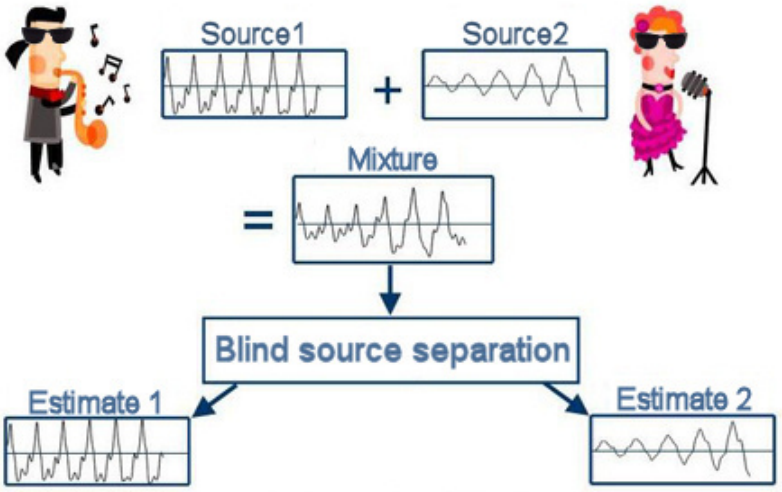
Real-time Blind Source Separation

A Feasibility Study using CUDA

10th semester project, AAU,
Applied Signal Processing and Implementation
Spring 2009



Group 1042
Søren Reinholt Søndergaard
Martin Brinch Sørensen

AALBORG UNIVERSITET

Aalborg University
Institute for Electronic Systems
Fredrik Bajers Vej 7
DK-9100 Aalborg
Phone: (+45) 9635 8080
http://es.aau.dk

Title:
Real-time Blind Source Separation
A Feasibility Study using CUDA

Theme:
Master Thesis

Project period:
P9, Fall Term 2008
P10, Spring Term 2009

Project group:
ASPI 09gr1042

Participants:
Søren Reinholt Søndergaard
Martin Brinch Sørensen

Supervisor:
Kjeld Hermansen

Co-Supervisor:
Alexandre David

Copies: 5

Number of pages: 201

Appendices hereof: 38

Attachment: 1 CD-ROM

Completed: 03-06-2009

Abstract:

In many signal processing applications it is of interest to separate a number of mixed signals. Often the knowledge of the sources and the mixing model is very limited making the separation "blind". This report focuses on the feasibility of implementing a real time Blind Source Separation (BSS) of speech signals using higher order statistics (HOS) on an NVIDIA GPU.

Intially a two input two output (TITO) model is presented and the BSS problem is reduced to an estimation of the filters in this model. A method for estimating the filters based on HOS is presented and subsequent simulations show that a signal to interference ratio of 10 dB is obtainable. Following the simulations the complexity of the method is examined and it is shown that the trispectra estimates used to estimate the filters are the most complex. Using the initial complexity and assuming that the NVIDIA GPU can be utilized fully, it is calculated that the BSS method needs to run 130 times faster to execute in real time. As a consequence the trispectra estimates are examined further to lower their complexity. By taking advantage of how the trispectra estimates are used in the BSS method, it is possible to reduce the complexity by a factor of 263. Part of the method is then implemented using the CUDA programming language. Following an optimization of the implementation an execution time is measured and it is estimated that a filter update rate of 1.19 times per second is achievable.

The conclusion is that the achievable update rate is not sufficient for a real time execution on the platform with the current implementation, as an update rate of 25 times per second is the target.

Preface

This report is the documentation for the 9th and 10th semester ASPI Master Thesis concerning "Real-time Blind Source Separation Feasibility Study using CUDA" at the Institute of Electronics at Aalborg University (AAU).

The report is prepared by group 09gr1042 and spans from the 1st of September 2008 to 3rd of June 2009.

The original project proposal for BSS was presented by Kjeld Hermansen, Associate Professor at AAU. Kjeld Hermansen and Alexandre David, Associate Professor at AAU, functioned as supervisors for this project.

The report is split into four parts. The introduction where the project objectives are described. The theory part where the theory behind blind source separation is presented. The simulations and verification part where the before mentioned theory is simulated to verify its functionality. And the implementation part, where the feasibility of a real time execution of the blind source separation is examined.

The accompanying CD contains a copy of this report, MATLAB implementations used in the simulations and the software implementations of the CUDA program.

The notation used for mathematical expression and an explanation of abbreviation can be found in appendix C and D.

Søren Reinholt Søndergaard

Martin Brinch Sørensen

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

In many of todays signal processing applications a signal consisting of several mixed signals may appear. Typically it is the source signals that are of interest and the problem is that these signals are not always available in the demixed form. In existing applications many methods are employed to avoid mixing the signals or mixing them in specific ways that allow easy separation. In telecommunications, for instance, it is widely known that different bands of the frequency spectrum are assigned to specific applications such as television, AM and FM broadcasts etc. The separation of the signals is then done via bandpass filters attenuating all the unwanted frequencies. Another method is time division. Instead of using different frequencies several transmitters can transmit at the same frequency, but only one at a time. Time slots of a specified length are then assigned to the transmitters which will take turns transmitting. Both methods can also be combined to allow division between a multiple signals as is the case in GSM systems. These three methods are also known as signal separation by domain separation and are very common ways of solving the signal separation problem.

The real separation problem arises when it is not possible to control the mixing of the source signals. To make matters worse the source signals may also be subject to changes from the channel, making it harder to find the source signals even if it is possible to separate the signals. If the source signals could be found from the mixed signal this would have many uses in practical application beyond the ones exemplified above. For instance it would be possible to use only one microphone to record a meeting or an interview with several speakers and then separate each speaker from the mixed signal. It could also be used in hearing aids to allow their users to distinguish several speakers from each other and thus focus on one particularly speaker - more commonly known as the cocktail party problem - or separating signals in a electromyogram (EMG)/electroencephalogram (EEG).

The process of separating source signals when these are not known is also known as blind source separation (BSS). The only assumption made about the source signals is that these are statistically independent. Methods to address the BSS problem have been developed, but it is still a field of ongoing research to improve current methods or create new ones.

Most research papers present the general mixing channel, where there are N sources, s_n, that are received by an array of M sensors, x_m. In the channel between the sources and the sensors the sources are mixed in a way that can be modelled by equation 1.1 [5, p. 2].

$$x_m(t) = \sum_{n=1}^{N} \sum_{k=0}^{K-1} h_{mnk} \cdot s_n(t-k) + v_m(t) \tag{1.1}$$

The mixed signal is a linear combination of filtered versions of the original source signals. Where h_{mnk} describes the filter coefficients and $v_m(t)$ is additive sensor noise. In this model h_{mnk} is assumed to be stationary, but this is not always the case as for the cocktail party problem. It is more likely that the sources and the sensors are not stationary in the room and therefore the filters for the room are often assumed to be quasi-stationary. Also the filters may be infinitely long, but in practice they are assumed to be of finite length.

For the general model there are four assumptions that are often used in published papers:

- Instantaneous mixing: h_{mnk} is assumed to be a constant value with no delay. This results in what is known as instantaneous mixing.
- Delayed sources: h_{mnk} is assumed to be a delay filter only.
- Convolved mixing: h_{mnk} is an abitrary filter, this is the most common model.
- Over- and under-determined sources: It is normally assumed that the number of sources is equal to or less than the number of sensors. This type of system can be solved using linear methods. However, BSS methods that deal with separating under-determined systems, where the number of sources exceeds the number of sensors also exist.

One system model that is often used in research papers is the Two Input Two Output mixing model (TITO) which is illustrated in figure 1.1.

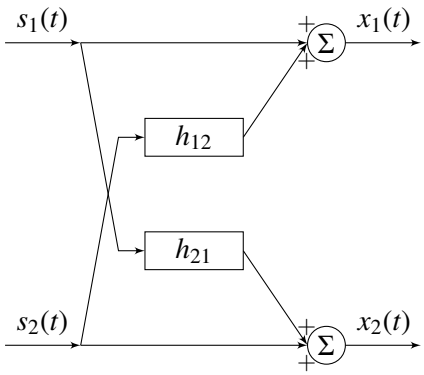


Figure 1.1: Two-channel version of the signal model presented in [5, p. 7]. The recorded signal at each sensor is a superposition of the primary signal and a number of secondary signals.

And the following system equations are used for the TITO model:

$$x_1(t) = \quad s_1(t) \quad + h_{12} * s_2(t) \tag{1.2}$$

$$x_2(t) = \quad h_{21} * s_1(t) \quad + s_2(t) \tag{1.3}$$

For this system model there are a number of common assumptions:

- The filters are FIR, meaning that they are finite in length.
- The system is not under-determined so it can be solved by linear methods.
- Convolved mixing for the filters h_{12} and h_{21} and instantaneous mixing for h_{11} and h_{22} (not depicted in figure 1.1)

The TITO model is used in most research papers concerning BSS and is therefore also used as a system model in the project, as are the above mentioned assumptions.

To recover the source signals the channels must be demixed by the use of the inverse model. Finding the inverse TITO model is a solvable problem, but stability problems may arise if not done properly and these must be taken care of. The problem is then reduced to estimating the unknowns in the TITO model, i.e. the filters h_{12} and h_{21} and this is the problem most research papers are focused on when presenting different BSS methods.

Many different methods exists for solving the BSS problem and they typically have different assumptions about the mixing model. The sparseness method for estimating the filters is normally employed when the system is assumed to be under-determined. Some of the sparseness methods also require that the sources do not overlap in the time-frequency domain. The sparseness methods generally only work well when only delay filters are present in the systems. However, when reverberation is present the time frequency representation of the signal becomes less sparse and the sparseness methods become less than optimal to use. If the reverb is more mild the methods can be combined with Inter Component Analysis (ICA) to get better results [5, pp. 12-13]. Other BSS methods use the concept of clustering where the system model is determined by clustering data with respect to amplitude and delay.

The TITO model is, however, not under-determined and as such there are other and better methods for solving this system. Many of these are based on the statistical properties of the source signals, mainly that the source signals are independent or at least uncorrelated. Some of these methods use second order statistics (SOS) to solve the BSS problem, but these also requires some assumptions about the system in order to be utilized for BSS. Such assumption could be that the filters are minimum phase. If this is the case it is then possible to estimate the filter coefficients in the TITO model according to [5]. Unfortunately it is not reasonable to assume that the filters are minimum phase. Methods that allow for non-minimum phase system are therefore more reasonable alternatives.

One way to handle the non-minimum phase problem is to move away from SOS to higher order statistics (HOS). Going from the power spectra of SOS to the bispectra and trispectra of HOS adds phase information to the system. The only assumptions about the system when using HOS methods are that the sources are independent and non-Gaussian.

Normally it is enough to use third order statistics to preserve phase information, but one problem arises as the third order cumulant spectrum describes the deviation from a non-symmetric PDF. As a lot of distributions have symmetric PDFs their 3 order spectrum would simply be zero and this is generally not acceptable in most application. Because of this, fourth order statistics are

normally used instead in BSS. The problem with using fourth order statistics is that it is rather computational intensive compared to using SOS.

One method for solving the BSS problem using fourth order statistics is presented by Shamsunder and Giannakis in [8]. The method uses slices from the auto- and cross-trispectra of the output signals, x_1 and x_2, to estimate the filters in the system model. As this method was the initial focus for the project and it produces results with around 10 dB signal to interference ratio (SIR) restoration of the original signals, this method for doing the BSS is used in this project.

Chapter 2

Scope of the Project

This project is done as part of the Applied Signal Processing and Implementation (ASPI) specialization. Because of this the weight of the project is on taking an existing algorithm, optimizing it and implementing it to run on a specific hardware platform. The project is therefore divided into parts that reflect this.

The first part is concerned with the mathematical theory behind the the existing algorithm presented by Shamsunder and Giannakis in [8]. This gives the basis for doing the initial implementation and computational optimization on the algorithm.

The second part is verifying the theory by implementing it in a high level language, in this case MATLAB, and running simulations of this implementation. This gives a baseline to compare any optimizations that are performed, either for speed, numerical precision or resource consumption.

The third part is the implementation of the algorithm on a hardware platform. First the complexity of the algorithm is analyzed and the algorithm is modified to reduce the complexity. Next the algorithm is implemented in a low level language to run on the hardware platform. The platform used in this project is an NVIDIA Graphic Processing Unit (GPU) and the programming language used is the CUDA, a framework developed by NVIDIA for running applications on their GPUs.

The overall goal of this project can be summarized to the following:
To investigate if it is possible to make a BSS on speech signals using HOS that can run in real time on an NVIDIA GPU.

The cocktail party problem is used as the reference application for evaluating the real time implementation, i.e. the sources are speech signals.

2.1 Assumptions and Limitations of the system model

For this project the following assumptions/limitations are applied to system model:

- Two Input Two Output (TITO) system model: This limitation is mainly to reduce the problem size. The algorithms can be extended to larger system models if necessary [8, p. 521], but only the TITO model is used in the project.
- The signals are additive: This is a needed assumption about the model in order to make it possible to find the inverse system.
- Quasi-stationary filters: The filters in the TITO model must be stationary for the time it takes to estimate them.
- The sources are non-Gaussian: This assumption is related to the trispectra of the signals. If the sources are Gaussian their trispectra would be zero and and it would not be possible to estimate the filters in the TITO model.
- No additive noise added in the channel/mixing model: This limitation is due to the weight of the algorithm being on the BSS and not reduction of additive noise. Numerical noise will still be present due to finite word length of the implementation.
- The sources are independent: This assumption is necessary to be able to estimate the filters - the multilinearity property can not be applied if the sources are not independent.

Part II

Theory behind the Blind Source Separation

Chapter 3

Inverting the TITO model

As the problem for doing the BSS was divided into two parts. Inverting the system model and estimating the filters in the system model. The first part is handled in this chapter, and it is assumed that the filters are known. In the introduction the TITO (two input two output) mixing model, was introduced. And it was decided to use this model to describe the mixing process of two source signal, that then needs to be separated using BSS. The first question is if it is at all possible the separate the signals again if everything is known about the mixing process.
In the time domain model of the mixing system can be described as equation 3.1 and 3.2.

$$x_1(t) = s_1(t) + h_{12} * s_2(t) \tag{3.1}$$
$$x_2(t) = s_2(t) + h_{21} * s_1(t) \tag{3.2}$$

These equations can be rewritten into matrix form to give equation 3.3, that describes the mixing process.

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} 1 & h_{12} \\ h_{21} & 1 \end{bmatrix} * \begin{bmatrix} s_1(t) \\ s_2(t) \end{bmatrix} \tag{3.3}$$

Reconstructing the signals s_1 and s_2 from equation 3.3 in the time domain is rather problematic, because of the convolution between the filter and the source signals. But by using the convolution theorem the convolution can in the frequency domain be performed as a multiplication. Performing a Fourier transform on equation 3.3 gives equation 3.4.

$$\begin{bmatrix} X_1(\omega) \\ X_2(\omega) \end{bmatrix} = \begin{bmatrix} 1 & H_{12}(\omega) \\ H_{21}(\omega) & 1 \end{bmatrix} \cdot \begin{bmatrix} S_1(\omega) \\ S_2(\omega) \end{bmatrix}$$
$$\bar{X}(\omega) = \bar{\bar{H}}(\omega) \cdot \bar{S}(\omega) \tag{3.4}$$

The original source signal can now be reconstructed from x_1 and x_2 by inverting the matrix $\bar{\bar{H}}(\omega)$ as described by 3.5

$$\bar{S}(\omega) = \bar{\bar{H}}(\omega)^{-1} \cdot \bar{X}(\omega) \tag{3.5}$$

As the matrix for $\bar{\bar{H}}$ only contains 2x2 elements the inverse can be calculated as:

$$\bar{\bar{H}}^{-1}(\omega) = \frac{1}{1 \cdot 1 - H_{12}(\omega) \cdot H_{21}(\omega)} \begin{bmatrix} 1 & -H_{12}(\omega) \\ -H_{21}(\omega) & 1 \end{bmatrix} \tag{3.6}$$

An equivalent solution model can be made for equation 3.6 for demixing the TITO model. This solution model can be seen in figure 3.1.

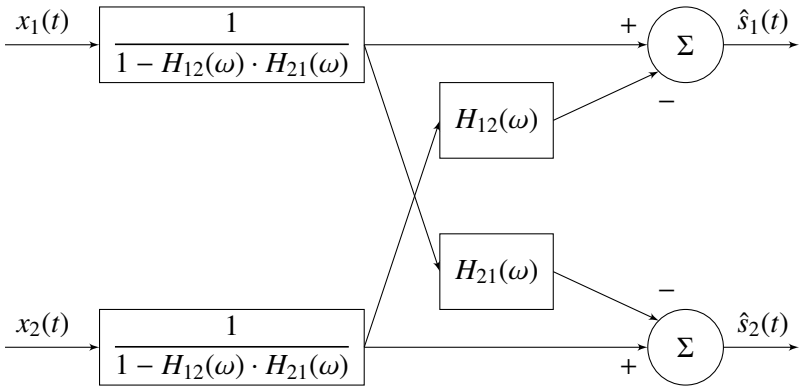


Figure 3.1: Solution model for demixing the TITO mixing model

Figure 3.1 can be simplified to figure 3.2

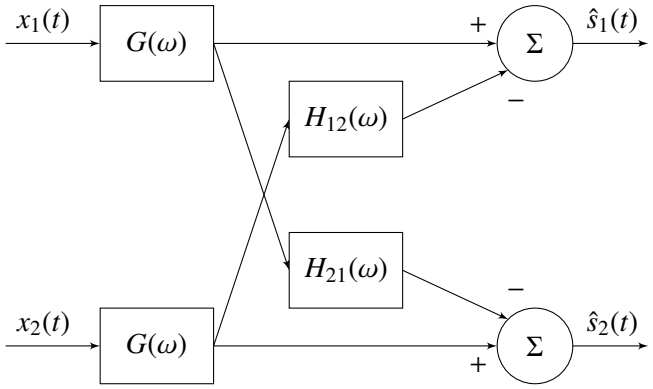


Figure 3.2: Simplefied solution model for making the inverse

One criterion for this model to function is that all the filters in figure 3.2 must be stable. The filters H_{12} and H_{21} are assumed to stable. The problem is the filter G, therefor steps needs to be taken in order to ensure that this filter is indeed stable. The transfer function for G is listed in equation 3.7, where it's inverse filter K is also defined.

$$G(\omega) = \frac{1}{1 - H_{12}(\omega) \cdot H_{21}(\omega)} \tag{3.7}$$

$$K(\omega) = \frac{1}{G(\omega)} \tag{3.8}$$

$$= 1 - H_{12}(\omega) \cdot H_{21}(\omega) \tag{3.9}$$

If the inverse of $G(\omega)$ $K(\omega)$ is evaluated, then it would always be stable under the assumption that H_{12} and H_{21} are stable (eqn. 3.9). Now that a stable filter $K(\omega)$ has been established the question is whether the inverse filter G is stable. In general there are three types of filters minimum, maximum and none-minimum phase (none-minimum phase being a combination of a minimum and maximum). Figure 3.3 illustrates examples of zero pole plots for a minimum, maximum and none-minimum phase filters.

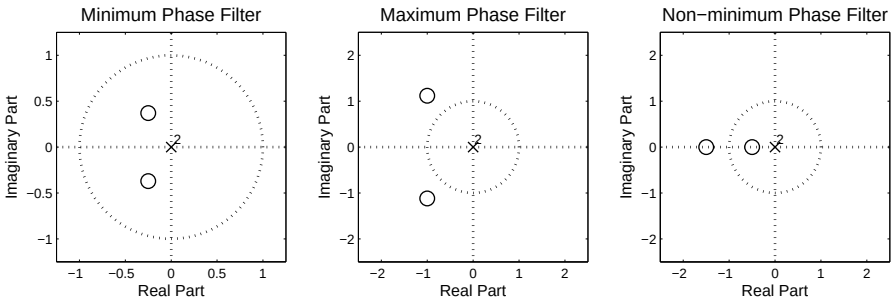


Figure 3.3: Zero pole plots of a minimum, maximum phase and non-minimum filter

The maximum or none-minimum phase filters have zeros located outside the unit circle. These zero becomes a problem when trying to create the inverse filter of a maximum or none-minimum phase filter, as the zeros would have to be converted into poles making the inverse filter unstable. The solution for making this inverse filter stable is based on that all non-minimum and maximum phase filters can be divided into a minimum phase filter and an all pass filter. The next part describes how this is possible, and how this helps in inverting the filter.

Consider the filter in equation 3.10.

$$H(z) = \frac{1 + 3z^{-1}}{1 + \frac{1}{2}z^{-1}} \tag{3.10}$$

The filter described by equation 3.10 is a maximum phase filter, which can be seen from the fact that the zero is not located inside the unit circle in zero pole plot in figure 3.4. The idea is to use the all pass part of the filter to move the zero located outside the unit circle into the unit circle, thereby creating a minimum phase filter that is invertible.

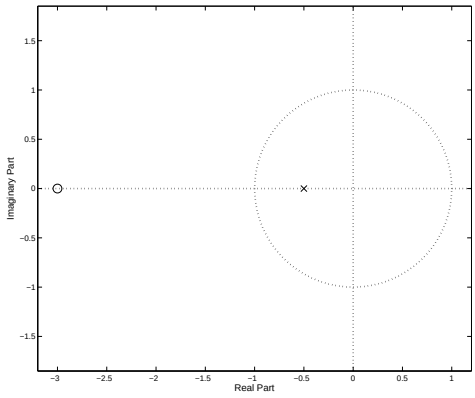


Figure 3.4: Zero pole plot of the H(z) example filter

The filter in equation 3.10 is in equation 3.11 described by a minimum phase filter and an all pass filter:

$$H(z) = H_{min}(z) \cdot H_{ap}(z) \tag{3.11}$$

In order to add the zero outside the unit circle the $H_{ap}(z)$ filters pole must be located at -3, and in order to preserve an magnitude of one it's zero must located at at the inverse of the pole at -$\frac{1}{3}$. So the filter $H_{ap}(z)$ would have a zero pole plot as illustrated in figure 3.5.

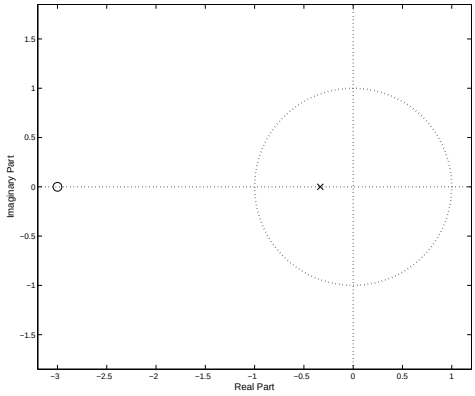


Figure 3.5: Zero pole plot of the all pass filter.

This zero pole plot would have a transfer function for $H_{ap}(z)$ as in equation 3.12.

$$H_{ap}(z) = \frac{1 + 3z^{-1}}{1 + \frac{1}{3}z^{-1}} \tag{3.12}$$

The remaining $H_{min}(z)$ filter would then have a zero pole configuration as in figure 3.6 and a transferfunction described by equation 3.13.

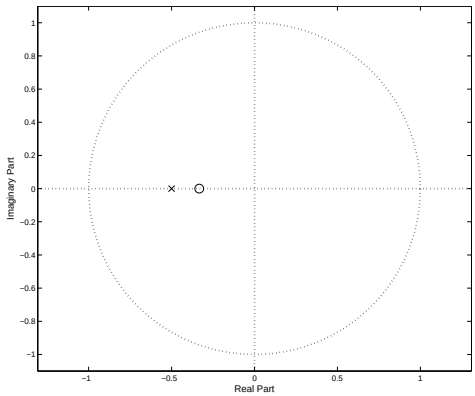


Figure 3.6: Zero pole plot of the minimum phase filter.

$$H_{min}(z) = \frac{1 + \frac{1}{3}z^{-1}}{1 + \frac{1}{2}z^{-1}} \tag{3.13}$$

Writing the original filters as a function of the allpass in equation 3.12 and equation 3.13 for the minimum filter gives equation 3.14.

$$\begin{aligned} H(z) &= H_{min}(z) \cdot H_{ap}(z) \\ &= \frac{1 + \frac{1}{3}z^{-1}}{1 + \frac{1}{2}z^{-1}} \cdot \frac{1 + 3z^{-1}}{1 + \frac{1}{3}z^{-1}} = \frac{1 + 3z^{-1}}{1 + \frac{1}{2}z - 1} \end{aligned} \tag{3.14}$$

This division of filter into a allpass and minimum phase filter can also shown mathematically

by dividing the filter into an AR and an MA process as in equation 3.15.

$$\begin{aligned} H(z) &= \frac{1+3z-1}{1+\frac{1}{2}z-1} \\ &= \frac{1}{1+\frac{1}{2}z-1} \cdot \left(1+3z^{-1}\right) \end{aligned} \tag{3.15}$$

The idea is to divide the MA process with it's inverse thereby creating the all pass-filter, which is performed in equation 3.16 and 3.17.

$$H(z) = \frac{1}{1+\frac{1}{2}z-1} \cdot \left(1+3z^{-1}\right) \cdot \frac{1+\frac{1}{3}z^{-1}}{1+\frac{1}{3}z^{-1}} \tag{3.16}$$

$$= \frac{1+\frac{1}{3}z^{-1}}{1+\frac{1}{2}z-1} \cdot \frac{1+3z^{-1}}{1+\frac{1}{3}z^{-1}} \tag{3.17}$$

Giving the same result as constructing the filters from the zero pole plots.

The next step is to find the inverse filter of $H(z)$, this is now matter of finding the inverse of H_{min} and H_{ap}. The H_{min} is done by replacing the poles with zeros and zeros with poles, effectively inverting the fraction in equation 3.12. The H_{ap} is more complicated, the magnitude and the phase characteristics of the filters can be seen in figure 3.7

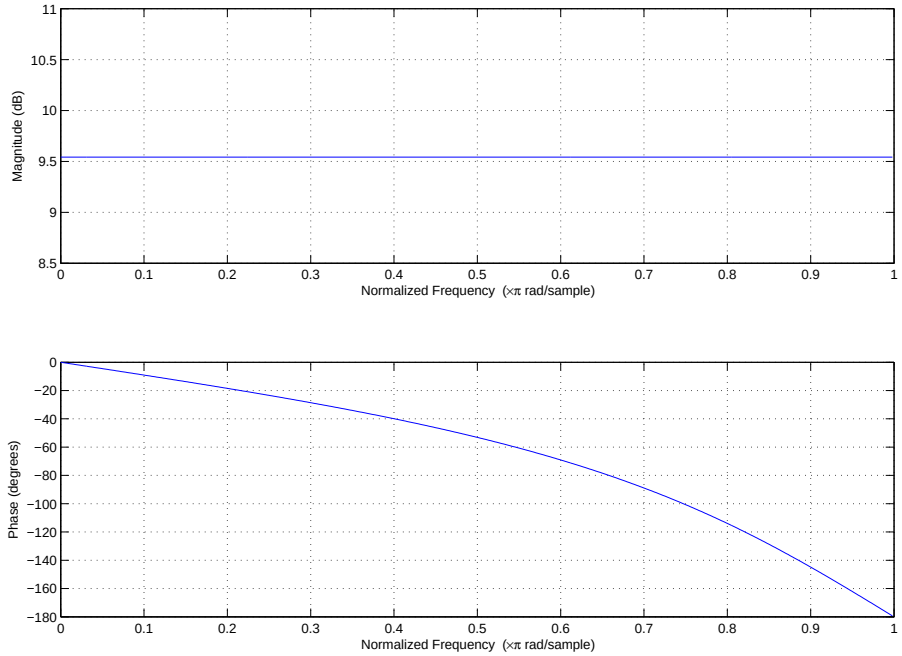


Figure 3.7: Magnitude phase plot of the all pass filter

Taking the inverse of this is a matter of inverting the magnitude plot and shifting the sign of the phase so the inverse H_{ap} would have a phase magnitude plot as illustrated in figure 3.8.

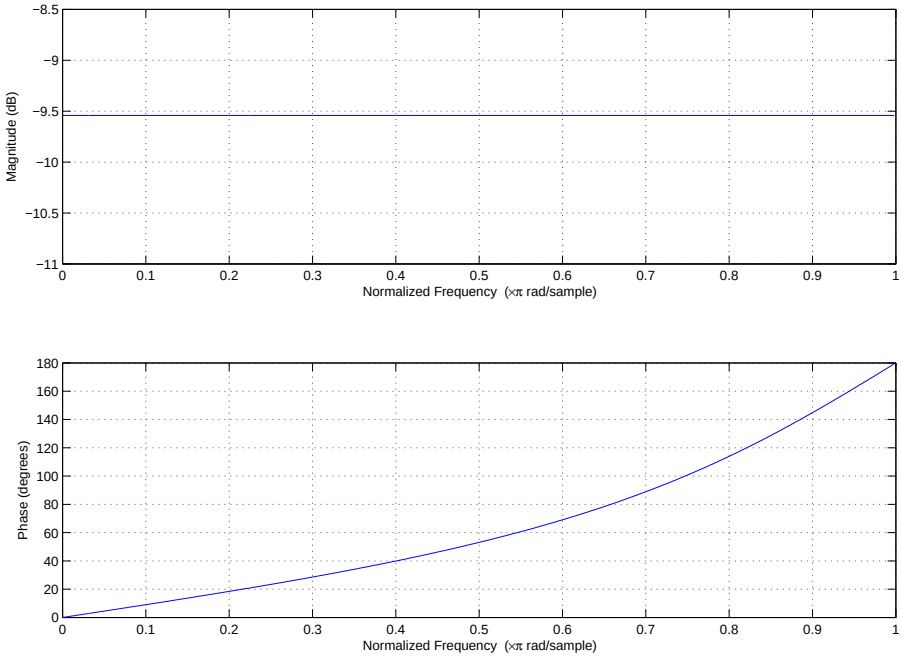


Figure 3.8: Magnitude phase plot of the inverted all pass filter

The problem is that it is not possible to construct a inverse all pass filter that is stable, because a positive phase implies that the output from the filter would be growing exponentially.
However if the filter is made none-causal by introducing a delay filter prior to the the all pass filter, the negative phase from the delay filter would cancel the positive phase of the inverted all pass filter and it would then become stable. This means that the inversion of the $H(z)$ becomes a function of H_{min} and H_{ap} and a delay filter H_{df}, which produces a stable filter if the delay in the delay filter is large enough to counteract the positive phase from the inverted allpass filter.

$$H^{-1}_{stable}(z) = H^{-1}_{min}(z) \cdot H^{-1}_{ap}(z) \cdot H_{df}(z) \tag{3.18}$$

Now one thing to notice is that dividing the filter into a all pass and a minimum phase part is no longer necessary as the unstable all pass filter is stabilized by the delay filter. Therefore the stable inversion of the $H(z)$ filter can be written as a inversion and a delay filter to make the inversion stable, as show in equation 3.19.

$$H^{-1}_{stable}(z) = H^{-1}(z) \cdot H_{df}(z) \tag{3.19}$$

Returning to the solution model it can be shown that the filter $G(\omega)$ can be made stable by adding a delay filter to it, this of course means that the system is not going to be causal but this is however acceptable, where a non stable system is not. Adding the delay filters to the original solution model 3.2 gives a system model illustrated in figure 3.9.

