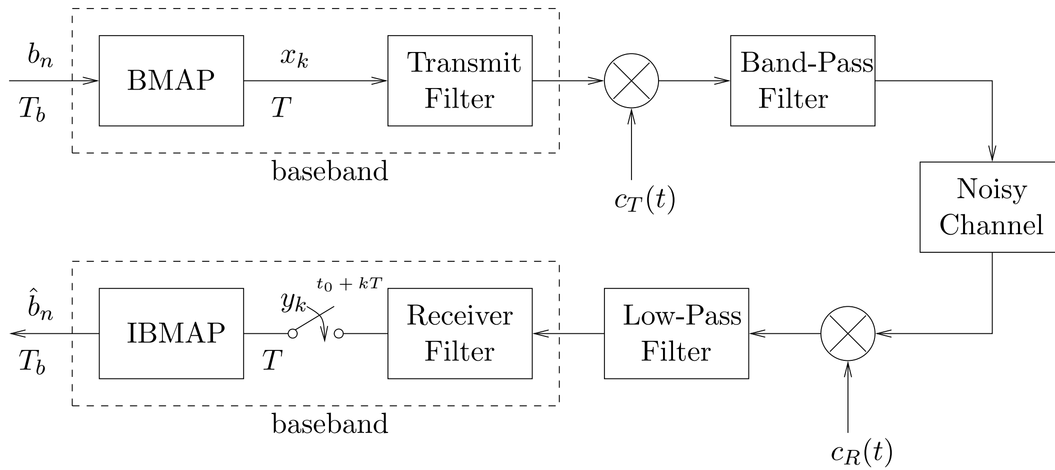

Figure 2.1: Passband communication system.

At the transmission side the *Bit mapper* (BMAP) takes a series of encoded bits $\{b_n\}$ and maps it to a channel symbols x_k. This operation can be made in different ways. The most common mapping is the *Gray encoding* where adjacent (M-ary) signal amplitudes differs by one binary digit. The two most widespread used modulation schemes in wireless communications are Quadrature Amplitude Modulation (QAM) and Phase Shift Keying (PSK). In QAM the symbol x_k takes the form

$$x_k = a_k + jb_k \tag{2.2}$$

where a_k and b_k belong to $\{\pm 1, \pm 3, \ldots, \pm(M-1)\}$. On the other hand in PSK the symbol takes the form

$$x_k = e^{j\alpha_k} \tag{2.3}$$

with $\alpha_k \in \{0, 2\pi/M, \ldots, 2\pi(M-1)/M\}$. In both cases M denotes the modulation order. Fig. 2.2 shows an example of constellations for such schemes.

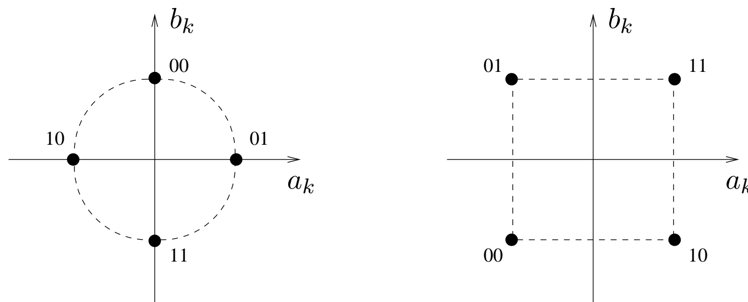

Figure 2.2: 4-PSK and 4-QAM modulation.

The information symbol x_k is then filtered by a *Transmit Filter* (a.k.a. Pulse shaping filter) with impulse response $g_T(t)$ obtaining a baseband signal $s_{BB}(t)$

$$s_{BB}(t) = \sum_k x_k g_T(t - kT) \tag{2.4}$$

where T denotes the symbol period.

After that it is modulated onto a sinusoid denoted by $c_T(t)$ with frequency f_T

$$c_T(t) = e^{j2\pi f_T t} \tag{2.5}$$

amounting the passband signal

$$s_{PB}(t) = Re\left\{s_{BB}(t)e^{j2\pi f_T t}\right\} \tag{2.6}$$

Before the signal $s_{PB}(t)$ is transmitted over the *Noisy Channel* it is fed into a *Band-Pass filter.*

At the receiver, the received signal is first processed by a pre-filter (not shown in Fig. 2.1) which purpose is to eliminate the out of band noise, and then down-converted to a baseband signal by using a local carrier $c_R(t)$ and a *Low-pass filter.* The output of this filter is then processed by a baseband receiver. We must now take a closer look on the signal processing performed in a passband receiver.

The output signal of the low-pass filter illustrated can be represented as the complex signal

$$y(t) = e^{j(2\pi\nu t+\theta)} s_{BB}(t) + \omega(t) \tag{2.7}$$

where

- $\omega(t)$ denotes a wide band low pass noise whose bandwidth is wider of the bandwidth of the signal $s_{BB}(t)$.
- $\nu = f_R - f_L$ is referred to as the carrier frequency offset. f_R is the frequency of the incoming carrier and due to the *Doppler effect* and the clock instabilities it may differ from f_T the frequency of the transmitted carrier [1].
- $\theta = \theta_L - \theta_R - 2\pi f_R \tau$ is a phase offset where θ_L and θ_R is the phase of the local carrier and the incoming carrier and τ is the timing offset.
- We have assumed that the low-pass filter has a unity frequency response for the low pass signal components.

The received signal is processed by the *Receiver filter* (a.k.a. *Matched filter*) whose impulse response is $g_R(t) = g_T(\tau_R - t)$ where τ_R is chosen such that $g_T(\tau_R - t)$ is a causal function. The last assumptions are made in order to allow the convolution $h(t) = (g_T * g_R)(t - \tau_R)$ to satisfy the Nyquist criterion [1], i.e.,

$$h(kT) = \begin{cases} 1 & \text{for } kT = 0 \\ 0 & \text{for } kT \neq 0 \end{cases}$$

The output of the received filter is given by

$$\tilde{y}(t) = (y * g_R)(t) = \int_0^{\infty} y(t-u) g_R(u) du + \tilde{\omega}(t) \tag{2.8}$$

where $\tilde{\omega}(t) = (n * g_R)(t)$. The following approximation holds:

$$\begin{aligned} \tilde{y}(t) &= \int_0^{\infty} \left[e^{j(2\pi\nu(t-u)+\theta)} s_{BB}(t-u) \right] g_R(u) du + \tilde{\omega}(t) \\ &= \sum_l x_l \int_0^{\infty} e^{j(2\pi\nu(t-u)+\theta)} g_T(t-u-lT-\tau_C) g_T(\tau_R - u) du + \tilde{\omega}(t) \\ &\approx e^{j(2\pi\nu t+\theta)} \sum_l x_l h(t-lT-\tau_R-\tau_C) + \tilde{\omega}(t) \end{aligned} \tag{2.9}$$

as long as the phase $\theta(t) = 2\pi\nu t + \theta$ varies only little over a time interval equal to the symbol period T.

Finally sampling $\tilde{y}(t)$ at the instants $t = t_0 + kT$ yields to a received sampled signal

$$y_k = e^{j(2\pi\nu(t_0+kT)+\theta)} \sum_l x_l h(\tau + (k-l)T) + \omega_k \tag{2.10}$$

where $\omega_k = \tilde{\omega}(t_0 + kT)$, $y_k = y(t_0 + kT)$ and $\tau = t_0 - \tau_R - \tau_C$ is the timing offset. If

$$t_0 = t_{ideal} = \tau_R + \tau_C \tag{2.11}$$

the timing offset is equal to 0 and, as a consequence of the Nyquist criterion, the expression can be simplified to

$$y_k = e^{j(2\pi\nu(t_{ideal}+kT)+\theta)} x_k + \omega_k \tag{2.12}$$

As we can see the symbol y_k depends only on the channel symbol x_k; otherwise it is in principle affected by all channel symbols. This effect is called "inter-symbol interference" (ISI) [1]. In many practical receivers a timing synchronization algorithm (see Fig. 2.3) tries to align the sampler to the incoming signal in order to avoid as possible ISI. However small deviations between t_0 and t_{ideal} are unavoidable. An alternative approach is not to adjust t_0 at all, but to use a free running sample clock instead. The timing offset τ is then estimated from the received samples, and the ISI is compensated by digital signal processing, for instance, using linear equalizer (LE) or differential feedback equalizer (DFE) [1].

In contrast with timing offsets, the carrier offsets, do not lead to inter-symbol interference as long as the phase offset remains small. Most of the passband receiver are equipped with algorithms to track the carrier offsets and the timing offsets, the following picture shows one possible architecture.

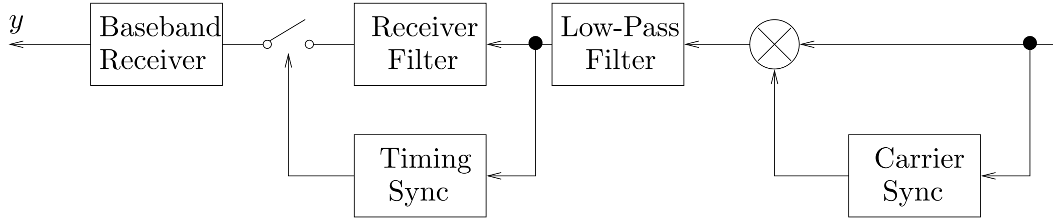

Figure 2.3: Timing and carrier synchronization in a passband receiver.

2.2 Constant phase model

We now introduce a first simple stochastic channel model for a single carrier passband communication system. The model is called *Constant phase model* and is defined as follows.

Def. *Let y_k be the received symbol and x_k the transmitted symbol at time index k, the Constant-phase model is*

$$y_k = x_k e^{j\theta} + \omega_k \tag{2.13}$$

where $\theta \in [-\pi, \pi)$ denotes a random phase and ω_k is a zero-mean complex Gaussian random variable with variance per component σ_ω^2, i.e $\omega_k \sim \mathcal{CN}(0, 2\sigma_\omega^2)$.

Since the model above is a stochastic model, all the variables involved are random variables.

The conditions under which this channel model is valid are the following.

a) A timing synchronization algorithm as depicted in Fig. 2.3 is adopted to track the timing offsets. This means that Eq. (2.12) represents a good description of the received symbol y_k.

b) The frequency offset $\nu = 0$.

c) The phase offset $\theta = \theta_L - \theta_R - 2\pi f_R \tau$ is constant.

As we will see, the Constant phase model can be a strong assumption in some communications systems. Indeed in this model it is assumed that the phase offset is constant, but this is typically not met. Often the phase offset undergoes random fluctuations due to different noisy sources and there exist more refined models that try to take this into account. Before presenting a more refined channel model we take a closer look to the reason of these fluctuations, the phase noise phenomenon.

2.3 Phase noise

Oscillators used in practical systems deviate from ideal spectral characteristics. Indeed the signals generated are not perfectly periodic. Due to this, unpleasant performance issues arise.

A/D and D/A converters used in digital receivers use sampling clocks (for sampling or sample and hold) which are derived from sinusoidal oscillators. Practical clocks are affected by phase and frequency instabilities called *phase noise*. In order to understand how phase noise degrades the performance of communications systems, we illustrate some examples.

The transition of an oscillator used as time reference can be affected by *timing jitter*. Indeed the spacing between those transitions is ideally constant but in practice they will be variable due to phase noise. This has a harmful effect on the sampling process as depicted in Fig. 2.4: the uncertainty in the sampling times translates directly to uncertainty in the sampled value.

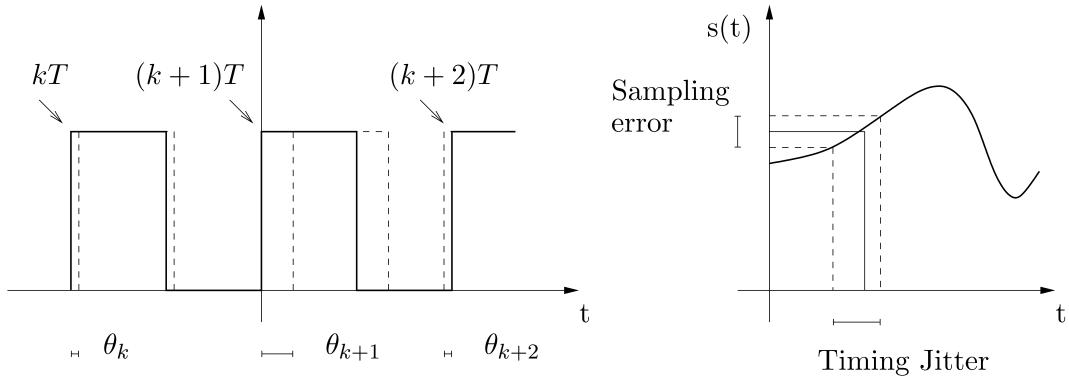

Figure 2.4: Timing jitter and sampling error due to timing jitter.

Phase noise also has a deteriorating influence on the down conversion in a pass-band receiver. The demodulators used in pass-band receivers, like the one that we have illustrated in the previous section, are classified as *coherent* because they use a carrier signal, which ideally should have the same frequency as the carrier at the transmitter, to demodulate the received signal. Ideal sinusoidal oscillators of frequency f_0 have a mathematically strict spectrum in the form of delta functions, centered at $-f_0$ and f_0. However, real oscillators seldom exhibit this kind of clean spectrum. They tend to have spreading of spectral energy around the carrier frequency. Due to this non ideality, background signals in the frequency band adjacent to the incoming data signal are down converted and interfere with the desired base-band signal. This phenomenon is known as *interchannel interference* and is shown in Fig. 2.5.

Finally from a communication system performance point of view, significant phase noise results in an irreducible error floor in bit error rate (BER) at high signal to noise ratio (SNR) [10] as illustrated in Fig. 2.6.

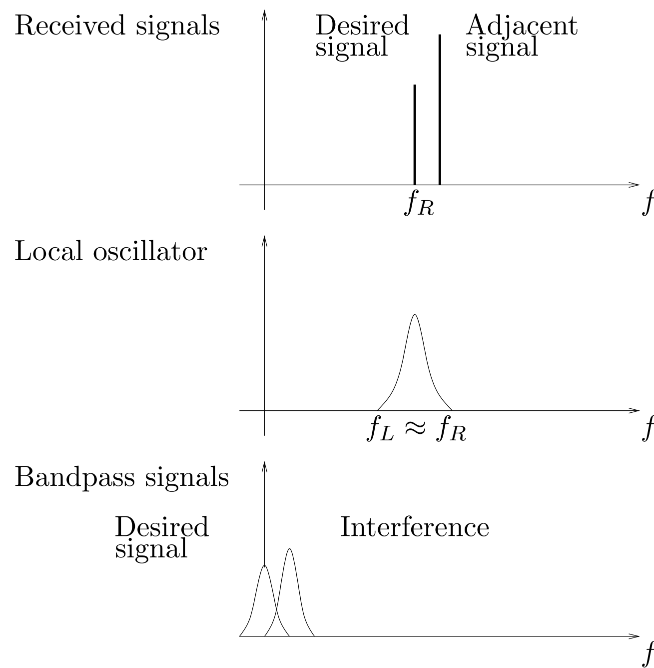

Figure 2.5: Interchannel interference.

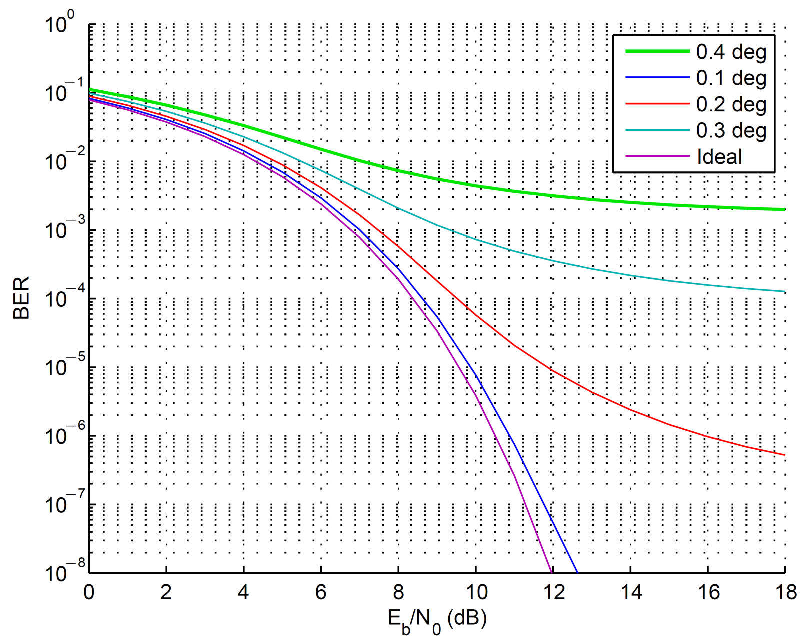

Figure 2.6: Phase noise causes a floor on the BER: uncoded BPSK for different levels of phase noise standard deviation in degrees [10].

Oscillators are typically perturbed by short-term and long-term instabilities. Long-term instabilities, also known as drifts or trends, may be due to aging of the resonator material. These usually very slow changes are much less critical than short-term instabilities, caused by noise sources such *thermal*, *shot*, and *flicker noise* in electronic components. The signal out from an oscillator may also interfere with other signals from the same electronic device. This kind of interference can often be avoided by carefully system design.

Since oscillators are ubiquitous in communication systems, phase noise has been studied intensively in the past. Nevertheless, a generally accepted model for phase noise does not seem to exist. Until now we have discussed the influence of this phenomenon in order to understand why a phase noise compensation is required. If one is interested in modeling the impact of phase noise in passband systems it is also of crucial importance to have a good understanding of the nature of the random fluctuations of phase in an oscillator. For an excellent description of the nature of the noise sources and on the way they lead to phase noise we refer to [8], for our purposes it is sufficient to have a description of the following stochastic model used in literature, the Wiener-Levy phase model [8].

2.4 Wiener-Levy phase model

The *Wiener-Levy phase model*, or *Random-walk phase model*, is a widely used stochastic channel model for phase noise in free running clocks. It is valid if the oscillator only contains white noise sources such as shot noise and thermal noise while other sources, like flicker noise, are not considered. It can be derived by extending the model (2.13) with random phase fluctuations.

Def. *Let y_k be the received symbol and x_k the transmitted symbol at time index k, the Wiener-Levy phase model is*

$$y_k = x_k e^{j\theta_k} + \omega_k \tag{2.14}$$

where

$$\theta_k = \theta_{k-1} + \Delta_k \tag{2.15}$$

denote a random phase and Δ_k is a zero-mean Gaussian random variable with variance σ_p^2, i.e., $\Delta_k \sim \mathcal{N}(0, \sigma_p^2)$. ω_k is a zero-mean complex Gaussian random variable with variance per component σ_ω^2.

In words: the phase process undergoes a Gaussian random walk process. The variable Δ_k is the *stepsize* of the walk and its variance sets the speed of the process. Fig. 2.7 shows a realization of the phase process.

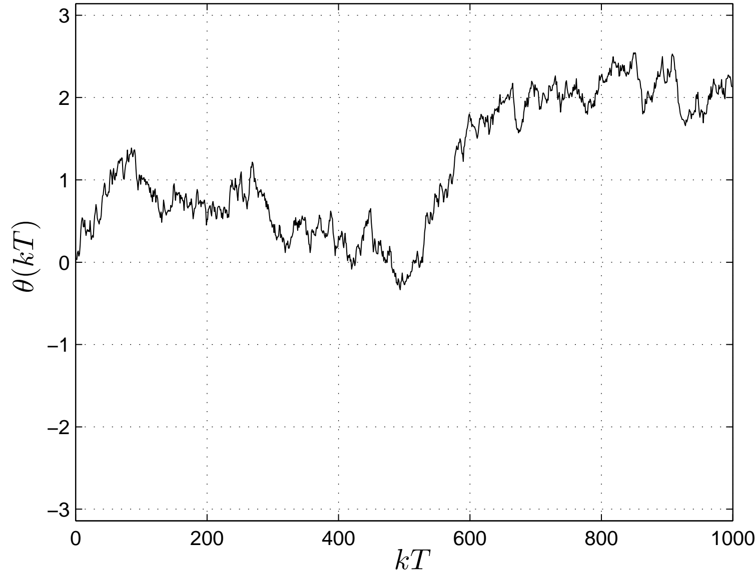

Figure 2.7: A realization of the phase process with phase noise variance $\sigma_p^2 = 5(^\circ)^2$.

A key property of such process is that its variance grows linearly with the time index [8], i.e.,

$$E[\theta_k^2] = k\sigma_p^2 \tag{2.16}$$

As we will see in Chapter 4, this property will be exploited in the development of the algorithms.

In summary the model describes the received symbols y_k accurately under the following assumptions.

a) A timing synchronization algorithm is adopted to track the timing offsets.

b) The frequency offset $\nu = 0$.

c) The fluctuations of the phase offset $\theta = \theta_L - \theta_R - 2\pi f_R \tau$ can be described as a random-walk process.

With respect to the last assumption, it is met, for example, if the oscillator at the transmitter and at the receiver are free-running clocks perturbed by white noise sources, the frequency f_R is approximately constant, and the relative distance between the transmitter and receiver is constant.

Chapter 3

Factor Graphs

This chapter seeks to give an introduction from a general point of view to *Factor Graphs* and the *Sum Product Algorithm* (SPA). A large variety of algorithms in coding, signal processing, and artificial intelligence may be viewed as instances of the SPA which operates by message passing on this graphical model. This elegant approach has been widely treated in literature after its introduction in 2001 by Kschinchang *et. al.* in their article [11]. To give a complete description of this graphical models is out of the scopes of this thesis. We only give some basic concepts and definitions. For further details about this topic we remind [11], [12], [13], [14]. In this chapter we will indicate the domain of the variables with capital letters and the variables with small letters.

3.1 Factor Graphs

A common situation in digital communications is to consider algorithms that deal with complicated global functions of many variables. By exploiting the way in which the global function factors into a product of simpler *local* functions, each of which depends on a subset of the variables, this algorithms can reach more computational efficiency. With factor graph approach this particularity is highlighted first by depicting the global function as a graph and then by developing an algorithm which works as a message passing on it. Let us dive right in and introduce the concept of factor graphs.

Let f be a real value function of many variables

$$f : X_1 \times X_2 \times \dots X_n \to \mathbb{R} \tag{3.1}$$

Suppose that f factors into a product of several local functions, each having some subset of $\{X_1 \times X_2 \times \dots X_n\}$ as argument, for instance suppose that it can be factorized in K factors such that

$$f(x_1, x_2, \dots, x_n) = \prod_{i=1}^{K} f_k(S_k) \tag{3.2}$$

where $S_k \subseteq \{X_1 \times X_2 \times \dots X_n\}$ is the k-th variable subset, and $f_k(\cdot)$ is a real-valued function.

Def. *A factor graph for f is a bipartite graph that expresses the structure of the factorization (3.2). A factor graph has a variable node for each variable x_i, a factor node for each local function $f_k(\cdot)$, and an edge-connecting variable node x_i to factor node $f_k(\cdot)$ if and only if x_i is an argument of $f_k(\cdot)$.*

Factor graphs are thus a standard bipartite graphical representation of a mathematical relation, in this case, the "is an argument of" relation between variables and local functions. For example consider the function

$$f(x_1, x_2, x_3, x_4) = f_A(x_1) f_B(x_1, x_2) f_C(x_1, x_3, x_4) \tag{3.3}$$

The corresponding factor graph results as in Fig. 3.1.

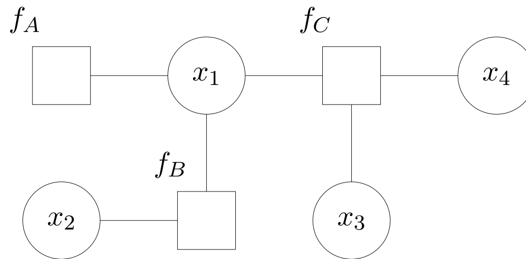

Figure 3.1: A factor graph representation of the function in (3.3).

3.2 The Sum Product Algorithm

In many situations, for instance when f represents a joint probability density function, we are interested in computing the marginal functions $g_i(x_i)$ with respect the variables involved. In general a marginal can be written as

$$g_i(x_i) = \int_{\sim\{x_i\}} f(x_1, x_2, \dots, x_n) \tag{3.4}$$

where the notation $\sim \{x_i\}$ refers to all the variables except x_i and the symbol $\int$ indicates the *summary operation.* For our purposes, depending on the nature of the variable, it denotes an integration over the whole range of a continuous variable or a summation for each element of the alphabet of a discrete variable.

The direct computation of a marginals could be cumbersome. The marginals of a function can be determined in a computationally elegant way by messages passing over the edges of the corresponding factor graph. This must be done according to the *Sum product rule* defined as follows.

Def. ***Sum product rule.*** *The message sent from a generic node v on a edge e is the product of the local function at v with all the messages received on edges other than e, summarized for the variable associated with e. If v is a variable node the local function corresponds to the unit function.*

For instance, let $\mu_{x_i \to f_i}(x_i)$ be the message sent from the variable node x_i to the factor f_i, and $\mu_{f_i \to x_i}(x_i)$ be the message sent from node f_i to node x_i. Also, let $n(v)$ denote the set of neighbors of the node v in a factor graph. The sum product rule leads to express this messages as follows:

$$\mu_{x_i \to f_i}(x_i) = \prod_{j \in n(x_i) \setminus \{f_i\}} \mu_{j \to x_i}(x_i) \tag{3.5}$$

$$\mu_{f_i \to x_i}(x_i) = \int_{\sim\{x_i\}} \left(f(S_i) \prod_{j \in n(f_i) \setminus \{x_i\}} \mu_{j \to f_i}(j) \right) \tag{3.6}$$

By adopting systematically this rule working on a factor graph we obtain the *Sum Product Algorithm* whose purpose is to compute the marginal functions for all the variables. The algorithm is reported in pseudo-code in the following [14].

The Sum Product Algorithm

- **Initialization**
  $\forall$ factor node f_i connected to a single variable node x_i transmit the message

  $$\mu_{f_i \to x_i}(x_i) = f_i(x_i)$$

  $\forall$ variable node x_i connected to a single factor node f_i transmit the message

  $$\mu_{x_i \to f_i}(x_i) = 1$$

- **Repeat**
  - select a node connected to D edges which has received incoming messages on at least $D - 1$ edges.
  - compute the outgoing message on the remaining edge by adopting the Sum product rule as in Eq. (3.5) and Eq. (3.6).
- **until** all messages are computed.
- **for** each variable nodes X_i compute the marginal by

  $$g_i(x_i) = \prod_{h \in n(x_i)} \mu_{h \to x_i}(x_i)$$

- **end for**

For example consider again the function in Eq. (3.3) represented as factor graph in Fig. 3.1 and assume that all the variables are continuous. The Sum product algorithm proceeds as follows:

1. Start from the nodes f_A, x_2, x_3, and x_4

$$\mu_{f_A \to x_1}(x_1) = f_A(x_1)$$

$$\mu_{x_2 \to f_B}(x_2) = 1$$

$$\mu_{x_3 \to f_B}(x_3) = 1$$

$$\mu_{x_4 \to f_B}(x_4) = 1$$

2. Messages sent from f_B and f_C to x_1

$$\mu_{f_B \to x_1}(x_1) = \int_{x_2} f_B(x_1, x_2)\mu_{x_2 \to f_B}(x_2)dx_2$$

$$\mu_{f_C \to x_1}(x_1) = \int_{x_2}\int_{x_3} f_C(x_1, x_3, x_4)\mu_{x_3 \to f_B}(x_3)\mu_{x_4 \to f_B}(x_4)dx_3dx_4$$

3. Node x_1 has received the messages from all its neighbors and it is ready to send back informations

$$\mu_{x_1 \to f_A}(x_1) = \mu_{f_B \to x_1}(x_1)\mu_{f_C \to x_1}(x_1)$$

$$\mu_{x_1 \to f_B}(x_1) = \mu_{f_A \to x_1}(x_1)\mu_{f_C \to x_1}(x_1)$$

$$\mu_{x_1 \to f_C}(x_1) = \mu_{f_A \to x_1}(x_1)\mu_{f_B \to x_1}(x_1)$$

4. Messages are transmitted from f_B and f_C to them extern neighbors

$$\mu_{f_B \to x_2}(x_2) = \int_{x_1} f_B(x_1, x_2)\mu_{x_1 \to f_B}(x_1)dx_1$$

$$\mu_{f_C \to x_3}(x_3) = \int_{x_1}\int_{x_4} f_C(x_1, x_3, x_4)\mu_{x_1 \to f_B}(x_1)\mu_{x_4 \to f_B}(x_4)dx_1dx_4$$

$$\mu_{f_C \to x_4}(x_4) = \int_{x_1}\int_{x_3} f_C(x_1, x_3, x_4)\mu_{x_1 \to f_B}(x_1)\mu_{x_3 \to f_B}(x_3)dx_1dx_3$$

5. The marginals of all the variables can be computed by the product of all the incoming messages

$$g_1(x_1) = \mu_{f_A \to x_1}(x_1)\mu_{f_B \to x_1}(x_1)\mu_{f_C \to x_1}(x_1)$$

$$g_2(x_2) = \mu_{f_B \to x_2}(x_2)$$

$$g_3(x_3) = \mu_{f_C \to x_3}(x_3)$$

$$g_4(x_4) = \mu_{f_C \to x_4}(x_4)$$

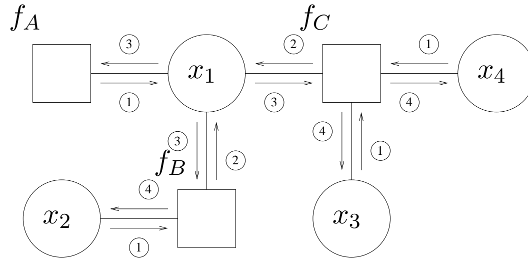

Figure 3.2: Sum Product Algorithm, message scheduling.

The particular message scheduling for the algorithm is shown in Fig. 3.2. Before continuing our discussion we make some observations.

- It can be shown that the SPA computes the exact marginals when the factor graph has *no cycles*, i.e., when the graph is a *tree*. When the graph has cycles the SPA has no natural initialization and termination. In this case the initialization can be forced by introducing artificial messages, while termination is achieved by simply aborting the SPA after some time. The resulting marginals are not exact but approximated.
- At Step 3) of the algorithm the node x_1 literally broadcasts its information to the neighbors. This situation is referred when the factor graph is a tree and x_1 is considered as the *root*. The messages, which have started the message passing from the leafs of the tree, begin to come back from the root to the leafs.

3.3 Factor Graphs for Statistical Inference

In our specific case we are interested in factor graphs for the phase estimation problem. Indeed the factor graph approach can be used to solve estimation problems and more generally, inference problems. In this context they are important first of all because they allow to reformulate several important algorithms with the same notation and terminology. Algorithms such as the *Viterbi Algorithm*, the *Kalman Filtering* and the *Forward-Backward Algorithm* can all be cast in the factor graph framework in a natural way [11], [12]. Furthermore they lead us to deliver new, optimal (sub-optimal) inference algorithms in a straightforward way. Applying the factor graph approach relies solely on local computations in basic blocks. For the details about how the phase estimation problem can be solved with the factor graph approach we refer to Chapter 5.

In general, in statistical inference we are interested in obtaining information regarding certain variable $\mathbf{x}$ on the basis of an observation $\mathbf{y}$. Factor graphs can help to solve the following common problems in communications:

- find the likelihood function $p(\mathbf{y}|M)$ of the channel model M;

- find the A Posteriori Distribution $p(\mathbf{x}|\mathbf{y}, M)$ of $\mathbf{x}$ given the observation $\mathbf{y}$ and the model M;
- find the Maximum A Posteriori (MAP) estimation of $\mathbf{x}$,

$$\hat{\mathbf{x}}^{MAP} = \arg\max_{\mathbf{x}} p(\mathbf{x}|\mathbf{y}, M)$$

  given the observation $\mathbf{y}$ and the model M.

To solve the first two situations the general idea is to create a factor graph of a factorization of the joint probability distribution and to implement the SPA on this graph. The third problem is solved by adopting the *Max Product Algorithm* which is an alternative to the SPA [11]. Otherwise one can think to develop another algorithm which works as message passing in a graph but is not an instance of the SPA. In these cases the factor graph approach may be useful to highlight particular nature of the problems and details (see Steepest ascent algorithm in Chapter 5).

Messages and their representation

Finally we give some considerations regarding the message representation. For discrete variables the messages can be represented by vectors of three types: probability mass functions, log-likelihoods, and log-likelihood ratios. For continuous variables, it is required to adopt an approximations method since the SPA can lead to intractable integrals over continuous ranges. Approximation methods can be quantization, parametric representations or particle representations [14]. As we will see in Chapter 5 different approximations lead to different algorithms.

We conclude the chapter by recalling that the SPA guarantees the exact marginals only for cycle-free factor graphs but, as expected, the most of the cases in communications are applications on cyclic factor graphs. This is naturally translated in iterative algorithms and it is in general accepted that the marginals obtained at convergence (the beliefs) are approximations of the true marginal a posteriori distributions. Empirical results support this claim.

Chapter 4

Maximum Likelihood estimators

After presenting the already existing system and the channel models chosen for this work, with this chapter we start the development of the phase estimation algorithms. Every oscillator used in passband communication systems suffers from an instability of their phase. This phenomenon is known as phase noise and, if left unaddressed, can lead to great degradation of the system performance. By using the popular statistical methodology known as *Maximum Likelihood* (ML) estimation [2] we will investigate and compare different estimators for a channel affected by phase noise.

The organization of the chapter is the following: first we briefly review the system model. Finally, depending on the statistical description adopted for the channel, we will obtain different ML estimators and their performance will be compared in different scenarios.

4.1 System Model description

Fig. 4.1 shows the general block diagram of the system under consideration in its baseband equivalent representation.

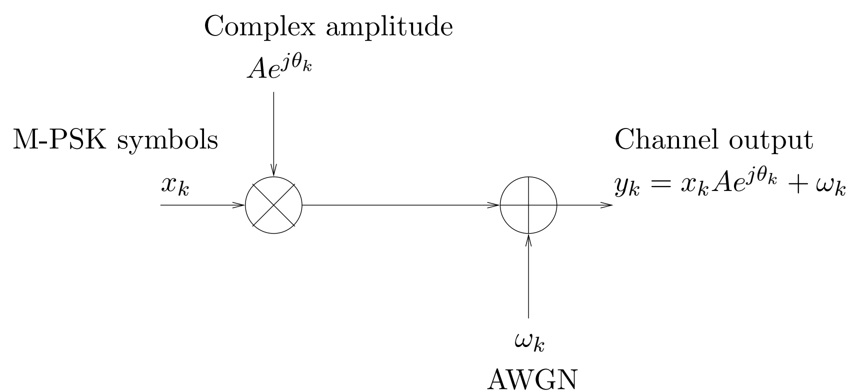

Figure 4.1: System model.

Information bits are modulated using a M-Phase Shift Keying (M-PSK) modulation to obtain the complex symbols $x_k = e^{j\alpha_k}$ where α_k can take values in $\{0, 2\pi/M, \ldots, 2\pi(M-1)/M\}$; M denotes the modulation order.

The symbols are then multiplied by the channel phasor $Ae^{j\theta_k}$ where θ_k is a random variable that accounts for the instabilities of the frequency source in the system, and A is the channel amplitude. The signal is then passed through an Additive White Gaussian Noise (AWGN) channel. In summary, the input-output relation for our system is:

$$y_k = x_k A e^{j\theta_k} + \omega_k \tag{4.1}$$

where the real amplitude A is assumed to be known, θ_k is the random phase that takes into account the phase noise, and finally, ω_k is a zero-mean complex Gaussian random variable with variance per component σ_ω^2. The symbols x_k are normalized by multiplying by a suitable factor so that $E[|x_k|^2] = 1$. The time index is denoted by k.

The phase estimation is performed over a sliding window of size $N = 2W + 1$ samples. The parameter W should depend on the coherence time of the channel. More precisely, let $\mathbf{x}$ be a vector of $N = 2W + 1$ transmitted symbols and $\mathbf{y}$ the observation at the receiver

$$\mathbf{x} = [x_{-W}, ..., x_0, ..., x_W]$$

$$\mathbf{y} = [y_{-W}, ..., y_0, ..., y_W]$$

The ML estimators seek to find the estimate of the phase θ_0 of the sample x_0 in the middle of the window from the whole observation $\mathbf{y}$.

4.2 The ML estimator

As we already told, the algorithm actually implemented for the phase estimation is equivalent to a ML phase estimator for a signal observed over a time interval in stationary Gaussian noise. In the following we briefly review how it is obtained by using the ML approach.

Usually in literature $\mathbf{x}$ is called the *missing data* while $\mathbf{y}$ is called the *incomplete data*. Finally we call $\kappa = (\mathbf{x}, \mathbf{y})$ the *complete data* [9], [7]. If θ is the parameter that we want to estimate, when κ is available, ML estimation of θ is obtained by maximizing the *likelihood function*

$$\hat{\theta}_{ML} = \arg\max_{\theta} p(\mathbf{x}, \mathbf{y}|\theta) \tag{4.2}$$

In particular the above definition tells that we are interested in the likelihood function as a function of the unknown parameter θ when the complete data κ acts as a *fixed* parameter. The principle of maximum likelihood requires us to choose as an estimate of the unknown parameter that value of θ for which the likelihood function assumes its largest value.

In a communication system the incomplete data is the only available information at the receiver side, i.e., no a priori information on the source distribution is available. In such situation ML estimation becomes:

$$\hat{\theta}_{ML} = \arg\max_{\theta} p(\mathbf{y}|\theta) \tag{4.3}$$

It is usually more convenient, for many situations, to consider the logarithm of the likelihood function, the *log-likelihood function*. Since the logarithm is a monotonic function, the maximization of the likelihood and log-likelihood functions is equivalent, that is, $\hat{\theta}_{ML}$ maximizes the likelihood function if and only if it also determines the largest value of the log-likelihood. In this thesis we will mainly assume the log-likelihood function. In this case the ML estimation criterion becomes:

$$\hat{\theta}_{ML} = \arg\max_{\theta} \log p(\mathbf{y}|\theta) \tag{4.4}$$

In order to determine an estimator one needs to assume a stochastic model for the unknown parameter. In the first ML estimator we assume that the phase θ obeys the *Constant phase model* that we briefly recall.

$$\theta_k = \theta \in [-\pi, \pi) \qquad \forall k \in [-W, \dots, W] \tag{4.5}$$

The above model is a *stochastic* model, so the variable θ is a *random* variable, we refer to Section 2.2 for further details.

The likelihood function is obtained by conditioning on the unknown parameter θ the distribution of the received samples $p(\mathbf{y})$ (a.k.a. *evidence* [1]). In this way the observation y_k is a function of the Gaussian variable ω_k. Thus the observation vector $\mathbf{y}$ has a multivariate Gaussian distribution. Since all the noise samples are independent and identically distributed (i.i.d.), the likelihood function can be factorized obtaining the following, well known, distribution:

$$p(\mathbf{y}|\theta) = \frac{1}{(2\pi\sigma_\omega^2)^{\frac{N}{2}}} \exp\left[-\frac{1}{2\sigma_\omega^2} \sum_{k=-W}^{W} |y_k - Ae^{j\theta} x_k|^2\right] \tag{4.6}$$

The log-likelihood function is straightforward and results in the following:

$$\log p(\mathbf{y}|\theta) = -\frac{N}{2} \log(2\pi\sigma_\omega^2) - \frac{1}{2\sigma_\omega^2} \sum_{k=-W}^{W} |y_k - Ae^{j\theta} x_k|^2 \tag{4.7}$$

The major issue now is to find a way to maximize the log-likelihood function. In this case we can apply the *likelihood equation* [3]:

$$\frac{\partial \log p(\mathbf{y}|\theta)}{\partial \theta} = 0 \tag{4.8}$$

In other situations, when a closed form does not exist, other approaches like the *gradient methods* are required, it will be discuss in the factor graph contest in the next chapter.

In summary by applying the Eq. (4.8) under the assumptions that:

- the received symbols are being corrupted by white Gaussian noise;
- the Constant phase model is adopted;

we obtain the first ML estimator that is:

$$\hat{\theta}_{ML} = \arg\left\{\sum_{k=-W}^{W} y_k \hat{x}_k^*\right\} \tag{4.9}$$

Before continuing our analysis we state that x_k is not known and is replaced in the previous equation by $\hat{x}_k$ that could denote the *pilot symbols* or the *estimates* of the transmitted symbols provided by other components of the receiver. In the particular case of the iterative algorithm used, during each iteration these estimates are updated and lead, hopefully, to a better channel estimation until convergence is reached.

Performance of the ML estimator

We start by reporting in the following picture a realization of the phase process and its ML estimate, see Fig. 4.2.

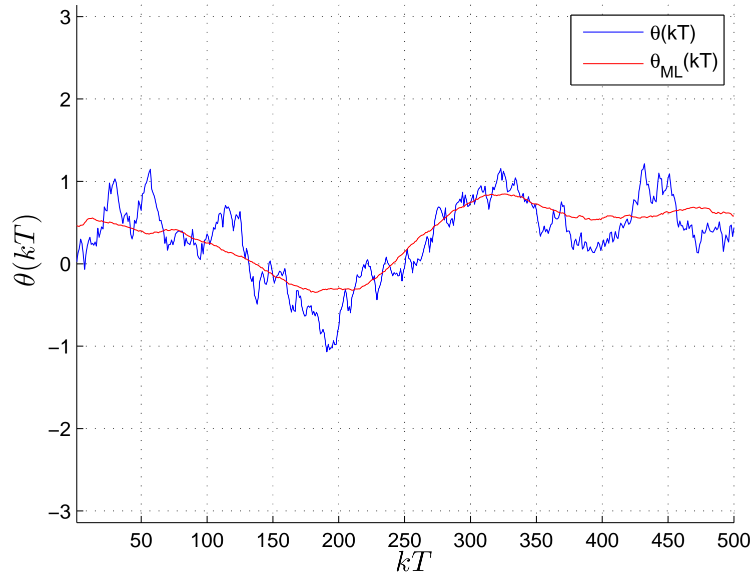

Figure 4.2: ML phase estimation: $SNR = 4$ dB, $\sigma_p^2 = 25(°)^2$, $N = 101$.

The previous figure graphically supports the idea that the estimation process is equal to the *moving average* of the phase process. For us this issue has an important role since it has a direct consequence on the implementation of the estimator.

The operation of moving average can be implemented by a FIR filtering. We indicate by $h_{ML} = [h_0, \ldots, h_N]$ the FIR filter, which in this case has a *rect* impulse response and when computes the average it does not favor any sample in the window. A block diagram of the ML estimator is shown in Fig. 4.3.

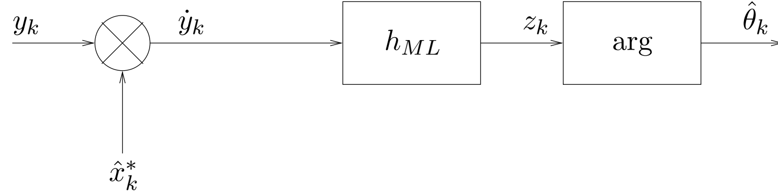

Figure 4.3: Maximum Likelihood phase estimation as a moving average.

The performance of the estimator are being evaluated in terms of the *Mean Square Error* (MSE) for the phase estimation. We recall that if θ is the unknown parameter and $\hat{\theta}$ its estimate, the MSE is defined as

$$MSE = E[|\theta - \hat{\theta}|^2] \tag{4.10}$$

In the following figures we present the MSE as function of the signal to noise ratio SNR (Fig. 4.4) and of the phase noise variance σ_p^2 (Fig. 4.5).

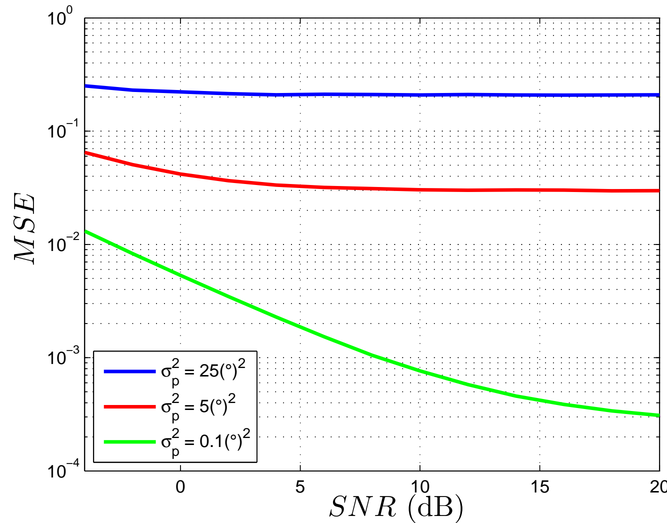

Figure 4.4: Mean Square Error as a function of SNR, $N = 101$.

By observing Fig. 4.4 we can see how the estimation strongly degrades as the phase noise increases. We are conscious that a phase process with variance $5(°)^2$ represents a very bad situation for a passband receiver; we can infer that in presence of strong phase noise there seems to exist an error floor for at high SNR.

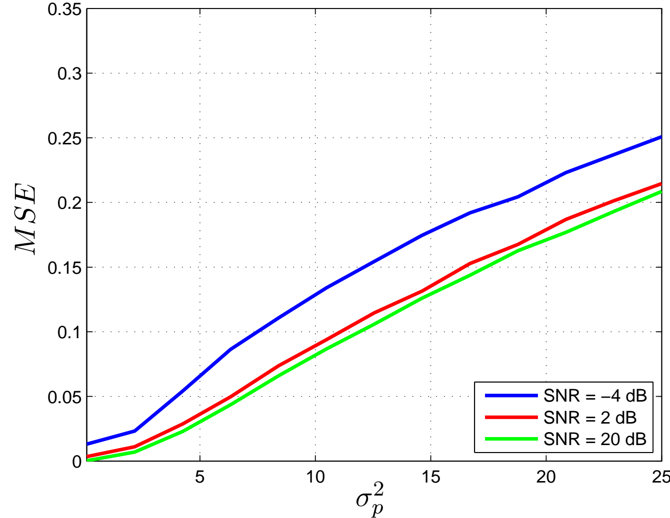

Figure 4.5: Mean Square Error as a function of σ_p^2, $N = 101$.

On the other hand we observe in Fig. 4.5 how the differences between the three cases proposed are less pronounced. In particular there exists just a small offset between the curves for $SNR = 2$ dB and $SNR = 20$ dB and it is still constant in the variation of the phase noise variance. This also confirms what we have already asserted and, in conclusion, we can say that the ML estimator seems to be less robust to the phase noise rather than the thermal noise.

The reason is that the Constant phase model can be a strong assumption in some communications systems. In this model it is assumed that the phase offset is constant, but this assumption is typically not met. Indeed it often undergoes random fluctuations as described by the Wiener-Levy phase model (which was the case of Fig. 4.3, Fig. 4.4, and Fig. 4.5). Due the nature of the *real* phase noise, the performance of this estimator is strongly dependent on the window size N, hence by the value of W. Indeed in the previous simulations this parameter was fixed on purpose and equal to $N = 101$ samples. This choice was suggested by the intuition that the designer may be induced to use a larger window in order to fight the channel variability. To validate this thesis we have investigated the MSE as a function of the SNR for different window sizes. The phase noise is fixed and has variance $\sigma_p^2 = 25(^{\circ})^2$. For the same purposes we present also the MSE versus the phase noise variance in the case of a very low thermal noise, $SNR = 20$ dB.

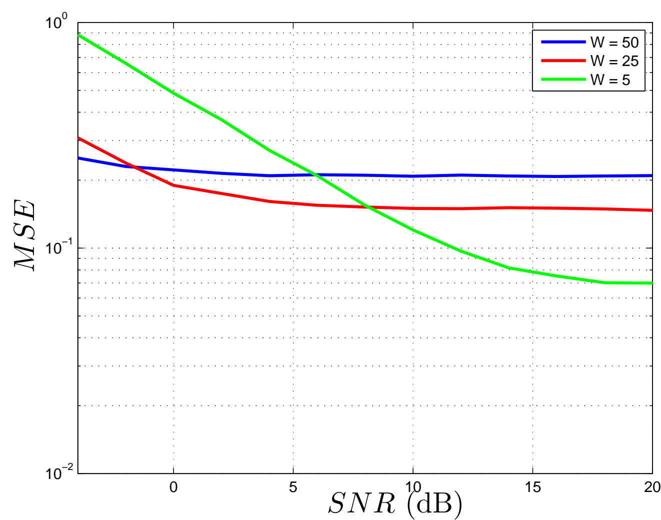

Figure 4.6: Mean Square Error as a function of SNR, for different window sizes, $\sigma_p^2 = 25(^\circ)^2$.

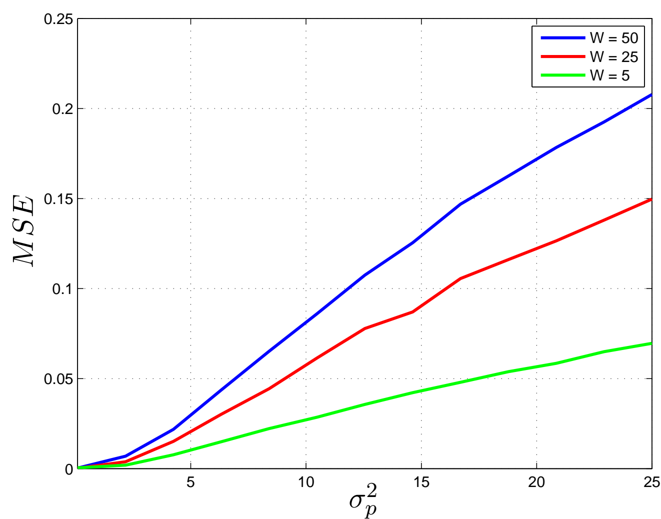

Figure 4.7: Mean Square Error as a function of σ_p^2, for different window sizes, $SNR = 20$ dB.

As we can see the performance of the ML estimator in terms of MSE clearly depends on the window size. In Fig. 4.6 it can be observed that for high SNR values (greater then 10 dB) the smaller window leads to a better estimation. This is not valid for low SNR where the larger window works better. In Fig. 4.7 it is evident how in presence of a good thermal noise environment a smaller window size determines best results. The reason for this is that in the presence of strong phase noise, non-adjacent symbols are weakly correlated, so the width of the selected window should be quite small. On the other hand thermal noise has an important role. In case the thermal noise is dominant over the phase noise, it may be necessary to choose a wide window so that the estimator can average over a larger number of samples.

After these first results, in order to determine an improvement for the phase estimation, we decide to adopt a more refined statistical model for the phase noise. In Section 4.3 another ML estimator will be obtained whereas in Section 4.4 through a qualitative study, in which is investigated the behavior of the first estimator for different window sizes, we will obtain a way to make an "optimal" choice for the window parameter W. With a slight abuse of notation we refer to these estimators as the *ML estimator improved* and the *ML estimator with optimal window.*

4.3 The ML estimator improved

In order to improve the phase estimation some considerations on the received symbol y_k are first performed, after that we will adopt the Wiener-Levy phase model and through an approximation another estimator will be obtained.

We start by multiplying the received sample by $\hat{x}_k^*$ in order to get rid of the dependency of the data.

$$\dot{y}_k = y_k \hat{x}_k^* = |\hat{x}_k|^2 A e^{j\theta_k} + \omega_k \hat{x}_k^* = A e^{j\theta_k} + \dot{\omega}_k \tag{4.11}$$

This operation is equivalent to a rotation for the symbol y_k and it does not change the statistical properties of ω_k so the resulting $\dot{\omega}_k$ is still a complex Gaussian variable. Note that, as mentioned before, $\hat{x}_k$ could be a pilot symbol or the estimate of a data symbol.

Now we write the general channel phase θ_k as a function of the desired phase θ_0. By considering the Wiener-Levy phase model (see Eq. (2.14)), the phase noise in the sliding window can be written as follows:

$$\theta_k = \theta_0 + \sum_i sgn(i)\Delta_i \tag{4.12}$$

where $i \in [-k, ..., 0]$ if $k < 0$ or $i \in [0, ..., k]$ if $k > 0$.

This lets us introduce an approximation by assuming that the variable $\sum_i sgn(i)\Delta_i$ has a small value compared to one:

$$e^{j\theta_k} = e^{j(\theta_0 + \sum_i sgn(i)\Delta_i)} = e^{j\theta_0} e^{j\sum_i sgn(i)\Delta_i} \approx e^{j\theta_0}\left(1 + j\sum_i sgn(i)\Delta_i\right) \tag{4.13}$$

With the above simplification the rotated received symbol $\dot{y}_k$ becomes:

$$\begin{aligned}\dot{y}_k &= Ae^{j\theta_0}\left(1+j\sum_i sgn(i)\Delta_i\right)+\dot{\omega}_k = \\ &= Ae^{j\theta_0}+jAe^{j\theta_0}\left(\sum_i sgn(i)\Delta_i\right)+\dot{\omega}_k = \\ &= Ae^{j\theta_0}+\eta_k \end{aligned} \tag{4.14}$$

where we have denoted as η_k a new equivalent noise that we need to statistically characterize in order to find the new ML estimator for θ_0.

The equivalent noise η_k has the following form:

$$\eta_k = jAe^{j\theta_0}\left(\sum_i sgn(i)\Delta_i\right)+\dot{\omega}_k \tag{4.15}$$

As we can see, it is a function of two independent Gaussian random variables (namely the total phase offset $\sum_i sgn(i)\Delta_i$ and AWGN process $\dot{\omega}_k$).