The delay filters are also added to the filters h_{12} and h_{21}, as they would need the same delay to ensure that the demixing is done correctly.

Adding the delay filters to equation for $G(omega)$ of the system gives equation 3.20

$$G(\omega) = \frac{H_{df}(\omega)}{1 - H_{12}(\omega) \cdot H_{21}(\omega)} \tag{3.20}$$

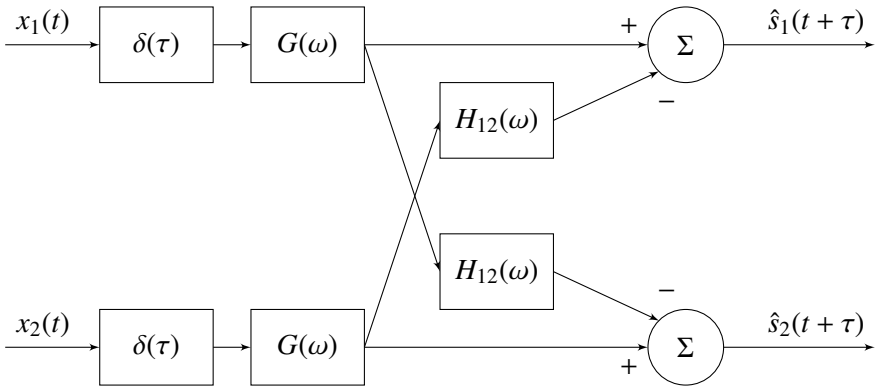


Figure 3.9: Simplefied solution model for making the inverse

Going back to the solution equation for the TITO model it can now be rewritten to equation 3.21.

$$\begin{aligned} \bar{S}(\omega) &= \frac{H_{df}(\omega)}{1 - H_{12}(\omega) \cdot H_{21}(\omega)} \begin{bmatrix} 1 & -H_{12}(\omega) \\ -H_{21}(\omega) & 1 \end{bmatrix} \cdot \bar{X}(\omega) \\ \bar{S}(\omega) &= \bar{\bar{H}}(\omega)^{-1} \cdot H_{df}(\omega) \cdot \bar{X}(\omega) \end{aligned} \tag{3.21}$$

Which is stable for a large enough delay. Now that it has been proven mathematically that it is possible to demix the signals if a delay is added to the system. It is possible to move on to the estimation of the h_{12} and h_{21} filters.

Chapter 4

Estimating the Filters in the TITO model

This chapter contains the estimation of the filters in the TITO model which would make it possible to invert the mixing of the source signals. A method for estimating this is proposed in [8] while some details and procedures are described in other papers referred to from [8]. As this method uses higher order spectra's for the estimation, the reader is encouraged to read appendix A ,which covers the basic theory of HOS, theory that is used in this chapter.

The system equations for the TITO model is in the frequency domain defined as:

$$X_1(\omega) = S_1(\omega) + H_{12}(\omega) \cdot S_2(\omega) \tag{4.1}$$
$$X_2(\omega) = S_2(\omega) + H_{21}(\omega) \cdot S_1(\omega) \tag{4.2}$$

Because of the multilinearity properties of spectra discussed in appendix A the following equations holds true for the n-th order spectra for the TITO model:

$$C_{x_1^n}(\bar{\omega}) = C_{s_1^n}(\bar{\omega}) + M_{h_{12}^n}(\bar{\omega}) \cdot C_{s_2^n}(\bar{\omega}) \tag{4.3}$$
$$C_{x_1^n}(\bar{\omega}) = C_{s_2^n}(\bar{\omega}) + M_{h_{21}^n}(\bar{\omega}) \cdot C_{s_1^n}(\bar{\omega}) \tag{4.4}$$

where:
$\bar{\omega} = \omega_1, \omega_2, \ldots, \omega_{n-1}$.

Similarly the cross spectrum of the $(n-1)$th order polyspectra of x_1 and the first order spectra of x_2 can be defined as:

$$C_{x_1^{n-1}x_2}(\bar{\omega}) = M_{h_{11}^{n-1}h_{21}}(\bar{\omega}) \cdot C_{s_1^n}(\bar{\omega}) + M_{h_{12}^{n-1}h_{22}}(\bar{\omega}) \cdot C_{s_2^n}(\bar{\omega}) \tag{4.5}$$

This can also be done for the reverse case, the cross spectrum between the $(n-1)$th polyspectra of x_2 and the first order spectra of x_1:

$$C_{x_2^{n-1}x_1}(\bar{\omega}) = M_{h_{22}^{n-1}h_{12}}(\bar{\omega}) \cdot C_{s_2^n}(\bar{\omega}) + M_{h_{21}^{n-1}h_{11}}(\bar{\omega}) \cdot C_{s_1^n}(\bar{\omega}) \tag{4.6}$$

Equations 4.3, 4.4 , 4.5 and 4.6 gives four equations with four unknowns, s_1, s_2, h_{12} and h_{21} (h_{11} and h_{22} are equal to one).

The next step is to determine the order, n, of the cumulant spectra to solve the equations. In order to reconstruct the true phase at least the bispectrum (n = 3) must be used. If the bispectrum of x_1 is used in equation 4.5 and x_2 in 4.6 the resulting order of the cross spectra will be $n = 4$, also it was noted in the introduction that the bispectrum ($n = 3$) has problem related to the PDF of the source signals, that are not allowed to be symmetric PDF if the bispectrum is used.

Using the trispectrum for the system equations gives the equations in 4.10:

$$\begin{aligned}
C_{x_1^4}(\bar{\omega}) &= C_{s_1^4}(\bar{\omega}) + M_{h_{12}^4}(\bar{\omega}) \cdot C_{s_2^4}(\bar{\omega}) && (4.7)\\
C_{x_2^4}(\bar{\omega}) &= C_{s_2^4}(\bar{\omega}) + M_{h_{21}^4}(\bar{\omega}) \cdot C_{s_1^4}(\bar{\omega}) && (4.8)\\
C_{x_1^3 x_2}(\bar{\omega}) &= M_{h_{11}^3 h_{21}}(\bar{\omega}) \cdot C_{s_1^4}(\bar{\omega}) + M_{h_{12}^3 h_{22}}(\bar{\omega}) \cdot C_{s_2^4}(\bar{\omega}) && (4.9)\\
C_{x_2^3 x_1}(\bar{\omega}) &= M_{h_{22}^3 h_{12}}(\bar{\omega}) \cdot C_{s_2^4}(\bar{\omega}) + M_{h_{21}^3 h_{11}}(\bar{\omega}) \cdot C_{s_1^4}(\bar{\omega}) && (4.10)
\end{aligned}$$

where:
$\bar{\omega} = \omega_1, \omega_2, \omega_3$.

From appendix A it is know that a moment spectrum can be written as a product of the Fourier transforms of its operands. If this is done for the trispectra of the filters in equations 4.7 to 4.10 the following can be obtained:

$$\begin{aligned}
C_{x_1^4}(\bar{\omega}) &= C_{s_1^4}(\bar{\omega}) + H_{12}(\omega_1) \cdot H_{12}(\omega_2) \cdot H_{12}(\omega_3) \cdot H_{12}^*(\omega_1 + \omega_2 + \omega_3) \cdot C_{s_2^4}(\bar{\omega}) && (4.11)\\
C_{x_2^4}(\bar{\omega}) &= C_{s_2^4}(\bar{\omega}) + H_{21}(\omega_1) \cdot H_{21}(\omega_2) \cdot H_{21}(\omega_3) \cdot H_{21}^*(\omega_1 + \omega_2 + \omega_3) \cdot C_{s_1^4}(\bar{\omega}) && (4.12)\\
C_{x_1^3 x_2}(\bar{\omega}) &= H_{11}(\omega_1) \cdot H_{11}(\omega_2) \cdot H_{21}(\omega_3) \cdot H_{11}^*(\omega_1 + \omega_2 + \omega_3) \cdot C_{s_1^4}(\bar{\omega}) + \\
&\quad H_{12}(\omega_1) \cdot H_{12}(\omega_2) \cdot H_{22}(\omega_3) \cdot H_{12}^*(\omega_1 + \omega_2 + \omega_3) \cdot C_{s_2^4}(\bar{\omega}) && (4.13)\\
C_{x_2^3 x_1}(\omega) &= H_{22}(\omega_1) \cdot H_{22}(\omega_2) \cdot H_{12}(\omega_3) \cdot H_{22}^*(\omega_1 + \omega_2 + \omega_3) \cdot C_{s_2^4}(\bar{\omega}) + \\
&\quad H_{21}(\omega_1) \cdot H_{21}(\omega_2) \cdot H_{11}(\omega_3) \cdot H_{21}^*(\omega_1 + \omega_2 + \omega_3) \cdot C_{s_1^4}(\bar{\omega}) && (4.14)
\end{aligned}$$

If ω_3 is set to zero, the fourth order moment spectra of the filter can be reduced to the third order moment spectra of the filter multiplied with the DC amplification of the filter ($H(0)$). Remembering that $H_{22}(\omega)$ and $H_{11}(\omega)$ are equal to one, the equations in 4.11 to 4.14 can be reduced to:

$$\begin{aligned}
C_{x_1^4}(\bar{\omega}) &= C_{s_1^4}(\bar{\omega}) + H_{12}(\omega_1) \cdot H_{12}(\omega_2) \cdot H_{12}(0) \cdot H_{12}^*(\omega_1 + \omega_2) \cdot C_{s_2^4}(\bar{\omega}) && (4.15)\\
C_{x_2^4}(\bar{\omega}) &= C_{s_2^4}(\bar{\omega}) + H_{21}(\omega_1) \cdot H_{21}(\omega_2) \cdot H_{21}(0) \cdot H_{21}^*(\omega_1 + \omega_2) \cdot C_{s_1^4}(\bar{\omega}) && (4.16)\\
C_{x_1^3 x_2}(\bar{\omega}) &= H_{21}(0) \cdot C_{s_1^4}(\bar{\omega}) + H_{12}(\omega_1) \cdot H_{12}(\omega_2) \cdot H_{12}^*(\omega_1 + \omega_2) \cdot C_{s_2^4}(\bar{\omega}) && (4.17)\\
C_{x_2^3 x_1}(\bar{\omega}) &= H_{12}(0) \cdot C_{s_2^4}(\bar{\omega}) + H_{21}(\omega_1) \cdot H_{21}(\omega_2) \cdot H_{21}^*(\omega_1 + \omega_2) \cdot C_{s_1^4}(\bar{\omega}) && (4.18)
\end{aligned}$$

where:
$\bar{\omega} = \omega_1, \omega_2, 0$.

Which can be rewritten to:

$$C_{x_1^4}(\bar{\omega}) = C_{s_1^4}(\bar{\omega}) + M_{h_{12}^3}(\omega_1,\omega_2)\cdot H_{12}(0)\cdot C_{s_2^4}(\bar{\omega}) \tag{4.19}$$
$$C_{x_2^4}(\bar{\omega}) = C_{s_2^4}(\bar{\omega}) + M_{h_{21}^3}(\omega_1,\omega_2)\cdot H_{21}(0)\cdot C_{s_1^4}(\bar{\omega}) \tag{4.20}$$
$$C_{x_1^3x_2}(\bar{\omega}) = H_{21}(0)\cdot C_{s_1^4}(\bar{\omega}) + M_{h_{12}^3}(\omega_1,\omega_2)\cdot C_{s_2^4}(\bar{\omega}) \tag{4.21}$$
$$C_{x_2^3x_1}(\bar{\omega}) = H_{12}(0)\cdot C_{s_2^4}(\bar{\omega}) + M_{h_{21}^3}(\omega_1,\omega_2)\cdot C_{s_1^4}(\bar{\omega}) \tag{4.22}$$

The next step is to eliminate the trispectra's of the sources ($C_{s_1^4}(\bar{\omega})$ and $C_{s_2^4}(\bar{\omega})$) from the equations and derive expressions for $M_{h_{12}^3}(\omega_1,\omega_2)$ and $M_{h_{21}^3}(\omega_1,\omega_2)$.

Isolating $C_{s_1^4}(\omega)$ in equations 4.19 to 4.22 gives:

$$C_{s_1^4}(\bar{\omega}) = C_{x_1^4}(\bar{\omega}) - M_{h_{12}^3}(\omega_1,\omega_2)\cdot H_{12}(0)\cdot C_{s_2^4}(\bar{\omega}) \tag{4.23}$$
$$C_{s_1^4}(\bar{\omega}) = \frac{C_{x_2^4}(\bar{\omega}) - C_{s_2^4}(\bar{\omega})}{M_{h_{21}^3}(\omega_1,\omega_2)\cdot H_{21}(0)} \tag{4.24}$$
$$C_{s_1^4}(\bar{\omega}) = \frac{C_{x_1^3x_2}(\bar{\omega}) - M_{h_{12}^3}(\omega_1,\omega_2)\cdot C_{s_2^4}(\bar{\omega})}{H_{21}(0)} \tag{4.25}$$
$$C_{s_1^4}(\bar{\omega}) = \frac{C_{x_2^3x_1}(\bar{\omega}) - H_{12}(0)\cdot C_{s_2^4}(\bar{\omega})}{M_{h_{21}^3}(\omega_1,\omega_2)} \tag{4.26}$$

Setting the right hand sides equal to each other (equation 4.23 to 4.25 and 4.24 to 4.26) eliminates $C_{s_1^4(\omega)}$ and isolating $C_{s_2^4(\omega)}$ the equations can be reduced to:

$$\begin{aligned} C_{x_1^4}(\bar{\omega}) - M_{h_{12}^3}(\omega_1,\omega_2)\cdot H_{12}(0)\cdot C_{s_2^4}(\bar{\omega}) &= \frac{C_{x_1^3x_2}(\bar{\omega}) - M_{h_{12}^3}(\omega_1,\omega_2)\cdot C_{s_2^4}(\bar{\omega})}{H_{21}(0)} \\ &\Updownarrow \\ C_{s_2^4}(\bar{\omega}) &= \frac{C_{x_1^3x_2}(\bar{\omega}) - H_{21}(0)\cdot C_{x_1^4}(\bar{\omega})}{M_{h_{12}^3}(\omega_1,\omega_2)\cdot(1 - H_{12}(0)\cdot H_{21}(0))} \end{aligned} \tag{4.27}$$

$$\begin{aligned} \frac{C_{x_2^4}(\bar{\omega}) - C_{s_2^4}(\bar{\omega})}{M_{h_{21}^3}(\omega_1,\omega_2)\cdot H_{21}(0)} &= \frac{C_{x_2^3x_1}(\bar{\omega}) - H_{12}(0)\cdot C_{s_2^4}(\bar{\omega})}{M_{h_{21}^3}(\omega_1,\omega_2)} \\ &\Updownarrow \\ C_{s_2^4}(\bar{\omega}) &= \frac{C_{x_2^4}(\bar{\omega}) - H_{21}(0)\cdot C_{x_2^3x_1}(\bar{\omega})}{1 - H_{12}(0)\cdot H_{21}(0)} \end{aligned} \tag{4.28}$$

Setting equations 4.27 and 4.28 equal to each other eliminates $C_{s_2^4(\omega)}$ and an expression for $M_{h_{21}^3}(\omega_1,\omega_2)$ can be found as:

$$\begin{aligned} \frac{C_{x_2^4}(\bar{\omega}) - H_{21}(0)\cdot C_{x_2^3x_1}(\bar{\omega})}{1 - H_{12}(0)\cdot H_{21}(0)} &= \frac{C_{x_1^3x_2}(\bar{\omega}) - H_{21}(0)\cdot C_{x_1^4}(\bar{\omega})}{M_{h_{12}^3}(\omega_1,\omega_2)\cdot(1 - H_{12}(0)\cdot H_{21}(0))} \\ &\Updownarrow \\ M_{h_{12}^3}(\omega_1,\omega_2) &= \frac{C_{x_1^3x_2}(\omega_1,\omega_2,0) - H_{21}(0)\cdot C_{x_1^4}(\omega_1,\omega_2,0)}{C_{x_2^4}(\omega_1,\omega_2,0) - H_{21}(0)\cdot C_{x_2^3x_1}(\omega_1,\omega_2,0)} \end{aligned} \tag{4.29}$$

The procedure for $M_{h_{21}^3}(\omega_1, \omega_2)$ is the same as for $M_{h_{12}^3}(\omega_1, \omega_2)$, by first isolating $C_{s_2^4}(\omega)$ and then isolating $C_{s_1^4}(\omega)$ resulting expression for $M_{h_{21}^3}(\omega_1, \omega_2)$ as:

$$M_{h_{21}^3}(\omega_1, \omega_2) = \frac{C_{x_2^3 x_1}(\omega_1, \omega_2, 0) - H_{12}(0) \cdot C_{x_2^4}(\omega_1, \omega_2, 0)}{C_{x_1^4}(\omega_1, \omega_2, 0) - H_{12}(0) \cdot C_{x_1^3 x_2}(\omega_1, \omega_2, 0)} \tag{4.30}$$

Similar equations can also be made for a minimum phase system where the powerspectrum of the filters are estimated instead of the bispectrum. However the problem with symmetric PDF's and non-minimum phase filters, makes these equations only usable under these conditions. The equations are described in equations 4.31 and 4.32.

$$M_{h_{12}^2}(\omega_1) = \frac{C_{x_1^2 x_1}(\omega_1, 0) - H_{21}(0) \cdot C_{x_1^3}(\omega_1, 0)}{C_{x_2^3}(\omega_1, 0) - H_{21}(0) \cdot C_{x_2^2 x_2}(\omega_1, 0)} \tag{4.31}$$

$$M_{h_{21}^2}(\omega_1) = \frac{C_{x_2^2 x_1}(\omega_1, 0) - H_{12}(0) \cdot C_{x_2^3}(\omega_1, 0)}{C_{x_1^3}(\omega_1, 0) - H_{12}(0) \cdot C_{x_1^2 x_2}(\omega_1, 0)} \tag{4.32}$$

The only unknowns in the derived equations for estimation of the bispectra of the filters are $H_{12}(0)$ and $H_{21}(0)$, i.e. the DC components ($\omega = 0$) of the Fourier transforms of the filters. An approach to estimate these DC components is derived in the next section.

4.1 Estimation of $\bar{\bar{H}}(0)$

In this section a method for estimating $\bar{\bar{H}}(0)$ is derived, the method is presented in [8].
If the powerspectrum of the output is evaluated in $\omega = 0$, it would be an estimator of what the DC component is in the mixed signals $x(t)$ are. If all possible power spectra's and cross spectra's of the mixed signals are constructed, it would give the matrix in equation 4.33.

$$\begin{bmatrix} C_{x_1^2}(\omega) & C_{x_2 x_1}(\omega) \\ C_{x_1 x_2}(\omega) & C_{x_2^2}(\omega) \end{bmatrix} = \begin{bmatrix} X_1^*(\omega) \cdot X_1(\omega) & X_2^*(\omega) \cdot X_1(\omega) \\ X_1^*(\omega) \cdot X_2(\omega) & X_2^*(\omega) \cdot X_2(\omega) \end{bmatrix} \tag{4.33}$$

The power spectrum of $x(t)$ can be expended into a function of h and $s(t)$.By using equations 4.1 and 4.2 from the TITO system model the first element in equation 4.33 can be rewritten into equation 4.34:

$$\begin{aligned} C_{x_1^2}(\omega) &= X_1^*(\omega) \cdot X_1(\omega) = \langle S_1^*(\omega) + H_{12}^*(\omega) \cdot S_2^*(\omega) \rangle \cdot \langle S_1(\omega) + H_{12}(\omega) \cdot S_2(\omega) \rangle \\ &= S_1^*(\omega) \cdot S_1(\omega) + S_1^*(\omega) \cdot H_{12}(\omega) \cdot S_2(\omega) + H_{12}^*(\omega) \cdot S_2^*(\omega) \cdot S_1(\omega) \\ &\quad + H_{12}^*(\omega) \cdot S_2^*(\omega) \cdot H_{12}(\omega) \cdot S_2(\omega) \end{aligned} \tag{4.34}$$

Equation 4.34 can be written in a matrix form as equation 4.35:

$$\begin{bmatrix} C_{x_1^2}(\omega) & \$ \\ \$ & \$ \end{bmatrix} = \begin{bmatrix} 1 & H_{12}(\omega) \\ \$ & \$ \end{bmatrix} \cdot \begin{bmatrix} S_1^*(\omega) \cdot S_1(\omega) & S_2^*(\omega) \cdot S_1(\omega) \\ S_1^*(\omega) \cdot S_2(\omega) & S_2^*(\omega) \cdot S_2(\omega) \end{bmatrix} \cdot \begin{bmatrix} 1 & \$ \\ H_{12}^*(\omega) & \$ \end{bmatrix} \tag{4.35}$$

where:
\$ can be any value.

Similarly the elements $C_{x_2^2}(\omega)$ $C_{x_1x_2}(\omega)$ and $C_{x_2x_1}(\omega)$ in 4.35 can be expanded in matrix form as:

$$\begin{bmatrix} \$ & \$ \\ \$ & C_{x_1^2}(\omega) \end{bmatrix} = \begin{bmatrix} \$ & \$ \\ H_{21}(\omega) & 1 \end{bmatrix} \cdot \begin{bmatrix} S_1^*(\omega) \cdot S_1(\omega) & S_2^*(\omega) \cdot S_1(\omega) \\ S_1^*(\omega) \cdot S_2(\omega) & S_2^*(\omega) \cdot S_2(\omega) \end{bmatrix} \cdot \begin{bmatrix} \$ & H_{21}^*(\omega) \\ \$ & 1 \end{bmatrix} \quad (4.36)$$

$$\begin{bmatrix} \$ & C_{x_2x_1}(\omega) \\ \$ & \$ \end{bmatrix} = \begin{bmatrix} 1 & H_{12}(\omega) \\ \$ & \$ \end{bmatrix} \cdot \begin{bmatrix} S_1^*(\omega) \cdot S_1(\omega) & S_2^*(\omega) \cdot S_1(\omega) \\ S_1^*(\omega) \cdot S_2(\omega) & S_2^*(\omega) \cdot S_2(\omega) \end{bmatrix} \cdot \begin{bmatrix} \$ & H_{21}^*(\omega) \\ \$ & 1 \end{bmatrix} \quad (4.37)$$

$$\begin{bmatrix} \$ & \$ \\ C_{x_1x_2}(\omega) & \$ \end{bmatrix} = \begin{bmatrix} \$ & \$ \\ H_{21}(\omega) & 1 \end{bmatrix} \cdot \begin{bmatrix} S_1^*(\omega) \cdot S_1(\omega) & S_2^*(\omega) \cdot S_1(\omega) \\ S_1^*(\omega) \cdot S_2(\omega) & S_2^*(\omega) \cdot S_2(\omega) \end{bmatrix} \cdot \begin{bmatrix} 1 & \$ \\ H_{12}^*(\omega) & \$ \end{bmatrix} \quad (4.38)$$

Combining equations 4.35 to 4.38 gives equation 4.39:

$$\begin{aligned} \begin{bmatrix} C_{x_1^2}(\omega) & C_{x_2x_1}(\omega) \\ C_{x_1x_2}(\omega) & C_{x_2^2}(\omega) \end{bmatrix} &= \begin{bmatrix} 1 & H_{12}(\omega) \\ H_{21}(\omega) & 1 \end{bmatrix} \cdot \begin{bmatrix} C_{s_1^2}(\omega) & C_{s_2s_1}(\omega) \\ C_{s_1s_2}(\omega) & C_{s_2^2}(\omega) \end{bmatrix} \cdot \begin{bmatrix} 1 & H_{21}^*(\omega) \\ H_{12}^*(\omega) & 1 \end{bmatrix} \\ \bar{\bar{C}}_{x^2}(\omega) &= \bar{\bar{H}}(\omega) \cdot \bar{\bar{C}}_{s^2}(\omega) \cdot \bar{\bar{H}}^H(\omega) \end{aligned} \quad (4.39)$$

From equation 4.33 it can be seen that $\bar{\bar{C}}_{x^2}(0)$ is equal to its own conjugate transpose, so $\bar{\bar{C}}_{x^2}(0)$ is hermitian, therefore the eigenvalue decomposition of $\bar{\bar{C}}_{x^2}(0)$ can be defined as in equation 4.40.

$$\bar{\bar{C}}_{x^2}(0) = \bar{\bar{U}} \cdot \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \cdot \bar{\bar{U}}^H \quad (4.40)$$

where:
$\bar{\bar{U}}$ is a orthogonal matrix that contains the the eigenvectors of $\bar{\bar{C}}_x(0)$
λ is the eigenvalues of $\bar{\bar{C}}_x(0)$

Comparing equation 4.39 to equation 4.40 it is seen that the right hand sides have similar structure provided the anti diagonal of $\bar{\bar{C}}_{s^2}(0)$ is zero. This is reasonable to assume as the sources s_1 and s_2 are independent, cf. section 2.1, and as such the cross spectra, $C_{s_1s_2}(\omega)$ and $C_{s_2s_1}(\omega)$, are zero. This means λ_1 and λ_2 are scaled versions of $C_{s_1^2}(0)$ and $C_{s_2^2}(0)$, respectively, while $\bar{\bar{U}}$ is some transformation of $\bar{\bar{H}}(0)$. Using this similar structure it may be possible to estimate $\bar{\bar{H}}(0)$.

Consider the transformation matrix $\bar{\bar{T}}$ defined in equation 4.41.

$$\bar{\bar{T}} = \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}} & 0 \\ 0 & \frac{1}{\sqrt{\lambda_2}} \end{bmatrix} \cdot \bar{\bar{U}}^H \quad (4.41)$$

This transformation would make the matrix $\bar{\bar{C}}_{x^2}(0)$ into identity $\bar{\bar{I}}$ as defined in equation 4.42.

$$\begin{aligned} \bar{\bar{C}}_{y^2}(0) &= \bar{\bar{T}} \cdot \bar{\bar{C}}_{x^2}(0) \cdot \bar{\bar{T}}^H \\ &= \bar{\bar{T}} \cdot \bar{\bar{U}} \cdot \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \cdot \bar{\bar{U}}^H \cdot \bar{\bar{T}}^H \\ &= \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}} & 0 \\ 0 & \frac{1}{\sqrt{\lambda_2}} \end{bmatrix} \cdot \bar{\bar{U}}^H \cdot \bar{\bar{U}} \cdot \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \cdot \bar{\bar{U}}^H \cdot \bar{\bar{U}} \cdot \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}}^* & 0 \\ 0 & \frac{1}{\sqrt{\lambda_2}}^* \end{bmatrix} \\ &= \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}} & 0 \\ 0 & \frac{1}{\sqrt{\lambda_2}} \end{bmatrix} \cdot \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}} & 0 \\ 0 & \frac{1}{\sqrt{\lambda_2}} \end{bmatrix} \\ &= \bar{\bar{I}} \end{aligned} \quad (4.42)$$

This is interesting because if the same is applied to 4.39 for $\omega = 0$ it would. It would also turn into identity if applied to equation 4.39.

$$\bar{\bar{I}} = \bar{\bar{T}} \cdot \bar{\bar{H}}(0) \cdot \bar{\bar{C}}_{s^2}(0) \cdot \bar{\bar{H}}^H(0) \cdot \bar{\bar{T}}^H \tag{4.43}$$

If it is assumed that the source signal are uncorrelated, $\bar{\bar{C}}_{s^2}(0)$, would only have values on the diagonal and these values would be the DC-component for the individual source signals. What the transformation matrix $\bar{\bar{T}}$ does is to scale the DC-component int source signals to one. So if a new matrix $\bar{\bar{G}}(0)$ is defined as in equation 4.44:

$$\bar{\bar{G}}(0) = \bar{\bar{T}} \cdot \bar{\bar{H}}(0) \tag{4.44}$$

Then $\bar{\bar{G}}(0)$ would be a "filter" that turns the source signals DC-component into identity and if it is possible to estimate this $\bar{\bar{G}}(0)$, it would be possible to possible to estimate $\bar{\bar{H}}(0)$ as:

$$\bar{\bar{H}}(0) = \bar{\bar{T}}^{-1} \cdot \bar{\bar{G}}(0) \tag{4.45}$$

The "filter" $\bar{\bar{G}}(0)$ has some special proberties. If $\bar{\bar{G}}(0)^H$ and $\bar{\bar{G}}(0)^{-H}$ is applied to equation 4.43, it results in equation 4.46.

$$\begin{aligned}
\bar{\bar{I}} &= \bar{\bar{G}}(0) \cdot \bar{\bar{C}}_{s^2}(0) \cdot \bar{\bar{G}}^H(0) \\
\bar{\bar{G}}^H(0) \cdot I \cdot \bar{\bar{G}}^{-H}(0) &= \bar{\bar{G}}^H(0) \cdot \bar{\bar{G}}(0) \cdot \bar{\bar{C}}_{s^2}(0) \cdot \bar{\bar{G}}^H(0) \cdot \bar{\bar{G}}^{-H}(0) \\
\bar{\bar{I}} &= \bar{\bar{G}}^H(0) \cdot \bar{\bar{G}}(0) \cdot \bar{\bar{C}}_{s^2}(0) \\
\bar{\bar{C}}_{s^2}^{-1}(0) &= \bar{\bar{G}}^H(0) \cdot \bar{\bar{G}}(0)
\end{aligned} \tag{4.46}$$

So by multiplying $\bar{\bar{G}}(0)$ hermitian with itself the result is the inverted $\bar{\bar{C}}_{s^2}(0)$. As $\bar{\bar{C}}_{s^2}^{-1}(0)$ only contains elements on the diagonal, it can be shown that the column vectors in $\bar{\bar{G}}(0)$ are actually orthogonal to each other which is proven in equation 4.47 to 4.51.

$$\bar{\bar{C}}_{s^2}^{-1}(0) = \begin{bmatrix} G_{11}^*(0) & G_{21}^*(0) \\ G_{12}^*(0) & G_{22}^*(0) \end{bmatrix} \begin{bmatrix} G_{11}(0) & G_{12}(0) \\ G_{21}(0) & G_{22}(0) \end{bmatrix} \tag{4.47}$$

$$\begin{bmatrix} C_{s_1^2}^{-1}(0) & 0 \\ 0 & C_{s_2^2}^{-1}(0) \end{bmatrix} = \begin{bmatrix} \bar{G}_1^H(0) \\ \bar{G}_2^H(0) \end{bmatrix} \begin{bmatrix} \bar{G}_1(0) & \bar{G}_2(0) \end{bmatrix} \tag{4.48}$$

$$\Downarrow$$

$$0 = \bar{G}_1^H(0) \cdot \bar{G}_2(0) \tag{4.49}$$

$$0 = \bar{G}_2^H(0) \cdot \bar{G}_1(0) \tag{4.50}$$

$$\Downarrow$$

$$\bar{G}_1(0) \perp \bar{G}_2(0) \tag{4.51}$$

It can also be shown from equations 4.52 to 4.59 that $G_{11}(0)$ and $G_{22}(0)$ only contains real values and $G_{12}(0)$, $G_{21}(0)$ are the complex conjugated of each other.

$$G_{11}(\omega) = G_{11}^*(\omega) \tag{4.52}$$

$$G_1^*(\omega) \cdot G_1(\omega) = (G_1^*(\omega) \cdot G_1(\omega))^* \tag{4.53}$$

$$G_{22}(\omega) = G_{22}^*(\omega) \tag{4.54}$$

$$G_2^*(\omega) \cdot G_2(\omega) = (G_2^*(\omega) \cdot G_2(\omega))^* \tag{4.55}$$

$$G_{12}(\omega) = G_{21}^*(\omega) \tag{4.56}$$

$$G_1^*(\omega) \cdot G_2(\omega) = (G_2^*(\omega) \cdot G_1(\omega))^* \tag{4.57}$$

$$G_{21}(\omega) = G_{12}^*(\omega) \tag{4.58}$$

$$G_2^*(\omega) \cdot G_1(\omega) = (G_1^*(\omega) \cdot G_2(\omega))^* \tag{4.59}$$

In order to estimate $\bar{\bar{G}}(0)$, the cross trispectrum of the transformed signal $y(t)$ is evaluated, the first step is the create the transformed signal $y(t)$ from $x(t)$ which can be done using the transformation matrix as in equation 4.60

$$\begin{bmatrix} Y_1(\omega) \\ Y_2(\omega) \end{bmatrix} = \bar{\bar{T}} \cdot \begin{bmatrix} X_1(\omega) \\ X_2(\omega) \end{bmatrix} \tag{4.60}$$

In order to estimate $\bar{\bar{G}}(0)$ it can according to [8, p. 522] be done by evaluating the "average" cross trispectrum of $y(t)$ at $\omega = 0$. The "average" cross trispectrum is defined in equation 4.61.

$$C_{y^4_{avg}}(0,0,0) = \begin{bmatrix} C_{y_1^4}(0,0,0) + C_{y_2^2 y_1^2}(0,0,0) & C_{y_1^3 y_2}(0,0,0) + C_{y_2^2 y_1 y_2}(0,0,0) \\ C_{y_1^2 y_2 y_1}(0,0,0) + C_{y_2^3 y_1}(0,0,0) & C_{y_1^2 y_2^2}(0,0,0) + C_{y_2^4}(0,0,0) \end{bmatrix} \tag{4.61}$$

$$= \begin{bmatrix} C_{avg_{11}} & C_{avg_{12}} \\ C_{avg_{21}} & C_{avg_{22}} \end{bmatrix} \tag{4.62}$$

From the multilinearity properties of cumulant spectra's (se appendix A.7.1) it is possible to rewrite equation 4.61 to a function of $C_{s^4}(0,0,0)$ and $G(0)$

$$\begin{aligned} C_{avg_{11}} &= G^*_{11} \cdot G_{11} \cdot G_{11} \cdot G_{11} \cdot C_{s_1^4} + G^*_{12} \cdot G_{12} \cdot G_{12} \cdot G_{12} \cdot C_{s_2^4} + \\ &\quad G^*_{21} \cdot G_{21} \cdot G_{11} \cdot G_{11} \cdot C_{s_1^4} + G^*_{22} \cdot G_{22} \cdot G_{12} \cdot G_{12} \cdot C_{s_2^4} \qquad (4.63) \\ C_{avg_{12}} &= G^*_{11} \cdot G_{11} \cdot G_{11} \cdot G_{21} \cdot C_{s_1^4} + G^*_{12} \cdot G_{12} \cdot G_{12} \cdot G_{22} \cdot C_{s_2^4} + \\ &\quad G^*_{21} \cdot G_{21} \cdot G_{11} \cdot G_{21} \cdot C_{s_1^4} + G^*_{22} \cdot G_{22} \cdot G_{12} \cdot G_{22} \cdot C_{s_2^4} \qquad (4.64) \\ C_{avg_{21}} &= G^*_{11} \cdot G_{11} \cdot G_{21} \cdot G_{11} \cdot C_{s_1^4} + G^*_{12} \cdot G_{12} \cdot G_{22} \cdot G_{12} \cdot C_{s_2^4} + \\ &\quad G^*_{21} \cdot G_{21} \cdot G_{21} \cdot G_{11} \cdot C_{s_1^4} + G^*_{22} \cdot G_{22} \cdot G_{22} \cdot G_{12} \cdot C_{s_2^4} \qquad (4.65) \\ C_{avg_{22}} &= G^*_{11} \cdot G_{11} \cdot G_{21} \cdot G_{21} \cdot C_{s_1^4} + G^*_{12} \cdot G_{12} \cdot G_{22} \cdot G_{22} \cdot C_{s_2^4} + \\ &\quad G^*_{21} \cdot G_{21} \cdot G_{21} \cdot G_{21} \cdot C_{s_1^4} + G^*_{22} \cdot G_{22} \cdot G_{22} \cdot G_{22} \cdot C_{s_2^4} \qquad (4.66) \end{aligned}$$

From equation 4.48 it was show that the column vector of $\bar{\bar{G}}(0)$ multiplied with its hermitian is the inverse of C_{s^2}.

$$\begin{aligned} C_{avg_{11}} &= \left(G^*_{11} \cdot G_{11} + G^*_{21} \cdot G_{21}\right) \cdot G_{11} \cdot G_{11} \cdot C_{s_1^4} + \\ &\quad \left(G^*_{12} \cdot G_{12} + G^*_{22} \cdot G_{22}\right) \cdot G_{12} \cdot G_{12} \cdot C_{s_2^4} \\ &= \left(\frac{1}{C_{s_1^2}}\right) \cdot G_{11} \cdot G_{11} \cdot C_{s_1^4} + \\ &\quad \left(\frac{1}{C_{s_2^2}}\right) \cdot G_{12} \cdot G_{12} \cdot C_{s_2^4} \qquad (4.67) \end{aligned}$$

If this is done for all the equations from 4.63 to 4.66 they would reduce to equations 4.68 to 4.71.

$$C_{avg_{11}} = \frac{C_{s_1^4}}{C_{s_1^2}} \cdot G_{11} \cdot G_{11} + \frac{C_{s_2^4}}{C_{s_2^2}} \cdot G_{12} \cdot G_{12} \tag{4.68}$$

$$C_{avg_{12}} = \frac{C_{s_1^4}}{C_{s_1^2}} \cdot G_{11} \cdot G_{21} + \frac{C_{s_2^4}}{C_{s_2^2}} \cdot G_{12} \cdot G_{22} \tag{4.69}$$

$$C_{avg_{21}} = \frac{C_{s_1^4}}{C_{s_1^2}} \cdot G_{21} \cdot G_{11} + \frac{C_{s_2^4}}{C_{s_2^2}} \cdot G_{22} \cdot G_{12} \tag{4.70}$$

$$C_{avg_{22}} = \frac{C_{s_1^4}}{C_{s_1^2}} \cdot G_{21} \cdot G_{21} + \frac{C_{s_2^4}}{C_{s_2^2}} \cdot G_{22} \cdot G_{22} \tag{4.71}$$

If equations 4.68 to 4.71 is written into matrix the result can be reduced into equation 4.76.

$$\bar{\bar{C}}_{y^4_{avg}}(0,0,0) = \begin{bmatrix} \frac{C_{s_1^4}}{C_{s_1^2}} \cdot G_{11} \cdot G_{11} + \frac{C_{s_2^4}}{C_{s_2^2}} \cdot G_{12} \cdot G_{12} & , \frac{C_{s_1^4}}{C_{s_1^2}} \cdot G_{11} \cdot G_{21} + \frac{C_{s_2^4}}{C_{s_2^2}} \cdot G_{12} \cdot G_{22} \\ \frac{C_{s_1^4}}{C_{s_1^2}} \cdot G_{21} \cdot G_{11} + \frac{C_{s_2^4}}{C_{s_2^2}} \cdot G_{22} \cdot G_{12} & , \frac{C_{s_1^4}}{C_{s_1^2}} \cdot G_{21} \cdot G_{21} + \frac{C_{s_2^4}}{C_{s_2^2}} \cdot G_{22} \cdot G_{22} \end{bmatrix} \tag{4.72}$$

$$= \frac{C_{s_1^4}}{C_{s_1^2}} \cdot \begin{bmatrix} G_{11} \cdot G_{11} & G_{11} \cdot G_{21} \\ G_{21} \cdot G_{11} & G_{21} \cdot G_{21} \end{bmatrix} + \frac{C_{s_2^4}}{C_{s_2^2}} \cdot \begin{bmatrix} G_{12} \cdot G_{12} & G_{12} \cdot G_{22} \\ G_{22} \cdot G_{12} & G_{22} \cdot G_{22} \end{bmatrix} \tag{4.73}$$

$$\bar{\bar{G}}(0) = \begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix} = \begin{bmatrix} \bar{G}_1(0) & \bar{G}_2(0) \end{bmatrix} \tag{4.74}$$

$$\bar{\bar{C}}_{y^4_{avg}}(0,0,0) = \frac{C_{s_1^4}}{C_{s_1^2}} \cdot \bar{G}_1(0) \cdot \bar{G}_1(0)^T + \frac{C_{s_2^4}}{C_{s_2^2}} \cdot \bar{G}_2(0) \cdot \bar{G}_2(0)^T \tag{4.75}$$

$$= \sum_{i=1}^{2} \frac{C_{s_i^4}}{C_{s_i^2}} \cdot \bar{G}_i(0) \cdot \bar{G}_i(0)^T \tag{4.76}$$

The result for the "averaged" trispectrum in equation 4.76 is now evaluated, if an eigenvalue decomposition of $\bar{\bar{C}}_{y^4_{avg}}(0,0,0)$ is performed it can be written as equations 4.77 and 4.78.

$$\bar{\bar{C}}_{y^4_{avg}}(0,0,0) = \bar{\bar{U}} \cdot \bar{\bar{\lambda}} \cdot \bar{\bar{U}}^H \tag{4.77}$$

$$= \begin{bmatrix} U_{11} & U_{21} \\ U_{12} & U_{22} \end{bmatrix} \cdot \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix} \tag{4.78}$$

Equation 4.78 can be expended into equation 4.79.

$$\bar{\bar{C}}_{y^4_{avg}}(0,0,0) = \begin{bmatrix} \lambda_1 U_{11}^2 + \lambda_2 U_{21}^2 & \lambda_1 U_{11} U_{12} + \lambda_2 U_{21} U_{22} \\ \lambda_1 U_{12} U_{11} + \lambda_2 U_{22} U_{21} & \lambda_1 U_{12}^2 + \lambda_2 U_{22}^2 \end{bmatrix} \tag{4.79}$$

As it is known that the column vectors in $\bar{\bar{G}}(0)$ are orthogonal and the column vectors of $\bar{\bar{U}}$ also are orthogonal. Then when comparing equation 4.72 to 4.79, it becomes obvious that $\bar{\bar{G}}(0)$ is given as the eigenvectors of $\bar{\bar{C}}_{y^4_{avg}}(0,0,0)$ with some constant that depends on λ_i and $\frac{C_{s_i^4}}{C_{s_i^2}}$. This scaling ambiguity k_i can be solved by the knowledge that the diagonal of $\bar{\bar{H}}(0)$ is one. From

equation 4.45 the relation between $\bar{\bar{H}}(0)$ and $\bar{\bar{T}}$ and $\bar{\bar{G}}(0)$ is as follows:

$$\bar{\bar{H}}(0) = \bar{\bar{T}}^{-1} \cdot \bar{\bar{k}} \cdot \bar{\bar{G}}(0) \tag{4.80}$$

$$\begin{bmatrix} 1 & h_{12} \\ h_{21} & 1 \end{bmatrix} = \begin{bmatrix} T_{11}^{-1} & T_{21}^{-1} \\ T_{12}^{-1} & T_{22}^{-1} \end{bmatrix} \cdot \begin{bmatrix} k_1 G_{11}(0) & k_2 G_{12}(0) \\ k_1 G_{21}(0) & k_2 G_{22}(0) \end{bmatrix} \tag{4.81}$$

The scaling can from the diagonal be solved as:

$$k_1 = \frac{1}{T_{11}^{-1} G_{11}(0) + T_{21}^{-1} G_{21}(0)} \tag{4.82}$$

$$k_2 = \frac{1}{T_{12}^{-1} G_{12}(0) + T_{22}^{-1} G_{22}(0)} \tag{4.83}$$

The next problem to handle is the shuffling of the eigenvectors from $\bar{\bar{C}}_{y^4_{avg}}(0,0,0)$, as there is no unique way of assigning them to columns of $\bar{\bar{G}}(0)$. The way they are assigned to $\bar{\bar{G}}(0)$ is also dependent on the matrix $\bar{\bar{T}}$. Which again is dependent on how the eigenvalues from the eigenvalue decomposition of $\bar{\bar{C}}_{x^2}(0)$ is assigned. Giving a total of four different results dependent on how the eigenvectors are organized from the two eigenvector decompositions. But there is only one correct organization of the columns of $\bar{\bar{G}}(0)$, but this cannot be determined, without knowing something about the filters.

There is therefore two ways of constructing the $\bar{\bar{G}}(0)$ matrix ($\bar{\bar{G}}_1$ and a flipped version $\bar{\bar{G}}_2$) as well as two ways of constructing $\bar{\bar{T}}$ named ($\bar{\bar{T}}_1$ and $\bar{\bar{T}}_2$). The connection between $\bar{\bar{G}}_1$ and a flipped version $\bar{\bar{G}}_2$ is stated in equation 4.84 and 4.85.

$$\bar{\bar{G}}_1 = \begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix} \tag{4.84}$$

$$\bar{\bar{G}}_2 = \begin{bmatrix} G_{12} & G_{11} \\ G_{22} & G_{21} \end{bmatrix} \tag{4.85}$$

Also the connection between $\bar{\bar{T}}_1$ and $\bar{\bar{T}}_2$ can from equation 4.41 be derived as equation 4.87 and 4.89, that shows what happens if the if the eigenvalues and the eigenvectors are swapped.

$$\begin{aligned} \bar{\bar{T}}_1 &= \bar{\bar{\lambda}}_1^{-\frac{1}{2}} \cdot \bar{U}_1^H \\ &= \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}} & 0 \\ 0 & \frac{1}{\sqrt{\lambda_2}} \end{bmatrix} \cdot \begin{bmatrix} U_{11}^* & U_{21}^* \\ U_{12}^* & U_{22}^* \end{bmatrix} \end{aligned} \tag{4.86}$$

$$= \begin{bmatrix} \frac{U_{11}^*}{\sqrt{\lambda_1}} & \frac{U_{21}^*}{\sqrt{\lambda_1}} \\ \frac{U_{12}^*}{\sqrt{\lambda_2}} & \frac{U_{22}^*}{\sqrt{\lambda_2}} \end{bmatrix} = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} \tag{4.87}$$

$$\begin{aligned} \bar{\bar{T}}_2 &= \bar{\bar{\lambda}}_2^{-\frac{1}{2}} \cdot \bar{U}_2^H \\ &= \begin{bmatrix} \frac{1}{\sqrt{\lambda_2}} & 0 \\ 0 & \frac{1}{\sqrt{\lambda_1}} \end{bmatrix} \cdot \begin{bmatrix} U_{12}^* & U_{22}^* \\ U_{11}^* & U_{21}^* \end{bmatrix} \end{aligned} \tag{4.88}$$

$$= \begin{bmatrix} \frac{U_{12}^*}{\sqrt{\lambda_2}} & \frac{U_{22}^*}{\sqrt{\lambda_2}} \\ \frac{U_{11}^*}{\sqrt{\lambda_1}} & \frac{U_{21}^*}{\sqrt{\lambda_1}} \end{bmatrix} = \begin{bmatrix} T_{21} & T_{22} \\ T_{11} & T_{12} \end{bmatrix} \tag{4.89}$$

The consequence of swapping the eigenvalues and the eigenvectors for $\bar{\bar{T}}$ is that the rows of $\bar{\bar{T}}$ are going to be swapped. If this swapped transformation matrix is applied to $\bar{x}(t)$, it would result

in the rows of $\bar{y}(t)$ being swapped as well, effectively swapping $y_1(t)$ and $y_2(t)$. This swapping of the output vectors would then result in the two filters in $\bar{\bar{H}}(0)$ being swapped. So placing the swapping the eigenvectors in $\bar{\bar{T}}$ would change the swap the places of $H_{12}(0)$ and $H_{21}(0)$ in the $\bar{\bar{H}}(0)$ matrix. The last thing to investigate is the effect of the columns of $\bar{\bar{G}}$ being swapped. As the effect on $\bar{\bar{H}}$ is evaluated the first thing is to establish is the inverse of the matrix $\bar{\bar{T}}$, which is derived in equation 4.91.

$$\bar{\bar{T}}_1^{-1} = \frac{1}{T_{11}T_{22} - T_{12}T_{21}} \begin{bmatrix} T_{22} & -T_{12} \\ -T_{21} & T_{11} \end{bmatrix} \tag{4.90}$$

$$= \frac{1}{\det(T)} \begin{bmatrix} T_{22} & -T_{12} \\ -T_{21} & T_{11} \end{bmatrix} \tag{4.91}$$

Now $\bar{\bar{H}}$ can be derived using equation 4.80. As there is two different scenarios ($\bar{\bar{G}}_1$ and $\bar{\bar{G}}_2$). $\bar{\bar{G}}_1$ where the columns are not swapped is the first to be evaluated, which is done in equation 4.92 and 4.93.

$$\bar{\bar{T}}_1^{-1} \cdot \bar{\bar{k}} \cdot \bar{\bar{G}}_1 = \frac{1}{\det(T)} \begin{bmatrix} T_{22} & -T_{12} \\ -T_{21} & T_{11} \end{bmatrix} \cdot \begin{bmatrix} k_1 G_{11} & k_2 G_{12} \\ k_1 G_{21} & k_2 G_{22} \end{bmatrix} \tag{4.92}$$

$$= \frac{1}{\det(T)} \begin{bmatrix} k_1(T_{22}G_{11} - T_{12}G_{21}) & k_2(T_{22}G_{12} - T_{12}G_{22}) \\ k_1(-T_{21}G_{11} + T_{11}G_{21}) & k_2(-T_{21}G_{12} + T_{11}G_{22}) \end{bmatrix} \tag{4.93}$$

As the diagonal in $\bar{\bar{H}}(0)$ is know to be one, k_1 and k_2 in equation 4.93 are scaled to achieve this.

$$\bar{\bar{H}}_{11} = \begin{bmatrix} 1 & \frac{(T_{22}G_{12} - T_{12}G_{22})}{(-T_{21}G_{12} + T_{11}G_{22})} \\ \frac{(-T_{21}G_{11} + T_{11}G_{21})}{(T_{22}G_{11} - T_{12}G_{21})} & 1 \end{bmatrix} \tag{4.94}$$

The same procedure is now performed on $\bar{\bar{G}}_2$ where the columns are swapped.

$$\bar{\bar{T}}_1^{-1} \cdot \bar{\bar{k}} \cdot \bar{\bar{G}}_2 = \frac{1}{\det(T)} \begin{bmatrix} k_1(T_{22}G_{12} - T_{12}G_{22}) & k_2(T_{22}G_{11} - T_{12}G_{21}) \\ k_1(-T_{21}G_{12} + T_{11}G_{22}) & k_2(-T_{21}G_{11} + T_{11}G_{21}) \end{bmatrix} \tag{4.95}$$

Scaling k_1 and k_2 so there is one on the diagonal gives:

$$\bar{\bar{H}}_{12} = \begin{bmatrix} 1 & \frac{(T_{22}G_{11} - T_{12}G_{21})}{(-T_{21}G_{11} + T_{11}G_{21})} \\ \frac{(-T_{21}G_{12} + T_{11}G_{22})}{(T_{22}G_{12} - T_{12}G_{22})} & 1 \end{bmatrix} \tag{4.96}$$

Comparing equation 4.96 with equation 4.94, it can be concluded that if the columns of $\bar{\bar{G}}(0)$ are swapped the filters in the output are going to be inverted.

The conclusion from the ambiguity when organizing the eigenvalues/eigenvectors is the that the estimated DC-coefficients for the filters can be swapped with each other and/or inverted, giving a total of four possible solutions for the DC-coefficient of the filters. If no information about the DC-coefficients are know priori then there is no way of solving this problem. If however it is know priori that the DC-coefficient is smaller then one, which would be reasonable to assume for most channel models, the inversion of the filters can be handled and there is only two possible solutions. Both solutions could be tried and the one with the lowest correlation between the two sources would most likely be the correct solution.

If the estimated DC-coefficient for $\bar{\bar{H}}$ can be assign to a filter it is now possible from equation 4.30 to estimate the bispectrum of the filters. The last step would therefore be to recover the filter coefficient from these bispectrum.

4.2 Reconstruction of the Filter

As the algorithm presented in [8] does not provide the impulse response of filters, but the third order moment spectrum of the filters, it is necessary to reconstruct the impulse response from these. In the paper from [8, 520] it is suggested to use an algorithm presented by [4] to derive this impulse response.

The algorithm in [4] works by reconstructing the phase response and the magnitude response from third order momentspectrum sepratly. These two estimates are then combined to create the frequency responce of the filter. The last step is therefor to apply the inverse Fourier Transform to derive the impulse response of the filter.

4.2.1 Estimation of the Phase Response

The third order moment spectra of the filter can in the frequency domain be calculated as equation 4.97:

$$C_{3H}(\omega_1, \omega_2) = H(\omega_1) \cdot H(\omega_2) \cdot H^*(\omega_1 + \omega_2) \tag{4.97}$$

The derivation of this can be seen in appendix A.3 and appendix A.7.1. Consequently the phase for the spectrum in equation 4.97 is derived as equation 4.98.

$$\phi_{3h}(\omega_1, \omega_2) = \phi_h(\omega_1) + \phi_h(\omega_2) - \phi_h(\omega_1 + \omega_2) \tag{4.98}$$

Equation 4.98 can be rewritten to make it into matrix form, but several simplifications are applied first. First simplification is the first entry of ϕ_h ($\omega = 0$) is the DC component and it would be reasonable to assumed that it has a phase of zero, i.e. $\phi_h(0) = 0$, and can therfor be removed. Also all calculations in equation 4.97 where either ω_1 or ω_2 are zero will result in zero phase, meaning they can also be removed. Furthermore the symmetry conditions for cumulant spectras, that also applies to the third order moment spectra, (see appendix A.5 for cumulant sym. cond.). Allows for the removal of redundant component from the third order moment spectra reduces the number of entries from $\phi_{3h}(\omega_1, \omega_2)$ nedded to estimate the original phase. If it is assumed that the size of the Fourier transform have an even number of frequency bins, i.e. N is even, and all the simplifications are taken into considerations equation 4.97 can on matrix form be rewritten as equation 4.99.

$$\begin{bmatrix} \phi_{3h}(1,1) \\ \phi_{3h}(1,2) \\ \phi_{3h}(1,3) \\ \vdots \\ \phi_{3h}(1,N-1) \\ \phi_{3h}(2,2) \\ \vdots \\ \phi_{3h}(\frac{N}{2}, N-\frac{N}{2}) \end{bmatrix} = \begin{bmatrix} 2 & -1 & 0 & 0 & \cdots & 0 & \cdots & 0 \\ 1 & 1 & -1 & 0 & \cdots & 0 & \cdots & 0 \\ 1 & 0 & 1 & -1 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 0 & 0 & \cdots & 0 & \cdots & 1 \\ 0 & 2 & 0 & -1 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 2 & \cdots & 0 \end{bmatrix} \cdot \begin{bmatrix} \phi_h(1) \\ \phi_h(2) \\ \phi_h(3) \\ \phi_h(4) \\ \vdots \\ \phi_h(\frac{N}{2}) \\ \vdots \\ \phi_h(N-1) \end{bmatrix} \tag{4.99}$$

$$\bar{\phi}_{3h} = \bar{\bar{A}}_\phi \cdot \bar{\phi}_h \tag{4.100}$$

To find the phase of the filter ($\bar{\phi}_h$). Equation 4.101 provides the optimum solution in the least square sense for solving the phase of the filter.

$$\bar{\phi}_h = \left(\bar{\bar{A}}_\phi^T \bar{\bar{A}}_\phi\right)^{-1} \bar{\bar{A}}_\phi^T \cdot \bar{\phi}_{3h} \tag{4.101}$$

There is however one problem with this approach as principal value of the phase in $\bar{\phi}_h$ lies in the interval $[-\pi;\pi[$, then from equation 4.98, the principal value of the phase in $\bar{\phi}_{3h}$ must lies in the interval $[-3\pi;3\pi[$, but when estimated from the third order moment spectrum, it would only be in the interval $[-\pi;\pi[$. This means that there is a phase ambiguity in 4.101 that must be addressed to ensure that the result lies in the $[-\pi;\pi[$ interval, which is done by adding a term that corrects the phase error in $\bar{\phi}_{3h}$. This corrections is also referred to as phase unwrapping.

$$\bar{\phi}_{3h} + 2\pi \cdot \bar{k} = \bar{\bar{A}}_\phi \cdot \bar{\phi}_h \tag{4.102}$$

Where:
$\bar{k} \subset [-1,\ 0,\ 1]$

The next step is to determine $\bar{k}$. This can be done by using the less accurate Bartelt-Lohman-Wirnitzer algorithm, that does not requires phase unwrapping. Accuracy is however not a problem as the only interest is to use it to estimate $\bar{k}$ and this allows an error up to π before accuracy becomes a problem. Based on equation 4.98 the algorithm is as follows:

$$\phi_{3h}(\omega_1,\omega_2) = \phi_h(\omega_1) + \phi_h(\omega_2) - \phi_h(\omega_1+\omega_2) \tag{4.103}$$

By setting $\omega_2 = \omega_2 - \omega_1$ and $\omega_1 = 1$ in equation 4.103 yields:

$$\phi_{3h}(1,\omega_2-1) = \phi_h(1) + \phi_h(\omega_2-1) - \phi_h(\omega_2) \tag{4.104}$$

Equation 4.104 can be rewritten to equation 4.106, which is a recursive algorithm for estimating the phase.

$$\phi_h(\omega_2) = \phi_h(1) + \phi_h(\omega_2-1) - \phi_{3h}(1,\omega_2-1) \tag{4.105}$$

Compared to equation 4.98 the algorithm only works on the upper part of the moment spectrum as ω_1 is fixed to one. Assuming that the phase of the first component, $\phi_h(1)$, is equal to zero equation 4.117 is a matrix solution for determining the phase for $\phi_h(2)$ to $\phi_h(N)$:

$$\begin{bmatrix} \phi_{3h}^{upper}(1,1) \\ \phi_{3h}^{upper}(1,2) \\ \vdots \\ \phi_{3h}^{upper}(1,\frac{N}{2}-1) \end{bmatrix} = \begin{bmatrix} -1 & 0 & \cdots & 0 & 0 \\ 1 & -1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} \hat{\hat{\phi}}_h(2) \\ \hat{\hat{\phi}}_h(3) \\ \vdots \\ \hat{\hat{\phi}}_h(\frac{N}{2}) \end{bmatrix}$$

$$\bar{\phi}_{3h}^{upper} = \bar{\bar{G}}_\phi \cdot \hat{\hat{\bar{\phi}}}_h$$

$$\hat{\hat{\bar{\phi}}}_h = \bar{\bar{G}}_\phi^{-1} \cdot \bar{\phi}_{3h}^{upper} \tag{4.106}$$

The error $E_{\phi_h(1)}$ introduces in equation 4.106 by assuming that $\phi_h(1) = 0$ would accumulate over frequency as defined in equations 4.107 to 4.111.

$$\begin{aligned}
\hat{\bar{\phi}}_h(1) = &\quad 0 + E_{\phi_h(1)} &&= E_{\phi_h(1)} && (4.107)\\
\hat{\bar{\phi}}_h(2) = &\quad \hat{\bar{\phi}}_h(1) + \hat{\bar{\phi}}_h(1) - \phi_{3h}^{upper}(1,1) &&= \hat{\bar{\phi}}_h(2) + 2 \cdot E_{\phi_h(1)} && (4.108)\\
\hat{\bar{\phi}}_h(3) = &\quad \hat{\bar{\phi}}_h(1) + \hat{\bar{\phi}}_h(2) - \phi_{3h}^{upper}(1,2) &&= \hat{\bar{\phi}}_h(3) + 3 \cdot E_{\phi_h(1)} && (4.109)\\
\hat{\bar{\phi}}_h(4) = &\quad \hat{\bar{\phi}}_h(1) + \hat{\bar{\phi}}_h(3) - \phi_{3h}^{upper}(1,3) &&= \hat{\bar{\phi}}_h(4) + 4 \cdot E_{\phi_h(1)} && (4.110)\\
\vdots &\quad \vdots &&\vdots &&\\
\hat{\bar{\phi}}_h(\tfrac{N}{2}) = &\quad \hat{\bar{\phi}}_h(1) + \hat{\bar{\phi}}_h(\tfrac{N}{2}-1) - \phi_{3h}^{upper}(1,\tfrac{N}{2}-1) &&= \hat{\bar{\phi}}_h(\tfrac{N}{2}) + \tfrac{N}{2} \cdot E_{\phi_h(1)} && (4.111)
\end{aligned}$$

If it is assumed that the last entry in equation 4.107, consists mainly of the error $E_{\phi_h(1)}$. Dividing this entry with $\frac{1}{N/2}$ would be a better estimate for the error then setting it is zero, which was the first assumption. Correcting the error in each step can be using equation 4.112

$$\begin{bmatrix} \hat{\phi}_h(1) \\ \hat{\phi}_h(2) \\ \vdots \\ \hat{\phi}_h(\frac{N}{2}-1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cdots & 0 & -\frac{1}{N/2} \\ 1 & 0 & \cdots & 0 & -\frac{2}{N/2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -\frac{N/2-1}{N/2} \end{bmatrix} \cdot \begin{bmatrix} \hat{\bar{\phi}}_h(2) \\ \hat{\bar{\phi}}_h(3) \\ \vdots \\ \hat{\bar{\phi}}_h(\frac{N}{2}) \end{bmatrix} \tag{4.112}$$

$$\hat{\bar{\phi}}_h = \bar{\bar{F}}_\phi \cdot \hat{\bar{\phi}}_h \tag{4.113}$$

Combining equations 4.106 and 4.112 the phase response of the filter can be written as:

$$\begin{aligned}
\hat{\bar{\phi}}_h &= \bar{\bar{F}}_\phi \cdot \hat{\bar{\phi}}_h \\
&= \bar{\bar{F}}_\phi \cdot \bar{\bar{G}}_\phi^{-1} \cdot \bar{\phi}_{3h}^{upper} \\
&= \left[\bar{\bar{F}}_\phi \cdot \bar{\bar{G}}_\phi^{-1} \ \bar{\bar{0}}\right] \cdot \bar{\phi}_{3h} \\
&= \bar{\bar{D}}_\phi \cdot \bar{\phi}_{3h}
\end{aligned} \tag{4.114}$$

From this estimate it is now possible to estimate the phase of bispectrum in the interval $[-3\pi; 3\pi[$ as opposed to $[-\pi; \pi[$ in equation 4.100. Before solving the least squares solution it is necessary to estimate $\bar{k}$ which can be done by rewriting equation 4.102 and inserting the estimate of $\bar{\phi}_h$:

$$\hat{\bar{k}} = \text{round}\left(\frac{\bar{\bar{A}}_\phi \cdot \hat{\bar{\phi}}_h - \bar{\phi}_{3h}}{2 \cdot \pi}\right) \tag{4.115}$$

Rounding the right hand side ensures that the elements of $\hat{\bar{k}}$ are integers. It is now possible to determine the phase of the filter with the phase ambiguities resolved using least squares:

$$\bar{\phi}_h = \left(\bar{\bar{A}}_\phi^T \bar{\bar{A}}_\phi\right)^{-1} \bar{\bar{A}}_\phi^T \cdot \left(\bar{\phi}_{3h} + 2\pi \cdot \hat{\bar{k}}\right) \tag{4.116}$$

This concludes the estimation of the phase response, the next step is to estimate the magnitude response.