In order to statistically characterize it, we start by evaluating its mean and its statistical power:

$$E[\eta_k]=0 \qquad \forall\, k \tag{4.16}$$

$$E\left[|\eta_k|^2\right]=A^2|k|\sigma_p^2+\sigma_\omega^2=\sigma_{\eta,k}^2 \tag{4.17}$$

We observe a first difference respect to the first phase model. In this case we obtain for the new noise η_k a power profile that is time variant within a window as shown in Eq. (4.17). The reason is that we have have assumed a model for θ_k whose variance grows linearly with the time index. In particular, as we move far from the sample in the middle of the window, the statical power increases. For example Fig. 4.8 reports a power profile of η_k for $\sigma_p^2 = 10$ $(°)^2$ and $SNR = 2$ dB.

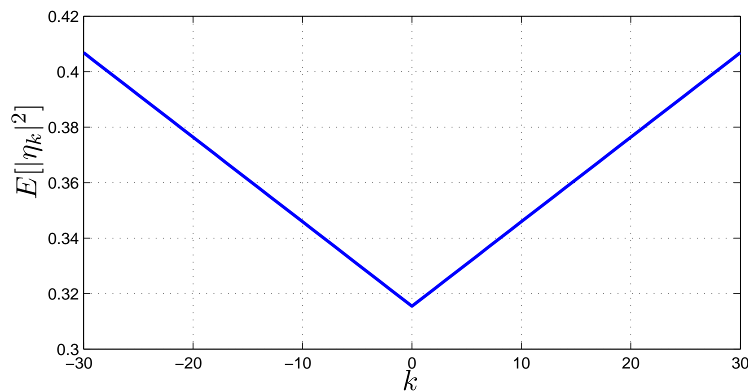

Figure 4.8: Power profile of η_k for $\sigma_p^2 = 10$ $(°)^2$ and $SNR = 2$ dB.

We remark that for the Constant phase model the noise power profile is constant along all the window size.

We can also evaluate the covariance matrix $\mathbf{\Sigma}_\eta$ that is equal to the autocorrelation matrix $\mathbf{R}_\eta$ since η_k is a zero-mean process. Let be $\boldsymbol{\eta}$ a vector of noise samples:

$$\boldsymbol{\eta} = [\eta_{-W}, ..., \eta_0, ..., \eta_W]^T$$

the correlation matrix is:

$$\mathbf{R}_\eta = E\left[\boldsymbol{\eta}\boldsymbol{\eta}^H\right] \begin{bmatrix} A^2W\sigma_p^2 + \sigma_\omega^2 & 0 & \dots & \dots & 0 \\ 0 & \ddots & & & 0 \\ \vdots & & \sigma_\omega^2 & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \dots & \dots & 0 & A^2W\sigma_p^2 + \sigma_\omega^2 \end{bmatrix}.$$

Since it is a diagonal matrix we can conclude that the process η_k is *white* and still has a Gaussian distribution because is obtained by a sum of independent Gaussian random variables.

As we already did for the estimator of Section 4.2, we adopt the ML approach and, by conditioning on the value θ_0, we see that the rotated received symbol $\dot{y}_k$ is a function of two independent random variables, thus the observed vector $\dot{\mathbf{y}}$ has in general a *multivariate* Gaussian distribution. The likelihood function is:

$$p(\dot{\mathbf{y}}|\theta_0) = \frac{1}{[(2\pi)^N det(\mathbf{\Sigma}_\eta)]^{\frac{1}{2}}} \exp\left[-\frac{1}{2}(\dot{\mathbf{y}} - \mathbf{m})^T \mathbf{\Sigma}_\eta^{-1}(\dot{\mathbf{y}} - \mathbf{m})\right]$$

where the mean vector $\mathbf{m}$ is:

$$\mathbf{m} = E[\dot{y}_k][11\dots1]^T = Ae^{j\theta_0}[11\dots1]^T$$

Since the symbols are independent, the likelihood function can be factorized as:

$$p(\dot{\mathbf{y}}|\theta_0) = \frac{1}{[(2\pi)^N det(\mathbf{\Sigma}_\eta)]^{\frac{1}{2}}} exp\left[-\frac{1}{2}\sum_{k=-W}^{W}\frac{|\dot{y}_k - Ae^{j\theta_0}|^2}{\sigma_{\eta,k}^2}\right]$$

By taking the natural logarithm we obtain the log-likelihood:

$$\log p(\dot{\mathbf{y}}|\theta_0) = -\frac{1}{2}\log((2\pi)^N det(\mathbf{\Sigma}_\eta)) - \frac{1}{2}\sum_{k=-W}^{W}\frac{|\dot{y}_k - Ae^{j\theta_0}|^2}{\sigma_{\eta,k}^2}.$$

As it is known the probability density function $p(\dot{\mathbf{y}}|\theta_0)$ is maximized when the log-likelihood is maximized and, since the latter is a concave function, we can find the

new ML estimator by writing $\partial \log p(\dot{\mathbf{y}}|\theta_0)/\partial\theta_0 = 0$. In this particular case we report all the steps:

$$\begin{aligned}
\frac{\partial \log p(\dot{\mathbf{y}}|\theta_0)}{\partial\theta_0} &= -\frac{1}{2}\sum_{k=-W}^{W}\frac{\partial}{\partial\theta_0}\frac{|\dot{y}_k - Ae^{j\theta_0}|^2}{\sigma^2_{\eta,k}} = \\
&= -\frac{1}{2}\sum_{k=-W}^{W}\frac{\partial}{\partial\theta_0}\frac{(\dot{y}_k - Ae^{j\theta_0})(\dot{y}^*_k - Ae^{-j\theta_0})}{\sigma^2_{\eta,k}} = \\
&= -\frac{1}{2}\sum_{k=-W}^{W}\frac{\partial}{\partial\theta_0}\frac{|\dot{y}_k|^2 - \dot{y}_k Ae^{-j\theta_0} - \dot{y}^*_k Ae^{j\theta_0} + A^2}{\sigma^2_{\eta,k}} = \\
&= -\frac{jA}{2}\sum_{k=-W}^{W}\frac{\dot{y}_k e^{-j\theta_0} - \dot{y}^*_k e^{j\theta_0}}{\sigma^2_{\eta,k}} = \\
&= A\sum_{k=-W}^{W}\frac{\Im\left[\dot{y}_k e^{-j\theta_0}\right]}{\sigma^2_{\eta,k}} = 0 \qquad (4.18)
\end{aligned}$$

The last equation is satisfied when:

$$e^{-j\theta_0}\left(\sum_{k=-W}^{W}\frac{\dot{y}_k}{\sigma^2_{\eta,k}}\right) \in \mathbb{R} \qquad (4.19)$$

In particular:

$$\arg\left\{\sum_{k=-W}^{W}\frac{\dot{y}_k}{\sigma^2_{\eta,k}}\right\} - \theta_0 = 0 + m\pi \qquad m \in \mathbb{Z} \qquad (4.20)$$

Finally by considering only the solution for $m = 0$ and denoting θ_0 as $\hat{\theta}_{ML_i}$, the ML *estimator improved*, is:

$$\hat{\theta}_{ML_i} = \arg\left\{\sum_{k=-W}^{W}\frac{y_k \hat{x}^*_k}{\sigma^2_{\eta,k}}\right\} \qquad (4.21)$$

We can observe how this second ML estimator is quite similar to the previous, the only difference is that in this case are present the coefficients $1/\sigma^2_{\eta,k}$ within the sum. As a consequence, the filter h_{ML} (see Fig. 4.3) of the second estimator has an impulse response related to the power profile $\sigma^2_{\eta,k}$, indeed each coefficient can be obtained as follows:

$$h_{ML,k} = \frac{1}{\sigma^2_{\eta,k}} \qquad k = [-W, ..., W] \qquad (4.22)$$

For example Fig. 4.9 depicts the impulse response of the FIR filter corresponding to the noise power profile in Fig. 4.8. The meaning is that the second ML estimator acts as a *weighted moving average* for the phase process, where each phase sample is weighted with a coefficient related to its position within the observation window.

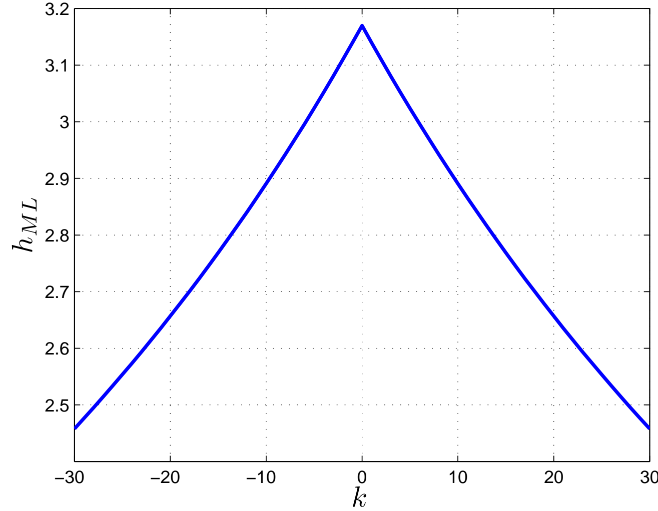

Figure 4.9: Estimator filter h_{ML} for $SNR = 2$ dB and $\sigma_p^2 = 10(°)^2$.

By observing Fig. 4.9 it is easy to understand that the estimator computes a weighted moving average privileging the sample in the middle of the window, i.e., the sample whose phase we try to estimate, because of the particular properties of the Wiener-Levy phase process.

Before doing a performance comparison between the two ML estimators we prefer to focus on another issue, the behavior of the first estimator for different values of the window parameter W. Section 4.5 will report the performance comparison.

4.4 The ML estimator with optimal window

We have seen that the choice of the window size N, i.e., of the window parameter W, is important for the performance of the first ML estimator and it should depend on both the phase noise and the thermal noise.

In this regard we report a experimental study done by a series of simulations in order to investigate the ML estimator's behavior for different W. These are the adopted simulation parameters:

- Number of uncoded BPSK symbols $L = 2000$.
- SNR range $[-4, \ldots, 10]$ dB.
- Phase noise variance σ_p^2 range $[0.1, \ldots, 25]$ $(°)^2$.
- Window size N range [1, ..., 201] corresponding to a range $[0, \ldots, 100]$ for the parameter W.

In all the simulations the phase noise random process θ_k is generated by means of a random walk. The performance are evaluated in terms of the MSE of the phase estimation. In the following sections we refer to the *optimal window parameter* W as the parameter that determines the smallest MSE among all the tests performed.

Optimal W as a function of the phase noise variance

In this first series of simulation is evaluated the optimal window parameter as a function of the phase noise variance σ_p^2. For each realization the MSE is computed for different parameters W and then the one that minimizes the MSE is chosen. We have done simulations for 8 different variances of the thermal noise. The results are shown in Fig. 4.10.

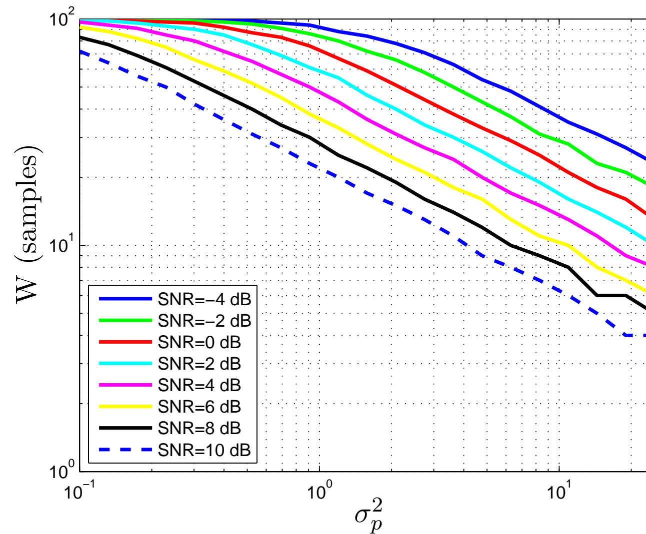

Figure 4.10: Optimal W as a function of σ_p^2 (logarithmic x,y-axis).

As expected we can observe that, as the phase noise variance increases the optimum window size decreases. The saturation phenomenon for small values of σ_p^2 is due to the chosen upper limit on the range of W. Anyway we observe that in case of weak phase noise the choice of a larger window determines in general a better estimation. We can also note that the slope of all the curves is $\approx 1/2$.

Optimal W as a function of the SNR

The next series of simulations is complementary to the previous set. In this case, we investigate the behavior of the estimator as function of the signal to noise ratio (SNR) for different phase noise scenarios (Fig. 4.11).

As in the previous plots, the curves have a decreasing trend but less pronounced. A saturation is also present for low SNR that confirms our hypothesis on the estimator behavior, in case the thermal noise is dominant over the phase noise it advised to average on as large a number of samples as possible.

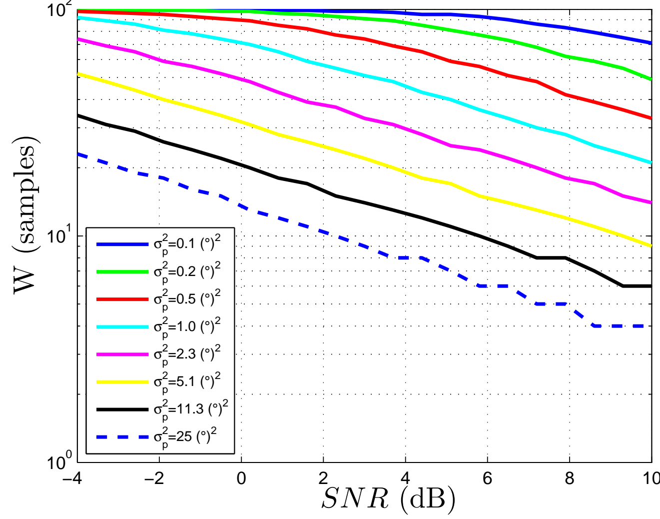

Figure 4.11: Optimal W as a function of the SNR.

Optimal W as a function of $\sigma_p^2/\sigma_\omega^2$

Since the estimator seems to have a similar behavior as a function of the phase noise and of the thermal noise variance, we seek to evaluate its performance as a function of a quantity that takes both into account, in particular we consider the ratio $\sigma_p^2/\sigma_\omega^2$. This choice is done keeping in mind the approximation of the Wiener-Levy phase model adopted in Section 4.3. Indeed consider the Eq. (4.17) re-written as follows:

$$\sigma_{\eta,k}^2 = \sigma_\omega^2 \left(A^2|k|\frac{\sigma_p^2}{\sigma_\omega^2} + 1 \right) \tag{4.23}$$

By reporting the latter in decibel we can observe an interesting issue:

$$(\sigma_{\eta,k}^2)_{dB} = (\sigma_\omega^2)_{dB} + \left(A^2|k|\frac{\sigma_p^2}{\sigma_\omega^2} + 1 \right)_{dB} \tag{4.24}$$

As we can see the approximation adopted suggests to have a further degradation factor in addition to the thermal noise. This factor depends both on the phase noise variance and the thermal noise, i.e., on the ratio $\sigma_p^2/\sigma_\omega^2$. There is also the scale term A^2 that we do not consider since in our simulations is set to 1.

To avoid the saturation, in the following simulations we have increased the window parameter range for the tests.

First we report in Fig. 4.12 the optimal W as a function of the ratio, plotted with a linear scale.

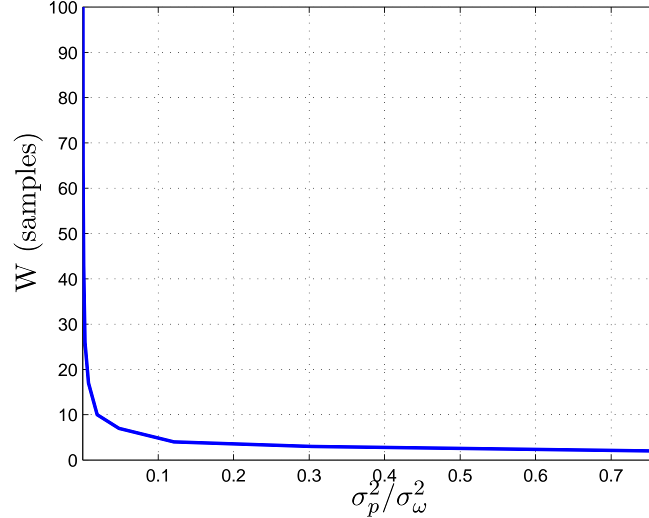

Figure 4.12: Optimal W as a function of the ratio $\sigma_p^2/\sigma_\omega^2$.

As we can see the quantities involved are related by some kind of inverse relation:

$$y = \frac{K}{x^n}$$

In this regard it could be useful to plot the same result with a logarithmic scale for both axis. This should enable us to evaluate the slope of the curve. Indeed by taking the logarithm we obtain:

$$\log y = \log \frac{K}{x^n} = \log K - n \log x$$

In this case the points should follow a straight line whose slope should be easy to evaluate, i.e., the value of n. This last issue is shown in Fig. 4.13.

By observing it, we can see how ever two decades on the x-axis the curve decreases by one decade on the y-axis. The slope of the curve is $n \approx 1/2$. Also by experimental tests we have found that the coefficient $K \approx 1.88$. In this way we can infer the relation between the "optimal" parameter W_{opt} and the ratio $\sigma_p^2/\sigma_\omega^2$ that is:

$$W_{opt} \approx \frac{1.88}{\sqrt{\sigma_p^2/\sigma_\omega^2}} \tag{4.25}$$

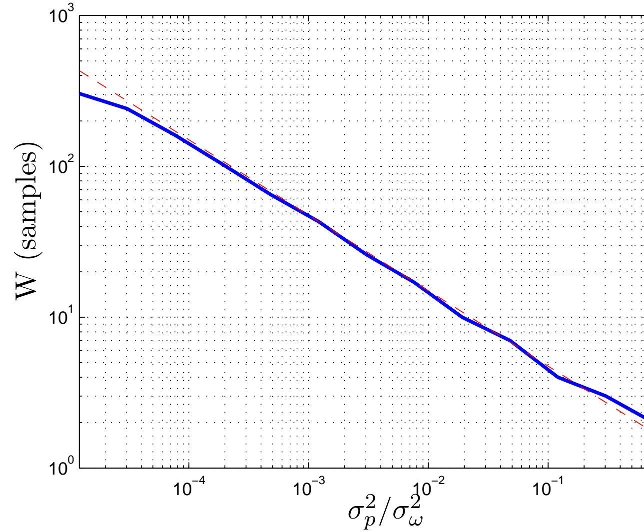

Figure 4.13: Optimal W as a function of $\sigma_p^2/\sigma_\omega^2$ (logarithmic scale).

Until now, in order to improve the ML phase estimation already implemented in the receiver, it has been assumed an approximation of the Wiener-Levy phase model and we have obtained respectively, a new ML estimator and a way to choose the optimal window size W_{opt}.

The two solutions proposed have in common the fact that they require more knowledge at the receiver, that is usually not provided in typical communication systems. Indeed while for the ML estimator improved the two parameters σ_p^2 and σ_ω^2 are necessary for the design of the h_{ML} filter, on the other hand, one can think to design a receiver able to evaluate the ratio $\sigma_p^2/\sigma_\omega^2$ and then the choice of the best window size will be straightforward by using Eq. (4.25).

In this regard we can assert that in our case, since the final goal of the global project is to demonstrate the system described in Chapter 1 by means of a demo in which a *Channel emulator* will be used, we can assume that these parameters are known. On the other hand there exist other phase estimation techniques based on *factor graph approach* which require the same knowledge. In the next chapter we will explore this last channel estimation techniques in a revisited fashion as a first step to find a parallel version of the algorithms, but before we report the performance of the two improvements of the ML estimator.

4.5 Performance evaluation and comparison

In this section we will evaluate the performance of three different ML estimators for a phase process corrupted by a phase noise. We assume that the phase noise is due to white noise sources in the local oscillator at the receiver. For this reason the

phase process is generated by means of a random walk (see Section 2.4 for further details). The three different ML estimators are:

- Maximum Likelihood estimator (ML)
- Maximum Likelihood estimator improved (MLi)
- Maximum Likelihood estimator with optimal W (ML_{opt})

We start by reporting the results of the three different approach on one phase realization. In this way we can have first indication on the three different behaviors.

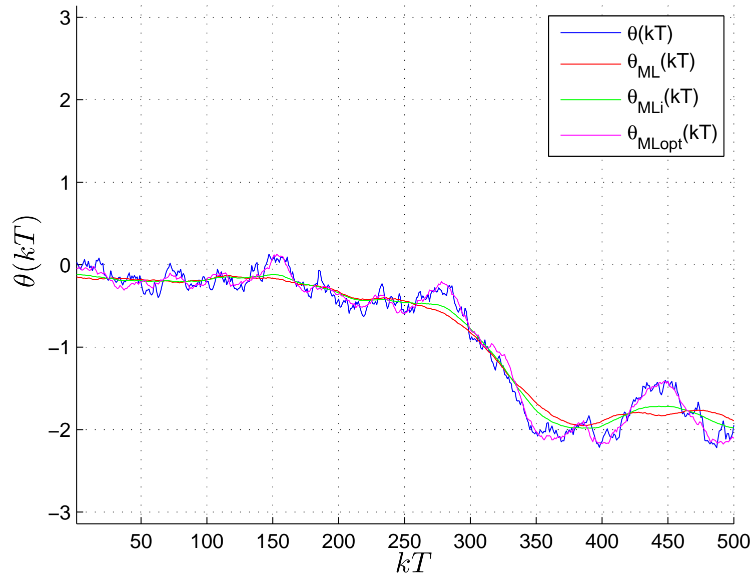

Figure 4.14: Maximum likelihood estimations, $SNR = 10$ dB, $\sigma_p^2 = 10(°)^2$, $N = 101$ samples, $W_{opt} = 8$ samples.

We can observed in Fig. 5.12 how the three approaches lead to different results and, in this particular case, the differences between them are very marked. The MLi estimator seems to determine a better estimation compared to the canonical ML estimator, indeed the resulting estimate process seems to follow the phase variation more closely than the latter. On the other hand by adopting the optimal window size the phase estimation quality increases very much, the resulting curve is very close to the real phase process. In order to confirm these initial insights we compute the MSE for the three estimates.

$$MSE_{ML} = 0.0303 \qquad MSE_{MLi} = 0.0184 \qquad MSE_{MLopt} = 0.0076$$

Basically we can assert that the strong differences are due to the different window sizes adopted in the estimations, i.e. $N = 101$ samples for ML estimator and MLi

estimator and $N = 8$ samples for the ML_{opt} estimator. The previous results are quite intuitive and depend on the particular scenario we have proposed. Indeed is present of a very strong phase noise ($\sigma_p^2 = 10(°)^2$) and, based on the previous considerations, is expected that the window size must be quite small in order to follow the variation of the random process.

In order to compare the real performance have been performed some simulations over a larger number of realizations. We start now with reporting the results obtained. After that some conclusions will be given.

MSE as a function of SNR

In this series of simulations we report the MSE versus the SNR in three different cases where the following phase noise variances were chosen:

$$\sigma_p^2 = 25\ (°)^2 \quad \sigma_p^2 = 5\ (°)^2 \quad \sigma_p^2 = 0.1\ (°)^2$$

As we can see the cases represent three very different phase noise scenarios. While a phase noise variance equal to $\sigma_p^2 = 0.1(°)^2$ is a very common situation for many passband receivers, a phase noise with a variance of $\sigma_p^2 = 5(°)^2$ is a typical situation in satellite communications where we have a strong phase noise and an accurate compensation is required. Finally a phase noise variance of $\sigma_p^2 = 25(°)^2$ is absolutely a disastrous situation for a receiver, but allow us to determine the robustness of the estimators to this impairment. In the following pictures we report the results of the simulations.

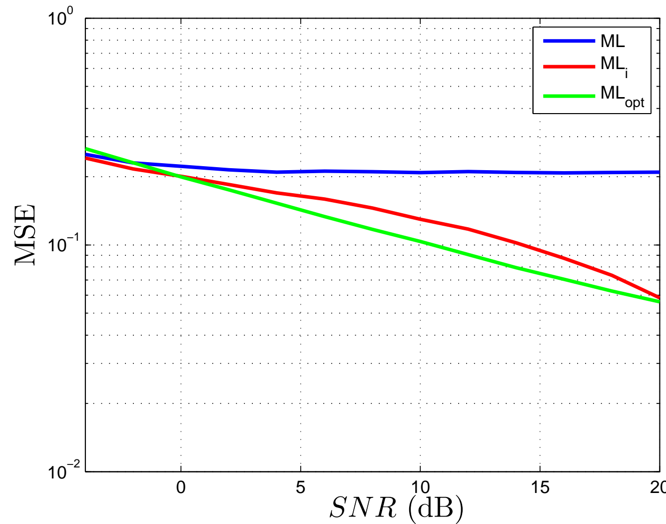

Figure 4.15: MSE as a function of SNR, $\sigma_p^2 = 25\ (°)^2$.

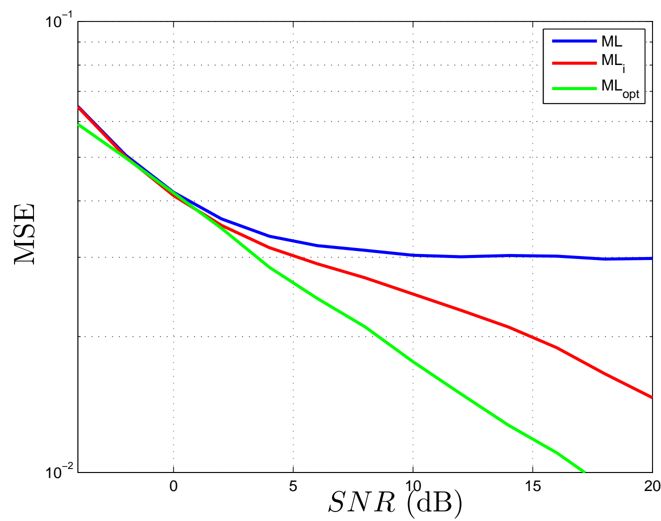

Figure 4.16: MSE as a function of SNR, $\sigma_p^2 = 5\ (^\circ)^2$.

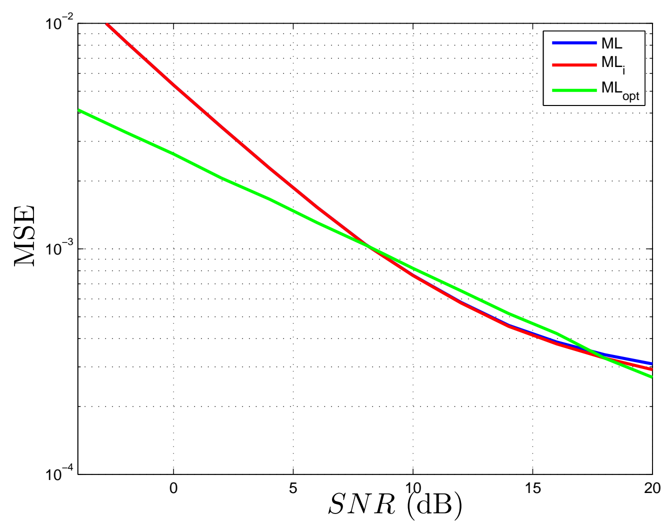

Figure 4.17: MSE as a function of SNR, $\sigma_p^2 = 0.1\ (^\circ)^2$.

From these first results we can see how the two alternative estimators determine in all cases a performance improvement with respect to the canonical ML estimator. In particular we can see that in a bad phase noise environment (Fig. 4.15) while the ML estimator is characterized by an error floor at high SNR the two new estimators improve their performance as the thermal noise decrease. This situation is also present in the second case depicted in Fig. 4.16 where the phase noise is still strong ($\sigma_p^2 = 5(°)^2$) but closer to a practical situation. Finally we observe how in presence of a weak phase noise the ML estimator and the MLi estimator lead to the same results while the choice of the optimal window works better at low SNR (see Fig. 4.16).

MSE as a function of σ_p^2

As already done previously, in the next series of simulations we evaluated the MSE versus σ_p^2 for different SNR. The SNR chosen are:

$$SNR = -4\ dB \quad SNR = 2\ dB \quad SNR = 20\ dB.$$

Regarding these choices we can say that they represent three totally different scenarios. The most common situation is to have a $SNR = 2$ dB while a $SNR = -4$ dB represents a bad thermal noise environment, but not impossible in some communication systems such as satellite ones. Finally to have a $SNR = 20$ dB is the limit case, even if it is a rare case, it will enable us to highlight some differences between the estimators.

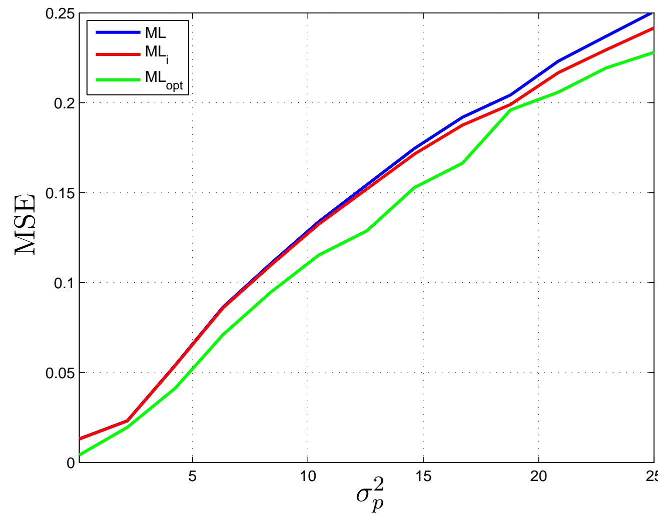

Figure 4.18: MSE as a function of σ_p^2, SNR = -4 dB.

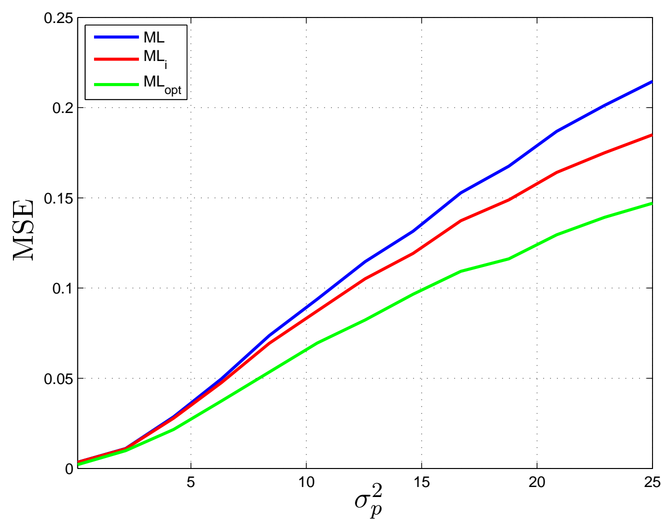

Figure 4.19: MSE as a function of σ_p^2, $SNR = 2$ dB.

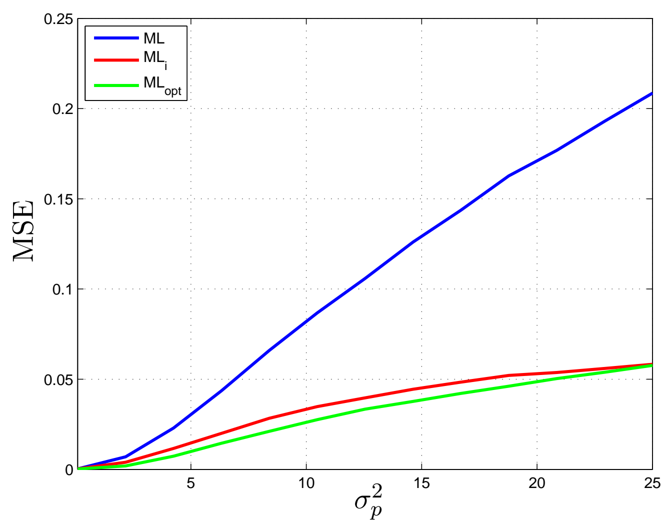

Figure 4.20: MSE as a function of σ_p^2, $SNR = 20$ dB.

By observing Fig. 4.18 we can see how the ML_{opt} estimator is more robust to the phase noise, while the second improvement found has just a slightly difference compared to the ML estimator. The most interesting situation is the one depicted in Fig. 4.19 and, as we can see, the performance improvement is present and very marked for both new estimators. In particular the ML estimator with the optimal window determines the best estimate in terms of mean square error. Finally in the limit case depicted in Fig. 4.20, even if it does not represent a common situation for practical communications systems, it is clear how the two new estimators work better for high SNR.

It should be interesting now to give a comparison between the two improvements found. Indeed by using the same knowledge at the receiver (the parameter σ_p^2 and σ_ω^2) one can think to determine the optimal window (Eq. (4.25)) and also design the h_{ML} filter such in Eq. (4.22) obtaining some kind of ML *estimator improved with optimal window size*. In the following we report the same typology of simulations done previously in which we compare this last estimator called $ML_{i,opt}$ with the M_{opt} because it has yielded the best improvement.

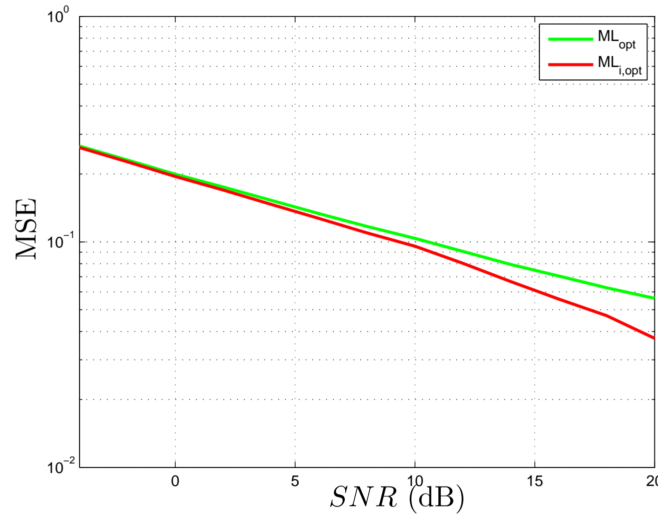

Figure 4.21: MSE as a function of SNR, $\sigma_p^2 = 25\ (^\circ)^2$.

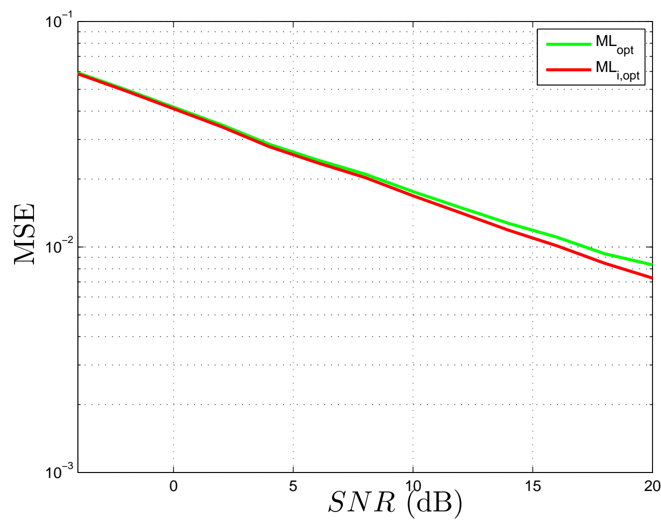

Figure 4.22: MSE as a function of SNR, $\sigma_p^2 = 5\ (^\circ)^2$.

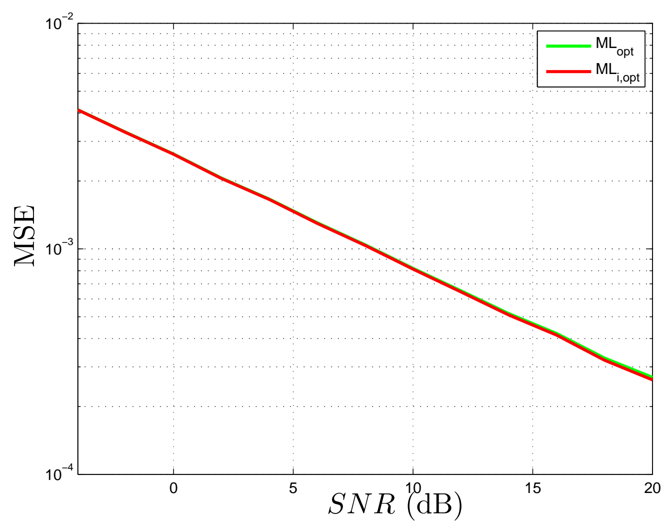

Figure 4.23: MSE as a function of SNR, $\sigma_p^2 = 0.1\ (^\circ)^2$.

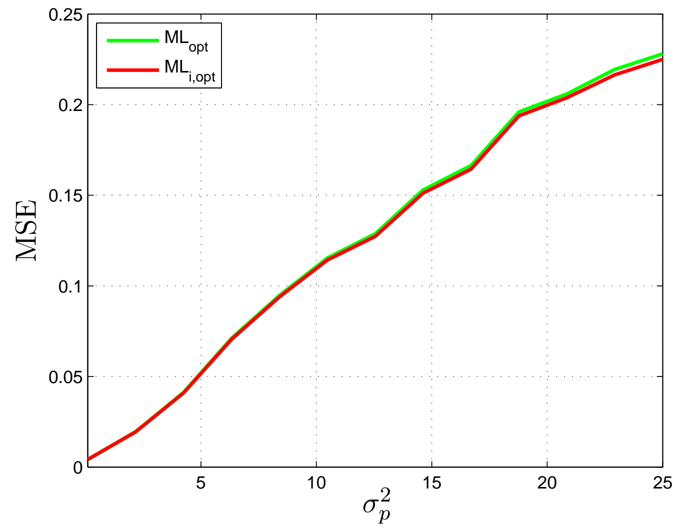

Figure 4.24: MSE as a function of σ_p^2, SNR = -4 dB.

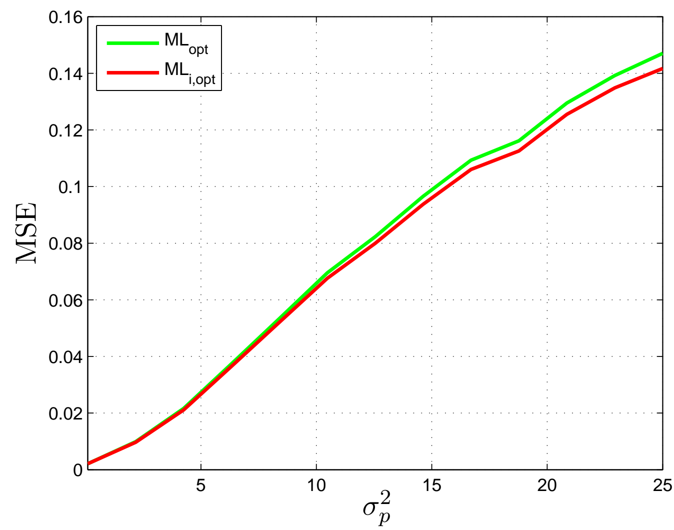

Figure 4.25: MSE as a function of σ_p^2, $SNR = 2$ dB.

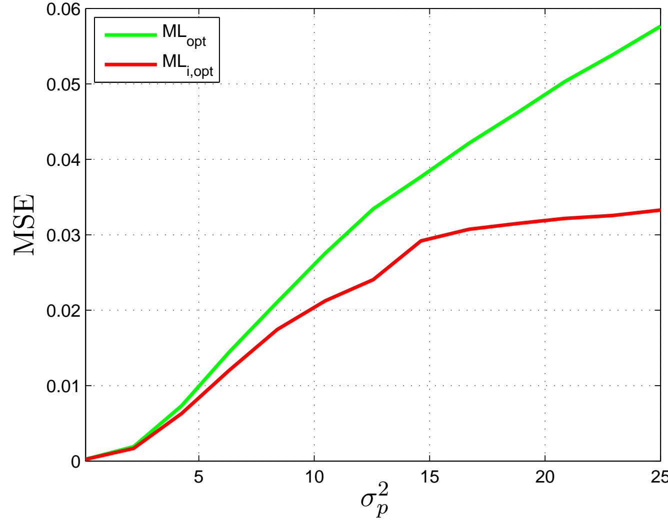

Figure 4.26: MSE as a function of σ_p^2, $SNR = 20$ dB.

We can see how the two estimators are quite equivalent in most of the proposed cases. Only in the limit case depicted in Fig. 4.26 the differences are marked. This is due to the peak of the h_{ML} that has a high amplitude for high SNR, in this way in the weighted moving average operation the estimator privileges the sample in the middle of the window. In the other cases the differences are negligible, this clearly reveals that the biggest performance improvement is obtained by reducing the observation window in the ML_{opt} estimator rather than designing a new filter as done in the ML_i estimator.

4.6 Conclusions

After have presented the system model and the already implemented ML estimator, which assumes the Constant phase model, we have developed two different improvements in order to obtain best performance for the phase estimation. We have called these new estimators with a slight abuse of notation respectively ML estimator with optimal window (ML_{opt}) and ML estimator improved (ML_i).

Indeed the already existing estimator can be viewed as a moving average and its performance depends on the observation window size. In presence of a strong phase noise the window size should be small because non adjacent symbols are strongly uncorrelated, on the other hand when the thermal noise is dominant over the phase noise it may be necessary to choose a large window so that the estimator can average over a larger number of samples.

By adopting an approximation of the more realistic Wiener-Levy phase model we have developed a new ML estimator where the design of the h_{ML} filter is done

taking into account both the phase noise and the thermal noise, in particular the coefficients of the filter are related to these quantities by the following relation:

$$h_{ML,k} = \frac{1}{\sigma^2_{\eta,k}} = \frac{1}{A^2|k|\sigma_p^2 + \sigma_\omega^2} \qquad k = [-W, ..., W]$$

In this way the filter has a piqued impulse response as depicted in Fig. 4.9 and the operation performed on the rotated received symbol is a *weighted* moving average. This can be viewed also as a smarter way to reduce the window size.

The same reasoning has allowed us to think our received symbol affected by an equivalent noise η_k made by the sum of two impairment, the thermal noise and a further degradation related to the ratio $\sigma_p^2/\sigma_\omega^2$ (see Eq. (4.24)). In this regard we have done an experimental study through a series of simulation campaigns in order to investigate the qualitative relation between the optimal window parameter W_{opt} (optimal in the sense that it determines the best phase estimation in terms of MSE among all the tested values) and the ratio $\sigma_p^2/\sigma_\omega^2$. As result we have inferred that these two quantities are related by the following inverse relation:

$$W_{opt} \approx \frac{1.88}{\sqrt{\sigma_p^2/\sigma_\omega^2}}$$

This means that, assuming to have the same knowledge at the receiver, we can think to re-design h_{ML} filter or to choose the optimal window parameter in order to improve the performance.

Simulations have shown that, comparing with the already existing ML estimator, there is a performance improvement for both cases and in particular the ML_{opt} estimator seems to lead best results and at the same time to be more robust to the phase noise.

Finally we have also experimentally demonstrated how the biggest improvement is achieved by adapting the window size to the particular scenario rather than designing a new h_{ML} filter for the moving average.

Chapter 5

Factor Graphs estimators

Factor graphs represent an unified approach for a large variety of topics in coding, signal processing, machine learning and statistics. In particular many algorithms in these fields may be viewed as instances of the *Summary Product algorithms*, i.e., the Sum Product algorithm (SPA) and the Max Product algorithm (MPA) that operate by a message passing in a factor graph.

In this chapter we will present two phase estimation algorithms already known in literature, revisited in order to obtain a parallel implementation. We will start with a short description of how our communications system can be translated to the corresponding factor graph representing the joint probability density function of the variables involved. Then we will focus on the *Channel model* block introducing a first simple model based on the Constant phase model. After that by assuming the Wiener-Levy phase model we will obtain two different algorithms working on a factor graph and some specific implementation details will be chosen keeping in mind the parallel implementation on the graphic card.

In order to be consistent with the notation of Chapter 3 we will indicate by capital letters the domain of the variables and with small letters the random variables. Finally, $p(x)$ could denote a probability density function (pdf) or a probability mass function (pmf) depending on the nature of the variable x.

5.1 System Model description

The input-output relation for our system is:

$$y_k = Ae^{j\theta_k}x_k + \omega_k \tag{5.1}$$

where the channel input symbol x_k belongs to a M-PSK constellation normalized by multiplying by a suitable factor so that $E[|x_k|^2] = 1$ while y_k denotes the corresponding received symbol. A is a real amplitude assumed to be known and θ_k is the unknown phase, a random variable that accounts for the total phase noise. Finally, ω_k is a zero-mean complex Gaussian random variable with variance per component σ_ω^2. The time index is denote by k.

We consider the transmission of frames of $N = 2W + 1$ symbols so we will use the following notation:

- The vector of the input symbols:
$$\mathbf{x} = [x_1, \ldots, x_N]$$
- The vector of the output symbols:
$$\mathbf{y} = [y_1, \ldots, y_N]$$
- The vector of the unknown phase samples:
$$\boldsymbol{\theta} = [\theta_1, \ldots, \theta_N]$$

Another general assumption is that the input symbols are protected by channel coding. In order to maintain the similarities with recent standards for satellite communication systems, like DVB-S2, we will refer to a Low Density Parity Check (LDPC) code.

In the development of the algorithm we will evaluate the stochastic models presented in Chapter 2.

Constant phase model

$$\theta_k = \theta \in [-\pi, \pi) \qquad \forall k \in [1, \ldots, N]$$

where θ is an unknown constant.

Wiener-Levy phase model

$$\theta_k = \theta_{k-1} + \Delta_k \in [-\pi, \pi)$$

where Δ_k is the step-size of the walk and is a zero-mean Gaussian random variable with variance σ_p^2.

5.2 Factor Graph of the System

In general an algorithm for joint decoding and channel estimation may be derived from a factor graph of the code and the channel. This approach usually leads to an algorithm that represents an *approximation* of the symbol-wise MAP decoder [8]. In this case it can be obtained by maximizing the marginals for the symbol variables:

$$\hat{x}_k^{MAP} = \arg\max_{x_k} p(x_k|\mathbf{y}, \boldsymbol{\theta}) = \arg\max_{x_k} \int_{\sim\{x_k\}} p(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) \tag{5.2}$$

The function $p(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta})$ stands for the joint probability density function of $\mathbf{x}$, $\mathbf{y}$ and $\boldsymbol{\theta}$; it is a probability density function (pdf) in $\boldsymbol{\theta}$ and $\mathbf{y}$, due to their continuous nature, and a probability mass function (pmf) in $\mathbf{x}$.

The term approximation it is not used by change, indeed since the main factor graph of the system usually contains cycles, the resulting instance of the SPA leads to an iterative algorithm and does not compute the *exact* marginals of the variables involved.

The *general recipe* to obtain a message passing algorithm working on factor graph can be split in three basic steps:

1. The probability function $p(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta})$ must be represented by a factor graph.

2. Message type are chosen and message update rules are computed.

3. A message update schedule is chosen.

Usually the messages that are exchanging between the nodes are probability mass function for the discrete variables, or probability density function for the continuous random variables. As we will see earlier different representations and approximations are possible for the messages and the choice leads to different algorithms. Naturally also different strategies are possible in Step 3) and they result in different instances of the SPA.

For representing our system we use *conventional factor graph notation* used in [11] where the square-nodes represent factor nodes and circle-nodes represent variable nodes. The system described in Eq. (5.1) is translated into the factor graph shown in Fig. 5.1 which represents the joint probability function of all variables involved.

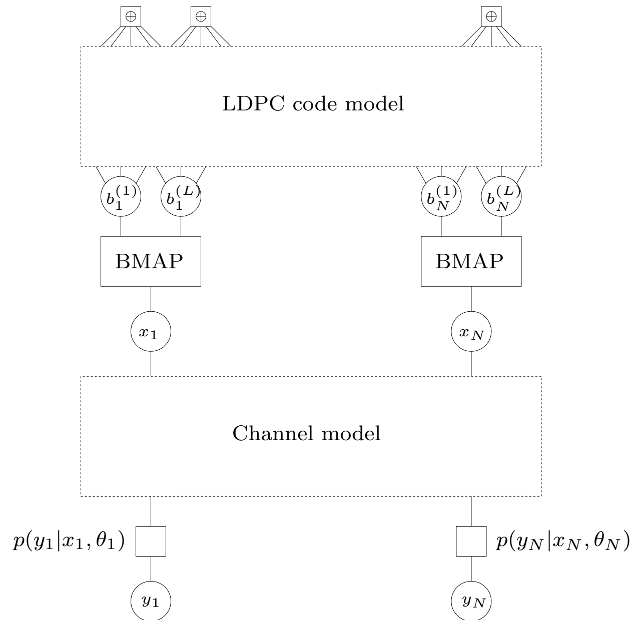

Figure 5.1: Factor graph of the system model.

The upper part of the graph is the indicator function of the LDPC code. We recall that by a factor graph for some code C, we mean a factor graph obtained from (some factorization of) the membership indicator function of C [13].

The block $BMAP$ indicates the bit mapping operation of a block of $L = \log_2 M$ bits to the symbol x_k. We have denoted by $b_k^{(l)}$, $k \in [1, \ldots, N]$, $l \in [1, \ldots, L]$ the information bits. This operation is deterministic.

The factor nodes in bottom row of the graph is the likelihood function of the whole observation vector $\mathbf{y}$ which, since all the samples are supposed to be i.i.d., can be nicely factorized. Each node represents the factor

$$p(y_k|x_k, \theta_k) = \frac{1}{\sqrt{2\pi\sigma_\omega^2}} e^{-\frac{|y_k - x_k A e^{j\theta_k}|^2}{2\sigma_\omega^2}} \tag{5.3}$$

We now focus on the *Channel model* block. First we will present a simple model by assuming the Constant phase model. This will be useful as an example on how a real algorithm can be developed by using the factor graph approach but, since a closed form solution exists for this problem, we will not pay too much attention on it. After that by using the Wiener-Levy phase model we will obtain a more defined factor graph contest for developing our algorithms.

We will see that for the finite-alphabet variables the messages that are exchanged by the nodes are probability mass functions and the message-update rules are handled by the standard sum product rule, which involved finite sums. Instead for continuous variable the messages are probability density functions and the sum product rule leads to intractable integrals which can be approximated in several ways, each approximations corresponds to a certain message type and results in a different algorithm.

5.3 A first simple model

In this paragraph is given an example of a first model in which is considered the Constant phase model. We will obtain a factor graph and an instance of the SPA algorithm working on it. In the following, input symbols will be replaced by their estimates.

We start by considering the joint probability density function of all the variables involved. Since no a priori information is available for the phase θ and the input symbols $\mathbf{x}$ we can write:

$$p(\mathbf{y}, \mathbf{x}, \theta) \propto p(\mathbf{y}|\mathbf{x}, \theta) \tag{5.4}$$

As already done for the ML estimator in Chapter 4, by conditioning on $\mathbf{x}$ and θ the probability density function of the random vector $\mathbf{y}$ is a Gaussian distribution

$$p(\mathbf{y}|\mathbf{x}, \theta) = \frac{1}{(2\pi\sigma_\omega^2)^{\frac{N}{2}}} \exp\left[-\frac{1}{2\sigma_\omega^2}\sum_{k=1}^{N}|y_k - x_k A e^{j\theta}|^2\right] \tag{5.5}$$

In order to find its representation as a factor graph we need to write it in a suitable form, i.e., finding a factorization. As we know the form above is obtained by exploiting the fact that all the noise samples are i.i.d., we come back to the initial general form:

$$\begin{aligned} p(\mathbf{y}|\mathbf{x},\theta) &= \prod_{k=1}^{N} \frac{1}{\sqrt{2\pi\sigma_\omega^2}} \exp\left[-\frac{1}{2}\frac{|y_k - x_k A e^{j\theta}|^2}{\sigma_\omega^2}\right] = \\ &= \prod_{k=1}^{N} f_k(y_k, x_k, \theta) \end{aligned} \tag{5.6}$$

Now that we have the joint probability function represented by a "nice" factorization in Fig. 5.2 we show the corresponding factor graph.

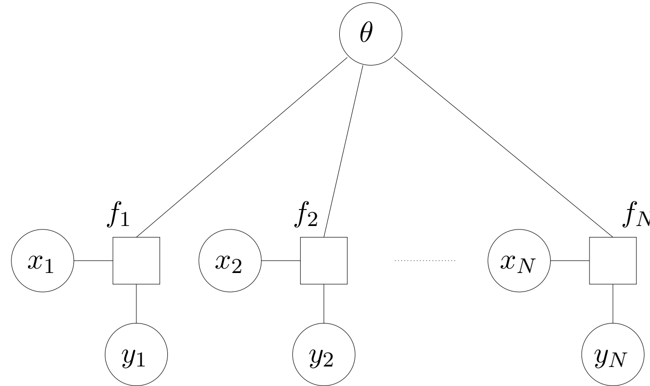

Figure 5.2: Factor graph for the phase estimation, Constant-phase model.

As a first observation we assert that since the resulting graph is cycle-free the SPA leads to the exact marginals for the variables after a finite number of iterations.

Straightforward application of the sum-product algorithm in the graph results as following.

Sum-product algorithm for the Constant phase model

1. Initialization

$$\mu_{x_i \to f_i}(x_i) = p(x_i) \qquad \forall\, i = 1, ..., N$$

$$\mu_{y_i \to f_i}(y_i) = 1 \qquad \forall\, i = 1, ..., N$$

2. Messages from factor nodes f_i toward variable node θ

$$\mu_{f_i \to \theta}(\theta) = \int_{\sim\{\theta\}} f_i(y_i, x_i, \theta)\mu_{x_i \to f_i}(x_i)\mu_{y_i \to f_i}(y_i) \qquad \forall\, i = 1, ..., N$$

3. Messages from θ toward factor nodes f_i

$$\mu_{\theta \to f_i}(\theta) = \prod_{j \neq i} \mu_{f_j \to \theta}(\theta) \quad \forall\, i, j = 1, ..., N$$

4. Messages from factor nodes f_i toward variable nodes x_i

$$\mu_{f_i \to x_i}(x_i) = \int_{\sim\{x_i\}} f_i(y_i, x_i, \theta)\mu_{\theta \to f_i}(\theta)\mu_{y_i \to f_i}(y_i) \quad \forall\, i = 1, ..., N$$

With the symbol $\int_{\sim\{x\}}$ we denote the summary operation over all variables except x.