4.2.2 Estimation of the Magnitude Response

The procedure for the magnitude response is more or less the same as for the phase response, but in order to replace the multiplications in equation 4.97 with additions, the logarithm and the absolute value is taken on both sides of the equation:

$$\begin{aligned} \ln|C_{3h}(\omega_1,\omega_2)| &= \ln|H(\omega_1)|\cdot\ln|H(\omega_2)|\cdot\ln\left|H^*(\omega_1+\omega_2)\right| \\ \mu_{3h}(\omega_1,\omega_2) &= \mu_h(\omega_1)+\mu_h(\omega_2)+\mu_h(\omega_1+\omega_2) \end{aligned} \tag{4.117}$$

Which can be written in matrix form as in equation 4.100:

$$\begin{bmatrix} \mu_{3h}(0,0) \\ \mu_{3h}(0,1) \\ \mu_{3h}(0,2) \\ \vdots \\ \mu_{3h}(0,N-1) \\ \mu_{3h}(1,1) \\ \vdots \\ \mu_{3h}(\frac{N}{2}-1,\frac{N}{2}-1) \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 & \cdots & 0 & \cdots & 0 \\ 1 & 2 & 0 & \cdots & 0 & \cdots & 0 \\ 1 & 0 & 2 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 0 & \cdots & 0 & \cdots & 2 \\ 0 & 2 & 1 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \\ 0 & 0 & 0 & \cdots & 2 & \cdots & 0 \end{bmatrix} \cdot \begin{bmatrix} \mu_h(0) \\ \mu_h(1) \\ \mu_h(2) \\ \vdots \\ \mu_h(\frac{N}{2}-1) \\ \vdots \\ \mu_h(N-1) \end{bmatrix}$$

$$\bar{\mu}_{3h} = \bar{\bar{A}}_\mu \cdot \bar{\mu}_h \tag{4.118}$$

In the least square sense this can be solved as:

$$\bar{\mu}_h = \left(\bar{\bar{A}}_\mu^T \bar{\bar{A}}_\mu\right)^{-1} \bar{\bar{A}}_\mu^T \cdot \bar{\mu}_{3h} \tag{4.119}$$

There is on problem with this approach, if the magnitude of one or more of the elements in $\bar{\mu}_{3h}$ are zero. Taking the logarithm of these entries would result in μ_h being an infinitely large negative number. This would dominate the results making the calculation in equation 4.119 problematic.

The solution is to remove the zero elements in $\bar{\mu}_{3h}$ and corresponding rows in $\bar{\bar{A}}_\mu$. By doing so some columns in $\bar{\bar{A}}_\mu$ may contain only zeros. If this is the case these columns should be removed, as they no longer have any impact on the calculations. Correspondingly the elements in $\bar{\mu}_h$ should be also be removed, but a record should be kept of which columns where removed, as these removed columns would corresponds to an entry in in the $\bar{\mu}_h$ that should have been minus infinite. These entries must be reinserted into $\bar{\mu}_h$ when the other magnitude values in $\bar{\mu}_h$ have been determined. The last step is to reconstruct $\hat{\bar{H}}$ by combining the results from equations 4.119 and 4.116 in the following equation:

$$\hat{\bar{H}} = \exp(\bar{\mu}_h)\cdot\exp(\mathrm{j}\cdot\bar{\phi}_h) \tag{4.120}$$

Now that the it is possible to estimate the filters the system model can be reverted in order to restore the original source signals.

Chapter 5

Conclusion

This concludes the theory behind the method presented by [8] for doing the BSS on the TITO model.

The following steps in the BSS process were identified for the method:

- Estimate the DC-gain of the filters.
- Estimate the bispectrum of the filters using HOS and the DC-gain.
- Estimate the filters from the bispectrum.
- Invert the TITO model using the estimated filters.

The theory behind the steps have been presented in the previous chapters. Only the step to estimate the DC-gain of the filters, also called the $\bar{\bar{H}}(0)$ estimation, proved to be troublesome. The problem stems from the two eigenvalue decompositions that give rise to four different estimates of the DC-gain. This can, however, be solved if some prior knowledge about the filters exist, in particular if the DC-gain is greater or smaller than one and it is known which of the filters have the largest DC-gain. This problem does not pose a problem for running simulations as the filters used for the TITO mixing are known in advance.

As the steps needed for doing the BSS have been presented, they are now simulated in order to verify that the described theory works. This is done in next part, Simulation and Verification of the Blind Source Separation.

Part III

Simulation and Verification of the Blind Source Separation

Chapter 6

Introduction

This part of the report contains a implementation and simulations of the BSS method for solving the TITO model described in the previous chapter. The method for doing the BSS was divided into four parts, the inverse filtering described in section 3, the minimum phase filter estimation and the non-minimum phase filter estimation both described in section 4. Reverse bispectrum algorithm in section 4.2 and the $\bar{\bar{H}}(0)$ estimation in section 4.1.

Figure 6.1 illustrates the different parts of the BSS method for the non-minimum filter estimation. If the same model is used for the minimum phase filter estimation the reverse bispectrum method must be replaced by an inverse Fourier transform and the non-minimum phase filter estimation is replaced by the minimum phase filter estimation.

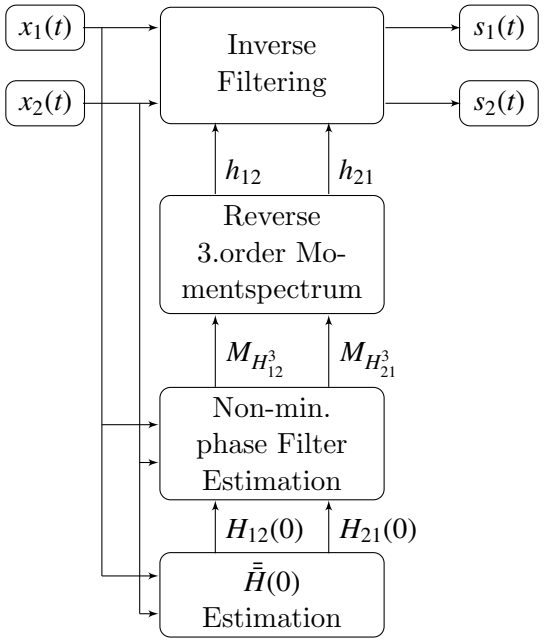


Figure 6.1: Illustration of BSS method presented in the theory section for solving the blind source separation

The model with minimum phase filter can only be accepted if both filters are known to be

minimum phase, as this cannot be known in advance the non-minimum phase filter model is the correct one to use. However both the non-minimum phase filter estimation part and the minimum phase filter estimation part are simulated.

Each individual part in the model are simulated separately in their respectively sections and the last section is a simulations of the entire model. Some simulations depends on other parts of the BSS model to be working or at least ideal values to be known, also some of the parts can further be exploded into subparts like the estimation of the trispectra, which is utilized in both the $\bar{\bar{H}}(0)$ estimation and the non-minimum filter estimation.

There are some prerequisites concerning the simulation of the above parts.

First the test signals $x_1(t)$ and $x_2(t)$ for the model needs to be created, as they are used to simulate several parts in the BSS model. Except the mixing model.

If ideal estimates of the filters are used for the simulation, the inverse filtering does not require any of the other parts to be working in order to simulate it. So this can be made as the second section.

In order to simulate the reverse third order moment spectrum, a third order moment estimator needs to be constructed in order to create the test vectors for this simulation. The third order moment estimator also needs to be tested to ensure proper functionality. This can be done by first implementing the minimum phase filter estimator, which uses the bispectrum (which is the third order moment spectrum without the mean). So by simulating this in the third section, functionality of the third order moment estimator can be ensured, and the reverse third order moment spectrum can simulated in the fourth section. One problem with this approach is that the minimum phase filter estimator needs an estimation of $\bar{\bar{H}}(0)$ in order to work proper, but as the simulations uses known filters, $\bar{\bar{H}}(0)$ is estimated from them instead of using the $\bar{\bar{H}}(0)$ estimation.

The fourth section is the Non-minimum phase filter estimation all the prerequisites except the $\bar{\bar{H}}(0)$ estimation and the trispectrum estimation have been meet. For the simulation an ideal estimation of $\bar{\bar{H}}(0)$ can by used again as the filters are known, but the trispectrum estimation needs to implemented for the simulation.

The fifth and sixth section contain the simulation of $\bar{\bar{H}}(0)$ estimator and the simulation of the BSS model.

Chapter 7

Test Signals

This chapter contains the design and implementation of the test signals that are to be used for the simulation. The TITO system model that was introduced earlier can be seen in figure 7.1 Equations 7.1 and 7.2 for the mixing the two signals $x_1(t)$ and $x_2(t)$, from the source signals $s_1(t)$

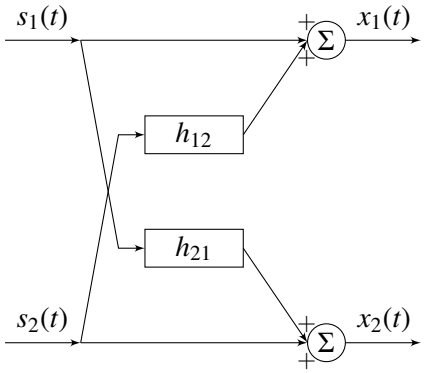


Figure 7.1: Two-channel version of the signal model presented in [8].

and $s_2(t)$ are the mathematically equivalent to figure 7.1.

$$x_1(t) = s_1(t) + h_{12}(t) * s_2(t) \tag{7.1}$$
$$x_2(t) = s_2(t) + h_{12}(t) * s_1(t) \tag{7.2}$$

In order to create the test signals $x_1(t)$ and $x_2(t)$, the only thing missing are the source signals, as the filters would normally be set for the specific simulation. For most of the simulations the source signals s_1 and s_2 are supposed to be random variables. When creating these, there are some considerations that needs to be addressed concerning the stochastic properties. More precisely the shape of PDF needs to differ from a normal distribution in certain ways.

If the signal $x(t)$ is the convolution between a filter and a random process $s(t)$ (see equation 7.3) and the random process is white to the fourth order e.g. the spectrum is flat. Then the

bispectrum and trispectrum of x is described by equation 7.4 and 7.5.

$$x(t) = h(t) * s(t) \tag{7.3}$$
$$C_{x^3}(\omega_1, \omega_2) = \gamma_s^3 \cdot H(\omega_1) \cdot H(\omega_2) \cdot H^*(\omega_1 + \omega_2) \tag{7.4}$$
$$C_{x^4}(\omega_1, \omega_2, \omega_3) = \gamma_s^4 \cdot H(\omega_1) \cdot H(\omega_2) \cdot H(\omega_3) \cdot H^*(\omega_1 + \omega_2 + \omega_3) \tag{7.5}$$

Where:
γ_S^3 is the skewness of the random variable s
γ_S^4 is the kurtosis of the random variable s

The reasoning behind these equations are described in appendix A.7.1). But basically it says the bispectrum and trispectrum becomes a function of the moment spectrum of the filter multiplied with a constant that depends on the kurtosis or the skewness of the white random process s. Therefore it is important that the random process has a skewness or kurtosis that is differs from zero, as this would result in the entire spectrum being zero.

The skewness and the kurtosis is a measurement of how a much a distribution differs from a normal Gaussian distribution. This means that as a PDF for the spectrum a normal distribution cannot be used as it is only white to the second order. For a PDF the random variables needs a skewness different from zero, e.g. the PDF is not allowed to be symmetric around the mean. It should also have a kurtosis (peak) that is higher (super Gaussian) or lower (sub Gaussian) the a normal distribution. One PDF that would satisfies these two conditions is an exponential distribution. Further more Matlab own random generator supports this distribution, making it an obvious choice as a PDF for the random processes s_1 and s_2.

As the source signals are not always white, the source signals are now colored by filtering it through an AR(2) process. [8, 523] uses two filters with poles located at -0.2 for s_1 and -0.3 for s_2, the same coloring filters are used for the test signal. The transfer function is for the test signal are shown in equation 7.6 and 7.7

$$s_1(t) = \frac{1}{1 + 0.2 \cdot z^{-1}} \cdot f_1 \tag{7.6}$$
$$s_2(t) = \frac{1}{1 + 0.3 \cdot z^{-1}} \cdot f_2 \tag{7.7}$$

Where:
f_1 is a random variable with a exponential distribution.
f_2 is a random variable with a exponential distribution.

Magnitude plots of the filters can be seen figure 7.2.

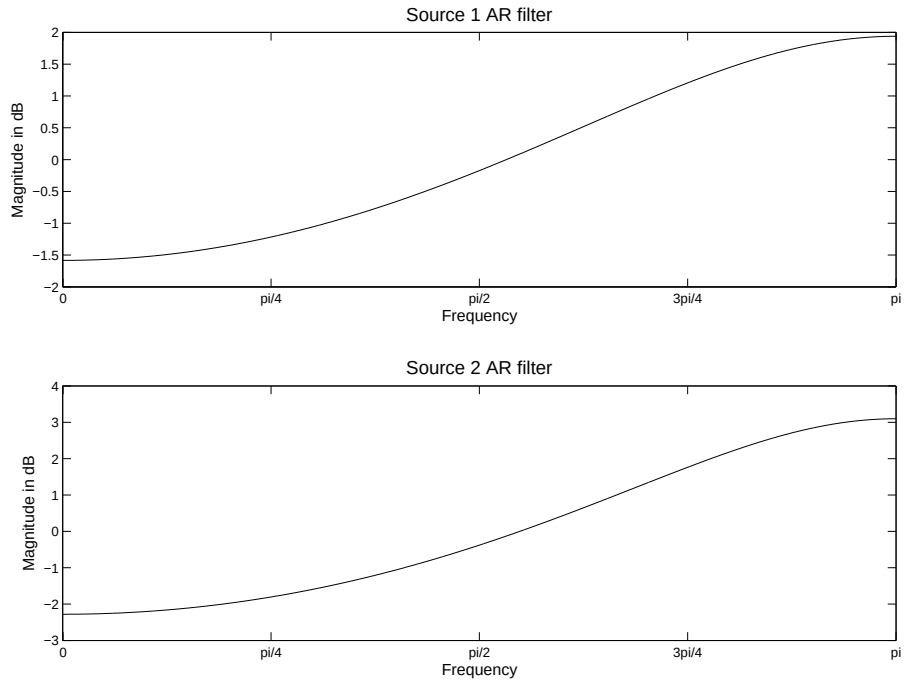


Figure 7.2: Ideal power spectra's of the AR(2) Source Signals

To calculate the convolutions the **filter** function is used in Matlab. For both the coloring and the mixing of the signals.

Now that the test signals $x_1(t)$ and $x_2(t)$ have been created, the next step to make a implementation of the inverse filtering in order to simulate it. Although the mixing model is still used for this simulation, other source signals then the ones devised here are used to better illustrated its functionality. An example file on how the signals are created can be found on the accompanying CD as: `/Matlab Code/Source Signals/SourceSignals.m`.

Chapter 8

Inverse Filtering

This chapter contains a simulation of the inverse filtering of the TITO model, the simulation and implementation are performed in MATLAB. In order to make the inverse filtering the assumption is that the filters h_{12} and h_{21} are known or can be estimated. In section 3 on page 11 the theory behind the inverse filtration is description. Based on this description the following procedure is implemented and simulated in MATLAB to evaluate the inverse filtering.

1. The input signals x_1 and x_2 are divided into frames.
2. The frames are zero padded and time shifted to accommodate non-causal filters.
3. The frames are Fourier transformed via an the **fft** function in MATLAB.
4. The estimated original signal is restored using the equation 8.1 within a **for** loop for each value of ω

$$\begin{bmatrix} \hat{S}_1(\omega) \\ \hat{S}_2(\omega) \end{bmatrix} = \begin{bmatrix} 1 & H12(\omega) \\ H21(\omega) & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} X_1(\omega) \\ X_2(\omega) \end{bmatrix} \tag{8.1}$$

5. The estimated original frames ($\hat{S}_1(\omega)$ and $\hat{S}_2(\omega)$) are inverse Fourier transformed using **ifft** in MATLAB
6. The frames are added together with an overlap between the frames corresponding to amount of zero padding performed in step 2.

The MATLAB implementation of the above step by step instructions can be found on the accompanying CD in `/Matlab Code/Inverse Filtration/invfilt.m`. To ensure functionality the above implementations functionality is simulated in MATLAB using know test signals.

8.1 Simulation of the Inverse Filtering

For the simulations the source signals are changed to square signals instead of using the stochastic signals as this is not needed. The square signals have the ability to spreads out in frequency

domain, and makes it easier to verify by plots that the source signals are restored. The mixed signals are generated using the procedure for the TITO model described in chapter 7, the filters used for this procedure have the following coefficients:

$$h_{12} = [0.3,\ 0.8,\ 0.4] \tag{8.2}$$

$$h_{21} = [1,\ 0.5,\ 0.2] \tag{8.3}$$

The zero pole plot of these filters can be seen in figure 8.1

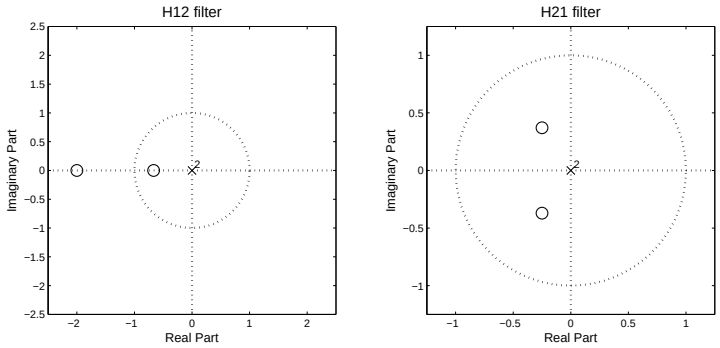


Figure 8.1: Zero pole plot of the filters used for the simulation of the Inverse Filtering Algorithm

From the zero pole plots it can be determined that h_{12} is a non-minimum phase filter, which facilitates the need for a delay in the processing according to chapter 3. This delay is created by using a frame length of 256 data points which is delayed by adding 128 data points to the header of the frame and 128 data point is added to tail of the frame. This results in a frame length for the Fourier transform of 512, and a overlap of the processed frames of 50 % .

The result from the simulation can be seen in figure 8.2, where the original signals, the mixed signals and the demixed signals are displayed.

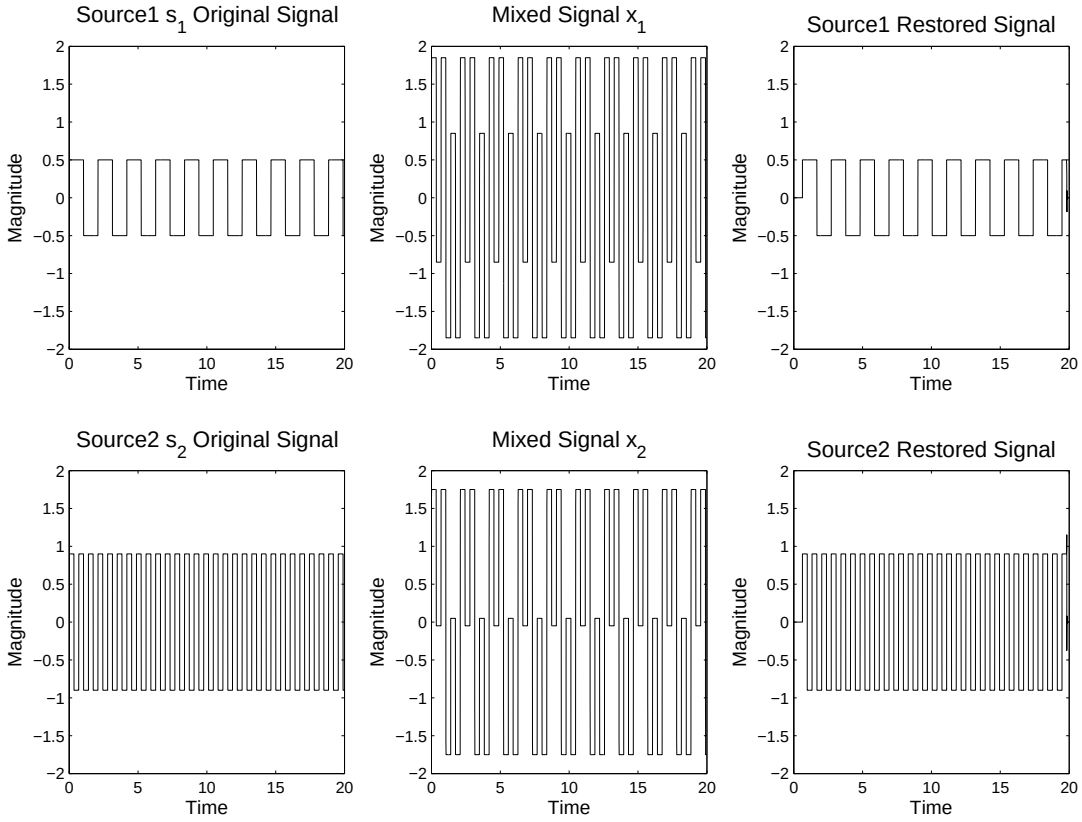


Figure 8.2: Demixing of the sources using a squared signals as a source

The error is not noticeable in the plot besides for the first frame and the last frame, which are missing half of the previous and the next frame so this is expected. Figure 8.3 shows a plot of the original and the demixed signals in the first column and in the right column the power difference between the two signals are plotted, to remove the known error the first half and last half frame is removed.

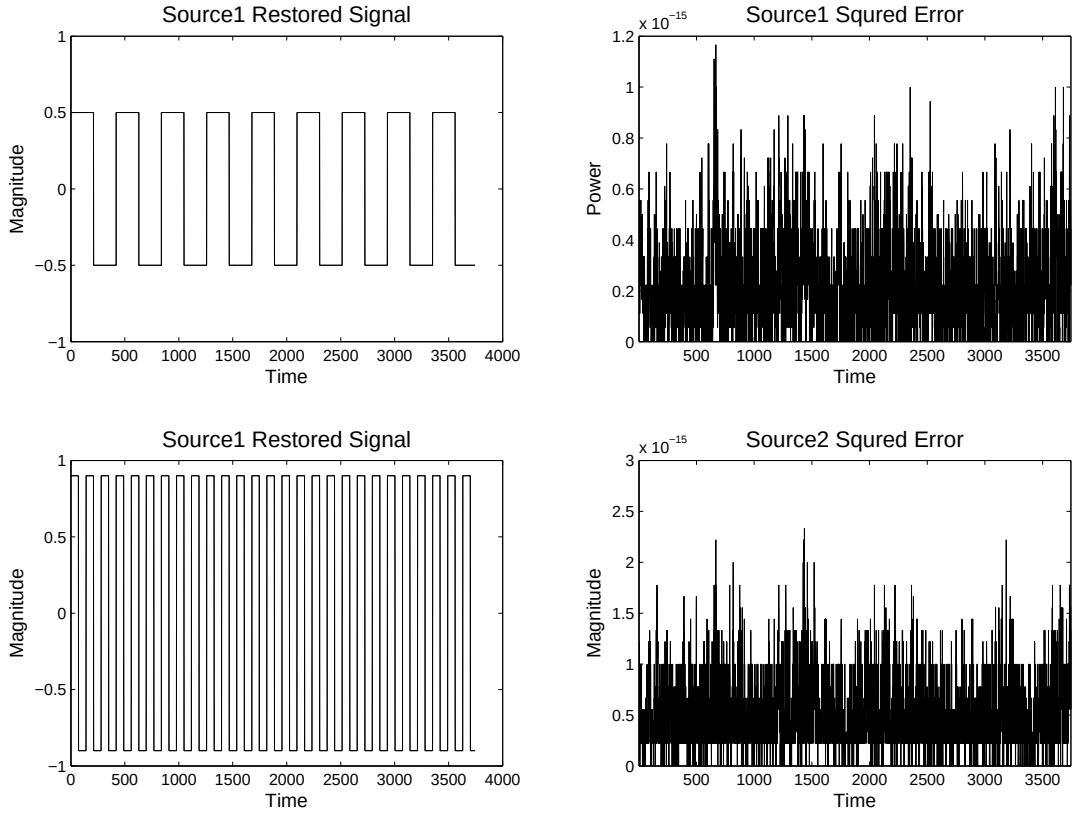


Figure 8.3: The original signal and the restored signal from the simulation of the inverse filtering, and the error between them, with first and last half frame removed

The mean squared error in the frame is mostly below 10^{-15}, which would correspond to a signal to noise ratio (SNR) of around 150 dB if the power of the signal would be around 1. Actually calculating the SNR gives a SNR for both channel above 300 dB. Matlab operates internally with double precision floating points. This means that for each numbers, there are 52 bits to represent a single value. Rounding errors would occur in last bit and from 52 bits it is possible to create around $4.5 \cdot 10^{15}$ values. This means that from the decimal point eventual rounding errors would occur at the 53'th bit. In the decimal system the rounding error would similarly occur at 2^{-53}, which is approximately at 10^{-16} from the decimal point. If the largest number processed is at around 10^0 the signal to numerical noise (SNNR) would be around 320 dB. So the observed noise can be more or less attributed to numerical noise alone. The SNR is so high that it does not introduce any problems into the system. So the conclusion of the simulation is that the functionality of the inverse filtering is acceptable and the noise introduced by the inverse filtering is neglectable . This concludes the simulation of the inverse filtering, the next step is to simulate the minimum phase filter estimation.

Chapter 9

Minimum Phase Filter Estimation

Section 4 describes two methods for estimating the filters h_{12} and h_{21}. One using the bispectrum and one using the trispectrum, as the method using the bispectrum estimates the powerspectrum of the filters. It serves as a good starting point as the results would be easy to compare to the actual powerspectrum of the filters, and the method is appropriate to use if the filters are known to be minimum phase. As the filter estimation only really consist of an bispectrum estimator and the $\bar{\bar{H}}(0)$ estimation as ideal values are used for the $\bar{\bar{H}}(0)$ estimation. The only thing needed is an estimator for a bispectrum. Therefore this needs to implemented and simulated in MATLAB before the minimum phase filter estimation can be simulated.

9.1 Bispectrum Estimation

The bispectrum also know as the third order cumulant spectrum is calculated from the third and lower order moment spectra's. Appendix A.3 introduces two methods for estimating a moment spectra's, referred to as the direct method and the indirect method. These two methods are essentially the same as the direct method and the indirect method when estimating a power-spectrum. The direct method for calculating a powerspectrum is done by taking the Fourier transform of the signal, and then squaring and taking absolute value. In the indirect method for calculating a powerspectrum, the autocorrelation is first calculated from the signal and Fourier transform is taken of the autocorrelation to create the powerspectrum.

For higher order moment spectra the procedure is more or less the same. In the direct method the first step is to calculate the Fourier transform of the inputs signals and the momentspectrum is then calculated in the frequency domain using equation A.61 on page 175. In the indirect method the n-th order moment sequence is first calculated and then a (n-1)th dimensional Fourier transform is performed on the sequence, which create the n-th order moment spectrum. Both methods should under the same conditions give the same results, therefore both are going to be implemented for comparison.

On step that is common for both methods are the moment to cumulant transformation as described in equation A.55 on page 174. But for the bispectrum estimations, this can be done

by removing the mean from the input signal and calculating the moment spectrum from these mean free signals instead. Therefore this becomes the first step for both methods, and the following steps becomes the same as for the moment spectrum estimation.

There is however one problem with both methods. Time frequency uncertainty relations introduces an error in the spectrum when a limited frame length is used. This means that the results becomes increasingly unreliable, when the frequency goes towards zero. The problem can be seen directly in the way the bispectrum is calculated in the direct method. Because the mean is subtracted, before a Fourier transform is performed, the first frequency bin is always zero. This is of course not true as the n-th order white signal the spectrum should always be flat, but in order to archive this, the frame length should go towards infinite, which is not possible to implement.

The solution for the direct method is utilizing a smoothing of the bispectrum and similarly for the indirect method not to use the entire cumulant sequence, when doing the Fourier transform, thereby creating a smoothing effect.

The steps used to implement the two methods in MATLAB are now presented and the two methods are then evaluated by simulations.

9.1.1 Direct Method for Estimating the Bispectrum

The direct method used for estimating the bispectrum is based [1, pp 124-127] and appendix A.3.

1. Remove the mean from the input frame, in order to make the transition from moment spectra to cumulant bispectra.

2. Calculate the discreet Fourier transform (DFT) off the input frame(s).

$$X(\omega) = \sum_{t=0}^{N-1} x(t) \cdot \exp\left(-\frac{2\pi j}{N}(t\omega)\right) \tag{9.1}$$

   If cross spectra's is calculated, then a DFT is performed for each individual input signal. MATLAB's own fast Fourier transform (FFT) function is used to performed the DFT.

3. The bispectrum is calculated from equation 9.2. This is made as a general function, which assumes that all the input signals differs from each other, effectively giving a cross bispectrum between three different signals.

$$C_{xzy}(\omega_1, \omega_2) = \frac{1}{N} Y(\omega_1) \cdot Z(\omega_2) \cdot X^*(\omega_1 + \omega_2) \tag{9.2}$$

   Where:
   $0 \leq \omega_1 \leq \pi,\ 0 \leq \omega_2 \leq \pi,\ \omega_1 + \omega_2 \leq \pi$
   $X(\omega) = Y(\omega) = Z(\omega) = 0|_{\omega > \pi}$ N is the number of samples in $x(t)$.

4. The last step is the smoothing of the bispectrum, instead of using equation 9.2 in the previous step, a smoothed version of the bispectrum using equation 9.3 could be used instead.

$$\hat{C_{xyz}}(\omega_1, \omega_2) = \frac{1}{(2L+1)^2} \sum_{k_1=-L}^{L} \sum_{k_1=-L}^{L} \frac{1}{N} Y(\omega_1 + k_1) \cdot Z(\omega_2 + k_2) \cdot X^*(\omega_1 + \omega_2 + k_1 + k_2) \quad (9.3)$$

Where:
L is the number of samples to smooth over.

The implementation for direct method can be found as a MATLAB file on the accompanying CD in `/Matlab Code/Bispectrum Estimation/bispec2.m`.

9.1.2 Indirect method for estimating the bispectrum

This method for estimating bispectrum via the indirect method is based on [1, pp 124-127] and appendix A.3.

1. Remove the mean from the signal, on which the bispectrum is calculated.
2. Calculate the third order cumulant sequence using equation 9.4

$$c_{xyz}(\tau_1, \tau_2) = \frac{1}{N} \sum_{t=s_1}^{s_2} x(t) \cdot y(t + \tau_1) \cdot z(t + \tau_2) \quad (9.4)$$

Where:
$\tau_1, \tau_2 = 0, \pm 1, \pm 2, \ldots, \pm$ maximum lag
$s_1 = max(0, -\tau_1, -\tau_2)$
$s_2 = min(N - 1,\ N - 1 - \tau_1,\ N - 1 - \tau_2\ )$

The maximum lag depends on the assumed system order of the filters, if no priory knowledge exist this would be the length of the input signals, but then the frequency time uncertainty relations should be taken into account. In this implementation it would typically be the maximum length of the filters h_{12} and h_{21} that is used.