If we think to this graph as a Channel block inserted in a general graph like the one depicted in Fig. 5.1, after these steps the block can propagate the "beliefs" of the variables x_k to the demapping and the decoding blocks. However we are interested to the phase estimation, and we know that it is possible to compute the marginal probability density function of θ as a product of the incoming messages to the corresponding variable node.

$$p(\theta) = \prod_{i=1}^{N} \mu_{f_i \to \theta}(\theta)$$

To obtain an estimation for θ by an usual criterion as the MAP criterion would be straightforward, but in this particular case the factor graph is not useful for this purpose. Indeed we observe that the integrals in Step 4) involved an integration over the continuous range of θ so, from an implementation point of view of the algorithm, they need to be approximated by numerical methods with the consequent loss of precision. In this example it is possible to obtain a closed form solution for the problem which corresponds to the ML estimator (see Eq. (4.9)). However in other situations there is no closed form solution for the problem, and in such cases one may apply numerical methods.

5.4 Factor graph of a Markovian channel

In this section we consider the Wiener-Levy phase model as channel model. In order to obtain the factor graph we require the a priori distribution of θ_k according to the new stochastic model. Since in this case the phase process is described by a random-walk, the channel, i.e., the phase θ_k, varies according to a *first-order Markov model.* If we denote with $\boldsymbol{\theta}$ the vector of the phase samples in a window of size N, its a priori distribution is:

$$p(\boldsymbol{\theta}) = p(\theta_1) \prod_{k=2}^{N} p(\theta_k|\theta_{k-1}) \tag{5.7}$$

The initial distribution $p(\theta_1)$ and the transition probabilities $p(\theta_k|\theta_{k-1})$ are required to be known at the receiver. In the case of the random walk model, if σ_p^2 is the phase noise variance, the transition probability takes form

$$p(\theta_k|\theta_{k-1}) = \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left\{\frac{(\theta_k - \theta_{k-1})^2}{2\sigma_p^2}\right\} \tag{5.8}$$

The initial distribution $p(\theta_1)$ instead is unknown, hence it is assumed to be uniform in $[-\pi, \pi)$.

By following the general approach we want to find a factorization of the joint probability density function suitable to be represented as a factor graph. In order to do this we report the following steps

$$\begin{aligned} p(\mathbf{y}, \mathbf{x}, \boldsymbol{\theta}) &\propto p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})p(\boldsymbol{\theta}) = \\ &= \left(\prod_{k=1}^{N} f_k(y_k, x_k, \theta_k)\right)\left(p(\theta_1)\prod_{k=2}^{N} p(\theta_k|\theta_{k-1})\right) \\ &= p(\theta_1) f_1(y_1, x_1, \theta_1) \prod_{k=2}^{N} f_k(y_k, x_k, \theta_k) p(\theta_k|\theta_{k-1}) \end{aligned} \tag{5.9}$$

The corresponding factor graph is depicted in Fig. 5.3.

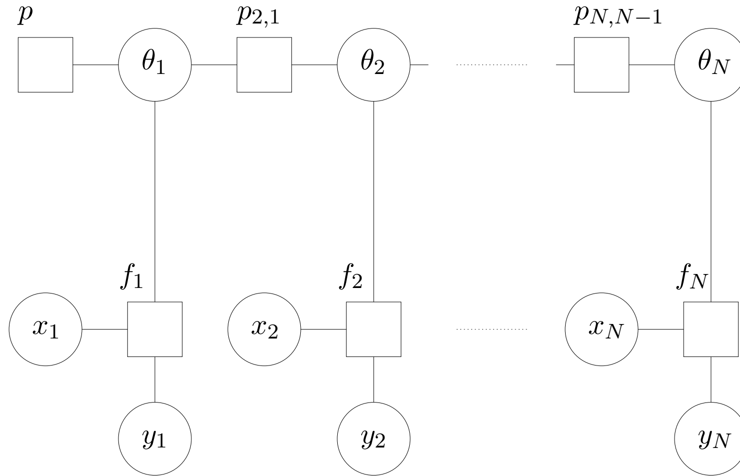

Figure 5.3: Factor graph for the phase estimation, Wiener-Levy phase model.

In the figure above we have denoted the factor nodes in the upper part of the graph by $p_{k,k-1} = p(\theta_k|\theta_{k-1})$. In addition we have also reported the factor node of the a priori distribution of θ_1 even if in our case it is assumed to be unknown.

Also in this case we observed that the graph depicting the factorization does not contain cycles, this means that the SPA developed on it will lead to the exact marginals for the variables. We report now the general SPA algorithm concerning this channel model.

Sum-product algorithm for the Wiener-Levy phase model

1. Initialization:

$$\mu_{x_i \to f_i}(x_i) = p(x_i) \quad \forall\, i = 1, \ldots, N$$

$$\mu_{y_i \to f_i}(y_i) = 1 \quad \forall\, i = 1, \ldots, N$$

$$\mu_{p \to \theta_1}(\theta_1) = p(\theta_1)$$

2. Messages from factor nodes f_i toward variable nodes θ_i:

$$\mu_{f_i \to \theta_i}(\theta_i) = \int_{\sim\{\theta_i\}} f_i(y_i, x_i, \theta_i)\mu_{x_i \to f_i}(x_i)\mu_{y_i \to f_i}(y_i) \quad \forall\, i = 1,\, \ldots,\, N$$

3. Forward messages and backward messages computed in parallel:

$$\mu_{\theta_1 \to p_{2,1}}(\theta_1) = \mu_{f_1 \to \theta_1}(\theta_1)\mu_{p \to \theta_1}(\theta_1)$$

$$\mu_{\theta_N \to p_{N,N-1}}(\theta_N) = \mu_{f_N \to \theta_N}(\theta_N)$$

$$\mu_{p_{i,i-1} \to \theta_i}(\theta_i) = \int_{\sim\{\theta_i\}} p_{i,i-1}\mu_{\theta_{i-1} \to p_{i,i-1}}(\theta_{i-1}) \quad \forall\, i = 2, \ldots, N$$

$$\mu_{p_{i,i-1} \to \theta_{i-1}}(\theta_{i-1}) = \int_{\sim\{\theta_{i-1}\}} p_{i,i-1}\mu_{\theta_i \to p_{i,i-1}}(\theta_i) \quad \forall\, i = N, \ldots, 2$$

$$\mu_{\theta_i \to p_{i+1,i}}(\theta_i) = \mu_{p_{i,i-1} \to \theta_i}(\theta_i)\mu_{f_i \to \theta_i}(\theta_i) \quad \forall\, i = 2, \ldots, N-1$$

$$\mu_{\theta_i \to p_{i,i-1}}(\theta_i) = \mu_{p_{i+1,i} \to \theta_i}(\theta_i)\mu_{f_i \to \theta_i}(\theta_i) \quad \forall\, i = N-1, \ldots, 2$$

4. Messages from variable nodes θ_i toward factor variables f_i:

$$\mu_{\theta_1 \to f_1}(\theta_1) = \mu_{p \to \theta_1}(\theta_1)\mu_{p_{2,1} \to \theta_1}(\theta_1)$$

$$\mu_{\theta_N \to f_N}(\theta_N) = \mu_{p_{N,N-1} \to \theta_N}(\theta_N)$$

$$\mu_{\theta_i \to f_i}(\theta_i) = \mu_{p_{i,i-1} \to \theta_i}(\theta_i)\mu_{p_{i+1,i} \to \theta_i}(\theta_i) \quad \forall\, i = 2,\, \ldots,\, N-1$$

5. Messages from factor nodes f_i toward variable nodes x_i:

$$\mu_{f_i \to x_i}(x_i) = \int_{\sim\{x_i\}} f_i(y_i, x_i, \theta_i)\mu_{\theta_i \to f_i}(\theta_i)\mu_{y_i \to f_i}(y_i) \quad \forall\, i = 1, \ldots, N$$

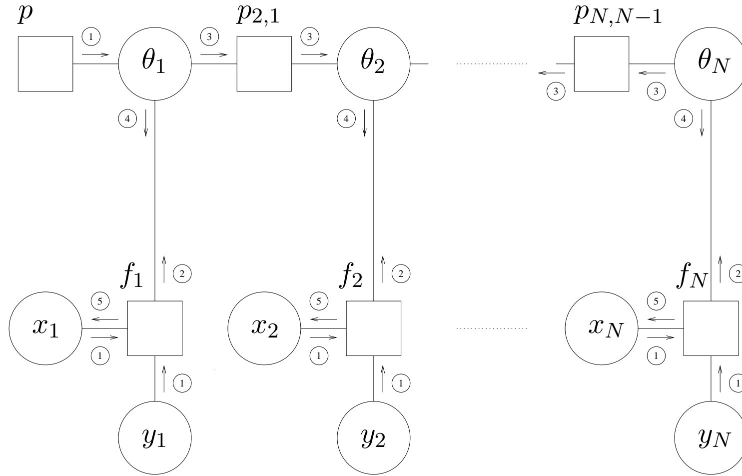

Figure 5.4: SPA message scheduling.

In Fig. 5.4 we have depicted the message scheduling. After these steps the block can propagate the "beliefs" of the variables x_k to the demapping and the decoding blocks. It is also possible to compute the marginal probability density function for all θ_i as a product of the incoming messages and obtaining a phase estimation by the *MAP* criterion.

$$\begin{aligned} p(\theta_1) &= \mu_{p\to\theta_1}(\theta_1)\mu_{f_1\to\theta_1}(\theta_1)\mu_{p_{2,1}\to\theta_1}(\theta_1) \\ p(\theta_N) &= \mu_{f_N\to\theta_N}(\theta_N)\mu_{p_{N,N-1}\to\theta_N}(\theta_N) \\ p(\theta_i) &= \mu_{f_i\to\theta_i}(\theta_i)\mu_{p_{i,i-1}\to\theta_i}(\theta_i)\mu_{p_{i+1,i}\to\theta_i}(\theta_i) \quad \forall\, i = 2,\ldots,N-1 \end{aligned}$$

$$\hat{\theta}_i^{MAP} = \arg\max_{\theta_i} p(\theta_i) \quad \forall\, i \tag{5.10}$$

Approximation methods

The summary operations in Steps 3) and 4) lead to intractable integrals since continuous variables are involved. In this case does not exist a closed form solution so we need an approximative integration methods. In the exhaustive work did in [8] more possibilities can be found in this regard. Among all the proposals we have picked:

- Quantization.
- Gradient algorithm.

They have been suggested by our final goal of finding a good parallel channel estimation algorithm running on the graphic card. In the following sections we will explain the resulting algorithms from a general point of view and we will motivate these choices. A detailed description of the parallel implementation will be give in Chapter 7.

5.5 Quantized Sum Product Algorithm

The most natural way to circumvent the integration problem is by quantization of the domains of the continuous variable. In this way we obtain the quantized version of the functions and the messages. Essentially the probability density functions are approximated with probability mass functions. The basic idea is that continuous variables are quantized and after that, the integrals over the whole range can be replaced by a finite sums. There exist several quantization techniques, and basically they differ depending on how the quantization levels are chosen.

The simplest method is the *uniform* quantization that leads the simplest numerical integration rule, i.e., the *rectangular rule.* As example consider the simple factor graph involving three variable nodes, namely x_1, x_2 and y, and a factor node $f(y, x_1, x_2)$ (see Fig. 5.5).

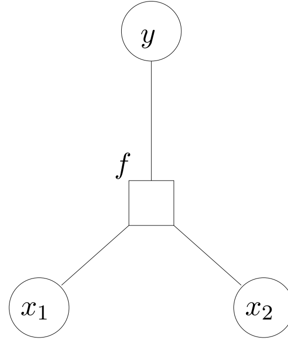

Figure 5.5: Factor graph of the function $f(y, x_1, x_2)$.

The variables x_1 and x_2 are continuous and we decide to quantize their domains with L levels. The general integral-product rule in the SPA algorithm can be evaluated as follow

$$\begin{aligned} \mu_{f\to y}(y) &= \int_{x_1}\int_{x_2} f(y, x_1, x_2)\mu_{x_1\to f}(x_1)\mu_{x_2\to f}(x_2)dx_1dx_2 = \\ &\approx \sum_{i=0}^{L-1}\sum_{i=0}^{L-1} f(y, x_1^{(i)}, x_2^{(i)})\mu_{x_1\to f}(x_1^{(i)})\mu_{x_2\to f}(x_2^{(i)}) \end{aligned} \tag{5.11}$$

where $x_k^{(i)}$ is the i-th quantization level of x_k. In this way the integrand

$$f(y, x_1, x_2)\mu_{x_1\to f}(x_1)\mu_{x_2\to f}(x_2)$$

is approximated as a piecewise constant function and the messages are represented by their function values at the quantization levels. This last assumption is shown in Fig 5.6.

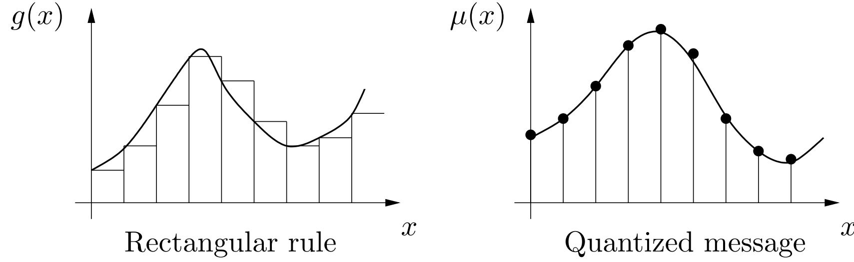

Figure 5.6: Rectangular integration rule and quantized message.

In our particular case the continuous variables that need to be quantized are θ_k. We assume that the channel phase samples take value on the following finite set:

$$\theta_k \in \left[-\pi, -\pi + \frac{2\pi}{L}, \dots, \pi \frac{L-2}{L} \right] \tag{5.12}$$

In order to obtain the quantized version of the previous algorithm we only need to replace the continuous variable with their quantized versions and the message update rules will be obtain by applying the sum product rule for discrete variables.

The algorithm obtained with the quantized version of the variable represents an instance of the SPA algorithm with only discrete variables. This means that this approach becomes "optimal", in the sense that it approaches the performance of the *exact* SPA, for a sufficiently large number of quantization levels, at the cost of an increasing computational complexity. Indeed the drawback of this technique, and of the quantization techniques in general, is that the complexity and the memory requirements scale in the number of quantization levels L.

The reason why we have chosen this particular approximation technique is the knowledge of the device (the graphic card) we will use to implement the algorithm. The resulting algorithm has a high computational complexity correlated to the number of quantization levels. On the other hand the GPU seems to allow a good performance improvement for several types of applications.

Another specialization we have thought regards the computation of the forward and backward messages. We have called this *Markov Chain update rule* and it is described in the next section.

5.5.1 Markov Chain update rule

By adopting the quantization method the messages can be represented in a computer program by discrete vectors:

$$\boldsymbol{\mu}(x) = \left[\mu(x^{(1)}), \dots, \mu(x^{(L)}) \right] \tag{5.13}$$

where L indicates the cardinality of the quantized domain decided for the variable x. Each element of the vector denotes the value of the message computed in the corresponding quantized value of x.

With this assumption what the nodes pass are the *true* messages, so the message-passing correspond to a real "belief" propagation. This means that the message-update rules are handled by standard sum product rule where the summary operation correspond to a finite sum.

We focus now on the i-th forward transaction of the Markov Chain depicted in the upper part of the graph. Due the Markovian nature of the system, one can think to obtain the messages passing through the chain by a simple evaluation of a one step transition of a Markov Chain. In Fig. 5.7 we have reported the general forward transition of the SPA with a simplified notation and where all the variables are supposed to be quantized.

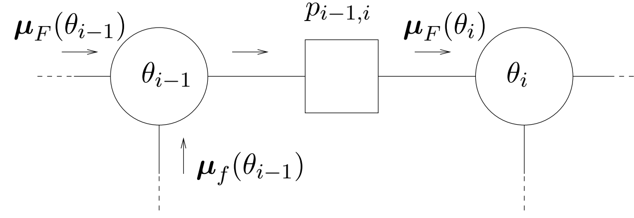

Figure 5.7: General forward message propagation of the quantized SPA.

The forward message $\boldsymbol{\mu}_F(\theta_i)$ is obtained by the sum product rule:

$$\boldsymbol{\mu}_F(\theta_i) = \sum_{j=1}^{L} p(\theta_i|\theta_{i-1}^{(j)})\mu_F(\theta_{i-1}^{(j)})\mu_f(\theta_{i-1}^{(j)}) \tag{5.14}$$

As well known, for a discrete Markov Chain, the first order transition matrix stores all the transition probability functions. By quantizing with L levels the continuous variables the matrix has dimension $L \times L$ and results as follows:

$$\mathbf{P} = \begin{bmatrix} p(\theta_i^{(1)}|\theta_{i-1}^{(1)}) & p(\theta_i^{(2)}|\theta_{i-1}^{(1)}) & \cdots & \cdots & p(\theta_i^{(L)}|\theta_{i-1}^{(1)}) \\ p(\theta_i^{(1)}|\theta_{i-1}^{(1)}) & \ddots & & & p(\theta_i^{(L)}|\theta_{i-1}^{(1)}) \\ \vdots & & p(\theta_i^{(j)}|\theta_{i-1}^{(j)}) & & \vdots \\ \vdots & & & \ddots & \vdots \\ p(\theta_i^{(1)}|\theta_{i-1}^{(L)}) & \cdots & \cdots & p(\theta_i^{(L-1)}|\theta_{i-1}^{(L)}) & p(\theta_i^{(L)}|\theta_{i-1}^{(L)}) \end{bmatrix}$$

In this way Eq. (5.14) can be write in vectorial notation as a vector $\times$ matrix product:

$$\boldsymbol{\mu}_F(\theta_i) = (\boldsymbol{\mu}_F(\theta_{i-1}) \cdot \boldsymbol{\mu}_f(\theta_{i-1}))\mathbf{P} \tag{5.15}$$

where the symbol $\cdot$ denotes the *pointwise product* between the two incoming messages (functions) in θ_{i-1}. The same considerations hold for the backward messages:

$$\boldsymbol{\mu}_B(\theta_{i-1}) = (\boldsymbol{\mu}_B(\theta_i) \cdot \boldsymbol{\mu}_f(\theta_i))\mathbf{P} \tag{5.16}$$

The reason of the last considerations is that we hope it will result in a more suitable algorithm to be implemented in our computer simulations. In particular, the knowledge of the CUDA programming model suggests us that this method could exploit the GPU compute capability for the vector $\times$ matrix product computation since it is an operation that can be parallelized. More details about the implementation will be give in Chapter 7. In the following section we report the phase estimation algorithm with quantized variables and the Markov chain update rule.

5.5.2 Quantized SPA for phase estimation

We suppose that the observation vector $\mathbf{y}$ and the input symbol estimates vector $\hat{\mathbf{x}}$ are available as inputs for the algorithm. The latter is provided by the decoding block (see Fig. 5.1). The variable θ_i are assumed quantized with L levels and take values as in Eq. 5.12. No a a priori knowledge for θ_1.

1. Messages from factor nodes f_i toward variable nodes θ_i:

$$\boldsymbol{\mu}_f(\theta_i) = f(y_i, \hat{x}_i, \theta_i) \quad \forall\, i \in [1, \ldots, N]$$

2. Forward and backward messages computed by the Markov chain update rule:

$$\boldsymbol{\mu}_F(\theta_2) = \boldsymbol{\mu}_f(\theta_1)\mathbf{P}$$

$$\boldsymbol{\mu}_F(\theta_i) = (\boldsymbol{\mu}_F(\theta_{i-1}) \cdot \boldsymbol{\mu}_f(\theta_{i-1}))\mathbf{P} \quad \forall\, i \in [3, \ldots, N]$$

$$\boldsymbol{\mu}_B(\theta_{N-1}) = \boldsymbol{\mu}_f(\theta_N)\mathbf{P}$$

$$\boldsymbol{\mu}_B(\theta_{i-1}) = (\boldsymbol{\mu}_B(\theta_i) \cdot \boldsymbol{\mu}_f(\theta_i))\mathbf{P} \quad \forall\, i \in [1, \ldots, N-2]$$

3. Marginals of the θ_i variables as pointwise product of the incoming messages:

$$p(\theta_1) = \boldsymbol{\mu}_B(\theta_1) \cdot \boldsymbol{\mu}_f(\theta_1)$$

$$p(\theta_i) = \boldsymbol{\mu}_F(\theta_i) \cdot \boldsymbol{\mu}_B(\theta_i) \cdot \boldsymbol{\mu}_f(\theta_i) \quad \forall\, i \in [2, \ldots, N-1]$$

$$p(\theta_N) = \boldsymbol{\mu}_F(\theta_N) \cdot \boldsymbol{\mu}_f(\theta_N)$$

4. Phase estimation by using the MAP criterion:

$$\hat{\theta}_i^{MAP} = \arg\max_{\theta_i} p(\theta_i) \quad \forall\, i \in [1, \ldots, N]$$

After these steps we obtain a phase estimation, i.e, the phase sample θ_i is associated to a quantized value in the range. It is clear that the estimation quality depends on the number of quantization levels adopted and a quantization error floor is unavoidable. For a performance evaluation of the estimator we refer to Section 7.3.

5.6 Steepest Ascent algorithm

A probability density function can be represented by a single or a set of values describing its properties. The typical example is the Gaussian distribution which can be completely described by the knowledge of its mean and its variance. However a general function $f(x)$ can be represented with a single value such as:

- its mode;
- its mean;
- its median;
- a sample value from f.

In our case we are interested in the *mode* of the function which is defined as follows:

$$x_{MAX} = \arg\max_x f(x)$$

The probability density function is then approximated by the *Dirac delta* $\delta(x - x_{MAX})$ as depicted in Fig. 5.8.

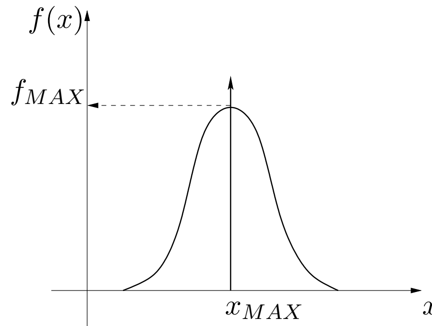

Figure 5.8: A probability density function and its approximation by a Dirac delta located at its mode.

Since the messages exchanged in our factor graph are basically probability density functions this assumption leads us to represent them as *single values.* Consider the example done in Section 5.5 and illustrated in Fig. 5.5, the sum product rule becomes

$$\mu_{f\to y}(y) = f(y, \hat{x}_1, \hat{x}_2) \tag{5.17}$$

where $\hat{x}_1$ and $\hat{x}_2$ are the modes of the two incoming messages for the factor node f, i.e., $\mu_{x_1\to f}(x_1)$ and $\mu_{x_2\to f}(x_2)$. Note that we have used the *hat* notation since they are the hard estimates of the variables x_1 and x_2.

We can think now that for each variable node in the graph, if its marginal function is available, its mode correspond to the MAP estimation of the variable. We already know that, if a closed-form solution for the MAP estimate does not exist, we need a method to find it. It might be found iteratively by a *gradient method* and, as we can see, this approach can also be viewed as a message passing working on the corresponding factor graph [8]. We briefly review this family of algorithms.

5.6.1 Gradient algorithms

Consider a general function $f(\mathbf{x})$ defined $\mathbb{R}^n \to \mathbb{R}$. One method for *optimizing* a function, which means in general to minimize or maximize it, is to iterate in such a way that:

$$f(\mathbf{x}^{(j+1)}) < f(\mathbf{x}^{(j)})$$

unless:

$$f(\mathbf{x}^{(j+1)}) = f(\mathbf{x}^{(j)})$$

in which case an extremum point is reached. The index j denotes the j-th iteration.

A common way to accomplish this is to update the argument vector $\mathbf{x}^{(j)}$ in each iteration by applying the following general rule:

$$\mathbf{x}^{(j+1)} = \mathbf{x}^{(j)} + \lambda_j \mathbf{g}_j$$

where λ_j is a scalar called *step size*, and $\mathbf{g}_j$ is a direction of motion, selected so that the successive steps decrease the function f. Depending on the vector $\mathbf{g}_j$ selected, we can get different gradient algorithms like *steepest descent(ascent)*, *conjugate-gradient descent(ascent)*, or other. Before specializing this general framework in our case it is worth to point out that iterating the general rule will not necessarily reach the global minimum of the function. For example consider the function illustrated in the next figure.

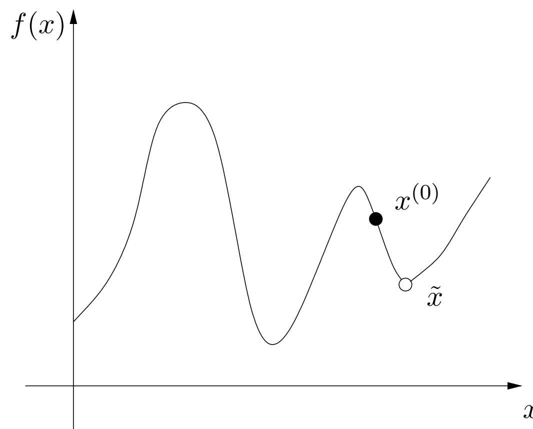

Figure 5.9: A function with a local and a global minimum.

It is clear that depending on the starting point $x^{(0)}$ this method can lead to different results. In the situation depicted in Fig. 5.9 the algorithm will reach the local minimum $\tilde{x}$ and not the global minimum.

5.6.2 A parallel implementation

The final goal for the phase estimation is reached by maximizing the joint probability function in $\boldsymbol{\theta}$, i.e. finding the phase vector that leads to its largest value. A widely

used method for maximizing a function f is the *steepest ascent*, which is applicable to differentiable functions.

Let us consider the joint probability function where $\mathbf{y}$ and $\hat{\mathbf{x}}$ act as parameters, so that we can write $p(\mathbf{y}, \hat{\mathbf{x}}, \boldsymbol{\theta}) = f(\boldsymbol{\theta})$. By adopting the steepest ascent algorithm the following general update rule is obtained:

$$\boldsymbol{\theta}^{(j)} = \boldsymbol{\theta}^{(j-1)} + \lambda_j \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})|_{\boldsymbol{\theta}^{(j-1)}} \tag{5.18}$$

Alternatively, since the logarithm is a monotonic function, we can also consider the rule:

$$\boldsymbol{\theta}^{(j)} = \boldsymbol{\theta}^{(j-1)} + \lambda_j \nabla_{\boldsymbol{\theta}} \log f(\boldsymbol{\theta})|_{\boldsymbol{\theta}^{(j-1)}} \tag{5.19}$$

Due to the particular nature of the problem we know that the joint probability density function can be nicely factorized in order to be represented as a factor graph. This fact can also be exploit to find a parallel phase estimation algorithm based on the steepest ascent. In order to do this we focus on the generic θ_i variable node illustrated in Fig. 5.10.