3. Apply a 2 dimensional window function to the cumulants sequence.
This is partly done in the previous step as the restricting τ has the same effect as not restricting it and applying a rectangular window. From appendix A.6 it is noted that window functions used differ even for rectangular windows (no window) when n-dimensional FFT's (with n higher then two) are used. The window function is implemented as described in appendix A.6.

4. Zero pad to obtain a appropriate Fourier length.
Certain Fourier lengths makes it possible to make more efficient DFT implementations. The zero padding makes it possible to archive these efficient DFT's. Zero padding is normally done by adding zeros to the end of a sequence. This is however not the correct method when a cumulant sequence is zero padded. The padding should be done so that

the point with $\tau = 0$ always stays in the center of the matrix. If the array has an even number of entires (no exact center) the point with $\tau = 0$ should be offset to $\tau > 0$ side of the array.

5. Apply a 2 dimensional Fourier transform on the cumulant sequence, using equation 9.5.

$$C_{xyz}(\omega_1, \omega_2) = \sum_{\tau_1} \sum_{\tau_2} c_{xyz}(\tau_1, \tau_2) \cdot \exp\left(j(\tau_1\omega_1 + \tau_2\omega_2)\right) \tag{9.5}$$

Instead of implementing equation 9.5 MATLAB's own **fftn** function for an n - dimensional DFT is used. It is more efficient then doing the above calculation, though it should be noted that **fftn** assumes a different arrangement of τ_1 and τ_2, this is corrected by doing **ifftshift** on the cumulant sequence, which rearranges the array before the n - dimensional DFT is applied.

The indirect method can be found as a Matlab m-file on the accompanying CD in `/Matlab Code/Bispectrum Estimation/bispec.m`, also a Matlab function that constructs up to three dimensional window functions is located in `/Matlab Code/Bispectrum Estimation/window.m`.

Now that the two methods for estimation a bispectrum have been presented, their functionality needs to be verified by a MATLAB simulation using know test signals. The first thing to verify is equality between the direct and indirect method. If the two methods gives similar results, under the same conditions it does not matter from a functional point, which of the two methods are used. Therefore it becomes a matter of execution speed when selecting which method to use.

In chapter 16.2 on page 90 the complexity of the above implementations are listed. As the filter length is known and is smaller then the length of the signals, then from an execution time perspective, it would makes most sense to use the indirect method for the simulations, however both methods are intially simulated to see how they performe.

9.1.3 Simulations of the bispectrum estimators

The first thing to establish is equality between the two methods under the same conditions, meaning no smoothing. This is done be providing the same test signal to both methods and comparing the output. If a test vector of length n is used, the maximum number of lags for the cumulant sequence would contain $(2n-1)$ times $(2n-1)$ values. The resulting bispectrum would contain the same, as both methods are used without smoothing the smoothing parameter in the direct method is set to 0, so no smoothing is performed.

As the three input vectors a random signal with a exponential distribution is used with a length of 40 samples. No window function is used so not to interfere with the results and the length of the Fourier transform should be twice the size of the signal length in each dimension, resulting in a matrix of size 80x80. The two resulting bispectra's are compared by calculating the mean square error (MSE) between the two bispectra's. This term is used loosely as it is not an actual error but a difference between the two methods. To calculate the actual MSE one would need to know the correct bispectrum, without using either of the two preseneted methods. Equation

9.6 is used to calculate the MSE between the two methods.

$$MSE = \sum_{\omega_1} \sum_{\omega_2} \left| \left(C_{x^4}^{DM}(\omega_1, \omega_2) - C_{x^4}^{IDM}(\omega_1, \omega_2) \right)^2 \right| \tag{9.6}$$

where:
DM is the direct method.
IDM is the indirect method.

To get a better estimate of the difference between the two methods the above simulation is repeated 100 times and the averaged MSE of these simulations is calculated. The test file can be found on the accompanying CD as `/Matlab Code/Bispectrum Estimation/test2.m`

The resulting MSE from the simulation is approximately $3.6 \cdot 10^{-33}$ giving a SNR of around 330 dB. Previously the SNNR was established to be around 320 dB Considering the low value of MSE and that it is below eventual rounding errors, it can be concluded that the two methods give the same results under the same conditions.

Previously it was noted that of the two methods the indirect method is the fastest, if the filter length is small compared to the dataset size. Normally large datasets from a stochastic source is desired as averaging over the results gives better estimates. To test the functionality of the indirect method a stochastic source Y that is white to the third order is put trough a filter H, which should give a bispectrum described by equation 9.7.

$$C_{x^3}(\omega_1, \omega_2) = \gamma_Y^3 \cdot H(\omega_1) \cdot H(\omega_2) \cdot H^*(\omega_1 + \omega_2) \tag{9.7}$$

Where:
γ_Y^3 is the skewness of the stochastic source Y

The stochastic source Y is generated as described in chapter 7, but the signal is not colored, so it remains n-th order white. The bispectrum of x is then evaluated at $C_{x^3}(\omega_1, 0)$, which should be the same as the powerspectrum of the filter multiplied with the DC- amplification ($H(0)$) of the filter and the skewness of the random variable Y. The filter H used for the test has the following coefficients:

$$h(t) = [1, .5, .2] \tag{9.8}$$

For the simulation the length of the stochastic source Y is 4000 samples and τ_1 ,τ_2 is varied between ±9 in order to create the smoothing. In order to make it easier to plot the resulting, the cumulant sequence is zero padded to archive a Fourier length of 24. Also the resulting estimation of the power spectrum of the filters is averaged over 100 Monte Carlo simulations to get a better result. The results from this simulation is illustrated in figure 9.1, where both the estimated and the ideal power spectra's are scaled so that the largest value equals one.

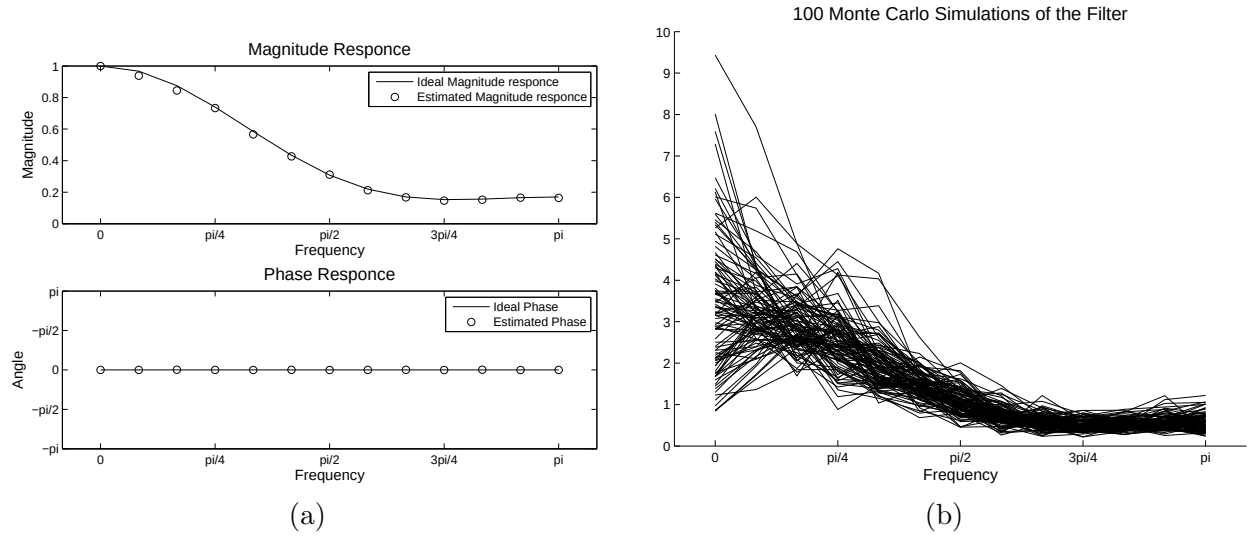


Figure 9.1: (a)Standard deviation and mean estimation for a low pass filters magnitude and phase response using 100 Monte Carlo simulations. (b) plot of the 100 estimations

The mean estimation of the filter over 100 simulations is rather close to the original. On plot (b) it can be seen that the variance decreases towards π, which is expected because of the before mentioned time frequency uncertainty. The phase show one thing of interest the system has a linear phase, as the powerspectrum should not contain any phase information, this is also what was expected. The test file for the simulation can be found on the accompanying CD as `/Matlab Code/Bispectrum Estimation/test1.m`

The same test is now performed using the direct method. The frame length is 128 samples and smoothing is performed over ± 10 samples and the averaging is also performed over 100 simulations. The test file can be found on the accompanying CD as `/Matlab Code/Bispectrum Estimation/test5.m`. The simulation results can be seen in figure 9.2.

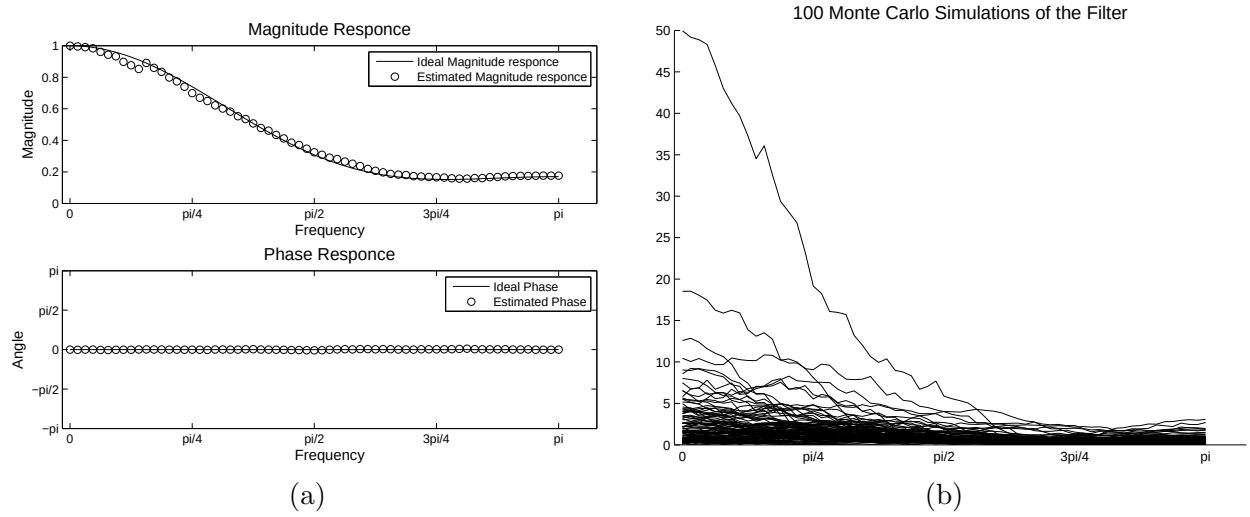


Figure 9.2: (a) Mean estimation of a low pass filters magnitude and phase response. The direct method is used for the estimation and it is averaged over 100 Monte Carlo simulations. (b) Plot of the 100 estimations

From figure 9.2 (b) it is clear that the variance is rather high compared to the indirect method. This actually means that some averaging over multiple frames is necessary in order to get a reasonable result. But it should be noted that the dataset used for the estimation in the direct method is around 30 times smaller, then the dataset used for the direct method. The phase estimation is very close to a linear phase, which again is expected. But the magnitude plot has discontinuity at around pi/6, which is caused by the smoothing function. Because of the time frequency uncertainty the magnitude would drop rather steep to zero when the frequency goes towards zero. If a smoothing is performed, then this sudden drop would be seen somewhere else on curve, which is what is seen in figure 9.2(a). Besides this jump the magnitude plot is rather close to the original and would pass as being acceptable.

The smoothing that is performed in the direct method is actually a 2 dimensional convolution with rectangular box. Instead of a rectangular box another 2 dimensional window could be used, this would give smoother transitions and eliminate the sudden drop in the magnitude, that was seen before. In figure 9.3 the simulations is performed again. But instead of doing the smoothing the bispectrum is convolved with a two dimensional Parzen window and a two dimensional optimum window. Constructing these windows are done the same ways as the windows for the indirect method are construceted, which is described in appendix A.6. The test file can be found on the accompanying CD as `/Matlab Code/Bispectrum Estimation/test4.m`.

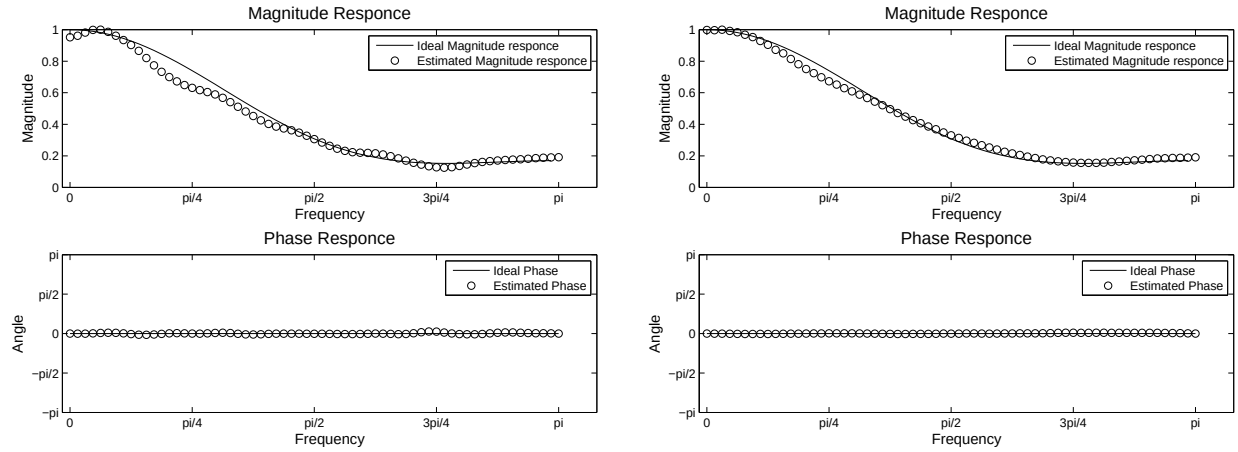
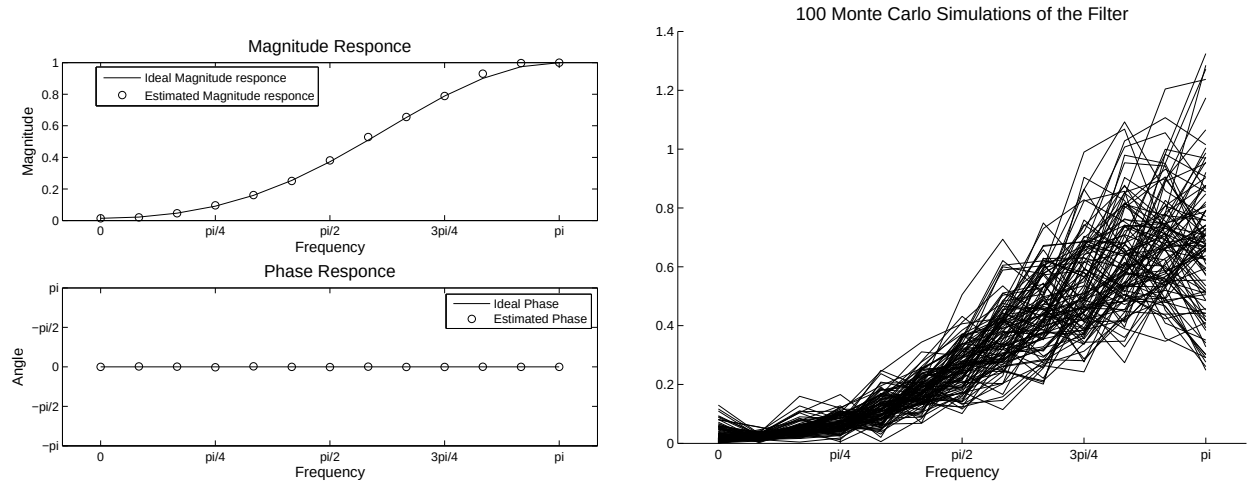


Figure 9.3: (a) Mean estimation of a low pass filters magnitude when the bispectrum is smoothed with a Parzen window (a) and a optimum window(b)

Using a Parzen window in figure 9.3 (a) removes the little jump in magnitude that was seen in figure 9.2 but the overall magnitude estimation became worse. If the optimum window is used the the overall estimation is better then with the Parzen window but not as good as with the original smoothing function. However the small jump that was seen in the original method is gone, instead it transformed to a gentle bump instead.

As both methods have until now only been tested with a low pass filter, which is particular troublesome, because of the time frequency uncertainty. The same simulations as before are now repeated with a high pass filter with filter coefficients $h(t) = [1, -.9, .14]$. The resulting power spectra for the indirect method can be seen in figure 9.4.

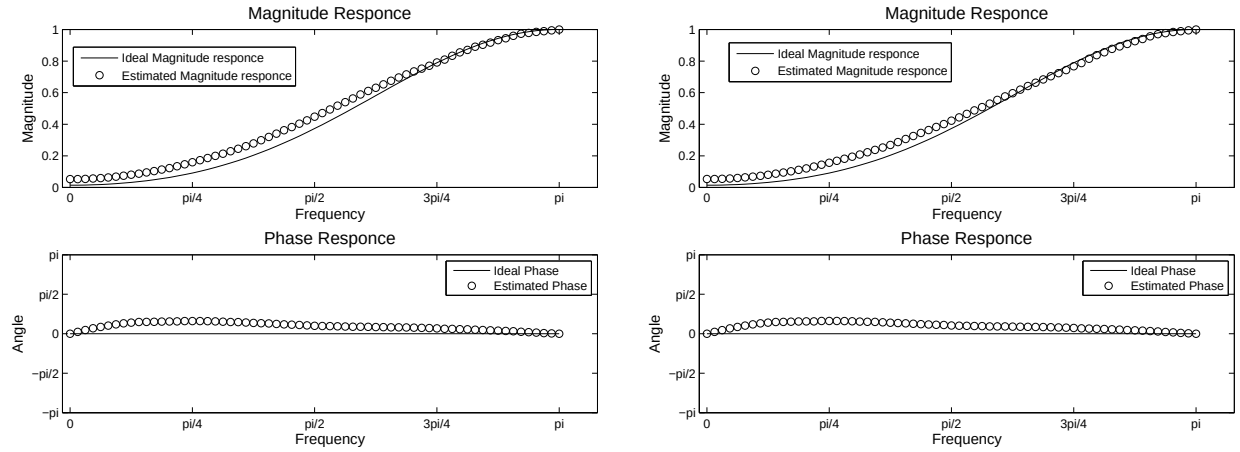


(a) (b)

Figure 9.4: Indirect method for estimating the powerspectrum of a high pass filter via the bispectrum. Figure (a) displays the standard deviation and mean estimation of the magnitude and the phase response using 100 Monte Carlo simulations. Figure (b) shows the results of each of 100 simulations for estimating the magnitude.

In figure 9.4 (a) The resulting magnitude spectrum estimating the filter is still acceptable and the phase is still linear. On plot of the 100 simulations (b) it can be seen that the deviation around π is smaller then it was for 0 in the low pass filter. This is expected as the frequency time uncertainty makes it easier to estimate high frequencies then low frequencies, when a limited frame length is used.

The results for performing the same test with the direct method can be seen in figure 9.5, where a optimum 2 dimensional window of size ±10(a) and ±15(b) is used for the smoothing and 128 samples is used for the frame length.

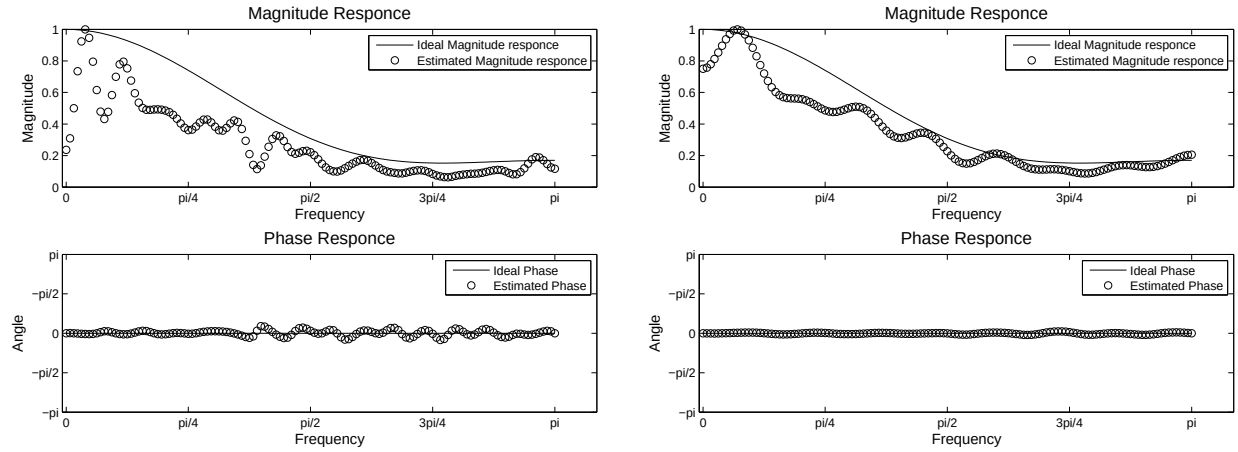


(a) (b)

Figure 9.5: Mean estimation of a high pass filters magnitude when the bispectrum is smoothed with an optimum window of size ±10(a) and ±15(b)

With a smoothing of ±10 the results are acceptable, but there is a bump around $\frac{6}{8}$ pi. If the

smoothing is increased to ±15 samples this bump disappears, at the cost of less precise, when estimating the lower frequencies. Until now the indirect method has proves superior to the direct method, but what happens if the indirect method is moved closer to the direct method by increased τ to ± 32 and ± 16 and decreasing the frame size to 256 samples. The plots of this can be seen in figure 9.6, where the low pass filter used is the one from equation 9.8 and the results are averaged over 100 simulations.



(a) (b)

Figure 9.6: Mean estimation of a low pass filters magnitude using the indirect method with lags at ±32(a) and ±16(b)

From these plots it becomes clear that the indirect method suffers, when the lags moves closer to the frame length. The Matlab files for performing the above simulations for the indirect method, can be found on the CD as `/Matlab Code/Bispectrum Estimation/test1.m`.

In this section several simulations for estimating a powerspectrum of a know filter from the bispectrum was performed. Two methods where presented for estimating a bispectrum, the direct method and the indirect method. Both methods where capable of giving good estimates of the powerspectrum, however one ting became clear. The direct method gives the best results with a relative long filter compared to the frame length and the indirect method gives the best results with a short filter length compared to the frame length. Also regarding execution time the same patter emerges short filter compared to the frames the indirect method is the most efficient and for a long filter compared to the frames the direct method is the most efficient.

Both methods give the same result, if they are performed under the same conditions (no smoothing), only confirming with the theory in appendix A.3 proved. As the succeeding simulations are performed on relative short filters, the prudent choice is to use the indirect method for estimation of the filters, as it does give the best results. However this choice has to be reevaluated in the implementation part, when other parameters becomes an issue.

9.2 Simulation of Minimum Phase System Identification

Now that a working bispectrum estimator has been implemented in the previous sections, it is possible, from equation 4.31 and 4.32 on page 22, to estimated the h_{12} and h_{21} filters in the TITO system model, if these filters are minimum phase filters. The equations are listed again in equation 9.9 and 9.10.

$$C_{h_{12}^2}(\omega_1) = \frac{C_{x_1^2 x_1}(\omega_1, 0) - H_{21}(0) \cdot C_{x_1^3}(\omega_1, 0)}{C_{x_2^3}(\omega_1, 0) - H_{21}(0) \cdot C_{x_2^2 x_2}(\omega_1, 0)} \tag{9.9}$$

$$C_{h_{21}^2}(\omega_1) = \frac{C_{x_2^2 x_1}(\omega_1, 0) - H_{12}(0) \cdot C_{x_2^3}(\omega_1, 0)}{C_{x_1^3}(\omega_1, 0) - H_{12}(0) \cdot C_{x_1^2 x_2}(\omega_1, 0)} \tag{9.10}$$

The above functions are implemented in a **for** loop for all values of ω. The $H_{12}(0)$ and $H_{21}(0)$ values are calculted from the known filters, so ideal values are used here. The simulation is performed using the test signal that was constructed in section 7 and the source signals are now coloured also described in this section. The filters used for the simulation are listen in equation 9.11 and 9.12

$$h_{12} = [1, -.9, .14] \tag{9.11}$$

$$h_{21} = [1, .5, .2] \tag{9.12}$$

A zero pole plot of these filters can be seen in figure 9.7, and they are both minimum phase.

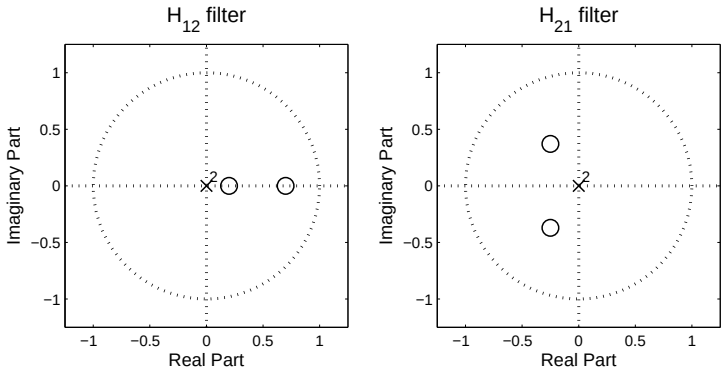


Figure 9.7: Zero Pole plots of the filters

To estimate the bispectrum the indirect method described earlier is used with a τ_1 and a τ_2 that varies between ±9 and a frame size of 4000 samples. The cumulant sequence is zero padded up to a length of 24 samples, resulting in a powerspectrum estimatio of the filters that is 24 values long. For the first simulation no window function, other then the one created from the restricted lags, is used.

The estimated filters and the true powerspectrum as well as the phase estimation from the first simulation can be seen in figure 9.8.

The estimation of the powerspectrum in the above plot (a) are not as good as the ones performed in the previous section. It should however be noted that the source signals in those plots where white, and in these plot they are coloured, which will affect the estimations. In figure 9.8 (c)

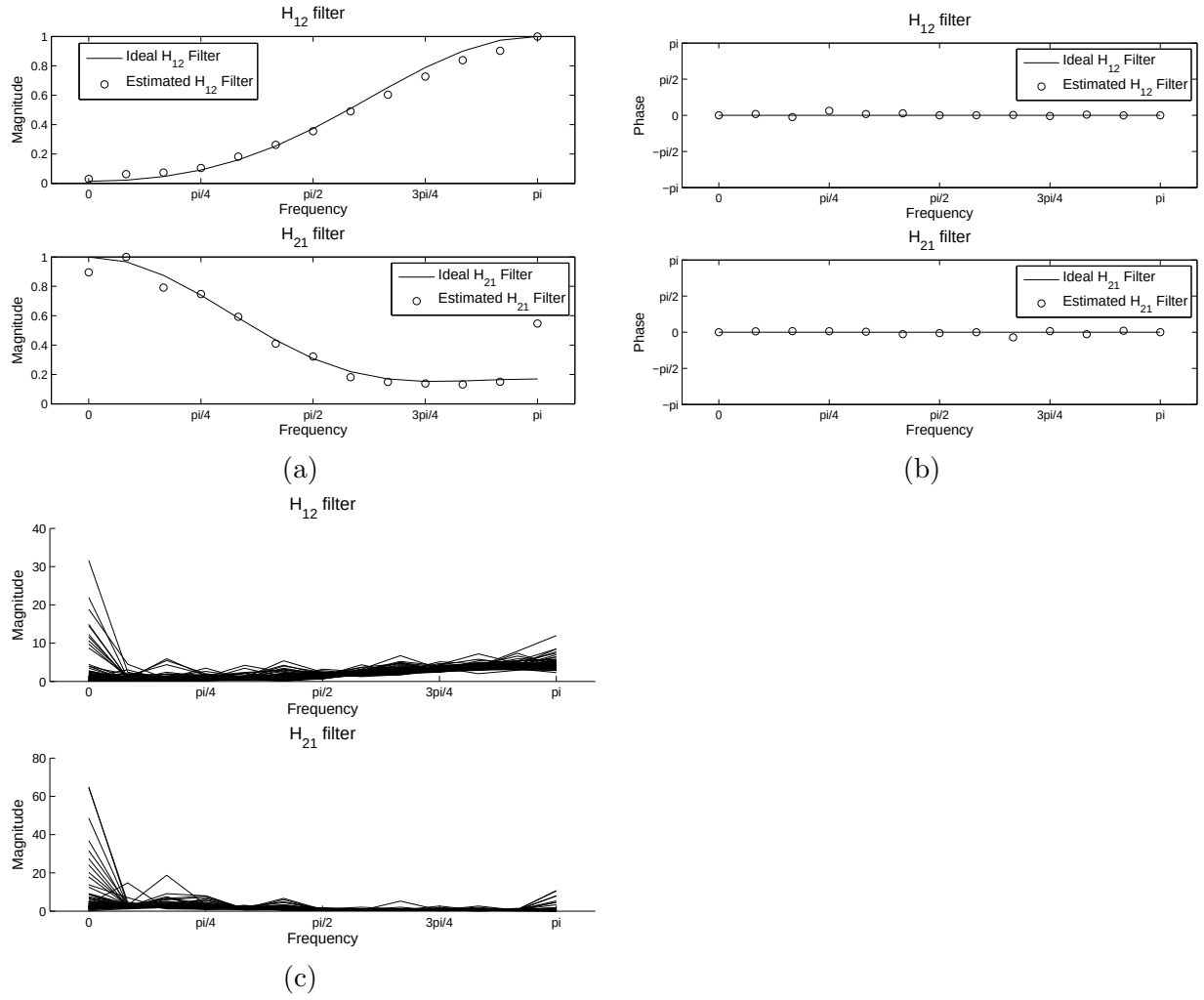


Figure 9.8: (a) Magnitude Estimation of the two filters, (b)Phase Estimation of the two filters and (c) the 100 estimations

where the 100 estimations of the filters are plotted, there are some large outliers in the plots and these would also affect the spectrum estimation. One way of removing these outliers could be by removing filter estimation with large power compared to the other estimations. Another way is to try using another window functions in the bispectrum estimation.

The simulations is performed again with a Parzen and optimum window to see if this helps, the phase estimation is dropped as it does not provide any useful information anyway other than the phase is mostly linear, which it should be.

The outliers becomes smaller when the two other windows are used. But the magnitude estimation of the filters is not acceptable for the Parzen Window, the optimum window however produces the best results so far. This means that the filter estimation performs best with an optimum window. As the minimum phase estimation of the TITO model is acceptable, moving on to the non-minimum phase estimation would be the next step. However the reverse third order moment spectrum needs to implemented and simulated before this is possible.