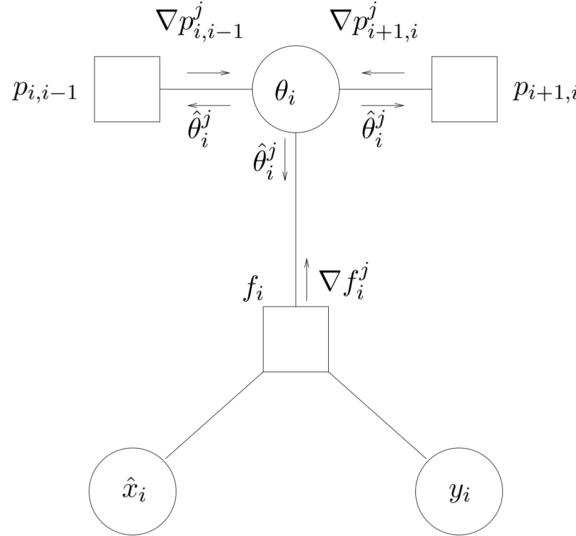

Figure 5.10: Local Steepest Ascent computation

The function depicted in the picture is:

$$g(\theta_i) = f_i(y_i, \hat{x}_i, \theta_i) p(\theta_i|\theta_{i-1}) p(\theta_{i+1}|\theta_i) \tag{5.20}$$

By taking the logarithm:

$$\log g(\theta_i) = \log f_i(y_i, \hat{x}_i, \theta_i) + \log p(\theta_i|\theta_{i-1}) + \log p(\theta_{i+1}|\theta_i) \tag{5.21}$$

Now one can think to apply the steepest ascent algorithm *locally* for the function in (5.21). Straightforward mono-dimensional application of the rules in Eq. (5.19) leads to:

$$\theta^{(j)} = \theta^{(j-1)} + \lambda_j \nabla_{\theta_i} \log g(\theta_i)|_{\theta_i^{(j-1)}} \tag{5.22}$$

The gradient is obtained by summation of three terms, each corresponding to one factor node:

$$\nabla_{\theta_i} \log g(\theta_i) = \frac{\partial \log f_i(y_i, \hat{x}_i, \theta_i)}{\partial \theta_i} + \frac{\partial \log p(\theta_i|\theta_{i-1})}{\partial \theta_i} + \frac{\partial \log p(\theta_{i+1}|\theta_i)}{\partial \theta_i} \tag{5.23}$$

The three terms constituting the gradient are the following:

$$\frac{\partial \log f_i(y_i, \hat{x}_i, \theta_i)}{\partial \theta_i} = \frac{\Im\left\{y_i \hat{x}_i^* A e^{-j\theta_i}\right\}}{\sigma_\omega^2} \tag{5.24}$$

$$\frac{\partial \log p(\theta_i|\theta_{i-1})}{\partial \theta_i} = -\frac{\theta_i - \theta_{i-1}}{\sigma_p^2} \tag{5.25}$$

$$\frac{\partial \log p(\theta_{i+1}|\theta_i)}{\partial \theta_i} = \frac{\theta_{i+1} - \theta_i}{\sigma_p^2} \tag{5.26}$$

The same reasoning can be applied to all the phase variables in the factor graph resulting in a local update rule for each of them. As we will see this consideration will be exploited in the parallel implementation on the graphic card. Indeed since we split a global optimization problem (Eq. (5.19)) in N local optimization problems we are applying the basic concept of *data parallelism* (see Section 6.2): working on different data with the same operations and at the same time.

Only one issue remains to be covered, the choice of the initial guess $\boldsymbol{\theta}^{(0)}$. As we told the starting point for a gradient algorithm is important for the final result. In this regard we decide to adopt as initial guess for our steepest ascent algorithm the *Maximum Likelihood estimate* of the phase vector (see Chapter 4). In this way we are quite sure the algorithm will start from a point near to the global maximum. Of course this does not represent the optimum choice, indeed in presence of a bad scenario the resulting poor ML estimate could force the algorithm to start from a point far from the desired solution.

Finally we will adopt for our algorithm a constant step size for each iteration. The parameter λ_j in Eq. (5.19) determines how far we move at step j and it should depend on the iteration index. Frequently, steepest ascent algorithms use $\lambda_j = \lambda$ for some constant λ. There exist some versions of the gradient algorithm that optimize the step size at each iteration but for our purposes it is sufficient to keep it fixed. An optimization of the step size for our particular case will be probably future work.

This algorithm does not represent an instance of the SPA working on a factor graph but, since it is suggested by the factorization of the joint probability density function, one can think to see it as a message passing algorithm on the corresponding graph. At each j-th iteration the variable nodes broadcast the current estimate $\theta_i^{(j)}$ to the neighbor factor nodes which reply with the corresponding term of the local gradient computed in the current estimated value. After that the local update rule is performed. The phase estimation is obtain when convergence is reached.

In the next section we report the specific algorithm we have developed for the phase estimation based on the steepest ascent method and in Fig 5.11 a details of particular message passing in the upper part of the graph.

5.6.3 Steepest Ascent algorithm for phase estimation

We suppose that the observation vector $\mathbf{y}$ and the input symbol estimates vector $\hat{\mathbf{x}}$ are available as inputs for the algorithm. The latter is provided by the decoding block (see Fig. 5.1). No a priori knowledge for θ_1 is available.

1. Initialization with ML estimation:

$$\boldsymbol{\theta}^{(0)} = \hat{\boldsymbol{\theta}}^{ML} = \arg\max_{\boldsymbol{\theta}} \log p(\mathbf{y}|\hat{\mathbf{x}}, \boldsymbol{\theta})$$

All nodes θ_i broadcast their initial guess $\theta_i^{(0)}$.

2. Iterative part $\forall\, i \in [1, \dots, N]$, set $j = 1$.

a) Each factor node computes its corresponding term of the local gradient:

$$\frac{\partial \log f_i(y_i, \hat{x}_i, \theta_i)}{\partial \theta_i}\Big|_{\theta_i^{(j-1)}}$$

$$\frac{\partial \log f_i(y_i, \hat{x}_i, \theta_i)}{\partial \theta_i}\Big|_{\theta_i^{(j-1)}}$$

$$\frac{\partial \log p(\theta_{i+1}|\theta_i)}{\partial \theta_i}\Big|_{\theta_i^{(j-1)}}$$

and sends back the result to its neighbors.

b) Each node θ_i updates its estimate by the local update rule:

$$\theta^{(j)} = \theta^{(j-1)} + \lambda_j \nabla_{\theta_i} \log g(\theta_i)|_{\theta_i^{(j-1)}}$$

c) Set $j = j + 1$ and back to a) until convergence.

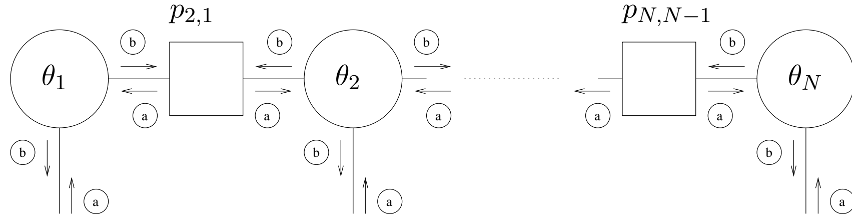

Figure 5.11: Steepest Ascent as message passing.

5.7 Performance comparison

Until now we have developed two different phase estimation algorithms working on the factor graph of the receiver. In order to obtain an indication on the quality of the estimate we report the results of a simulation campaign. Since it mimics the one did for the ML estimators we remind the general simulation parameters of Section 4.5. The specific parameters adopted for the two proposed algorithms are:

- Number of quantization levels for Quantized SPA, $L = 256$.
- Number of iteration for Steepest ascent, $N_{it} = 1000$.
- Fixed step size $\lambda = 0.03$. It has been experimentally evaluated.

Since all simulations reported in this thesis were done with the graphic card the parameter chosen have not resulted in a particular long simulation time.

In order to have a first qualitative evaluation of the two different methods, we report in the following the phase estimation performed on one realization of the phase process (Fig. 5.12).

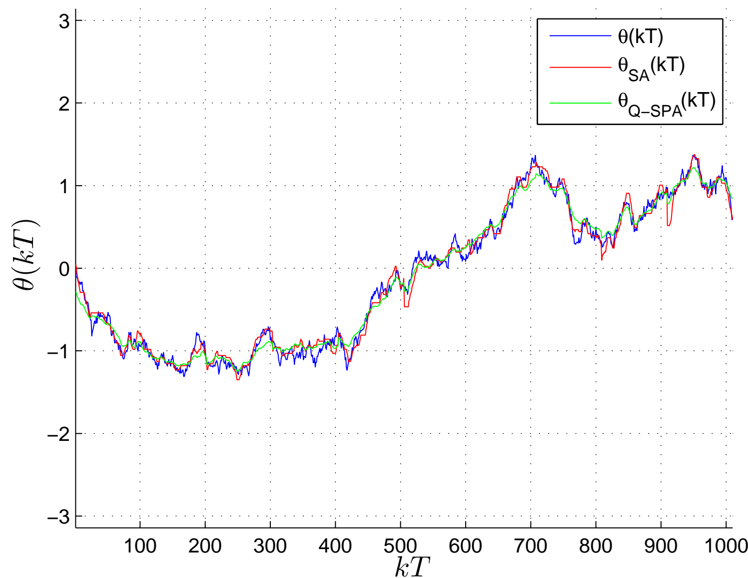

Figure 5.12: Factor graph estimations, $SNR = 7$ dB, $\sigma_p^2 = 7(°)^2$, $N = 101$ samples.

As it can be observed, the two estimates lead to similar results, as the MSE are respectively:

$$MSE_{Q-SPA} = 0.00943 \qquad MSE_{SA} = 0.0099$$

In order to have valid indications on the estimators performance, we have carried out a similar study to the one for the ML estimators of Chapter 4. The MSE is evaluated as a function of the thermal noise (SNR) and the phase noise (σ_p^2) for different scenarios.

MSE as a function of SNR

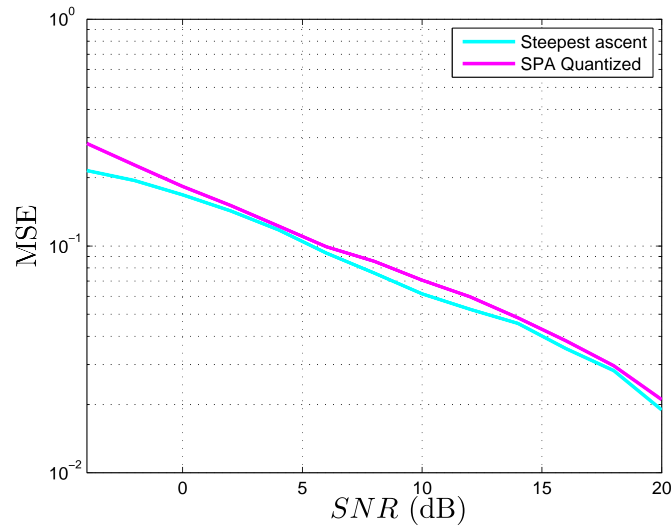

Figure 5.13: MSE as a function of SNR, $\sigma_p^2 = 25(^\circ)^2$.

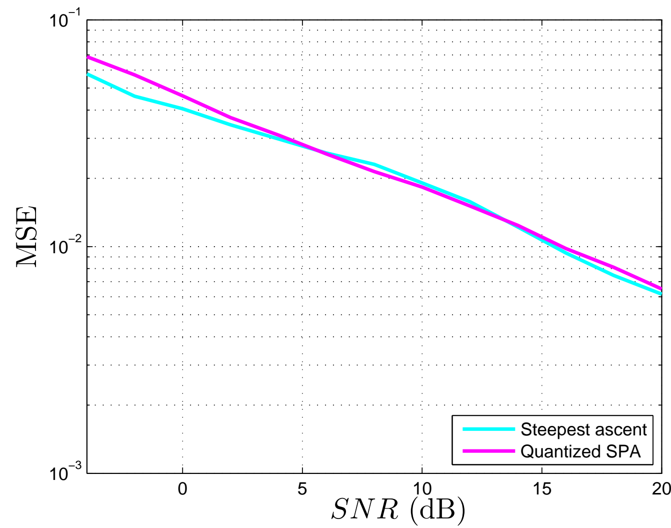

Figure 5.14: MSE as a function of SNR, $\sigma_p^2 = 5(^\circ)^2$.

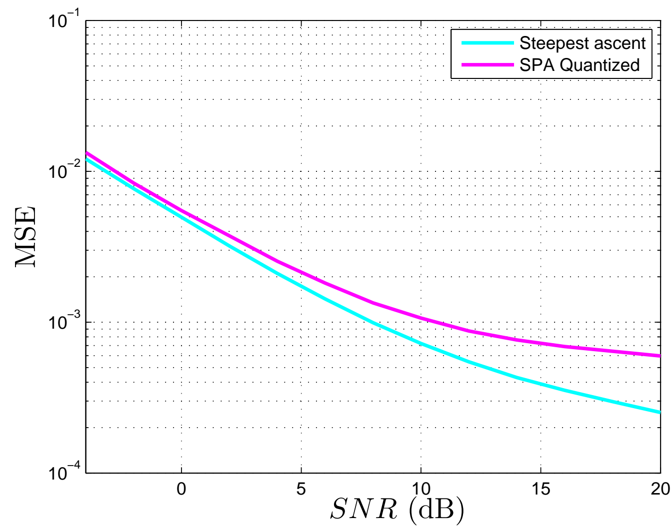

Figure 5.15: MSE as a function of SNR, $\sigma_p^2 = 0.1(°)^2$.

MSE as a function of σ_p^2

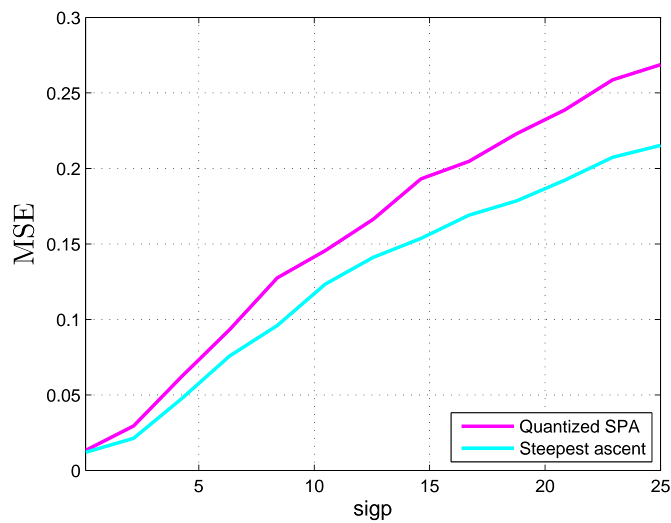

Figure 5.16: MSE as a function of σ_p^2, SNR = -4 dB.

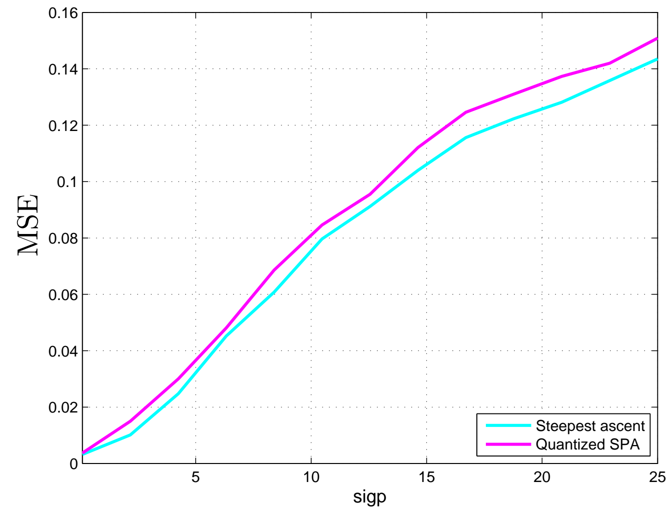

Figure 5.17: MSE as a function of σ_p^2, $SNR = 2$ dB.

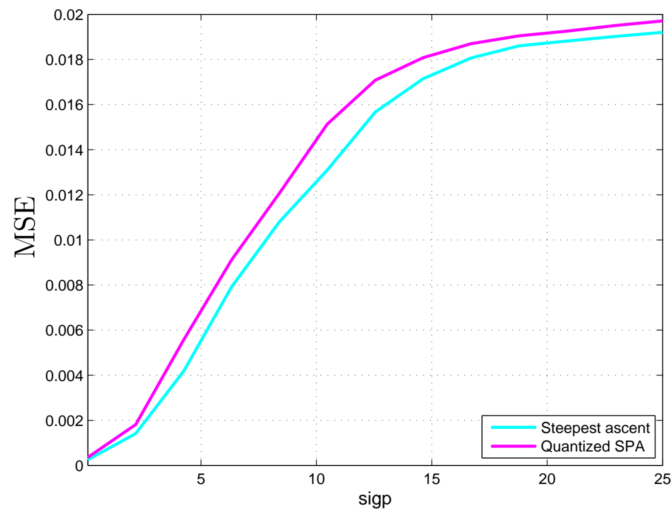

Figure 5.18: MSE as a function of σ_p^2, $SNR = 20$ dB.

As we can see the performance of the two estimators for different SNR are quite similar (see Fig. 5.13 - Fig. 5.13). The reason is that they are different approximations of the same MAP criterion applied for phase estimation. It can be observed that for low SNR the Steepest Ascent algorithm leads to a slightly better estimation in all the cases proposed. This issue added to the well known low complexity of the algorithm lead us to prefer it rather than the Quantized SPA for bad thermal noise environment. In Fig. 5.15 it can be observed how for high SNR the unavoidable quantization error in the quantized algorithm becomes dominant over the estimation error and leads to a worst estimate for the process. This is due to the fact that, in such kind of good scenario, the resulting quantized estimate it is not sufficiently refined and, in order to increase the estimation quality, for low phase noise it is required to increase the number of quantization levels. The next series of pictures (Fig. 5.16 - Fig. 5.18) confirms these observations and reveals that the Steepest Ascent algorithm is more robust to the phase noise than the Quantized SPA.

5.8 Conclusions

In this chapter we have presented two different phase estimation algorithms, both obtained through the factor graph approach.

First of all we have presented the system model and shown how it can be nicely translated into a factor graph depicting the joint probability density function of all the variables involved (Fig. 5.1). After that, through a first simple example where we have adopted the Constant phase model, we have shown how, in that particular case, since there exists a closed form solution (equal to the ML estimator), the factor graph approach does not result useful.

We have passed to a more refined stochastic model, i.e. the Wiener-Levy phase model, and we have seen how its Markovian nature can be highlighted by depicting it as a factor graph (Fig. 5.3). We have reported the general Sum Product Algorithm working on the graph and the message schedule that, since the graph is cycle-free, leads to the exact marginals for each variable.

Since the SPA requires the computation of intractable integrals over continuous range we need to approximate them. Keeping in mind the final parallel implementation on the graphic card, among all the proposals in [8], we have chosen two different approximation methods: *Quantization* and *Gradient algorithm*.

For both methodologies some tricks were devised in order to obtain a suitable parallel implementation. Without going into details on how the algorithm will be finally implemented on the GPU (purpose of Chapter 7), we have presented the *Markov Chain update rule* for the Quantized SPA. In this way the forward and backward messages are obtained by vector $\times$ matrix products which are operations suitable to be parallelized.

Regarding the Gradient based algorithm, we have developed an iterative *Steepest Ascent* algorithm for the phase estimation which performs a local and distributed optimization on each phase sample. In this way we have split a global problem

in N sub-problems applying the basic concept of *data parallelism* (see Section 6.2), work on different data with the same operation independently and at the same time. The resulting algorithm, in contrast with the Quantized SPA, does not represent an instance of the SPA but can be viewed as a message flow through the graph.

Finally as already done for the ML estimators we have presented a simulation campaign where the performance of the two factor graph estimators have been compared in terms of MSE. In particular as a function of the SNR and the phase noise variance σ_p^2 for different scenarios. We have observed how the two estimators lead to quite similar results since they are different approximation of the same MAP criterion for the phase estimation. However in some limit cases the unavoidable quantization error for the Quantized SPA determines a slightly worst estimate than the Steepest Ascent.

Chapter 6

An introduction to CUDA

With CUDA is referred at the same time to a general purpose parallel programming architecture and a programming model and instruction set provided by Nvidia in November 2006. Without the ambition to be exhaustive, with this chapter we give an introduction to CUDA in order to provide a basic background for the understanding of the next chapter where is described the parallel implementation of the channel estimation algorithms.

We start by reporting the reasons why the modern graphic cards can be view as massively parallel processors and a short description of the CUDA hardware architecture. After that the basic concepts of the programming model will be presented. The description in the following is based on [18] and [16], we refer the reader to them for further details.

6.1 GPUs as Massively Parallel processors

Since the early 2000 years, the semiconductor industry has settled two main roads for designing microprocessors. The *multi-core* trajectory seeks to maintain the execution speed of sequential program while moving into multiple cores. Starting with two-core processors, the number of cores approximately doubling with each semiconductor process generation. In contrast, the *many-cores* trajectory, is more focused on the execution throughput of parallel execution. The many-cores began as a large number of much smaller cores, and, once again, the number of cores doubles with each generation.

The *Graphic Processor Unit* (GPU) belong to this latter family and, since 2003, they have led the race of floating point performance. As shown in Fig. 6.1, while the performance improvement of general-purpose microprocessor has slowed significantly, the GPUs have continued to improve relentlessly. As we can see the ratio between the GPU and CPU, in terms of floating point computation throughput, is about 10 to 1. This is not necessarily the achievable applications speeds but is merely the raw speed that the execution resources can potentially support.

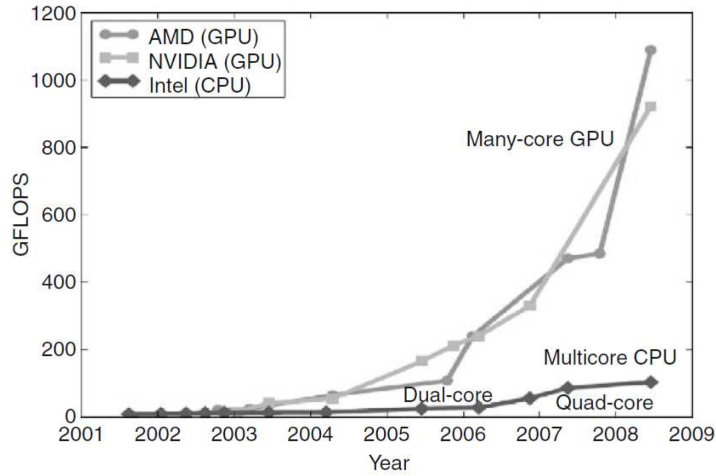

Figure 6.1: Performance gap between GPUs and CPUs [18].

The reason for this large performance gap lies in the differences in the fundamental design philosophies between the two types of processors. The design of a modern CPU is optimized for sequential code by using sophisticated control logic. Furthermore, large cache memories are provided to reduce the instruction and data access latencies of high complex applications. The actual general purpose multi-core processors have four large processor cores designed to deliver strong sequential code performance. On the other hand, the design philosophies of GPUs is driven by the fast growing video game industry, which makes pressure for the ability to perform a massive number of floating-point calculations per video frame in advanced video games. This has resulted in a maximization of the chip area dedicated to floating point calculations. This differences are schematically showed in Fig. 6.2.

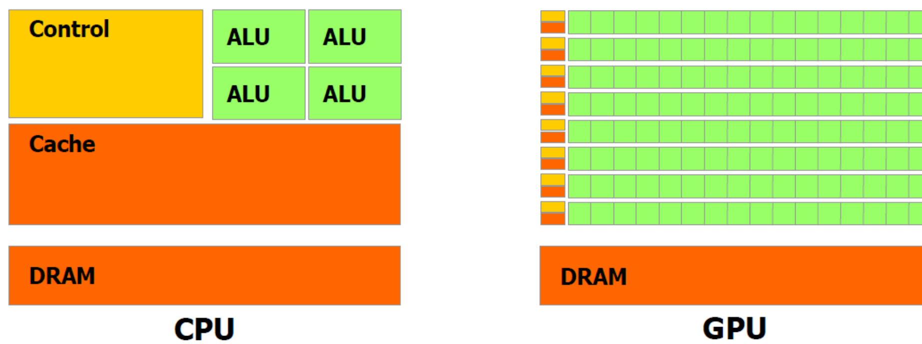

Figure 6.2: Different architectures for CPU and GPU [16].

Architecture of a CUDA GPU

A modern CUDA GPU is organized into an array of highly threaded *streaming multiprocessors* (SMs). In Fig. 6.3 two SMs form a building block, the number of SMs in a specific building block can vary from one generation of CUDA GPUs to another. Each streaming multiprocessor has a number of *streaming processors* (SPs) that share control logic and instruction cache. Since each SP is massively threaded it can run thousands of threads per application. A good application typically runs 5000-12000 threads simultaneously on these chips. We note that actual CPUs, depending on the model, supports from 2 to at most 8 threads.

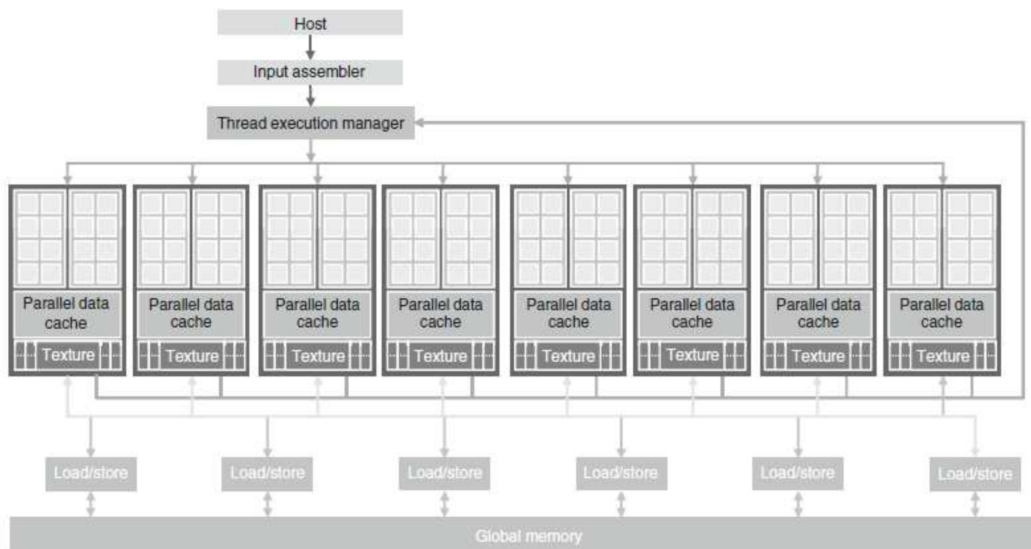

Figure 6.3: Architecture of a CUDA-capable GPU [18].

Each graphic card has its own graphics double data rate DRAM (GDDR) referred to as *global memory*. This memory differs from the system DRAM on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics. While for graphic applications it holds video images, textures for 3D models etc., for computing it acts as a very high bandwidth off-chip memory. Modern GPUs can transfer data from and to this memory at more than 140 GBps.

The communications bandwidth between the CPU mother board and the GPU through the PCI EXPRESS interface is much lower, and it may seem like a limitation in some cases. For this reason the data transfers between the CPU and the GPU should be minimized in the applications.

6.2 Programming Model, basic concepts

In this section are presented the basic concepts of the CUDA software model. Our purpose is not to give a detailed description of the programming model proposed by Nvidia, but just to indicate the basics. For further readings about this topic we refer to [16]. Before describing the programming model we want to discuss the concept of *data parallelism* since it has an important role in the development of the parallel algorithms.

Data parallelism

With data parallelism is referred the property of an algorithm whereby many arithmetic operations can be safely performed on the data structures independently and at the same time. This concept can be nicely illustrate by the *matrix - matrix multiplication* example[18].

Let be $\mathbf{A}$ and $\mathbf{B}$ two matrices and $\mathbf{C}$ the matrix obtained by the matrix multiplication of the two. The operation is schematically showed in Fig. 6.4.

$$\mathbf{AB} = \mathbf{C}$$

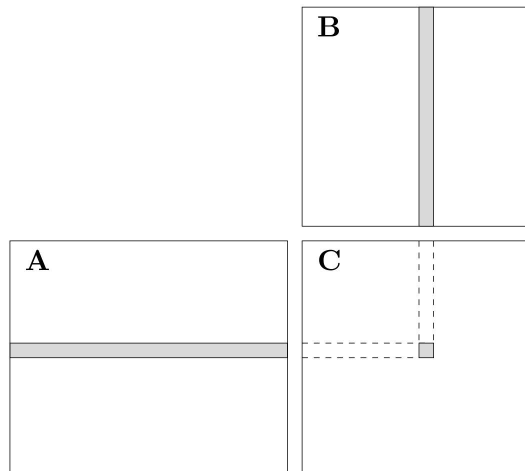

Figure 6.4: Data parallelism in matrix multiplication.

Each element of the result is obtained by performing the *dot product* between a row of the input matrix $\mathbf{A}$ and a column of the input matrix $\mathbf{B}$. This dot product operations can be simultaneously performed for each element of $\mathbf{C}$ and note that non of the dot products affects the result of each other. Therefore matrix multiplication of large dimensions can have a large amount of data parallelism. The *Markov Chain update rule* (Section 5.5.1) has been suggested by these considerations.

Heterogeneous Programming

In CUDA jargon a system consists of a *host*, which is a traditional central processing unit (CPU), and one or more *devices*, which are massively parallel processors equipped with a large number of execution units.

A CUDA program is constituted of one or more parts that are executed on either the host or a device. The code that exhibits small or no data parallelism is implemented on the host. In contrast the phases that are suitable to be parallelized are implemented in the device code and executed on the GPU. A CUDA program is a unified source code gathers both the host code and the device code. While the host code is written in canonical ANSI C, the device code is written in *CUDA C* which is an extension of the ANSI C with keywords for labeling data parallel functions and their associated data structures.

The typical execution of a CUDA program is shown in Fig. 6.5. The execution starts on the host. When a *kernel function* is launched the execution is moved to a device where a large number of threads run. All the kernels are collected in *grid*. How this grid is organized will be discuss later. Finally when all the threads have completed the execution the corresponding grid terminates, and the program continues on the host until another kernel is invoked.

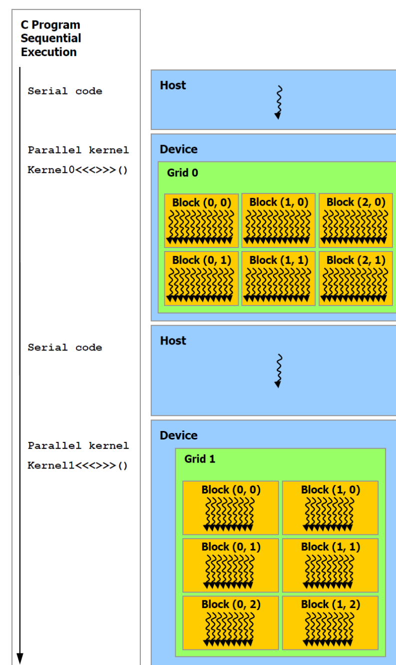

Figure 6.5: Heterogenous programming model, serial code executed on the host and parallel code executed on the device [16].

Kernels

CUDA C extends the canonical ANSI C and leads the programmer to define new functions that, when invoked, are executed N times in parallel by N different CUDA threads. This functions are known as *kernels* and they are showed on Fig. 6.6.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

Figure 6.6: Kernel definition and invocation in CUDA C [16].

In the upper part is reported the kernel definition. As we can see it is defined like a normal C function with in addition the __**global**__ specifier. It indicates the function being declared is a CUDA kernel function and it will be executed only on the device. It can only be called from the host to generate a grid of threads on a device.

In CUDA a kernel specifies the code to be executed by all threads during a parallel phase. In the example showed in Fig. 6.6 the kernel performs the pointwise vector addition between two input vectors, namely **A** and **B** and stores the result vector **C**. In particular each thread computes the addition between two entries and stores the result in the corresponding cell of the output array.

Thread hierarchy

Since all the threads execute the same kernel code it is required a mechanism to allow them to distinguish themselves and direct themselves toward the particular parts of the data portion that they are designated to work on. Each thread that executes the kernel is associated to a unique *thread ID* that is accessible within the kernel through the built-in **threadIdx** variable (see Fig. 6.6). It is a three dimensional vector so that each thread can be identified by a one-dimensional, two-dimensional or three-dimensional index. In this way they can form one-dimensional, two-dimensional or three-dimensional *blocks* of threads. This provide a natural way to match the computation across the data in a domain such a vector, matrix or volume.

The index of a thread and its relative thread ID are related by the following relations depending on the block dimension.

Block dim.	**index**	**thread ID**
1 D	x	x
2 D	(x, y)	$x + yD_x$
3 D	(x, y, z)	$x + yD_x + zD_xD_y$

Table 6.1: Relation between thread index and thread ID.

In the table above D_x, D_y denote the x and y block dimensions. There exists also D_z which is not shown.

Since all threads of a block reside on the same SP they must share the limited memory resources of that core. For this reason the number of threads per block is limited. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Blocks are organized into a one-dimensional or two-dimensional *grid* of thread blocks as illustrated by Fig. 6.7. The particular choice of the number of blocks in a grid is usually suggested by the size of the data being processed or the number of processors in the system.

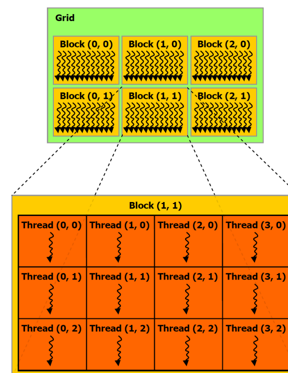

Figure 6.7: Grid of blocks [16].

The number of CUDA threads that executes that kernel is specified inside the **main()** function by using the *execution configuration syntax* indicated in Fig. 6.6 with the $<<< \ldots >>>$ brackets. The first parameter denotes the grid dimensions, i.e., the number of blocks, while the second parameter denotes the block dimension, i.e., the number of threads constituting each block. Similarly to threads, each block in the grid can be identified by a one-dimensional or two-dimensional index accessible within the kernel through the built-in **blockIdx** variable. The dimension of the block is accessible within the kernel through the built-in **blockDim** variable.

Memory hierarchy

CUDA supports several types of memory that can be used by the programmers to achieve high performance and high execution speed in their kernel. Fig. 6.8 shows these memories. In the bottom of the figure we have the already introduced global memory and the new *constant* memory. They can be written and read by the host by calling specific Application Programming Interface (API) functions provided by Nvidia. While the global memory can be written and read by the device, the constant memory represents a short-latency, high bandwidth, *read-only* memory particularly useful when all the threads need to simultaneously access to the same data.

Registers and *shared memory* are on-chip memories. Data stored in these type of memory can be accessed and manipulated at very high speed. While registers are allocated to individual threads shared memory is allocated to blocks. All the threads within a block can access to the variables stored in shared memory resulting in a efficient means for threads cooperation. Indeed they can share input data and intermediate results of the work. Sharing data through shared memory requires the synchronization of the execution to coordinate memory accesses. In CUDA C specific synchronization points in the kernel are achieved by calling the **__syncthreads()** function. Basically it acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Finally for efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (like a L1 cache).

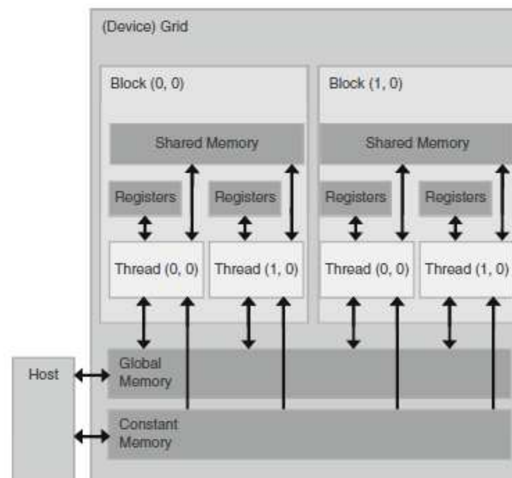

Figure 6.8: CUDA memories [18].

The description reported in this chapter represents only a glimpse to the CUDA architecture and the CUDA programming model. Further details can be found in the official documentation provided by Nvidia [16], [17], and also in [18] and [19].

Chapter 7

Parallel implementation on Graphic Card

The purpose of this chapter is to describe the implementation on the graphic card. As we will see some of the channel estimation algorithms developed are more suitable than others for parallel implementation. Some of the recommendations proposed in [17] and [16] have been adopted in order to obtain a good parallelization.

The chapter starts with a briefly illustration of the basic concepts to take under consideration when approach the development of algorithms on the graphic card and some technical specifications of the device we have used, the TESLA C2070 computing system. After that some details on how we have implemented in parallel the channel estimation algorithms will be give and finally is presented a performance comparison between the serial implementation on the CPU and the parallel implementation on the GPU.

7.1 Programming Guidelines and the Device

In our experience we have tried to obtain the biggest benefit as possible from the parallel implementation by assuming three basic guidelines:

- Finding ways to parallelize sequential code.
- Maximize parallel execution.
- Optimize memory usage as much as possible.

To get the improvement from CUDA we need to focus first on finding a way to parallelize our sequential algorithm, i.e., to exploit the data parallelism. The amount of performance benefit for an application depends entirely on the extent to which it can be parallelized. Code that cannot be sufficiently parallelized should run on the host otherwise it will result in excessive overhead due to transfers between the host and the device.

The maximum speedup expected by parallelizing portions of a serial program is stated by the *Amdahl's law* [17]. It says that the maximum improvement in terms of speed S of a program is

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{7.1}$$

where P is the fraction of the total serial execution time taken by the portion of code that can be parallelized, and N is the number of processors over which the parallel portion of the code runs. As we can see if we increase the number of processors the equation can be approximated into $S = 1/1 - P$. In this case if the portion of the program to parallelize is 3/4 the maximum speedup obtained over serial code is just 4. The key point asserted by the law is that the greater P, the greater is the speedup obtained. In addition, if P is small, increasing N does just a little improvement. So to get the larger lift, it is suggested to spend more effort on finding a good way to parallelize the algorithm, that is, to increase P.

To maximize the utilization we need to maximize the parallel execution. This means that the application should be structured in a way that it exposes as much parallelism as possible but also efficiently maps this parallelism to the various components of the system. At high level the application should optimize the parallel execution between the host and the devices. Basically it should assign to each processor the type of work it can do best: serial workloads to the host, parallel workloads to the device. It could happen that for a parallel workload the parallelism is broken because some threads need to synchronize in order to share data with each other. In the case the threads belong to the same block they should use the `__synchthreads()` function and share data through the shared memory. Otherwise, if they belong to different blocks, they must share data through global memory with the consequent loss of performance. The occurrence of this last kind of situation must be minimized by mapping the algorithm to the CUDA programming model in such a way that the computations that require inter-thread communication are performed within a single block as much as possible.

This also brings us to evaluate the third rule about the memory usage. The first step in maximizing overall memory throughput for the application is to minimize data transfers with low bandwidth. Basically this means that the data transfers between the host and the device since they have much lower bandwidth than data transfers in the device (see Section. 6.2). One way to accomplish this is to move code from the host to the device, even if it means running kernels with low parallelism. Intermediate data structures may be created in device memory, operated on by the device, and finally destroyed without being copied to host memory.

Further improvements can be also achieved by maximizing the usage of on-chip memories such as shared memory and constant memory. In particular the shared memory is equivalent to a user-managed cache on which the application can explicitly allocate and access.

The typical programming pattern is to have each thread of a block that:

- Loads data from the device memory to shared memory.
- Synchronizes with all the other threads of the block so that each one can safely read shared memory locations that were populated by other threads.
- Processes the data in shared memory.
- Synchronizes again if necessary to make sure that shared memory has been updated with the results.
- Writes the results back to the device memory.

The Device

In order to obtain good results it is also necessary to know the specific properties of the graphic card used. In our case the code has been developed to be executed on a Nvidia TESLA C2070. This card is based on the current CUDA architecture codenamed *Fermi*. The Fermi based GPUs, implemented with 3.0 billion transistors, features 448 CUDA cores. Each CUDA core executes a floating point or integer instruction per clock for a thread. All cores are organized in 14 Streaming Multiprocessors of 32 cores each. This GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-EXPRESS.

For our purposes it is important to know the properties of our board in terms of the CUDA programming model, i.e, maximum number of threads, of blocks etc.. The `DeviceQuery` application provided with the *Nvidia GPU Computing SDK code samples* returns all the specifics of the card. From the output of such program we can obtain the following useful information.

Total amount of global memory	5636554752 bytes
Multiprocessors x Cores/MP	4 x 32 = 448 (Cores)
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Maximum number of threads per block:	1024
Maximum sizes of each dimension of a block:	1024 x 1024 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1

Table 7.1: TESLA C2070 properties.

7.2 Parallel implementation of the algorithms

By following the basic guidelines described above we have developed our channel estimation algorithms in order to match as well as possible the properties illustrated in Tab. 7.1. In total have been implemented the following phase estimation algorithms:

- ML estimators.
- Quantized SPA.
- Steepest Ascent algorithm.

For each of them has been thought a different implementation and some specific tricks. In the following section we report the details.

7.2.1 Parallel Maximum Likelihood estimators

All the maximum likelihood estimators developed in Chapter. 4 are based on the moving average operation which is performed by a FIR filtering. Hence we are focused on the implementation of the *convolution* operation on the graphic card. In this regard we have developed two different parallel algorithms:

- Parallel time domain convolution.
- Parallel Overlap save method.

Parallel time domain convolution

A discrete time-invariant linear system, i.e., a filter, is shown in Fig. 7.1 where $\mathbf{x}$ and $\mathbf{y}$ are respectively the finite length input data sequence and output data sequence. The impulse response of the system is denoted by $\mathbf{h}$.

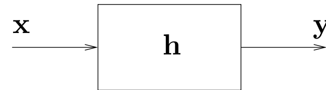

Figure 7.1: Discrete time linear system (filter).

For the system depicted the relation between a finite length input sequence $\mathbf{x}$ and the output sequence $\mathbf{y}$ is given by the convolution operation:

$$y(k) = \sum_{i=0}^{M-1} x(k-i)h(i) \qquad k \in [0, \ldots, L+M-2] \tag{7.2}$$

where L is the length of the input and M the length of the filter. Since the input sequence has finite length, in order to adopt the equation above in a computer program it is usual to perform a zero padding at the beginning and at the end with

$M-1$ zeros. In the following the input data sequence is assumed to be padded. The output data sequence has length $N_y = L + M - 1$.

Every sample of the output is obtained by a *dot product* between the impulse response **h** and a portion of the input signal of the same length, the whole input sequence is then spanned in order to obtain all the output samples. The following picture shows this operation in a graphic representation for an impulse response of length $M = 5$.

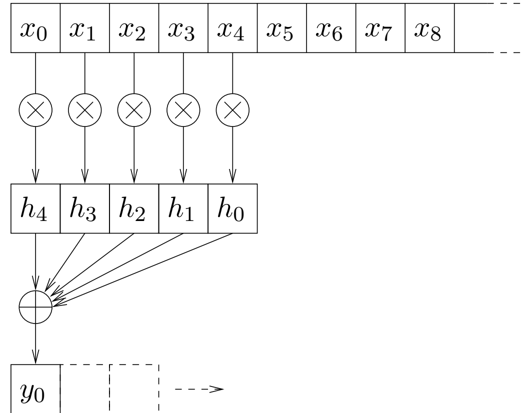

Figure 7.2: Serial convolution in time domain, $M = 5$.

In order to obtain a parallel implementation of this algorithm we need to individuate how to carry out the same operations on different data independently, i.e., to reveal the *data parallelism* of the algorithm. By observing Fig. 7.2 one can think to perform the same dot product operation simultaneously on different portions of the input data, the only requirement to consider is that these portions must not be overlapped in order to work independently. This last assumption is shown in Fig. 7.3.

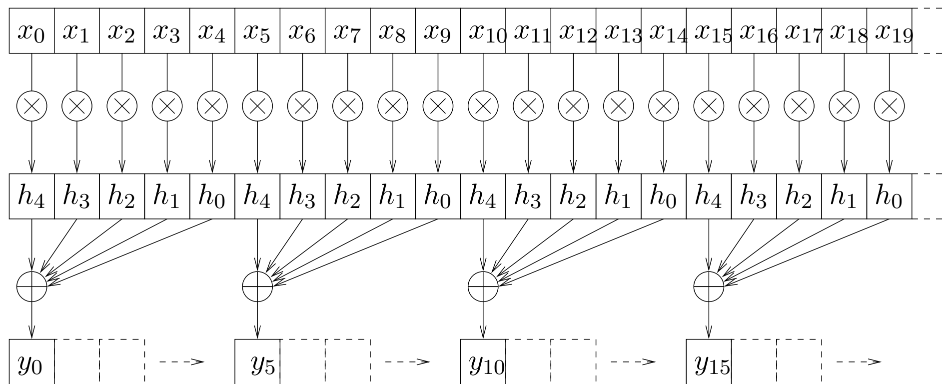

Figure 7.3: Parallel convolution in time domain, $N_b = 4$, $N_{threads} =$ M.

At this point the parallel implementation on the CUDA GPU is straightforward. We have N_b mono-dimensional blocks each one constituted by a number of threads equal to the length of the impulse response M. By working on different non-overlapped portions of the input, each block executes the dot product operation between the impulse response and its own portion of the input, independently, and at the same time. In this way instead of obtaining just one sample of the output (y_0 in Fig. 7.2) we can obtain many samples as the number of blocks, (y_0, y_5, y_{10}, y_{15} in Fig. 7.3). After that each block needs to be shifted by one sample and the next series of output samples is computed. This can be done for a number of shifts equal to the length of the impulse response. After that indeed each block would start to re-compute the output samples already computed by other blocks. This can be overcome by shifting the whole input data by a number of sample equal to $M \times (N_b - 1)$. The procedure is repeated until the input sequence is finished.

We focus now on the execution of one block, the dot product operation. The computation consists of two steps. First, we multiply corresponding elements of the two input vectors, after that we sum all the partial results to produce a single scalar output. For example for the first block in Fig. 7.3 we get an equation like the following

$$[x_0, x_1, x_2, x_3, x_4] \cdot [h_4, h_3, h_2, h_1, h_0] = x_0h_4 + x_1h_3 + x_2h_2 + x_3h_1 + x_4h_0 = y_0$$

The first step can be done with each thread of the block multiplies a pair of corresponding entries. The partial result is a vector and we decide to store it in a temporary buffer in shared memory. At this point of the algorithm we need to sum all the temporary values that have been placed in the buffer. There is a potentially danger situation that occurs when a thread seeks to read an entry in the cache before each thread has finished the pointwise multiplication. To guarantee that all the threads writes to the buffer completely before anyone tries to read we need to use the **__syncthreads()** function that assures every thread within a block has completed its instruction before its call.

Now that we have guaranteed that the temporary buffer has been filled, we need to sum the values in it. In general the process of taking an input array and performing some computations that produce a smaller array of results is called *reduction*. The serial way to accomplish this reduction would be having one thread iterates over the buffer in shared memory and calculates a running sum. This will take a time proportional to the length of the array. However this reduction can be done in parallel and takes a time that is proportional to the *logarithm of the length* of the array.

By adopting the *sequential addressing summation reduction* [21], [19] each thread adds two of the values in buffer and stores the result back to the buffer. Since each thread combines two entries into one we complete this step with half as many entries as we started with (see Fig. 7.4). In the next step this process is performed on the remaining half and repeated until only one entry remain, i.e. the dot product result. A limitation is that the length of the two vectors must be a power of 2.

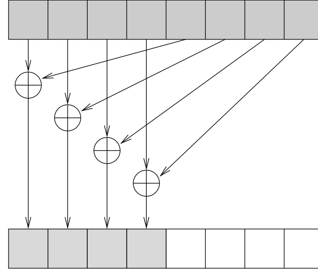

Figure 7.4: One step of the sequential addressing summation reduction.

Parallel Overlap save method

For very long input sequence the convolution can be computed on blocks of the input signal and by recombining the partial results. These techniques are based on the *Fast Fourier Transform* (FFT) algorithm and one of them is the *Overlap save method* [1]. The adoption of this algorithm has been suggested first of all because it is very suitable to be parallelized and by the availability of the very efficient CUFFT library. The availability of this library has allowed us to focus on the organization of the data structures in memory.

In this algorithm the input sequence is divided in overlapped blocks of length L where each block has the last $M-1$ samples in common with the first $M-1$ samples of the successive block. The convolution between each block and the impulse response is performed by using the FFT algorithm. After that the $iFFT$ is performed on the result and only the last $K = L-M-1$ samples is saved obtaining K samples of the output.

Let be $\mathbf{x}_i^{(L)}$ the i-th block of L samples and $\mathbf{h}$ the impulse response, the algorithm works on each block as follows:

- Padding of the impulse response in order to reach a length of L:

$$\mathbf{h}^{(L)} = [h_0, h_1, \ldots, h_{M-1}, \overbrace{0, \ldots, 0}^{L-M\ zeros}]$$

- L points FFT on the padded impulse response and on the i-th block:

$$\mathbf{H} = FFT[\mathbf{h}^{(L)}]$$

$$\mathbf{X}_i^{(L)} = FFT[\mathbf{x}_i^{(L)}]$$

- Point-wise multiplication:

$$\mathbf{Y}_i^{(L)} = \mathbf{H} \cdot \mathbf{X}_i$$

- Inverse transform and save the partial result, the first $M-1$ samples are neglected:

$$\mathbf{y}_i^{(K)} = iFFT[\mathbf{Y}_i^{(L)}] = [\overbrace{\#, \ldots, \#}^{M-1}, y_{M-1}, \ldots, y_{L-1}]$$

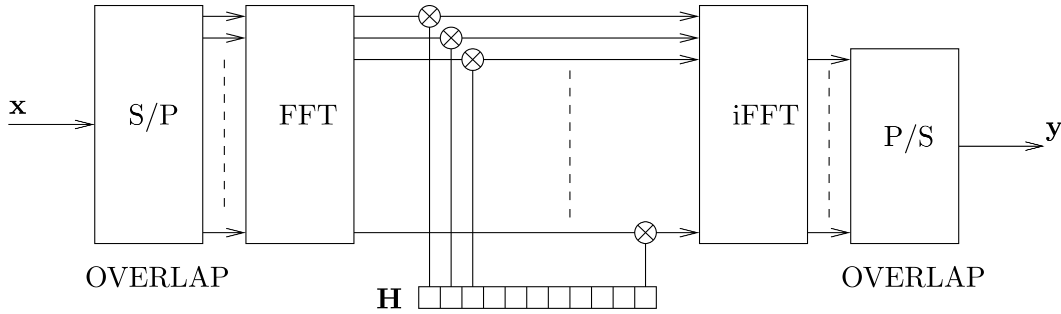

Figure 7.5: Overlap save method

The block diagram of the algorithm implemented on the graphic card is shown in Fig. 7.5. First of all is performed an *overlapped* serial to parallel conversion of the input sequence. The data are organized in a matrix where each column of length L has in common $M-1$ samples. After that the L dimensional FFT is performed on the graphic card by the *CUFFT* library on each column of the matrix and on the (padded) impulse response. The pointwise multiplication is executed between each column of the transformed matrix and the transformed impulse response $\mathbf{H}$ in parallel. We have a number of blocks equal to the number of columns, each of them consists of L threads. Each block works independently on one columns and each thread multiplies its corresponding entries. The CUFFT library is used also to perform efficiently the $iFFT$ on the result of the multiplication, and finally a parallel to serial conversion is performed disregarding the first $M-1$ samples of each column.

7.2.2 Parallel Quantized SPA

In order to exploit the computational capability of the card we have decided to parallelize the Quantized SPA in the number of quantized levels. Our hope is to obtain a good estimation quality by choosing a relative high number of levels, and at the same time to keep small the execution time. If we denote with L the number of uniform levels, we can think that each thread within a block is focused on the computation of one element of the resulting quantized marginal function for each phase sample variable.

Since it is important to minimize the memory transfers between the host and the device the algorithm works on the whole observation vector $\mathbf{y}$ and the input symbols estimate vector $\hat{\mathbf{x}}$, the result will be the whole estimated phase vector $\hat{\boldsymbol{\theta}}$. This means that, if N is the observation window size and N_s the total number of received symbols we launch $N_b = \lceil N_s/N \rceil$ blocks each one with L threads.

The particularity of our implementation is the Markov Chain update rule described in Section 5.5.1. Since all messages are represented as vectors, the forward and backward messages through the Markov Chain are evaluated by a vector $\times$ ma-

trix multiplication as follows:

$$\boldsymbol{\mu}_F(\theta_i) = (\boldsymbol{\mu}_F(\theta_{i-1}) \cdot \boldsymbol{\mu}_f(\theta_{i-1}))\mathbf{P} \tag{7.3}$$

$$\boldsymbol{\mu}_B(\theta_{i-1}) = (\boldsymbol{\mu}_B(\theta_i) \cdot \boldsymbol{\mu}_f(\theta_i))\mathbf{P} \tag{7.4}$$

where $\mathbf{P}$ denotes the $L \times L$ first order transition matrix and stores all the transition probabilities for all the L states of the chain.