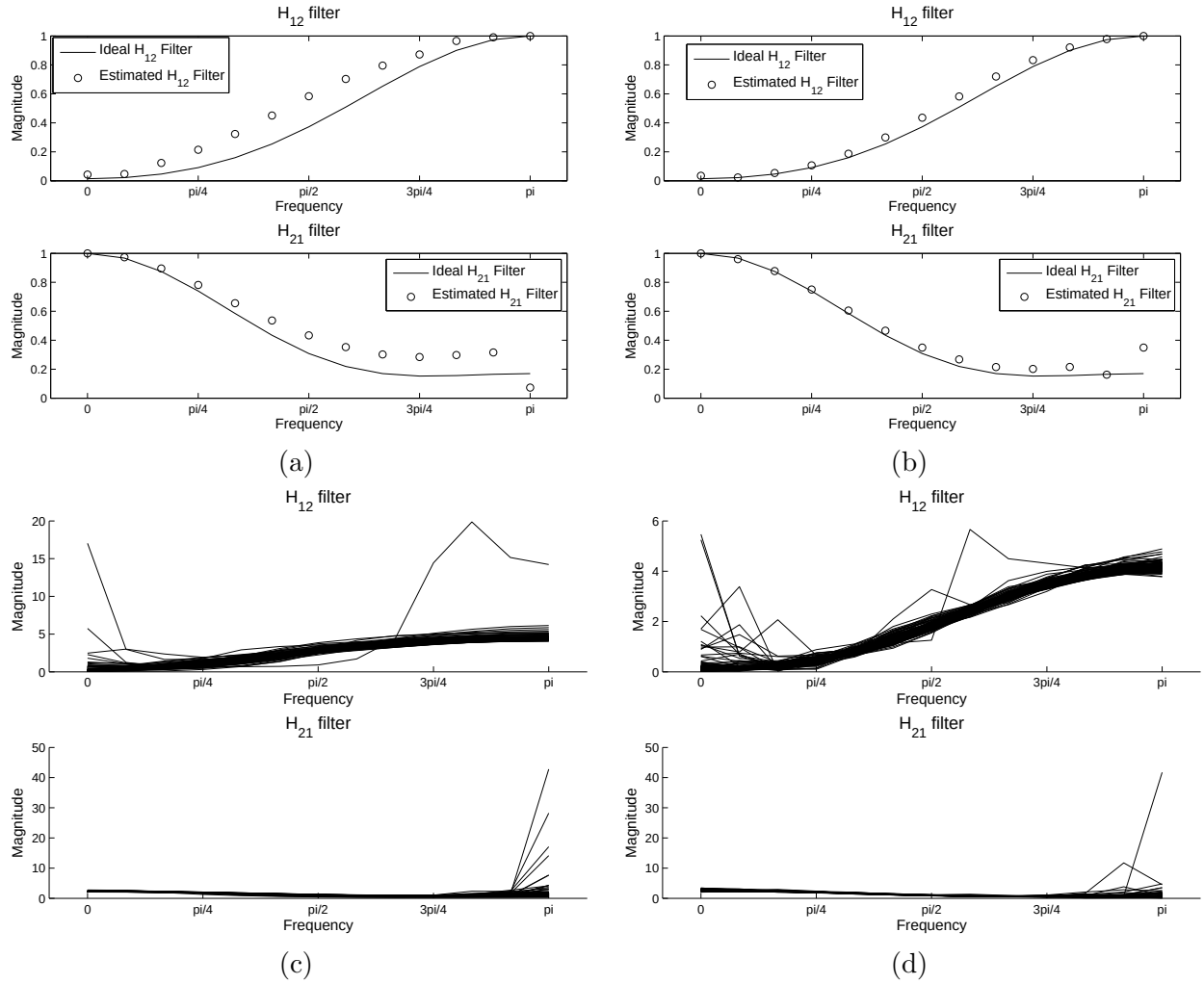


Figure 9.9: Magnitude Estimation of the two filters using (a) an Parzen window and (b) an optimum window. The 100 estimations of the filters using (c) an Parzen window and (d) an optimum window

Chapter 10

Reverse Third Order Moment Spectrum

The non-minimum filter estimation that is simulated in the next section gives the third order moment spectrum of the filters, but in order to use the results a reverse third order moment spectrum estimator is needed, to recreate the filters correctly if they are non-minimum phase. This method takes a moment spectrum and converts it back into the Fourier transform of the original signal, though with a phase and magnitude ambiguity, that needs to be addressed.

As described in the theory in section 4.2 the Fourier transform of the filter is estimated by reconstructing the phase and the magnitude from the bispectrum of the signal. This done in two separate operations in the theory section, therefore this is also implemented as two separate operations in the implementation. The first step is to estimate the original phase of the filter.

10.1 Estimating the Phase Response of the Filter

The step for estimating the phase response is as follows:

1. Construct the $\bar{\bar{A}}_\phi$ matrix as described in equation 4.100 on page 29. The size of the matrix depends on the dimension of the moment spectrum. If this is NxN the dimension of $\bar{\bar{A}}_\phi$ is $(N-1)\text{x}\left(\frac{N}{2}\right)^2$ if N is even and $(N-1)\text{x}\frac{(N-1)(N+1)}{4}$ if N is uneven.
2. Construct the $\bar{\phi}_{3h}$ vector containing the phase information from the bispectrum as described in equation 4.100 on page 29.. The length of $\bar{\phi}_{3h}$ vector is $\left(\frac{N}{2}\right)^2$ if N is even. If the matrix size is uneven the length is $\frac{(N-1)(N+1)}{4}$. To determine the phase the **angle** function from Matlab is used on the complex valued entries in the moment spectrum.
3. Determine the phase ambiguity $\hat{k}$ by doing the following:
   - Constructing the matrix $\bar{\bar{G}}_\phi$ as defined in equation 4.106.

- Constructing the matrix $\bar{\bar{F}}_\phi$ as defined in equation 4.112.
- Calculate the matrix $\bar{\bar{D}}_\phi$ from equation 10.1:

$$\bar{\bar{D}}_\phi = \left[\bar{\bar{F}}_\phi \cdot \bar{\bar{G}}_\phi^{-1} \ \bar{\bar{0}}\right] \tag{10.1}$$

- From matrix $\bar{\bar{D}}_\phi$ and the $\bar{\phi}_{3h}$ it is possible to make an estimate of $\hat{k}$ using equation 10.2

$$\hat{\bar{k}} = \text{round}\left(\frac{\bar{\bar{A}}_\phi \cdot \bar{\bar{D}}_\phi \cdot \bar{\phi}_{3h} - \bar{\phi}_{3h}}{2 \cdot \pi}\right) \tag{10.2}$$

  The result of this is rounded using Matlab **round** function in order to round to the nearest integer.

4. Determine the phase of the filter using equation 10.3

$$\bar{\phi}_h = \left(\bar{\bar{A}}_\phi^T \bar{\bar{A}}_\phi\right)^{-1} \bar{\bar{A}}_\phi^T \cdot \left(\bar{\phi}_{3h} + 2\pi \cdot \hat{\bar{k}}\right) \tag{10.3}$$

10.2 Estimating the Magnitude Response of the filter

After the phase is established, the magnitude also needs to be established as well. The procedure is a little different from the phase estimation as the logarithm must be taken of the bispectrum as described on page 32. This means that if an entry in the bispectrum has a magnitude of zero, the resulting value becomes minus infinite. This is bad for the magnitude estimation as all the values then becomes minus infinite and it not possible to get meaningful results. Therefore any zeros needs to be handled, which is described in following step for calculating the magnitude.

1. Construct the matrix $\bar{\bar{A}}_\mu$ as described in equation 4.118 on page 32. The dimensions of the matrix is $(N)\text{x}\frac{N^2 + 2 \cdot N}{4}$ if N is even and $(N-1)\text{x}\frac{(N-1)(N+1) + 2 \cdot N}{4}$ if N is uneven.
2. Construct the magnitude vector $\bar{\mu}_{3h}$ as described in equation 4.118 on page 32. The length of the vector is $\frac{N^2 + 2 \cdot N}{4}$ if N is even and $\frac{(N-1)(N+1) + 2 \cdot N}{4}$ if N is uneven.
3. Take the logarithm of the $\bar{\mu}_{3h}$ vector.
4. Copy the $\bar{\mu}_{3h}$ magnitude vector into a new vector $\bar{\mu}_{3hredux}$ and remove any entries with negative overflow in the new vector. At the same time create a vector $\bar{y}$ of ones the same length as $\bar{\mu}_{3h}$, and negate the current entry to zero each time a entry in $\bar{\mu}_{3hredux}$ is removed.
5. The vector $\bar{y}$ is element wise multiplied with the rows in the matrix $\bar{\bar{A}}_\mu$. After this the matrix pruned by remove any columns and rows which sum is zero.
6. Calculate the magnitude response of the filter using least squares as in equation 10.4

$$\bar{\mu}_h = \left(\bar{\bar{A}}_\mu^T \bar{\bar{A}}_\mu\right)^{-1} \bar{\bar{A}}_\mu^T \cdot \bar{\mu}_{3hredux} \tag{10.4}$$

7. Reinsert the missing entries in $\bar{\mu}_h$ by determining rows in the original (before the pruning) $\bar{\bar{A}}_\mu$ matrix, which sum is zero. The value of the these entries should be negative overflow.

Now that the magnitude and frequency response of the filter have been estimated the last step is to calculate the Fourier transform of the filter by using equation 10.5.

$$\hat{\bar{H}}(\omega) = \exp\left(\hat{\bar{\mu}}_h\right) \cdot \exp\left(j \cdot \bar{\phi}_h\right) \tag{10.5}$$

The implementation in MATLAB can be found on the accompanying CD in `/Matlab Code/Trispectrum Estimation/revbispec.m` as an m-file.

10.3 Simulation of the Reverse Third Order Moment Spectrum

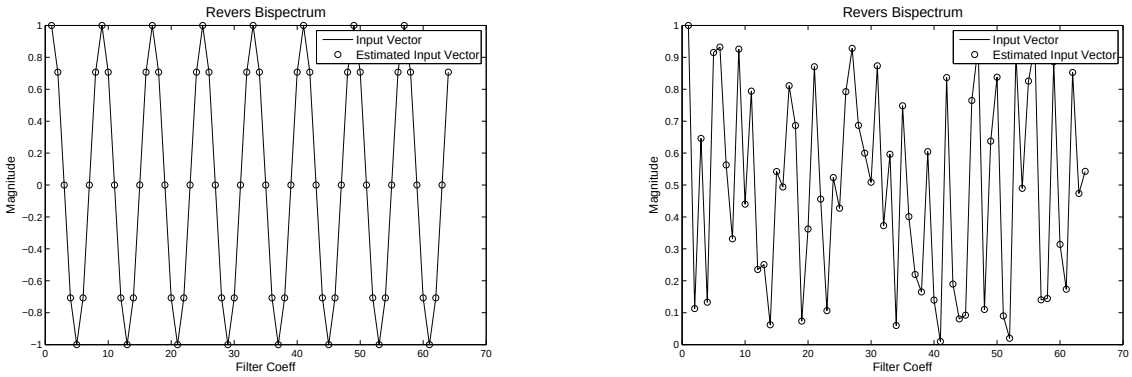
This section contains the simulation of the reverse third order moment spectrum which MATLAB is described in the previous section. For this simulation the direct method for creating a bispectrum that is implemented in section 9.1 is used. However the first step in the direct method, where the mean is subtracted, is removed from the implementation in order to create the third order moment spectrum instead of the bispectra. The test consist of creating a third order moment spectra from a test signal and then using the reverse bispectrum implementation to recreate this test signal again.

As the input signal does not need a to be a stochastic signal, the input signal is are created by using the **rand** function in MATLAB which is taken from a normal distribution, this would normally be a bad thing as the trispectrum ideally would be zero for this distribution, but as the input signal used is short chances are that distribution is completly Gaussian for this short sequence. A length of 64 samples for the input signal is found to have an acceptable execution time.

The MSE between the input signal and the estimated input signal is calculated and this is repeated 1000 time with different input signals and the mean MSE of these runs is calculated. One thing that the test must be able to handle is a circular shift and scaling of the estimated test signal, this is handled by scaling the largest value in the both the test vector and the estimated test signal to one and shifting the largest value to the first position in the array. The test m-file can be found in `/Matlab Code/Trispectrum Estimation/RevTest.m`.

The resulting MSE from the simulation using a random is $1.7728 \cdot 10^{-30}$, which gives a introduces noise around 300 dB. In order to test the ability to handle zeros in the bispectrum. The random input signal is replaced with a sinus function, and the test file is simulated again. The resulting MSE goes up a little to $2.9451 \cdot 10^{-29}$, with a SNR of around 290 dB. Figure 10.1 show a plot of one of the input signal and the reverse estimation of the input signal for both the random input vector and the sinus function.

In general the SNR is high regardless if random or sinus signals are used for the reverse third order moment spectrum. Numerical noise would be around 320 dB so the noise in the algorithm is not only numerical noise, however it is still so low that it is deemed to have acceptable results.



(a) (b)

Figure 10.1: Plot of input signal and the estimated input signal, when the bispectrum is calculated for the input signal and the reverse operation is performed, (a) is with sinus input vector, (b) is with random input vector

Chapter 11

Non-minimum Phase Filter Estimation

As it is possible to estimate the filters as minimum phase filters and recreate the filters in the Fourier domain via reverse third order moment spectrum. All the prerequisites for doing the non-minimum phase filter estimation have been meet. The only thing missing in order to perform the non-minimum phase filter estimation described by equation 4.29 and 4.30 on page 22, is a trispectrum estimator. This need to implemented and simulated before the non-minimum phase filter estimation can be simulated.

11.1 Trispectrum Estimation

This section is the implementation and simulation of a trispectrum estimator in MATLAB. As a bispectrum estimator already have been and simulated must of the most of the step is merely a matter of extending the bispectrum estimator with an additional vector. The only real difference comes form the moment to cumulant transformation, for the bispectrum estimator this could be done removing the mean from the input vectors. This is however not that simple when working with the trispectrum, as this transformation is only defined in the ”time” domain. This present a problem for the direct method as the fourth order moment spectrum must be inverse three dimensional Fourier transformed into the ”time” domain. Then the moment to cumulant transformation can be performed and the result must be transformed into the frequency domain again with a three dimensional Fourier transform. This is very cumbersome as the advantage with direct approach was that everything could more or less be done in the frequency domain.

The equation for the doing the moment to cumulant transformation for the fourth order is listed as equation A.56 in appendix A.1.8 on page 174. This equation is listed again in equation 11.1, the transformation is for zero mean signals, so the mean should still be subtracted from the input signals. If this moment to cumulant transformation could be performed in the frequency domain, the two consecutive cubic Fourier transformations would be avoided.

$$\begin{aligned} c_{xyzw}(\tau_1,\tau_2,\tau_3) &= m_{xyzw}(\tau_1,\tau_2,\tau_3) - m_{xy}(\tau_1)\cdot m_{zw}(\tau_3-\tau_2) \\ &\quad -m_{xz}(\tau_2)\cdot m_{yw}(\tau_3-\tau_1) - m_{xw}(\tau_3)\cdot m_{yz}(\tau_2-\tau_1) \end{aligned} \tag{11.1}$$

If an Fourier transform is applied to equation 11.1, it would result in equation 11.2

$$\begin{aligned} C_{xyzw}(\omega_1,\omega_2,\omega_3) &= M_{xyzw}(\omega_1,\omega_2,\omega_3) \\ &\quad -\mathscr{F}\left[m_{xy}(\tau_1)\cdot m_{zw}(\tau_3-\tau_2)\right] \\ &\quad -\mathscr{F}\left[m_{xz}(\tau_2)\cdot m_{yw}(\tau_3-\tau_1)\right] \\ &\quad -\mathscr{F}\left[m_{xw}(\tau_3)\cdot m_{yz}(\tau_2-\tau_1)\right] \end{aligned} \tag{11.2}$$

This removes the necessity for the inverse cubic Fourier transforms but the number of cubic Fourier transforms has risen to three. Equation 11.2 can be further reduced as in equation 11.3

$$\begin{aligned} C_{xyzw}(\omega_1,\omega_2,\omega_3) &= M_{xyzw}(\omega_1,\omega_2,\omega_3) \\ &\quad -M_{xy}(\omega_1) * \mathscr{F}\left[m_{zw}(\tau_3-\tau_2)\right] \\ &\quad -M_{xz}(\omega_2) * \mathscr{F}\left[m_{yw}(\tau_3-\tau_1)\right] \\ &\quad -M_{xw}(\omega_3) * \mathscr{F}\left[m_{yz}(\tau_2-\tau_1)\right] \end{aligned} \tag{11.3}$$

Already a reduction of the complexity has been achieved as the Fourier transform has gone from 3 dimensions to 2 dimensions and removing the need for a inverse Fourier transform. This new method is implemented into the direct method for doing the bispectra and the method is also extended to include one additional input signal.

11.1.1 Direct Method for Estimating the Trispectrum

The direct method used for estimating the trispectrum is based [1, pp 124-127] and appendix A.3.

1. Remove the mean from the input frames, in order to make part of the transition from moments to cumulants.
2. Calculate the discreet Fourier transform (DFT) off the input frames.

$$X(\omega) = \sum_{t=0}^{N-1} x(t)\cdot\exp\left(-\frac{2\pi j}{N}(t\omega)\right) \tag{11.4}$$

   If cross spectra's is calculated, then a DFT is performed for each individual input signal.
3. The trispectrum is calculated from equation 11.5. As this is made as a general function it is assumed that all the input signals differs from each other. Effectively giving a cross trispectrum between four different signals.

$$M_{xzyw}(\omega_1,\omega_2,\omega_3) = \frac{1}{N}Y(\omega_1)\cdot Z(\omega_2)\cdot W(\omega_3)\cdot X^*(\omega_1+\omega_2+\omega_3) \tag{11.5}$$

   Where:
   $0 \le \omega_1 \le \pi,\ 0 \le \omega_2 \le \pi,\ \omega_1+\omega_2 \le \pi$
   $X(\omega) = Y(\omega) = Z(\omega) = W(\omega) = 0|_{\omega>\pi}$ N is the length of the frames.

4. Make the moment to cumulant transformation as described by equation 11.3.
5. The last step is the smoothing of the spectrum using equation 11.6.

$$\hat{C}_{xyz}(\omega_1, \omega_2) = \frac{1}{(2L+1)^3} \sum_{k_1=-L}^{L} \sum_{k_2=-L}^{L} \sum_{k_3=-L}^{L} M_{xzyw}(\omega_1 + k_1, \omega_2 + k_2, \omega_3 + k_3) \tag{11.6}$$

Where:
L is the number of samples to smooth over.

As shown in the bispectrum estimation simulations, the last step can be replaced by 3 dimensional convolution with a window instead.

An implementation of direct method can be found as a Matlab on the accompanying CD in `/Matlab Code/Trispectrum Estimation/trispec2.m`.

11.1.2 Indirect method for estimating the trispectrum

This method for estimating trispectrum via the indirect method is also based on [1, pp 124-127] and appendix A.3.

1. Remove the mean from the input vectors.
2. Calculate the fourth order moment sequence using equation 11.7

$$c_{xyzw}(\tau_1, \tau_2, \tau_3) = \frac{1}{N} \sum_{t=s_1}^{s_2} x(t) \cdot y(t+\tau_1) \cdot z(t+\tau_2) \cdot w(t+\tau_3) \tag{11.7}$$

Where:
$\tau_1, \tau_2, \tau_3 = 0, \pm 1, \pm 2, \ldots, \pm$ maximum lag
$s_1 = max(0, -\tau_1, -\tau_2, -\tau_3)$
$s_2 = min(N-1,\ N-1-\tau_1,\ N-1-\tau_2\, N-1-\tau_3\ )$

3. Apply the moment to cumulant transformation described by equation 11.1.
4. Apply a 3 dimensional window function to the cumulants sequence.
Agian this is already done partly by using a limited range for τ, however the simulations of the minimum phase filter estimation showed better performance when using an optimum window.
5. Zero pad to obtain a appropriate Fourier length.
Certain Fourier lengths makes it possible to make more efficient DFT implementations. Zero padding is normally done by adding zeros to its ends, this is however not the approach, when a cumulant sequence is zero padded. The padding should be done so that that the point with $\tau = 0$ always stays in the center of the matrix. If the array has an even number of entires (no exact center) the point with $\tau = 0$ should be offset to $\tau > 0$ side of the array.

6. Apply a three dimensional Fourier transform on the cumulant sequence, using equation 11.8.

$$C_{xyzw}(\omega_1, \omega_2, \omega_3) = \sum_{\tau_1} \sum_{\tau_2} \sum_{\tau_3} c_{xyz}(\tau_1, \tau_2) \cdot \exp\left(j(\tau_1\omega_1 + \tau_2\omega_2)\right) \tag{11.8}$$

   Where:
   $\tau_1, \tau_2 = 0, \pm1, \pm2, \ldots, \pm$ number of filter coefficients

   Instead of implementing equation 9.5 MATLAB's own **fftn** function for an n - dimensional DFT is used. It is more efficient then doing the above calculation, though it should be noted that **fftn** assumes a different arrangement of τ_1, τ_2 and τ_3, this is corrected by doing **ifftshift** on the cumulant sequence, which rearranges the array before the three dimensional DFT is applied.

The indirect method can be found as a Matlab m-file on the accompanying CD in `/Matlab Code/Trispectrum Estimation/trispec.m`. Also a Matlab function that constructs the three dimensional window functions is located on the CD in `/Matlab Code/Trispectrum Estimation/window.m`.

Equality between the two methods was shown in bispectrum estimator, this will not be simulated for the trispectrum estimator. The focus would only be on the indirect trispectra estimators ability to estimate a filter.

11.2 Simulation of the Trispectrum Estimator

A similar equation as 9.7 for the bispectrum can be made for the trispectrum, when a stochastic signal is sent through a know filter.

$$C_{x^4}(\omega_1, \omega_2, \omega_3) = \gamma_Y^4 \cdot H(\omega_1) \cdot H(\omega_2) \cdot H(\omega_3) \cdot H^*(\omega_1 + \omega_2 +, \omega_3) \tag{11.9}$$

Where:
γ_Y^4 is the kurtosis of the random variable Y
x is the stochastic signal y convolved with the filter h
H is Fourier transform of a filter h

If the following vector in the trispectrum in equation (see 11.10) is evaluted it, it would contains the powerspectrum scaled by a factor of $H(0)^2$.

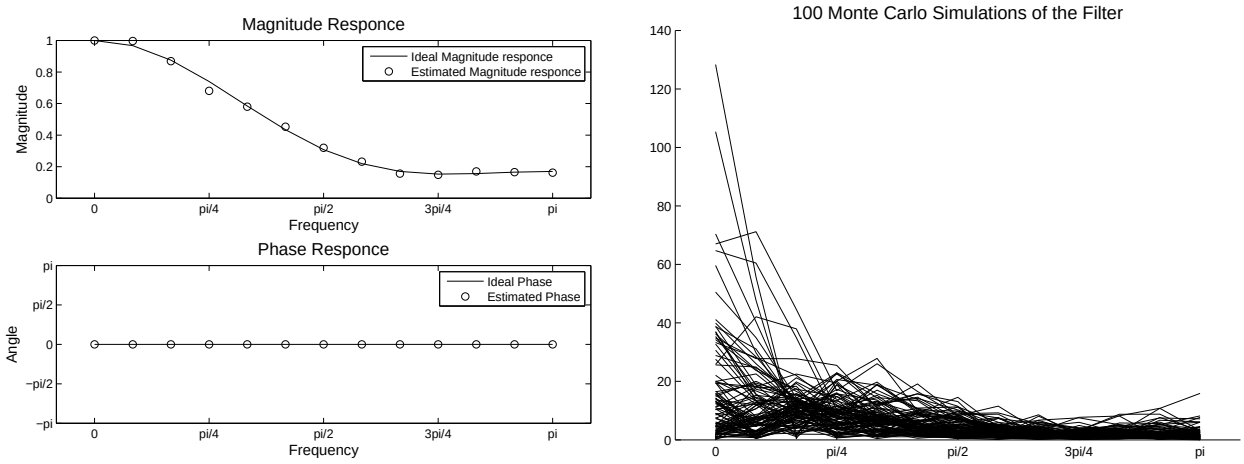
$$C_{x^4}(\omega_1, 0, 0) = \gamma_Y^4 \cdot H(\omega_1) \cdot H(0)^2 \cdot H^*(\omega_1) \tag{11.10}$$

This makes it possible to compare this vector from the trispectrum to the actual powerspectrum of the filter to ensure the functionality of the trispectrum estimator. The filter used for simulation is listed in equation 11.11

$$h(t) = [1,\ .5,\ .2] \tag{11.11}$$

As the result is evaluated as a powerspectrum, no non-minimum phase filters are used for the test. The signal Y is a stochastic signal with a exponential distribution as in the previous simulations. Lags are varied over ±9 and an 24 frequency bins are calculated. The resulting powerspectrum is

averaged over 100 simulations and the mean for the magnitude and phase of the estimated filter is plotted in figure 11.1. Please note that the two results, the ideal and estimated are scaled to each other. The MATLAB file for the simulation can be found in `/Matlab Code/Trispectrum Estimation/tritest2.m`



(a) (b)

Figure 11.1: (a)Mean estimation for a high pass filters magnitude and phase response using 100 Monte Carlo simulations (b) The 100 simulations

The magnitude plots are not as good as they where in the bispectra estimation. This is somewhat expected as an additional dimension is added, without it providing anything useful and it can be seen from plot (b), that the variance have also risen. But the result is acceptable. Therefore it is possible to proceed with the non-minimum phase filter estimation.

11.3 Simulation of the Non-Minimum Phase Filter Estimation

From chapter 4 it was found the third order momentspectrum of the filters in the TITO system model could be solved using equations 11.12 and 11.13.

$$M_{h_{12}^3}(\omega_1,\omega_2) = \frac{C_{x_1^3x_2}(\omega_1,\omega_2,0) - H_{21}(0)\cdot C_{x_1^4}(\omega_1,\omega_2,0)}{C_{x_2^4}(\omega_1,\omega_2,0) - H_{21}(0)\cdot C_{x_2^3x_1}(\omega_1,\omega_2,0)} \tag{11.12}$$

$$M_{h_{21}^3}(\omega_1,\omega_2) = \frac{C_{x_2^3x_1}(\omega_1,\omega_2,0) - H_{12}(0)\cdot C_{x_2^4}(\omega_1,\omega_2,0)}{C_{x_1^4}(\omega_1,\omega_2,0) - H_{12}(0)\cdot C_{x_1^3x_2}(\omega_1,\omega_2,0)} \tag{11.13}$$

As the reverse third order moment spectrum has been simulated and found working and the trispectrum estimator was simulated in the previous chapter and found acceptable. The only thing missing in the equations are estimates of $H_{12}(0)$ and $H_{21}(0)$. But as known filters are used in this simulation ideal estimates are instead. The equations 11.12 and 11.13 are implemented in MATLAB as two **for** loops, where the third order moment spectrum of the filter is calculated for each frequency bin.

For this simulation the filters stated in equation 11.14 and 11.15 are used.

$$h_{12} = [.3, .8, .4] \tag{11.14}$$
$$h_{21} = [1, .5, .2] \tag{11.15}$$

Their zero pole plots of the filters can be seen in figure 11.2. And from this it is obvious that h_{12} is an non-minimum phase filter, because of the pole located outside the unit circle. The test

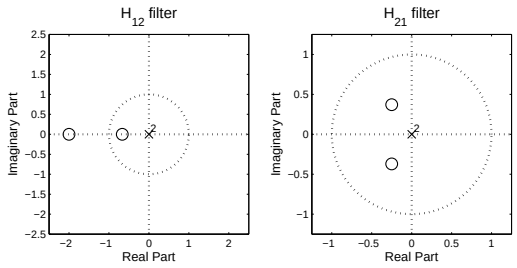


Figure 11.2: Zero Pole plot of the two filters used for the test.

signals are constructed as described in chapter 7 and the indirect method is used to estimated the trispectrum. An optimum window is used, as the results from the minimum phase filter estimation showed better performance when using it. The amount of lags used in the trispectrum method ranges from ±8 and the number frequency bins used is 20. The test-file can be found on the CD as `/Matlab Code/Trispectrum Estimation/tritest1.m`. The original and estimated filter coefficient, and their phase are illustrated in figure 9.8.

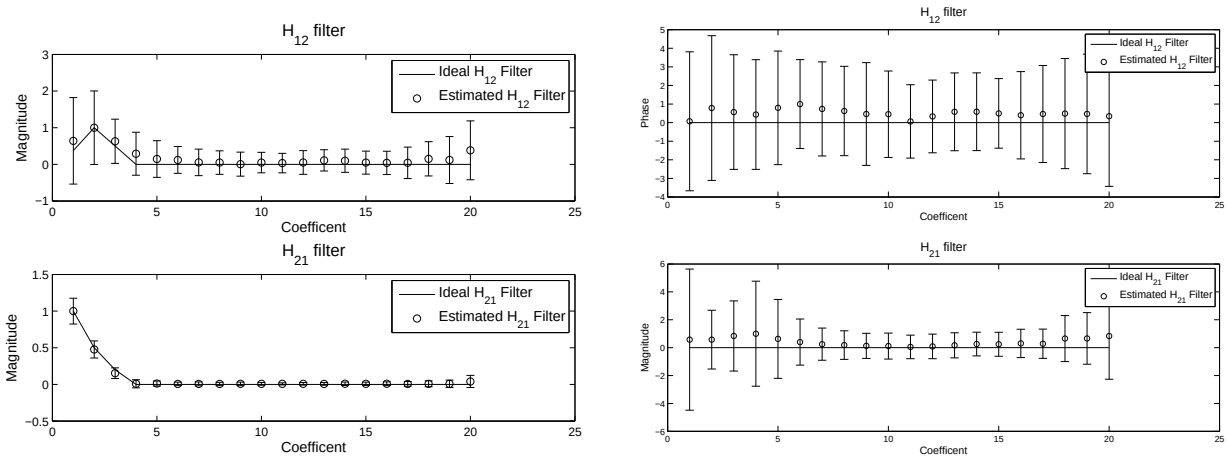


Figure 11.3: Result of the filter estimation.

The above plots of the estimation of the filter coefficients are deemed acceptable for the minimum phase filter h_{21}. The phase results would be ignored as one of the priory knowledges about the model is that only real filters are used. The filter coefficients for the non-minimum phase filter h_{12} is not as good. But would pass if the filter order is known to be 3. The last thing to simulate, before the final simulation of the whole BSS is the estimation of the DC-amplification in the filters the $\bar{\bar{H}}(0)$ estimation.