The j-th elements of the quantized forward message $\boldsymbol{\mu}_F(\theta_i)$ is obtained by the dot product of the incoming message and the j-th column of the matrix as shown in Fig. 7.6.

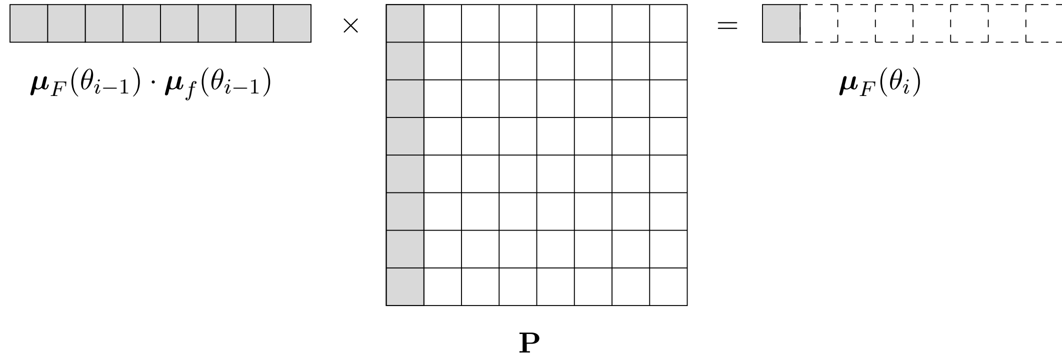

Figure 7.6: Dot product involved in the Markov chain update rule

Hence all dot products involved in the forward and backward recursion have been implemented in parallel by adopting the same principle described in Section 7.2.1 for the Parallel time domain convolution.

In summary the algorithm works as follows:

- Load the observation vector $\mathbf{y}$, the input symbols estimate vector $\hat{\mathbf{x}}$ and the first order transition matrix $\mathbf{P}$ in global memory.
- Each block by working on its specific portion of the inputs of size N:
  - computes the $\boldsymbol{\mu}_f(\theta_i)$ messages $\forall\, i \in [1, \ldots, N]$ and stores the messages in global memory. The marginals will be obtained starting from this partial results;
  - executes the forward and backward recursion by adopting the Markov chain update rule. Once a the i-th forward message has been computed updates the marginals for the phase variable θ_i in memory, the same holds for the i-th backward message;
  - when all the marginals have been computed performs a MAP estimation for each phase variable node θ_i.
- The estimated phase vector $\hat{\boldsymbol{\theta}}$ is stored in global memory.

Before continuing our discussion we anticipate that this algorithm has confirmed the well known high memory requirements for the quantization techniques. Indeed all messages must be stored in intermediate data structures until the phase estimation is performed. The amount of memory required depends both on the number of quantization levels L and the observation window size N. In our experience shared memory has not been used for this purpose because it is about 49 KByte in our specific graphic card (see Tab. 7.1) which is resulted not sufficient. For this reason all intermediate data structures have been moved in global memory and the shared memory has been exploited for the partial results of the Markov Chain update rule.

Furthermore some kind of bottle neck of the algorithm is the forward and backward recursion in the upper part of the factor graph which are inherently serial algorithms. Indeed in order to obtain each forward (backward) message we need to wait for the computation of the previous message, and this cannot be nicely parallelized in this specific implementation.

7.2.3 Parallel Steepest Ascent algorithm

Finally we give some details on the parallel implementation of the Steepest Ascent algorithm for the phase estimation, and we can assert that the most of the work has already been explained in Chapter 5. Indeed the specific algorithm developed in Section 5.6.2 represents already a parallel implementation suggested by the particular nature of the problem.

The possibility to work locally for each phase sample and to perform local optimizations perfectly matches with the CUDA programming model. As for the Quantized SPA we have implemented in parallel our algorithm in such a way that it works on the whole inputs by launching $N_b = \lceil N_s/N \rceil$ each one with N thread. Each threads computes locally and independently the following update rule for each sample θ_i:

$$\theta_i^{(j)} = \theta_i^{(j-1)} + \lambda_j \nabla_{\theta_i} \log g(\theta_i)|_{\theta_i^{(j-1)}} \tag{7.5}$$

where the local gradient $\log g(\theta_i)$ consists of the summation of the three terms:

$$\frac{\partial \log f_i(y_i, \hat{x}_i, \theta_i)}{\partial \theta_i} = \frac{\Im\left\{y_i \hat{x}_i^* A e^{-j\theta_i}\right\}}{\sigma_\omega^2} \tag{7.6}$$

$$\frac{\partial \log p(\theta_i|\theta_{i-1})}{\partial \theta_i} = -\frac{\theta_i - \theta_{i-1}}{\sigma_p^2} \tag{7.7}$$

$$\frac{\partial \log p(\theta_{i+1}|\theta_i)}{\partial \theta_i} = \frac{\theta_{i+1} - \theta_i}{\sigma_p^2} \tag{7.8}$$

evaluated at the current estimates $\theta_{i-1}^{(j-1)}$, $\theta_i^{(j-1)}$ and $\theta_{i+1}^{(j-1)}$. We are in presence of a *total parallelization* of the algorithm since it performs exactly the same operations independently and at the same time on different data. Furthermore the optimizations are completely performed in shared memory in order to improve the performance.

The parallel implemented algorithm is executed on the GPU as follows:

- Each block loads its corresponding portion of the initial guess $\hat{\boldsymbol{\theta}}^{(0)}$ from global memory to shared memory.
- Each thread iteratively update its corresponding phase estimate $\hat{\theta}_i^{(j)}$ by computing the local update rule (7.5).
- When convergence is reached each block stores the resulting portion of the phase estimate $\hat{\boldsymbol{\theta}}$ from shared memory to global memory.

We expected for this algorithm the biggest performance improvement among all the algorithms implemented. In order to validate this we report in the next section the performance comparison between the serial and the parallel implementation of our channel estimation algorithms.

7.3 Performance comparison

In this section is reported a performance comparison between the serial implementation running on the CPU and the parallel implementation running on the GPU for all the algorithms proposed. All the experiments have been executed on a workstation with a quad-core Intel Xeon E5620 CPU running at 2.4 GHz and provided with the Nvidia TESLA C2070 GPU. Tab. 7.2 summarizes the experimental setup.

	CPU	**GPU**
Platform	Intel Xeon E5620	Nvidia TESLA C2070
Language	C/C++	C/C++ & CUDA C
Number of cores	4	448
Clock freq.	2.40 GHz	1.15 GHz
OS	Ubuntu Server 10.04	Ubuntu Server 10.04

Table 7.2: Experimental setup

The metrics measured are the following:

- Execution time.
- Number of clock cycles.
- Effective Bandwidth.
- Average Throughput.

The *Execution time* is simply the measure of the time taken by the portion of code that is executed, this must be measured both for the CPU and the GPU. This metric is widely used in literature (for example in [20]) and it is useful to have a qualitative

indication on the performance. In this regard, while for measuring the execution time on the CPU are used the canonical C system timers, for code implemented on the graphic card it is suggested to use the *CUDA GPU Timers*. Indeed since many CUDA functions are asynchronous they return back the control to the host prior to completing their work. Also all the kernels launched are asynchronous so, to accurately measure the elapsed time, CUDA programming model provides a series of API functions in order to create and destroy events, record events and convert timestamp differences into floating point values in milliseconds. For details on the utilization of this functions and for a listing that illustrates their use we refer to [17].

Another metric we propose is the *Number of clock cycles*. In this way we can obtain an indication of the performance independent from the specific architecture used. It is simply the ratio between the execution time and the clock period of the main time reference in the core.

In general the bandwidth is defined as the rate at which data can be transferred and for us is the most important gating factors to obtain an indication of the performance. The bandwidth can be dramatically affected by the choice of memory in which data are stored, i.e., global memory rather than shared memory. To measure the performance it is useful to calculate the *Effective Bandwidth* achieved by the algorithm.

The Effective Bandwidth is calculated by timing the specific program activities and by knowing how data is accessed by the program. Basically we have to think to our algorithms as black boxes that read a certain amount of Bytes and return as output another quantity of data. The Effective Bandwidth is defined as follows [17]:

$$B_e = \frac{(B_R + B_W)}{T} \; [Bps] \tag{7.9}$$

Here B_R is the number of Bytes read by the algorithm and B_W is the number of Bytes written by the algorithm. T is the execution time in seconds.

Finally the *Average Throughput* is the average rate of useful data processed by the algorithm. It is measured in *bps* and in our case takes this form:

$$Thr = \frac{B_O}{E[T]} \; [bps] \tag{7.10}$$

where B_O is the number of useful bits returned as output by the function and $E[T]$ the average execution time in seconds.

Execution time and number of clock cycles

The next tables report the execution times and the number of clock cycles of the algorithms for different data input sizes in terms of M-PSK symbols. The time values are expressed in seconds.

Time domain convolution

Number of symbols	10e03	10e04	10e05	10e06	10e07
CPU	0.042	0.043	0.34	3.36	33.62
GPU	0.0032	0.0034	0.024	0.24	2.54

Overlap save method

Number of symbols	10e03	10e04	10e05	10e06	10e07
CPU	0.0014	0.0018	0.01	0.25	0.62
GPU	0.000061	0.000077	0.00032	0.0027	0.026

Quantized SPA

Number of symbols	10e03	10e04	10e05	10e06	10e07
CPU	1.74	17.43	173.06	1722.67	17309
GPU	0.23	1.17	10.06	100.11	1008.94

Steepest ascent algorithm

Number of symbols	10e03	10e04	10e05	10e06	10e07
CPU	0.2	1.97	19.69	197.24	1952.08
GPU	0.0043	0.0112	0.0902	0.912	9.5767

Table 7.3: Execution time for the algorithms.

Time domain convolution

Number of symbols	10e03	10e04	10e05	10e06	10e07
CPU	1.01e08	1.03e08	8.16e08	8.06e09	8.07e10
GPU	3.68e06	3.91e06	2.76e07	2.76e08	2.92e09

Overlap save method

Number of symbols	10e03	10e04	10e05	10e06	10e07
CPU	3.36e06	4.32e06	2.40e07	6.00e08	1.49e09
GPU	7.02e04	8.86e04	3.45e05	3.11e06	2.99e07

Quantized SPA

Number of symbols	10e03	10e04	10e05	10e06	10e07
CPU	4.18e09	4.18e10	4.15e11	4.13e12	4.15e13
GPU	2.65e08	1.35e09	1.16e10	1.15e11	1.16e12

Steepest ascent algorithm

Number of symbols	10e03	10e04	10e05	10e06	10e07
CPU	4.80e08	4.73e09	4.73e10	4.73e11	4.68e12
GPU	4.95e06	1.29e07	1.04e08	1.05e09	1.10e10

Table 7.4: Number of clock cycles.

From a first observation on the values reported we can assert that a performance improvement is present in all the cases. Furthermore the execution time of all the algorithms increase quite linearly with the input size (with the exception of the time domain convolution algorithms with short inputs). This is due to the fact that for both the implementations, in presence of long inputs, it is required to work on different portions and then to scan all the data with shift operations. The second table confirms these observations, just from an architecture-independent point of view.

After these first qualitative evaluations, in order to validate these results, we report a comparison in terms of Effective Bandwidth in the following section.

Effective bandwidth

Fig. 7.7 shows a comparison in terms of Effective Bandwidth. Bandwidth values are reported in MBps and, since the Overlap save method leads to an effective bandwidth of GBps we have depicted the graph with a logarithmic y-axes. The Effective Bandwidth values are in Tab. 7.5.

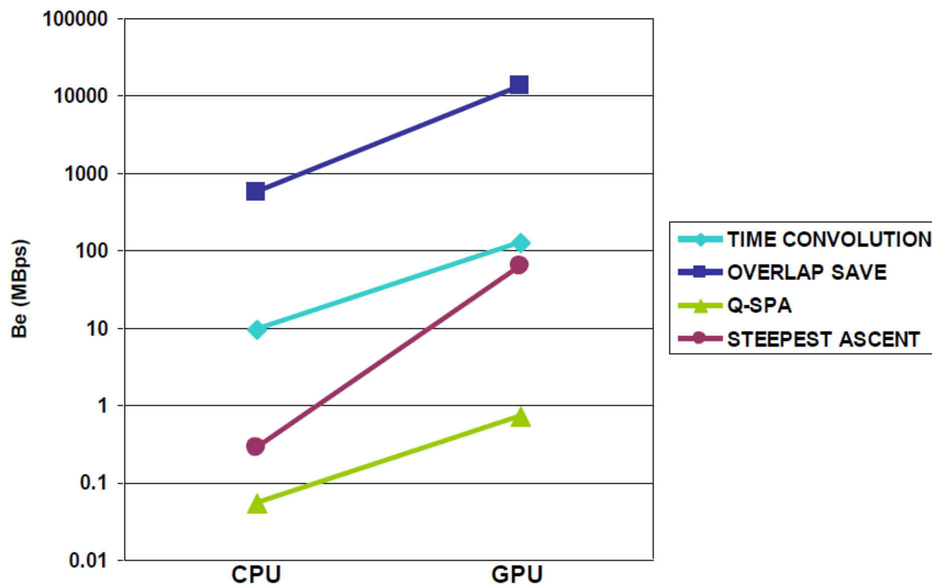

Figure 7.7: Effective bandwidth.

By evaluating the slope of the lines in Fig. 7.7 we have a visual indication on the real performance improvement obtained by the parallel implementation. It is clear that the Steepest Ascent algorithm achieved the biggest speedup by the execution on the GPU. Indeed the Effective Bandwidth goes from 0.28 MBps for the serial implementation on the CPU to 61.71 MBps for the parallel implementation on the GPU, for an improvement of more than 220 times. This was expected since, in the

development of the algorithm, the data parallelism has been exploited and this has resulted in a real *massive parallelization*. Regarding the other algorithms it is also present an improvement but not of the same order as for the Steepest ascent (see Tab 7.5).

	CPU	**GPU**	**Speedup**
Time domain convolution	9.5 MBps	125.5 MBps	up to 13 ×
Overlap save method	0.57 GBps	13.5 GBps	up to 23 ×
Quantized SPA	0.054 MBps	0.71 MBps	up to 13 ×
Steepest ascent algorithm	0.28 MBps	61.71 MBps	up to 220 ×

Table 7.5: Effective Bandwidth values.

In absolute terms of Effective Bandwidth, the Overlap save method achieves the best results among all the algorithms tested. In order to obtain a fair comparison, for the serial implementation on the CPU, the *FFTW* library has been used. This library is a standard choice for the computation of the FFT in C/C++ programs. With an effective bandwidth of 13 GBps the Overlap save method achieves about the 10% of the total bandwidth of the graphic card which is equal to 144 GBps. We recall that this last value it is not reffered to the effective speedup achievable since the real performance depends on the specific algorithm. However we assert that this result is due first of all, because the Overlap save method it is already a fast algorithm for the computation of the convolution, furthermore the utilization of the CUFFT library, which is probably optimized, helps in this sense. Another confirm obtained from this results is the high complexity of the quantization techniques. Indeed the Quantized SPA has the smallest effective bandwidth among all the algorithms. However an improvement is present also in this case. Finally excellent results has been obtained for the Time domain convolution with an Effective bandwidth of about 125 MBps.

Average Throughput

Finally in Tab. 7.6 we report the values of the Average Throughput for the different implementations.

	CPU	**GPU**	**Speedup**
Time domain convolution	19.2 Mbps	261.72 Mbps	up to 13 ×
Overlap save method	2.24 Gbps	54.160 Gbps	up to 23 ×
Quantized SPA	0.144 Mbps	1.9 Mbps	up to 13 ×
Steepest ascent algorithm	0.75 Mbps	164.56 Mbps	up to 220 ×

Table 7.6: Throughput of the algorithms.

In this way we can obtain an indication of the performance for the algorithm inserted in a real communication system like the one shown in the introduction chapter (see Section 1.3).

As we can see all the parallel implementations lead to good performance and the differences in terms of speedup between the serial implementation and the parallel implementation are equal to those observed for the Effective Bandwidth.

7.4 Conclusions

In this chapter we have reported some details on the parallel implementation for the algorithms proposed in this thesis. First of all we have reported the programming guidelines followed in the development and some informations about the specific graphic card used, the Nvidia TESLA C2070.

In order to evaluate the performance speedup obtained a comparison in terms of Execution time, Number of clock cycles, Effective Bandwidth and Average Throughput has been reported between the serial implementation on the CPU and the parallel implementation running on the GPU. The results lead us to make the following observations:

- A performance improvement is present in all the cases.
- The execution time increases, both for the serial and the parallel implementation, quite linearly with the input data size. This is due to the necessity of spanning the data for large inputs.
- The Steepest Ascent algorithm achieves the biggest speedup by the execution on the GPU (up to 220 $\times$ faster). This has been obtained by exploiting the data parallelism in the development of the algorithm. We can assert that in this specific case we are in presence of a *massive parallelization*.
- With an effective bandwidth of 13 GBps the Overlap save method achieves the best result in terms of absolute Effective bandwidth among all the algorithms tested.
- By comparing the Average Throughput we observe how the parallel implementation of all the algorithms leads to good performance. In our opinion this is an important result for the future implementation of the demo.

Chapter 8

Conclusions and future work

We recall that the goals of the work were the following.

1. Fine tuning and performance improvement of the already existing phase estimator.
2. Exploration of the factor graph approach in order to find algorithms that, at the same time, improve the estimation quality and are suitable to be parallelized on GPU.
3. Implementation of such algorithms on the graphic card and to have such systems that exploit the computational power as efficiently as possible.

We can assert that all the points have been covered. In this chapter we summarize the results obtained and give the future work for the project.

The first point has been addressed in Chapter 4 in which, through an approximation of the Wiener-Levy phase model, two different improvements for the existing ML phase estimator have been proposed, they are the *ML estimator improved* (ML_i) and the *ML estimator with optimal window (ML_{opt})*.

Both estimators require the same knowledge at the receiver, i.e, the parameter σ_p^2 and σ_ω^2. While the ML estimator improved is based on a new design for the h_{ML} filter used in the moving average operation, the ML estimator with optimal window seeks to adapt the observation window size to the particular scenario. The performance in term of MSE for the phase estimation have been evaluated and compared to the benchmark (the canonical ML estimator already implemented) and an improvement is present for both cases. In particular the so called ML_{opt} estimator seems to lead best results and, at the same time, to be more robust to strong phase noise. Finally it has been experimentally demonstrated how the biggest improvement is achieved by adapting the window size to the particular scenario rather than design a new h_{ML} filter for the moving average.

The factor graph approach for the phase estimation has been investigated in Chapter 5 in order to obtain a phase estimation algorithm that has best performance

compared to the benchmark, but also can be nicely implemented on the graphic card. In this regard, based on the work did in [8], we have presented the Sum Product Algorithm for the phase estimation working on the factor graph of the Wiener-Levy phase model and two appriximated algorithms have been developed in a new parallel fashion. They are respectively the *Quantized SPA* and the *Steepest Ascent algorithm*.

Indeed, since the SPA leads to intractable integrals, approximations are required. The particular approximation method chosen for representing the messages in the Sum Product Algorithm have been suggested by some considerations related to the final implementation on the graphic card. This has also led us to develop specific tricks, like the *Markov Chain update rule*, and particular message scheduling, like the one for the Steepest Ascent algorithm.

The performance of the two estimators proposed are quite similar in terms of MSE, only slightly differences can be observed due to the unavoidable quantization error in the Quantized SPA. The reason of this similarity is that both factor graph estimators represent different approximations of the Maximum a Posteriori (MAP) estimator for the phase process.

At this point we report a performance comparison for all the estimators developed in order to determine which of them leads to best results. The following pictures depict the two more realistic situations already evaluated in our previous simulation campaigns ($SNR = 2$ dB and $\sigma_p^2 = 5(°)^2$).

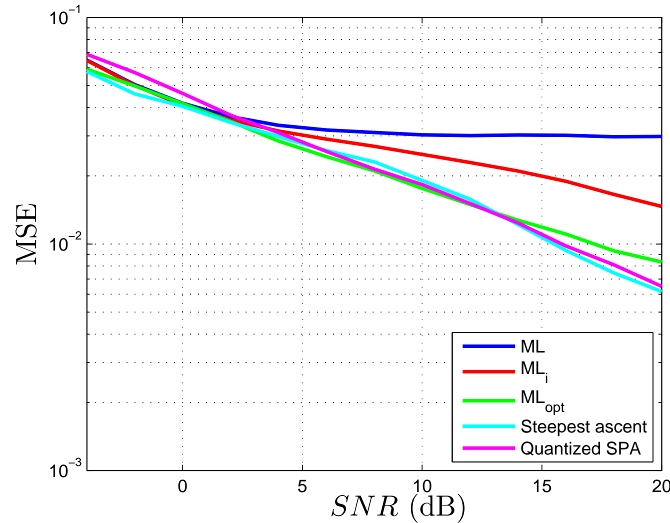

Figure 8.1: MSE as a function of SNR, $\sigma_p^2 = 5(°)^2$.

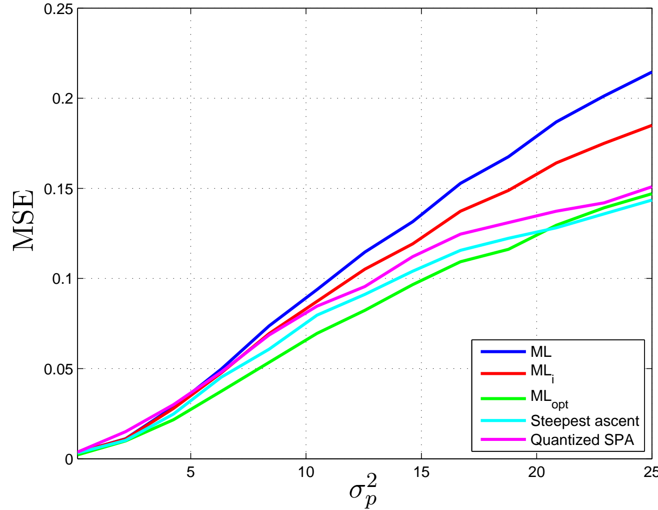

Figure 8.2: MSE as a function of σ_p^2, $SNR = 2$ dB.

From the figures above it is clear how all the estimators proposed have best performance compared to the existing estimator. In particular the ML_{opt} and the Steepest Ascent algorithm have comparable results for different thermal noise scenarios (see Fig. 8.1). Furthermore as depicted in Fig. 8.2 the ML with optimal window seems to be more robust to the strong phase noise compared with all the others.

Important results were finally obtained for the parallel implementation of the algorithms on the graphic card. Among all the implementations the biggest speedup obtained by the execution on the GPU regard the Steepest Ascent algorithm that achieves an improvement of up to 220 times compared to the serial implementation. The reason of this result lies in the development of the algorithm done in Chapter 5 where the concept of *data parallelism* has been exploited and a *massive parallelization* has been obtained.

Also for the other algorithms good performance improvement in terms of Effective bandwidth and Average Throughput have been achieved. In particular the Overlap save method reaches an Effective Bandwidth equal to more than 13 GBps, corresponding to an Average Throughput of about 54.160 Gbps. In our opinion this represents a good result for the future implementation of the multiuser receiver on the cluster server used in the demonstration.

Finally as future work for the project we proposed the following aspects to investigate.

- The two improvements found in Chapter 4 must be tested in the already existing simulator of the multiuser detector presented in Section 1.2. In this way it can be validate how the best phase estimation could lead to best performance in terms of bit error rate (BER) for the system.

- The Steepest Ascent algorithm has to be optimized. Indeed in our simulation the step size λ was fixed in each iteration. The resulting *Optimal Steepest Ascent algorithm* can achieve convergence in fewer iterations. We recall that in our case the number of iteration was quite big, $N_{it} = 1000$.

- Realization of a *Parallel Iterative Receiver* as the one described in Section 5.2 completely executed on the graphic card. This requires the parallel implementation of the LDPC decoder.

Bibliography

[1] N. Benvenuto, G. Cherubini - *Algorithms for communications systems and their applications* - John Wiley & Sons, 2002.

[2] S. M. Kay - *Fundamentals of statistical signal processing: estimation theory* - Enlewood Cliffs, NJ: Prentice-Hall, 1993.

[3] T. K. Moon, W. C. Stirling - *Mathematical methods and algorithms for signal processing* - Prentice-Hall, 2000.

[4] H. M. Taylor, S. Karlin - *An introduction to stochastic modeling, third edition* - Academic Press , 1998.

[5] J. G. Andrews - *Interference cancellation for cellular systems: a contemporary overview* - IEEE Wireless Communications, April 2005.

[6] M. Kobayashi, J. Boutros, and G. Caire. - *Successive interference cancellation with SISO decoding and EM channel estimation* - IEEE J. Select. Areas Communications, vol. 19, no. 8, pp.1450-1460, August 2001.

[7] J. Boutros - *A tutorial on iterative probabilistic decoding and channel estimation* - in IAP MOTION, Ghent (Belgium), January 2005.

[8] J. Dauwels - *On graphical models for communications and machine learning: algorithms, bounds, and analog implementation* - PhD Thesis at ETH Zurich, Diss. ETH No 16365, May 2006.

[9] A. P. Dempster, N. M. Laird, and D. B. Rubin, - *Maximum-likelihood from incomplete data via the EM algorithm* - j. Roy. Stat. Soc., vol. 39, no. 1, pp. 1-38, January 1977.

[10] R. Pulikkoonattu -*Oscillator phase noise and sampling clock jitter* - ST Microelectronics technical report, June 2007.

[11] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger - *Factor graphs and the Sum-Product algorithm* - IEEE Trans. Inform. Theory, vol. 47, no. 2, pp. 498-519, February 2001.

[12] H.-A. Loeliger - *An introduction to factor graphs* - IEEE Signal Proc. Mag., pp. 28-41, January 2004.

[13] H.-A. Loeliger, J. Dauwels, Junli Hu, S. Korl, Li Ping, F. R. Kschischang - *The factor graph approach to model-based signal processing* - Proceedings of the IEEE, vol. 95, no. 6, June 2007.

[14] H. Wymeersch - *Iterative receiver design* - Cambridge University Press, 2007.

[15] A. P. Worthen, W. E. Stark - *Unified design of iterative receivers using factor graphs* - IEEE Trans. Inform. Theory, vol. 47, no. 2, pp. 843-849, February 2001.

[16] NVIDIA Corporation - *CUDA C Programming Guide* - Version 3.2, August 2011.

[17] NVIDIA Corporation - *CUDA C Best Practices Guide* - Version 3.2, August 2011.

[18] David B. Kirk, Wen-mei W. Hwu - *Programming massively parallel processors, a hands-on approach* - Morgan Kaufmann Publishers, 2010.

[19] J. Sanders, E. Kandrot - *CUDA by example. An introduction to general-purpose GPU programming* - Morgan Kaufmann Publishers, 2011.

[20] Shuang Wang and Cheng, S. and Qiang Wu - *A parallel decoding algorithm of LDPC codes using CUDA* - Signals, Systems and Computers, 2008 42nd Asilomar Conference.

[21] Mark Harris - *Optimizing parallel reduction in CUDA* - Slide.