Chapter 12

H(0) Estimation

This chapter contains an simulation of the $\bar{\bar{H}}(0)$ estimation. The simulation is performed in MATLAB and the consequent MATLAB implementation is based on the theory presented in section 4.1. The $\bar{\bar{H}}(0)$ estimation presented in chapter 4.1 can be divided into the part which are shown in figure 12.1

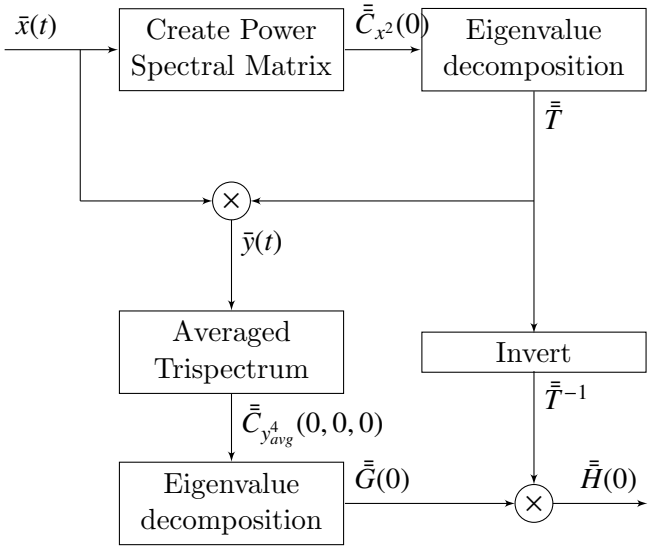


Figure 12.1: Illustration of mathematical model for solving the $\bar{\bar{H}}(0)$ estimation

There are some inherent problem with the first step. The first problem is that the values in the $\bar{\bar{C}}_x(0)$ matrix are always close to or zero as the mean is subtracted before the power spectrum is calculated, this is related to the frequency time uncertainty that have been mentioned before. The second problem is that the eigenvalue decomposition of $\bar{\bar{C}}_x(0)$ will always give only one eigenvalue, regardless if $\bar{\bar{C}}_x(1)$ or $\bar{\bar{C}}_x(2)$ etc. is used. Making it impossible to make the transformation matrix $\bar{\bar{T}}$.

The prof of the second problem is shown in equations 12.1 to 12.5

$$\bar{\bar{C}}_x(\omega) = \begin{bmatrix} X_1^*(\omega)\cdot X_1(\omega) & X_2^*(\omega)\cdot X_1(\omega) \\ X_1^*(\omega)\cdot X_2(\omega) & X_2^*(\omega)\cdot X_2(\omega) \end{bmatrix} \tag{12.1}$$

$$\text{Eigen}(\bar{\bar{C}}_x) \Rightarrow \det\begin{bmatrix} X_1^*\cdot X_1-\lambda & X_2^*\cdot X_1 \\ X_1^*\cdot X_2 & X_2^*\cdot X_2-\lambda \end{bmatrix} = 0 \tag{12.2}$$

$$0 = \left(X_1^*\cdot X_1-\lambda\right)\cdot\left(X_2^*\cdot X_2-\lambda\right)-\left(X_2^*\cdot X_1\right)\cdot\left(X_1^*\cdot X_2\right) \tag{12.3}$$

$$0 = \lambda^2-\lambda\cdot\left(X_1^*\cdot X_1+X_2^*\cdot X_2\right) \tag{12.4}$$

$$\lambda = X_1^*(\omega)\cdot X_1(\omega)+X_2^*(\omega)\cdot X_2(\omega) \tag{12.5}$$

In order to circumvent the first problem another frequency component that is close to the "DC"-frequency is used instead. The assumption is that the power level would also be close to that of the frequency bin representing DC. The amount the frequency component should be shifted is assess to be five frequency bins when a frame of 4000 samples are used for the powerspectrum.

To circumvent the second problem of one eigenvalue, it is necessary to averaged over several $\bar{\bar{C}}_x(0)$ estimations, to make a good eigenvalue decomposition. This averaging is also performed for $\bar{\bar{C}}_{y^4_{avg}}(0,0,0)$ in order to improve this eigenvalue decomposition as well. From these considerations and the figure 12.1 the MATLAB implementation is implemented in the following steps:

1. Create the power and cross -Spectrum matrix $\bar{\bar{C}}_x(4)$ for 100 frames and calculate the averaged $\hat{\bar{\bar{C}}}_x(4)$
2. Make an eigenvalue decomposition of $\hat{\bar{\bar{C}}}_x(4)$, this is done using MATLAB own functions for doing a eigenvalue decomposition.
3. Create the Transformation matrix $\bar{\bar{T}}$ from the eigenvectors and the eigenvalues as in equation 12.6

$$\bar{\bar{T}} = \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}} & 0 \\ 0 & \frac{1}{\sqrt{\lambda_2}} \end{bmatrix}\cdot\bar{\bar{U}}^H \tag{12.6}$$

4. Create $\bar{y}(t)$ from $\bar{\bar{T}}$ and $\bar{x}(t)$
5. Calculate the averaged trispectrum matrix $\hat{\bar{\bar{C}}}y(0,0,0)$ using the indirect method for a trispectrum described earlier and equation 12.7

$$C_{y^4_{avg}}(0,0,0) = \begin{bmatrix} C_{y_1^4}(0,0,0)+C_{y_2^2y_1^2}(0,0,0) & C_{y_1^3y_2}(0,0,0)+C_{y_2^2y_1y_2}(0,0,0) \\ C_{y_1^2y_2y_1}(0,0,0)+C_{y_2^3y_1}(0,0,0) & C_{y_1^2y_2^2}(0,0,0)+C_{y_2^4}(0,0,0) \end{bmatrix} \tag{12.7}$$

6. Make an eigenvalue composition of $\hat{\bar{\bar{C}}}y(0,0,0)$ to determine the eigenvectors $\bar{G}_1(0)$ and $\bar{G}_2(0)$
7. Calculate the scaling of $\bar{G}_1(0)$ and $\bar{G}_2(0)$ as:

$$k_1 = \frac{1}{T_{11}^{-1}G_{11}(0)+T_{21}^{-1}G_{21}(0)} \tag{12.8}$$

$$k_2 = \frac{1}{T_{12}^{-1}G_{12}(0)+T_{22}^{-1}G_{22}(0)} \tag{12.9}$$

8. Determine the $\bar{\bar{H}}(0)$ matrix as:

$$\bar{\bar{H}}(0) = \bar{\bar{T}}^{-1} \cdot \bar{\bar{k}} \cdot \bar{\bar{G}}(0) \tag{12.10}$$

The above implementation does not take into account the shuffling of the eigenvectors as described in the theory sections. Which gives the possibility that the DC- coefficients is assigned to the wrong filter and/or is inverted. The implementation in MATLAB, does not include any precautionary measure to handle this problem. But this is discussed further in the simulation section. The implementation can be found on the accompanying CD as: `/Matlab Code/H(0) Estimation/Hest.m`.

12.1 Simulation

The simulation is performed with frames with a length of 4000 samples the averaging for $\hat{\bar{\bar{C}}}_x(4)$ and $\hat{\bar{\bar{C}}}y(0,0,0)$ is performed over 100 frames, with a total of 400.000 samples for one estimation of $H(0)$. This simulation is performed 100 times and the estimated DC-coefficients for $H_{12}(0)$ and $H_{21}(0)$ are plotted as a function of the trials. The signals $x_1(t)$ and $x_2(t)$ are constructed as described in chapter 7 and the filter coefficients for $h_{12}(t)$ and $h_{21}(t)$ are [.3, .8, .4] and [1, .5, .2]. The results from the simulation can be seen in figure 12.2.

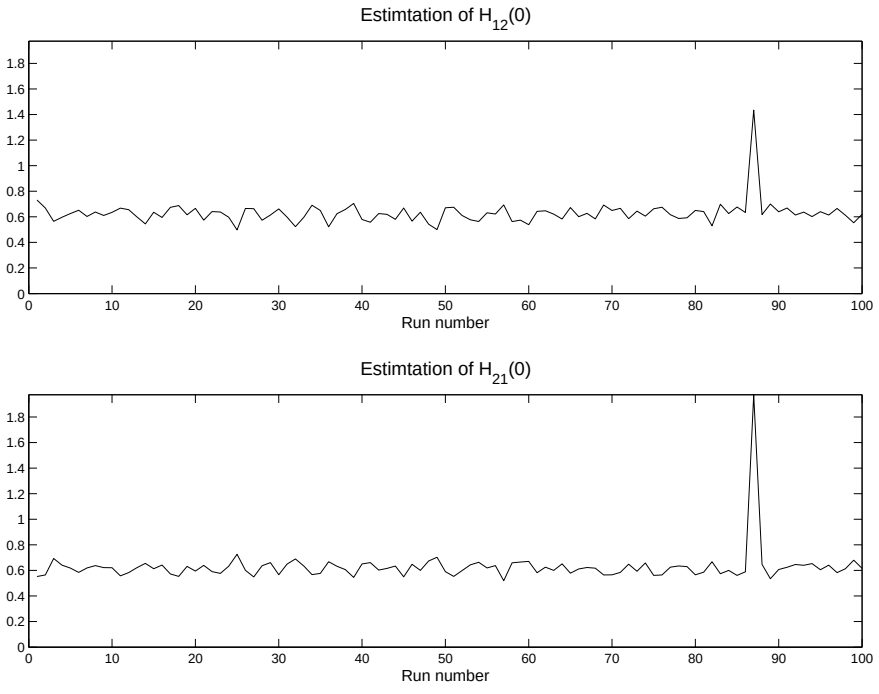


Figure 12.2: Estimation of the DC-coefficent for H_{12} and H_{21} over 100 runs.

The mean and the standard deviation in figure 12.2 is 0.630 ± 0.094 and 0.628 ± 0.142 for $H_{12}(0)$ and $H_{21}(0)$. As it is know that the mean of both filters should be above one all runs, except run number 87 which has values larger the one, must be inverted and swapped as the eigenvectors where not assigned correctly to $\bar{\bar{G}}$. This is corrected in figure 12.3.

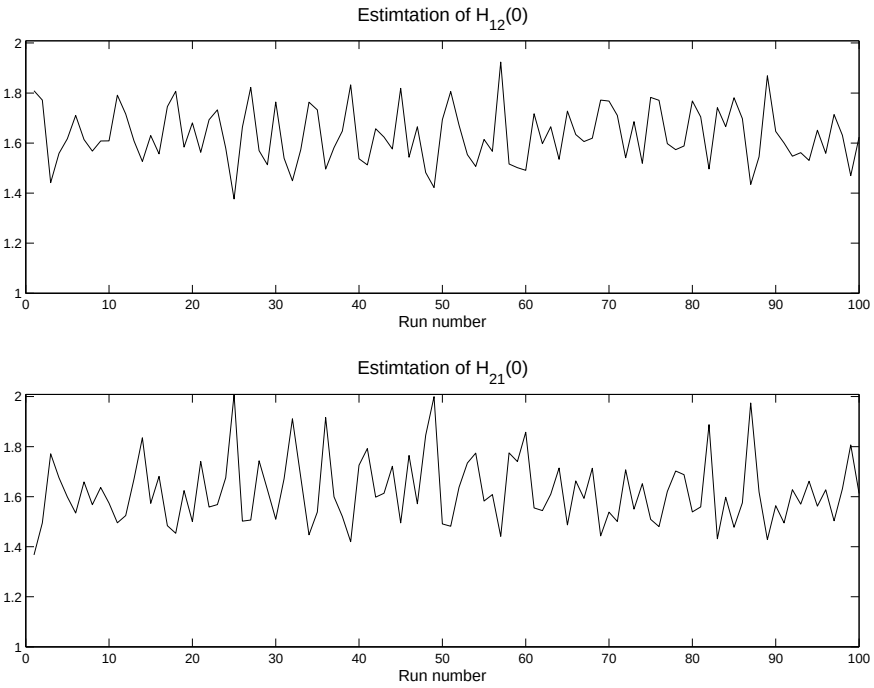


Figure 12.3: Estimation of the DC-coefficent for H_{12} and H_{21} over 100 runs.

The mean and the standard deviation in figure 12.3 is now 1.663 ± 0.111 and 1.622 ± 0.134 for $H_{12}(0)$ and $H_{21}(0)$. As the ideal $H_{12}(0)$ and $H_{21}(0)$ are know as 1.5 and 1.7, and that assigning the eigenvectors wrong for the $\bar{\bar{T}}$. would result in the values being swapped, the results are swapped each time that $H_{21}(0) < H_{12}(0)$, which results in figure 12.4.

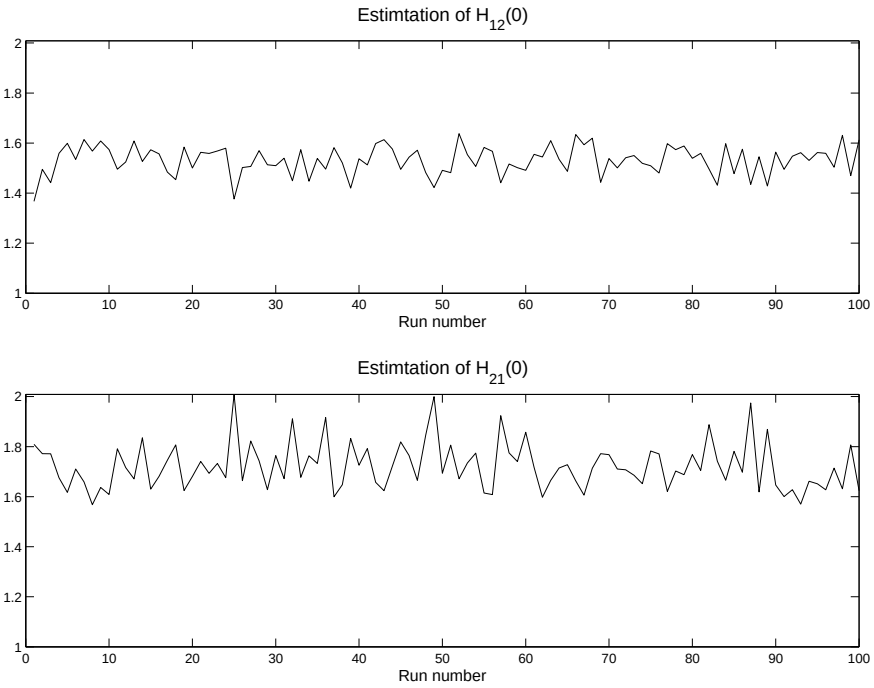


Figure 12.4: Estimation of the DC-coefficent for H_{12} and H_{21} over 100 runs.

Using priori the priori knowledge that $H_{12}(0)$ and $H_{21}(0) > 1$ and $H_{21}(0) < H_{12}(0)$. The mean and the standard deviation in figure 12.3 comes to 1.532 ± 0.058 and 1.722 ± 0.094 for $H_{12}(0)$ and $H_{21}(0)$. As the ideal values $H_{12}(0)$ and $H_{21}(0)$ are know as 1.5 and 1.7. The results are well within 5 %. For the above results a dataset of 40,000,000 is used, but using a data set of 400,000 the worst result from the 100 simulations gave a deviation of 20 %. The simulation file can be found on the accompanying CD as: `/Matlab Code/H(0) Estimation/Hesttest.m`.

12.2 Conclusion

From a 100 simulations $H_{12}(0)$ and $H_{21}(0)$ could be established within 5 % using priori knowledge about the filters. But the large dataset also indicates that it needs more data then the other simulations to give good estimates. The worst result from the 100 simulations gave up to 20 % deviation, it give rise to the question, how sensity the BSS is to these divations in the $\bar{\bar{H}}(0)$ estimation. Therefore the first thing to establish for the BSS simulation is a sensitivity simulation for $\bar{\bar{H}}(0)$, with deviation up to 20 % from the ideal value is simulated. As an working implementation for the $\bar{\bar{H}}(0)$ estimation now exists the last thing is to simulate the entire blind source separation algorithm.

Chapter 13

Blind Source Separation Simulation

This chapter contains a simulation of the BSS algorithm that was presented in the theory part of this project. The simulation is conducted in Matlab and includes all the parts, that have been simulated in the previous chapters.

The simulationg is conducted in two parts:
The first part is using an ideal estimation of $\bar{\bar{H}}(0)$ and making deviations, it is intially assumed that the worst case scenario is to subtract from one filter and add to the other filter, so a certain percentage is subtracted from one of the filters in $\bar{\bar{H}}$ and the same percentage is added to the other filter. This would intially give an overview of the sensitivity of this parameter, which showed a deviation of up to 20 % in the previous simulation.
The second part is a simulation using the implemeted version of the $\bar{\bar{H}}(0)$ estimator, that was simulated in the previous section.

In order get some idea what kind of results should be expected the Cramer Rao lower bound for BSS is calculated as an indicator of maximum obtainable signal to noise ration (SNR). From [10] a method for calculating this bound based on the TITO systen model is presented.

For this simulation the signal to interference ratio (SIR) is used to calculate the performance of the blind source separation. The reason for not using the SNR designation is that there is no added noise to the signal there is only interference from the other signal. But the methods for calculating the SNR is the same as for the SIR.

Equation 13.1 describes how the SIR is calculated.

$$\mathrm{SIR}_i = \frac{\mathrm{E}\left[(s_i(t))^2\right]}{\mathrm{E}\left[(s_i(t) - \hat{s}_i(t))^2\right]} \tag{13.1}$$

where:
$s_i(t)$ is the i'th source signal
$\hat{s}_i(t)$ is the restored i'th signal

The Cramer Rao lower bound (CRLB) can be calculated for the MSE as:

$$\begin{aligned} \mathrm{MSE}_i^{CRLB} &= \mathrm{E}\left[(s_i(t) - \hat{s}_i(t))^2\right] \\ &\geq \frac{N}{B \cdot T} \cdot \frac{\mathrm{Var}\left[s_i(t)\right]}{\mathrm{Var}\left[z_i(t)\right] \cdot \mathrm{Var}\left[s_i(t)\right]} \end{aligned} \tag{13.2}$$

Where:
N is the number of sources in the system
B is the inverse number of DFT frequencies
T is the number of samples used to estimate the system
$z_i(t)$ is defined in equation 13.3.

$$z_i(t) = \frac{\frac{d}{ds_i(t)} \mathrm{PDF}_{s_i}\left(s_i(t)\right)}{\mathrm{PDF}_{s_i}\left(s_i(t)\right)} \tag{13.3}$$

In order to make equation 13.2 comparable to the SIR method described in equation 13.1 it is rewritten as described by equations 13.4 to 13.6

$$\mathrm{SIR}_i^{CRLB} = \frac{\mathrm{E}\left[(s_i(t))^2\right]}{\mathrm{MSE}_i^{CRLB}} \tag{13.4}$$

$$\leq \frac{\mathrm{E}\left[(s_i(t))^2\right]}{\frac{N}{B \cdot T} \cdot \frac{\mathrm{Var}[s_i(t)]}{\mathrm{Var}[z_i(t)] \cdot \mathrm{Var}[s_i(t)]}} \tag{13.5}$$

$$\leq \frac{B \cdot T \cdot \mathrm{E}\left[(s_i(t))^2\right] \cdot \mathrm{Var}\left[z_i(t)\right]}{N} \tag{13.6}$$

All the variables in equation 13.6 are known except $z_i(t)$, in order to estimate it, the PDF of the sources signals needs to be known. In section 7 the source signals where created in MATLAB using the random function with a expential distribution. The MATLAB documentation describes that the exponential distribution has a PDF described by equation 13.7. As the PDF of the source signals have been established the derivative of the PDF can be found as equation 13.8. This makes it possible to determine $z_i(t)$ as equation 13.9

$$\mathrm{PDF}_{s_i}\left(s_i(t)\right) = \frac{1}{0.7} \cdot \exp\left\{\frac{s_i(t)}{0.7}\right\} \tag{13.7}$$

$$\frac{d}{ds_i(t)} \mathrm{PDF}_{s_i}\left(s_i(t)\right) = \frac{1}{0.49} \cdot \exp\left\{\frac{s_i(t)}{0.7}\right\} \tag{13.8}$$

$$z_i(t) = \frac{\frac{1}{0.49}}{\frac{1}{.7}} = \frac{1}{.7} \tag{13.9}$$

$$\mathrm{Var}\left[z_i(t)\right] = 0 \tag{13.10}$$

From equation 13.9 it becomes clear that because $z_i(t)$ is a constant the variance of it would be zero. This would affect equation 13.6 by making the upper bound infinitly large. It should be noted that the assumptions for this "upper" bound for the SIR only make the assumptions ,that the source signals are mutually independent and that they have the same distribution (PDF),

which also holds true for the assumptions made about the TITO system model. So the Cramer Rao upper bound does not give a reference point for how well the BSS method performes. Other publicized methods for doing BSS show improvement in the SIR from 3.1 dB up to 21 dB [5, 22] and the actual BSS method used in this project should show an improvement around 10 dB [8, 523] for the TITO model

Because of the reverse third moment spectrum the filter estimations can be circular shifted. Therefore the filters needs to be corrected in order to have the filter coefficient in a correct sequence. This problem is however rather difficult one to solve, but as the filters are known in the simulations, this is corrected manually and the shift is always constant when using the same filters. One possible solution to this problem would be to shift the filters until the cross correlation between the sources is minimized, but because of time constraints this solution has not been explored.

13.1 Simulation Description

This section contain the simulation that was described in the previous chapter. The actual implementation used can be found on the accompanying CD as */Matlab Code/SIR test/SIRtest1.m*. The simulation is performed in MATLAB as an off-line processing, so the data set as processed as a whole by the individual blocks like the $\bar{\bar{H}}(0)$ estimation, before the next block starts proccessing the data set. For the trispectrum estimation an optimum window is used, this restrict to some degree the amount of filter coefficient that can be estimated, but as the lags is set to supersede the number of filter coefficient with a factor of 3 this is not seen as a problem.

The simulation of the $\bar{\bar{H}}(0)$ estimation in section 12, showed up to 20 % deviation on this parameter, when using a dataset of 400.000 samples. Therefore the sensitivity of this paramter is the first thing to examine . A diviation of 0, .5, 1, 5, 10 , 20 % is applied as positive offset on h_{12} and as a negative offset on h_{21} and the resulting SIR is calculated. The same dataset is used for all offset SIR calculations in order to make the results comparable.

After the sensitivity estimation an BSS test which includes the $\bar{\bar{H}}(0)$ estimation is performed, this is repeated 10 times to see the results, as it is known that $\bar{\bar{H}}(0)$ estimation would change from one simulation to the next.

The estimated filters would have a length twice the size of any filters that can be estimated, because the size of spectrum would always be twice the size of the lags.This means that half of the filter coefficients can be removed before any scaling of the filter is performed. Also if the filter length is known only the exact number of filter coefficient from the estimated filter should be used in order to get a better scaling and demixing. All three methods are tested in the first test.

The steps in the two simulations are as follows:

1. Create test signals with 400000 samples as described in section 7, the filters used are

h_{12} = [1 .5 .2] and h_{21} = [.3 .8 .4]. When creating the test signals it is assured that the remain mean free, as this is the most likely scenario.

2. Estimated H(0), this is done by calculating it from the known filters or by using the MATLAB implementation from the previous test.

3. Estimate the filters moment spectra from 100 frames of 4000 samples using lags of ± 9 and an fftlength of 20

4. Calculated the filters coefficients using the reverse third order moment spectrum

5. Time shift the filters which is done manually. With the filters used only h_{21} needs to be shifted. The scaling is performed either on the whole filter length 20 coefficients, the largest filter (that can be estimated) with a length of 10 coefficients and the known filter with 3 coefficient. The scaling is performed from the estimated $H(0)$, by ensuring that the DC gain is that same as this estimated/ideal value.

6. The resulting 2 times 3 different filters are used for the demixing the signals, by using the inverse filtering.

7. The SIR is calculated for both signals and for the 3 different filters, as well as the SIR on the Mixed signal(the before SIR). Where s_1 is related to x_1 and s_2 is related to x_2 in order to document an improvement. When calculating the SIR the first half and last half frame from the inverse filtering should not be used, because of the overlapping frames.

13.2 Simulation Results

The simulation results from the sensitivity simulation for $\bar{\bar{H}}(0)$ can be seen in figure 13.1. From the first column and the second columns it is clear that the reasonable assumption, that the filterlength is the same as the number of lags used, doubles the SIR improvement. In general an improvement of around 12 dB can be achived up to an offset of 10 %. At an offset of 20 % the improvement is down to around 7 dB SIR. The last column in figure 13.1 is the SIR improvement if the filterlength is known to be three coefficients. Here the SIR improvements of up to 20 dB is achived. As most published method show improvements between 3.1 dB up to 21 dB, the BSS implementation seems to performe as it should. In the last two columns there seem to be a trend that the SIR improves if the offset is around 5 %, which was not expected. One likely canditate are that the multilinearity in the non-minimum filter estimation does not hold as the sources correlated with each other. Another candidate is that the trispectrum estimation is not good enough. In either cases another $\bar{\bar{H}}(0)$ estimation, then the correct one seems, to give better results.

The above test is repeated and the ideal H(0) estimator is replaced by the one implemented in the previous section. This test is run 10 times and estimated filters DC-coefficient for each run is displayed in figure 13.2 The resulting SIR for the corresponding runs can be seen in figure 13.3.

In general there would be a SIR around 10 dB improvement of the SIR of the filter length is assumed to be the number of lags examined. One thing that stands out is that the second

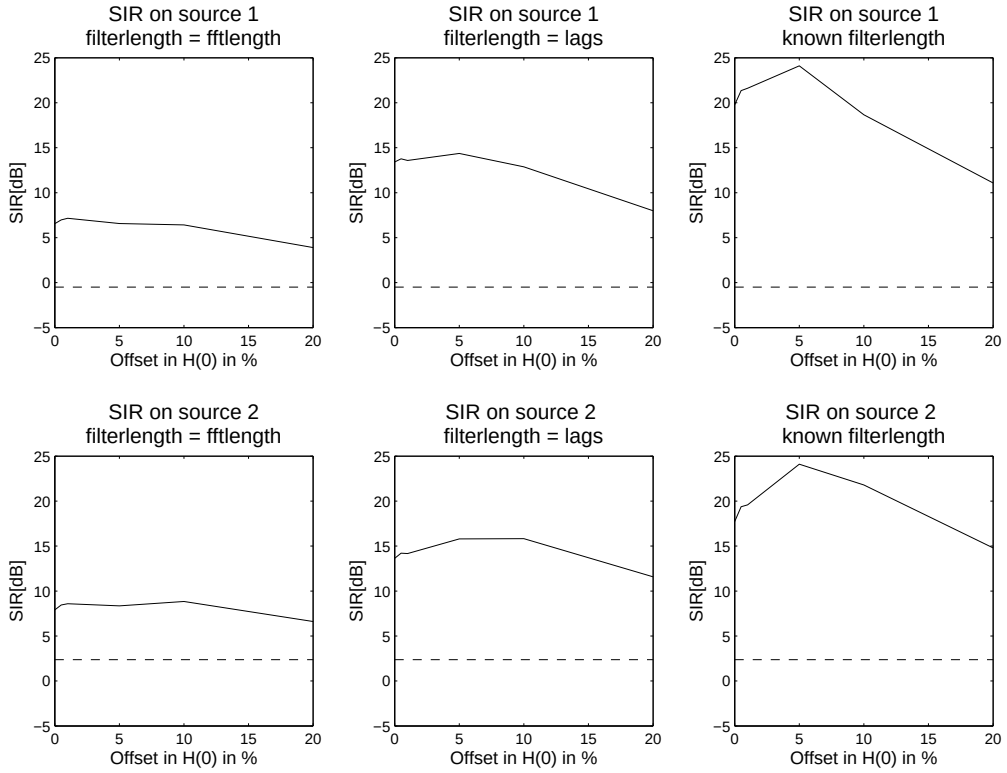


Figure 13.1: The solid line is the SIR as a function of the deviation of H(0) in percent. The dashed line indicates the SIR before the BSS

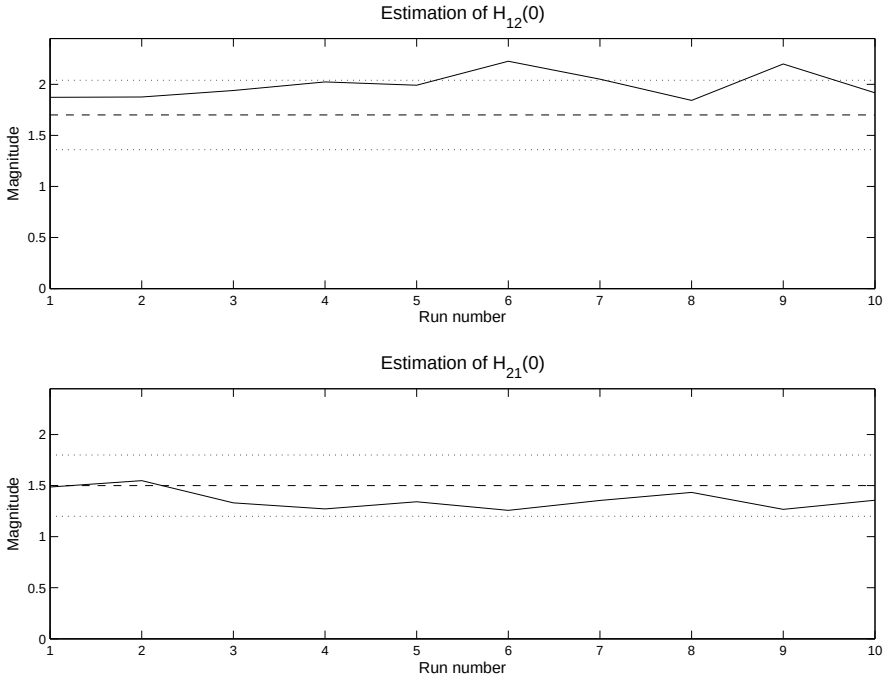


Figure 13.2: Estimation of the DC-coefficient for the 10 runs, the dashed line is the ideal filter coefficient, and the dotted lines represents 20 % offset.

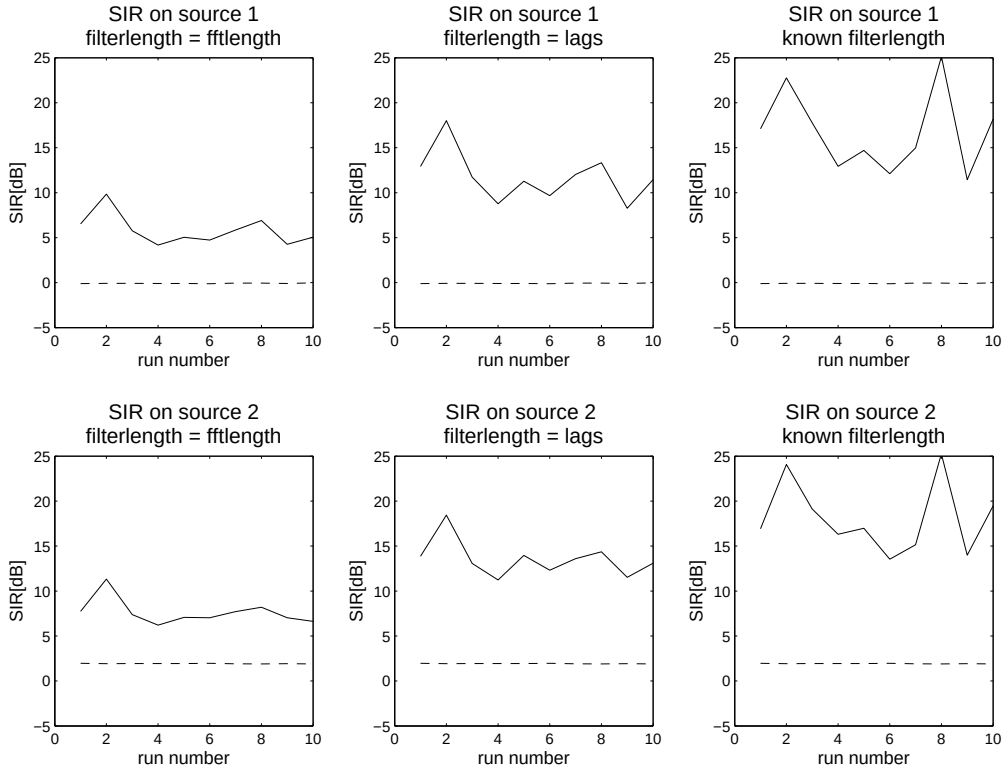


Figure 13.3: The solid line is the SIR at different runs. The dashed line indicates the SIR before the BSS was applied

run gives significant better results then the others. If this is correlated with figure 13.2, then a positive offset seems to improve the SIR. Again this is assumed to stem from the before mentioned candidates and therefore another $\bar{\bar{H}}(0)$ would give better results. If the first simulations is reapeted with the $\bar{\bar{H}}(0)$ with a positive offset for both filters, the resulting plots in figure 13.4.

This supports the theory that the trispectra is not entirely correct estimated, as a positive offset of 20 % in $\bar{\bar{H}}(0)$ improves the resulting SIR of up to 6 dB. This concludes the simulations of the BSS method, the method gave results that are comparable to other papers conserning BSS. One interresting thing that was observed was that the $\bar{\bar{H}}(0)$ estimation does not necessarily have to be the correct estimate as other values seems to give better results.

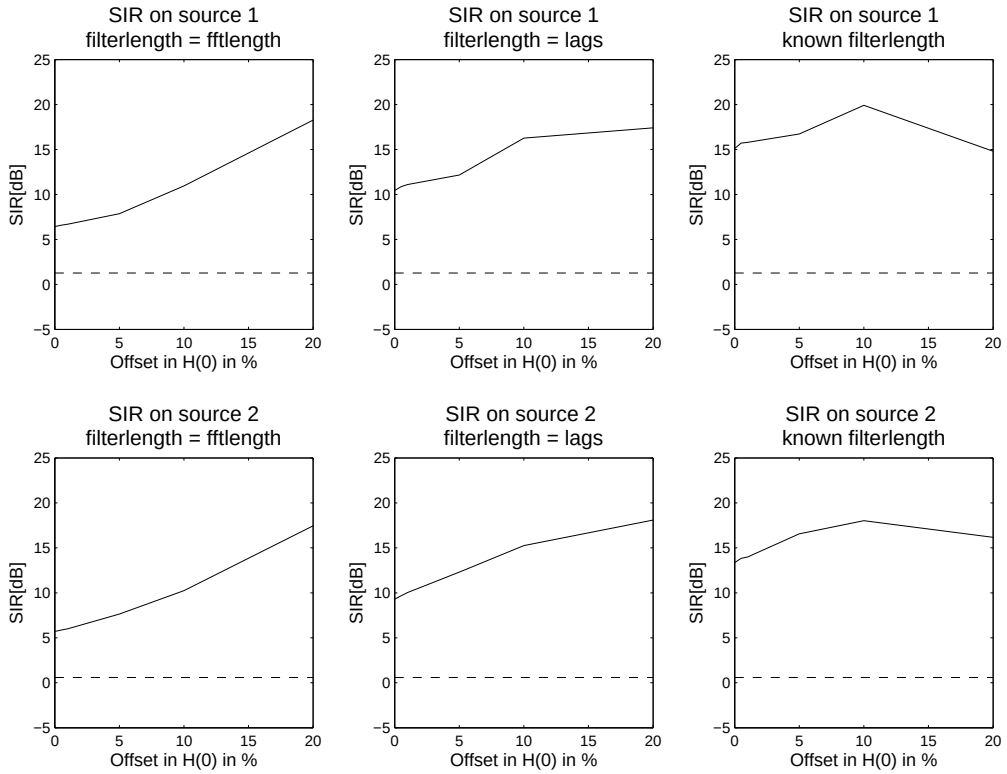


Figure 13.4: The solid line is the SIR as a function of the deviation of $\bar{\bar{H}}(0)$ in percent both filters have a positive offset in this plot. The dashed line indicates the SIR before the BSS

Chapter 14

Conclusion

In this part the BSS method was simulated in MATLAB. The BSS was divided into 4 parts that were simulated individually. Furthermore a simulation of the bispectrum estimation was also made to simulate the minimum phase filter estimation. While not used the purpose was to evaluate the direct method and the indirect method for estimation of cumulant spectra as well as create test vectors for the reverse third order moment spectra. The last section contained a simulation of the BSS method as a whole where the improvement in signal to interference ratio was calculated.

Chapter 8 was the inverse filtering, where the method discussed in chapter 3 was implemented and simulated. The simulations showed little to no error introduced by this method and the ability to handle non-minimum phase filters as well as minimum phase filters.

Chapter 9 was a simulation of the bispectrum and the minimum phase filter estimation. In the bispectrum estimation the direct method and the indirect method were presented and simulated. The conclusion on the two methods were: If the number of coefficients in the filter is short compared to the frame length the indirect method should be used. If on the other hand the number of filter coefficients are close to the frame length the direct method should be used. As the frames are relatively long compared to the filters the indirect method was used for estimation of the spectra in all the following simulations. The simulations of the minimum phase filter estimation showed it to be acceptable and the simulations of bispectrum estimation also showed that it was working to an acceptable degree.

Chapter 10 was the simulation of the method to find the reverse third order moment spectrum. The simulation showed that the method only introduced numerical noise. It should be noted that the error also includes the error from the bispectrum estimation and as such the method was found to operate well within acceptable limits. There was, however, a problem with the method as it can introduce scaling and a shift in the original signal. This is handled in the simulation by scaling and shifting by the largest value. However this presents a problem when using this method as the original signal is normally not known.

Chapter 11 was the simulation of the non-minimum phase filter estimation. In the simulation of the bispectra it was shown that the indirect method was the best for estimating of the

trispectrum for the simulations. The non-minimum filter estimation showed acceptable results especially if the filters length were known.

Chapter 12 was the simulation of the $H(0)$ estimation. Using the same dataset size as the rest of the simulations the simulations often showed deviations of up to 20 % and in some cases even more. If the results where averaged over 100 of these simulations the mean was within acceptable limits. However, this slow convergence rate will affect other parts of BSS method negatively and finding another method that converges faster would thus be a good choice.

Chapter 13 was the simulation of the entire BSS method. The simulation was divided into two parts; one to evaluate the sensitivity of the $H(0)$ parameter, because the simulation of the $H(0)$ estimation showed large deviations and one to simulate the entire BSS method including the $H(0)$ estimation. The simulation a general improvement in SIR of around 10-12 dB when the estimated filter order was three times higher than the correct filter order. The simulations also showed that making a positive offset on both $H(0)$ estimations, would improve the SIR by up to 5 dB. It was assumed that this is related to the trispectrum estimation not being good enough or the multilinearity conditions not holding. This would be worth looking further into, but was not possible within the time frame of this project.

This concludes the MATLAB simulation of the BSS method. Now that a working implementation in MATLAB has been made the next step is to implement it onto a hardware platform.

Part IV

Algorithm Implementation

Chapter 15

Introduction

This part concerns the implementation of the blind source separation method onto an architecture. The focus for this implementation is the execution speed as the feasibility of a real time implementation is investigated. As a reference application for this real time implementation a scenario with speech is used. The BSS method was presented in the previous parts and the system model of the BSS in figure 15.1 was also introduced.

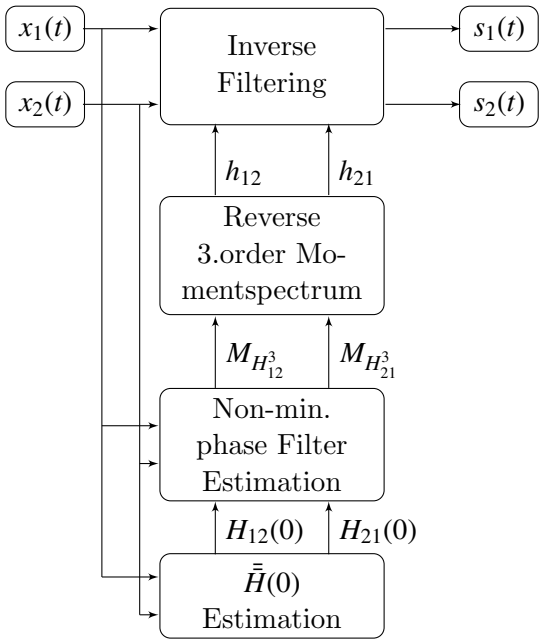


Figure 15.1: Illustration of the system model for solving the blind source separation.

Regarding execution speed there are two problematic areas in the BSS method:

The first is the execution speed of the inverse filtering. This is a hard real time problem as a datasets from x_1 and x_2 must be processed before the next datasets from x_1 and x_2 arrives. The second problem is the time it takes for the filters h_{12} and h_{21} to be estimated. The maximum allowed time for updating the filters is application dependant, but as a speech signals are used

as a reference application is used, it is possible to make an approximation.

Speech is used as the reference application when evaluating the real time potentials of BSS. In a scenario with different speakers, one would assume that they are not stationary sources, as they would probably be moving around. It is assumed that a filter update rate of 25 times per second is acceptable for such a scenario. In the simulations it was show that around 100 estimation of the filters where necessary to get a good estimate. This would translate into 4 seconds for the filters to be estimated from initialization of the BSS algorithm. However, if the source moves it as assumed that it would take less than 100 estimations to estimate the filter for this new location, as the old filter would provide a good enough starting point to handle the correction in a few iterations.

As the speech scenario is used as a reference application for the optimization, one important parameter is that the frame length should reflect the quasistationary properties of the speech signals. For speech and most audio applications this would give a frame length of around 20 milliseconds. Sampling the speech at 8 kHz, which is sufficient for speech signals, would give a frame size of around 160 samples. In order to get a power of two number, which scales better on most parallel implementations, this is lowered to 128 samples per frame.

The implementation starts with a complexity analysis of the inverse filtering and the different parts of the BSS method. This is done to determine which of these parts are the most complex and thus will the longest time to execute. After the identification of the most complex parts these are examined in detail and modified to lower the complexity. The last step is to implement the parts onto an architechture in order to get measure what the execution would be in a practical case.

The platform used is a Graphics Processing Unit (GPU) commonly found in PCs and normally used for accelerating graphical applications. The GPU used is from NVIDIA and supports the CUDA framework developed by NVIDIA. Some of the advantages of using this platform are:

- Easy to use as the CUDA framework is an extension to the commonly known C programming language.
- No or little management of the underlying hardware - this is managed through the operating system or the GPU itself.
- Offers tremendous computational power.
- Massively parallel architechture allowing large speedups if the algorithm is parallelizable.

The specific GPU used is the NVIDIA Geforce GTX 285 which is capable of around 1 TFLOPS. There is 1 GiB global memory available running at 1242 MHz DDR with a bus width of 512 bits. This results in a theoretical memory bandwidth of $2 \cdot 1.242 \cdot 10^9 \cdot 512/(1024^3 \cdot 8) = 148.06$ GiB/s.

Chapter 16

Complexity analysis

The analysis of the complexity is made on the MATLAB implementation used for the simulations. To measure the complexity small o-notation is used throughout this chapter. This is mainly used that the parameters for the reference application are known. For calculating the complexity the following constants are used for the o-notation in all the calculations.

n number of samples in a frame

f number of Fourier bins in the frequency domain

k number of lags used for the indirect method

L number of samples to smooth over for the direct method

A part from these constants the following simplifications are used when calculating the complexity.

- All arithmetic operations has a complexity of one operation, this includes logarithm, rounding operations, etc.
- Memory operations and calculating indexing operations are not included in the complexity calculations.
- Constant values that are not dependent on the e.g. the frame length are omitted, as they would appear as overhead and would not scale with the size of the input data.
- No assumptions is made about the operands in the arithmetic operations e.g. multiplying two complex numbers takes the same time as multiplying two real numbers.
- All Fourier transforms are treated as discreet Fourier transforms (DFT's), when calculating the complexity of this operation.
- Arithmetic functions that are independent from the input date e.g. Functions that does change the results with the input date, are removed as their complexity are only relevant for the initialization phase of the system and consequently have no subsequent impact on the execution time.

In order to compare the different parts in the BSS the complexity is only calculated for processing one frame of data. All the parts for the doing the non-minimum phase BSS is evaluated in the following sections. This means the inverse filtering, the reverse third order momentspectrum, the non-minimum phase estimation and the $\bar{\bar{H}}(0)$ estimation. This chapter also includes a complexity analysis of the bispectrum estimation as this complexity analysis is used to evaluate the difference in complexity for the direct and indirect method in a previous simulation chapter.

16.1 Complexity of Inverse Filtering

The complexity for the inverse filtering is based on the implementation present in section 8 on page 43. Each step in the implementation is evaluated for its complexity and these are added together to get an estimate of the total complexity for this part.

1. Dividing the samples into frames has a complicity of 2n memory moves, but is omitted because of it being memory moves.

2. The zero padding can be solved by indexing, if the time shift and the frame length remains constant, so the complexity is also zero for this step

3. The Fourier transform has a complexity of f^2 operations. If it is assumed that the number of Fourier bins is the same as the zero padded frame length.

4. Inverting the matrix has a complexity of 8 operations for inverting the matrix and this has to be repeated for all values of ω so the total complexity is $8 \cdot f$

5. The inverse Fourier transform has a complexity of f^2

6. Adding the overlap with the previous frame to the current depends on the amount of zero padding added so the number of operations is $f - n$

The total complexity for inverse filtering comes to: $2f^2 + 9f - n$ operations.

Please note that the frame length and the number of frequency bins are not the same length as number of frequency bins must be larger to allow for non-minimum filters as described in chapter 3. Assuming that doubling the frame length by zero padding allows for most non-minimum phase filters, the total complexity for doing the inverse filtering comes to: $4n^2 + 17n$ operations.

16.2 Complexity of the Bispectrum Estimation

This section evaluates the complexity of the direct method and indirect method for estimation of a bispectrum. This section is only used for the simulation part of the report as it is not part of the final BSS algorithm. A description of the MATLAB implementation can be found in section 9.1 on page 47.

16.2.1 Direct Method for calculating the bispectrum

1. The operations needed for removing the mean from the input vectors are n for the subtraction of the mean plus n operations for the calculations of the mean. So the total complexity is $3 \cdot 2n$ for all three input vectors.

2. The 1 dimensional DFT must be performed on the three input vectors in the worst case these are all different giving a complexity of $3 \cdot f^2$ for this step

3. The complexity for estimation of one point in the bispectrum is three multiplication and one complex conjugate giving a complexity of four operations, doing this for each point in the bispectrum gives an complexity of $3 \cdot f^2$ operations.

4. Smoothing the spectra by convolving with a window the equation (which is not listed in the simulations) for doing a 2 dimensional convolution is as follows:

$$C^S_{xyz}(\omega_1, \omega_2) = \sum_{\tau_a=-L}^{L} \sum_{\tau_b=-L}^{L} C_{xyz}(\omega_1, \omega_2) \cdot h_s(\omega_1 - \tau_a, \omega_2 - \tau_b). \tag{16.1}$$

   Where:
   h_s is a two dimensional window function with size $(2L + 1)$ times $(2L + 1)$. C^S_{xyz} is the smoothed cumulant spectra. The complexity for doing this smoothing operation is: $4(2L + 1)^2 \cdot f^2$ additions and $4(2L + 1)^2 \cdot f^2$ multiplications.

Note that the length of the Fourier transform must be at least the same length as the frame size so $f = n$. So the total complexity for the direct method comes to: $(6 + 8(2L + 1)^2)n^2 + 6n$ operations.

16.2.2 Indirect Method for calculating the bispectrum

1. The operations needed for removing the mean is n operations for the subtraction of the mean plus n for the calculations of the mean, which is performed on up to 3 input vectors. So the total complicity is $6n$ for the first step in procedure.

2. The operations needed for calculating the cumulants is $3n$ operations for calculating one specific cumulants, and this must repeated k times k. So the complexity is $3 \cdot k^2 \cdot n$

3. Applying the window to the cumulant sequence has a complexity of k^2 operations, the complexity for created the window is not added as this is a one time occurrence.

4. Zero padding this is more or less adding zeros to the array so that the size of the cumulants matches the wanted size of the Fourier transform. This is minor and is omitted from the final calculations. Because the initial array can be larger and the step be solved by indexing in the previous steps.

5. The 2 dimensional DFT, gives a complexity of k^2 for calculating a point in the DFT. This has to be repeated f^2 for the entire spectrum, giving a total complexity of $k^2 \cdot f^2$

Note that the Fourier transform does not need to be longer then k so $k \approx f$

The total complexity for the direct and the indirect method is:
Direct Method: $(6 + 8(2L + 1)^2)n^2 + 6n$
Indirect Method: $k^4 + k^2 + 3n \cdot k^2 + 6n$

The complexity is shown as a function k and n in figure 16.1 with a smoothing parameter of $L = 20$ in figure 16.2 the smoothing parameter is set to $L = 10$

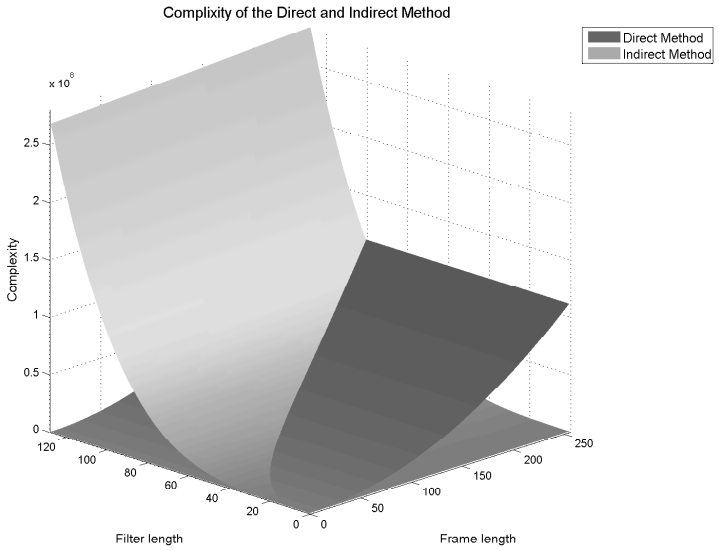


Figure 16.1: Complexity of the direct and indirect method for estimation of the bispectrum with a smoothing over ± 20 samples in the direct method.

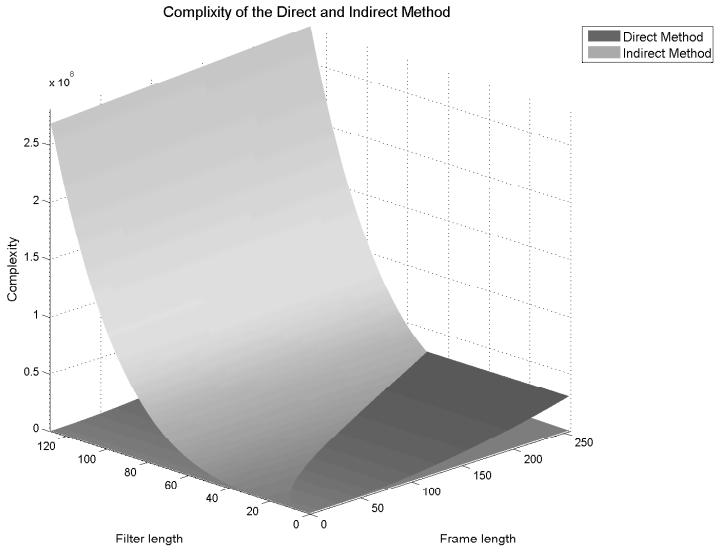


Figure 16.2: Complexity of the direct and indirect method for estimation of the bispectrum with a smoothing over ± 10 samples in the direct method.

From figure 16.1 and 16.2 it can be derived that if the number of filter coefficients k are small compared to the data length n. Then the indirect method is the best performing method. But if the number of filter coefficients are unknown, which would be the most likely case, then the direct method would be the best performing method. The smoothing factor should decrease when the frame length is small but it would not change the above observations about the complexity of the two methods.

16.3 Complexity of Reverse Third Order Momentspectrum

The implementation is presented in section 10 on page 59 the complexity is first calculated for the phase estimation and then for the magnitude estimation. This implementation contains many matrices that can be preallocated at the initialization of the system. if the frame length in does not change.

1. Construction of the matrix $\bar{\bar{A}}_\phi$ has a complexity of $(f-1)\cdot\left(\frac{f}{2}\right)^2$ for f being even, which it is assumed that it is. This matrix is not data dependent so it only needs to be constructed once, hence its complexity is zero.

2. Under the assumption that f is even. The construction of the $\bar{\phi}_{3h}$ vector consists of $\left(\frac{f}{2}\right)^2$ angle calculations.

3. Determining the phase ambiguity $\hat{k}$ has the following complexity.

- Constructing the matrix $\bar{\bar{G}}_\phi$ has a complexity of $(f-1)^2$, but as it is not data dependent its complexity is zero.
- Constructing the matrix $\bar{\bar{F}}_\phi$ has a complexity of $(f-1)^2$, but as it is not data dependent its complexity is zero.
- Calculating the matrix $\bar{\bar{D}}_\phi$ has complexity of a matrix inversion, a matrix matrix multiplication and a zero padding. But as there is no data dependency either it is only initialization complixity, so the complexity is zero.
- Estimate of $\hat{k}$ is done using equation 16.2.

$$\hat{\bar{k}} = \text{round}\left(\frac{\bar{\bar{A}}_\phi \cdot \bar{\bar{D}} \cdot \bar{\phi}_{3h} - \bar{\phi}_{3h}}{2 \cdot \pi}\right) \tag{16.2}$$

The initial thought was to reduce the data depend factors in the equation. But the $\bar{\bar{D}}_\phi$ matrix is zero padded, therefore it could be assumed that the matrix from the $\bar{\bar{A}}_\phi \bar{\bar{D}}$ multiplication does not have full rang. The illustration in figure 16.3 shows the presence of non zero elements in this matrix.

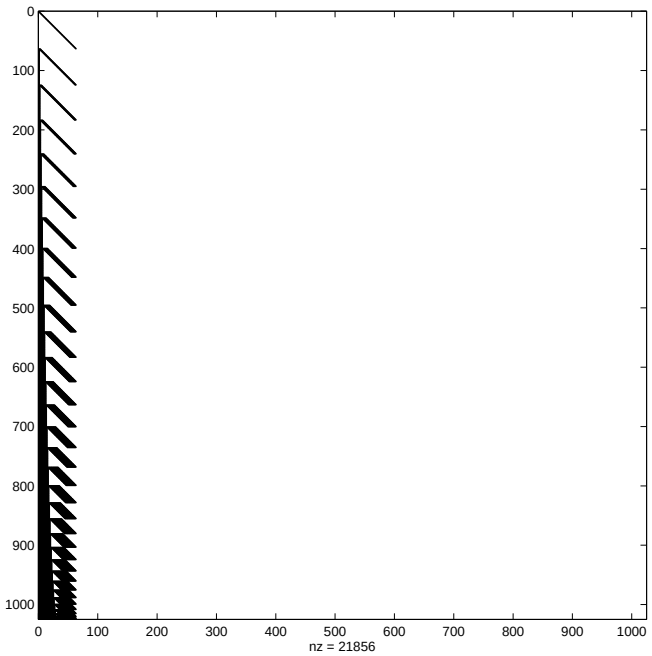


Figure 16.3: Illustration from the result of Matlabs spy function on the $\bar{\bar{A}}_\phi \cdot \bar{\bar{D}}$ matrix, where only non-zero elements are depicted and $f = 64$.

From figure 16.3 the matrix only contains elements within the $\frac{f^2}{2}$ x f region. So reducing this matrix with this size and also reducing the $\bar{\phi}_{3h}$ vector gives a rather large reduction in complexity and doing the subsequent subtraction is only a complexity of $\frac{n^2}{2}$ subtraction operations.

The complexity for the estimation of the phase ambiguity $\bar{k}$ comes to $\frac{f^2}{2} \cdot f$ operations for the $\bar{\bar{A}}_\phi \cdot \bar{\bar{D}}$ and $\bar{\phi}_{3h}$ matrix vector multiplication and $3 \cdot \frac{f^2}{2}$ operations for the subtraction, multiplication with $\frac{1}{2 \cdot \pi}$ and the round operation. Giving a total of $f^3 + \frac{3}{2} f^2$ operations

Total complexity of estimating $\hat{k}$ is : $f^3 + \frac{3}{2}f^2$

4. The last step in the phase estimation is equation 10.3. The equation consist of a matrix transposed, a matrix matrix multiplication, a matrix inversion. a matrix matrix multiplication, matrix vector multiplication and a vector vector additions and scaling.

   - There is no complexity for transposing the matrix $\bar{\bar{A}}_\phi$, this can be done by swapping rows with columns in the indexing. And the matrix is not data dependent, so the complexity is zero.
   - The first matrix multiplication ($\bar{\bar{A}}_\phi^T \bar{\bar{A}}_\phi$) has a complexity of $(f-1)^2 \cdot \left(\frac{f}{2}\right)^2$ multiplications and additions. But it is also not data dependent so the complexity is zero.
   - The inversion of the resulting matrix ($\left(\bar{\bar{A}}_\phi^T \bar{\bar{A}}_\phi\right)^{-1}$)gives $(f-1)^4$ [3, p. 295] arithmetic operations. But as the matrix that is being inverted is not data dependent the complexity is zero.
   - The following matrix matrix multiplication ($\left(\bar{\bar{A}}_\phi^T \bar{\bar{A}}_\phi\right)^{-1} \bar{\bar{A}}_\phi^T$) is also not data dependent so the complexity is zero.
   - Resolving the phase ambiguity of $\bar{\phi}_{3h}$ ($\bar{\phi}_{3h} + 2\pi \cdot \hat{\bar{k}}$) has a complexity of $\left(\frac{f}{2}\right)^2$ multiplications and additions.
   - The vector matrix multiplication ($\left(\bar{\bar{A}}_\phi^T \bar{\bar{A}}_\phi\right)^{-1}$) $\cdot$ ($\bar{\phi}_{3h} + 2\pi \cdot \hat{\bar{k}}$) has a complexity of $2 \cdot (f-1) \cdot \left(\frac{f}{2}\right)^2$ operations.

   That gives a total complexity of: $\frac{1}{2}f^3$ operations

The total complexity for estimating the phase comes to: $\frac{1}{4}f^2 + f^3 + \frac{3}{2}f^2 + \frac{1}{2}f^3$

Which reduces to: $\frac{3}{2}f^3 + \frac{7}{4}f^2$ operations

Now that the complexity for estimating the phase has been established the complexity for each step in determining the magnitude must be determined as well. The procedure for doing the pruning to handle negative overflow is done a little different, as the one used in the MATLAB implementation is not optimal with regards to number of operations used. Instead of removing values with negative overflow, they are set to zero and the matrix $\bar{\bar{A}}_\mu$ is not resized nor changed. The control vector still needs to be created the keep track of values that where set to zero.

1. Construct the matrix $\bar{\bar{A}}_\mu$ has a complexity of $f \cdot (\frac{1}{4}f^2 + \frac{1}{2}f)$. But as this is not data dependent the complexity becomes zero for this step.
2. Construct the magnitude vector $\bar{\mu}_{3h}$ by taking the logarithm to the values in the bispectra has a complexity ($\frac{1}{4}f^2 + \frac{1}{2}f$) operations, if f is even.
3. The next step is to find values in $\bar{\mu}_{3h}$ with negative overflow and set them to zero and at the same time create a control vector $\bar{y}y$ that contains a one at the indexes in $\bar{\mu}_{3h}$, where negative overflow occurred and zeros at all other indexes. This step differs from the previous implemented version in MATLAB and complexity for this step comes to $2 \cdot (\frac{1}{4}f^2 + \frac{1}{2}f)$ operations.

4. Pruning of the matrix $\bar{\bar{A}}_\mu$ is no longer necessary, so this step has a complexity of zero.
5. Calculating the magnitude response in equation 10.4 on page 60. The three matrices used on this operation does not change and are not data dependent anymore, so this step only contains a matrix vector multiplication. Which has a complexity of $f \cdot (\frac{1}{4}f^2 + \frac{1}{2}f)$ multiplications and additions operations. So total complexity comes to: $\frac{1}{2}f^3 + f^2$)
6. Last step is to find values in the output vector that should be set to negative overflow. This is done by multiplying the matrix $\bar{\bar{A}}_\mu$ with the control vector $\bar{y}y$ and all the indexes in the output vector where the value differs from zero, the corresponding index in the output vector should be set to negative overflow. This step has a complexity of $2 \cdot f \cdot (\frac{1}{4}f^2 + \frac{1}{2}f)$ operations for the matrix vector operation and f operations for setting negative overflow in the output vector. This last step is only necessary if there is negative overflow in the input vector $\bar{\mu}_{3h}$.

The total complexity for the estimation of the magnitude comes to two different scenarios:
If there is no entries in the $\bar{\mu}_{3h}$ vector with negative overflow the complexity comes to: $\frac{1}{2}f^3 + \frac{7}{4}f^2 + \frac{3}{2}f$ operations
In the other case when negative overflow occurs at some random entries then the complexity comes to: $f^3 + \frac{11}{4}f^2 + \frac{3}{2}f$. But as it is never know when negative overflow could occur the last complexity is assumed to be the correct one for the magnitude estimation. The complexity for combining the phase estimation and the magnitude estimation and performing the inverse Fourier transform to obtain the filter coefficients has a complexity of $3f + f^2$ operations.

Now that the complexity for both the phase and the magnitude has been estimated as well as the complexity for obtaining the filter coefficients the total complexity comes to: $\frac{5}{2}f^3 + \frac{9}{2}f^2 + 2f$ operations

This is based on the most cautious approach which handles negative overflow for all input vectors in the magnitude estimation. The reverse third order momentspectrum needs to performed on both filters, and it is know the fftlength could be the double the length of the frames size giving the following complexity for the reverse third order momentspectrum: $20n^3 + 18n^2 + 4n$

16.4 Complexity of Non-minimum Phase Filter Estimation

The largest complicity in the non-minimum phase filter estimation is located in the estimation of the eight trispectra's. There is two different methods to calculated the trispectra as there also where for the bispectrum, so both methods are going to be evaluated before the total complexity of the algorithm is calculated.

16.4.1 Complexity for Calculating the Trispectrum

Both methods are evaluated for complexity before a decision is made which one to use in the BSS. **Direct Method for calculating the Trispectrum**

The complexity for each step in estimating the trispectrum is listed in this subsection.

1. The operations needed for removing the mean from the four input vectors is n for the subtraction of the mean plus n for the calculations of the mean. So the total complexity is $8n$ for the first step in procedure.
2. The 1 dimensional DFT must be performed on the input vectors in the worst case these are all different giving a complexity of $4 \cdot f^2$ for this step
3. The complexity for estimation of a point in the fourth order moment spectrum is 4 multiplication and one complex conjugate giving a complexity of 5, doing this for each point in the bispectrum gives an complexity of $5 \cdot f^3$ operations.
4. The moment to the cumulant transformation is from equation 11.3. Made up of three cross power spectras $M_{xy}(\omega_1)$. convolved with a matrix $\mathscr{F}[m_{zw}(\tau_3 - \tau_2)]$. Creating the cross power spectras has a complexity of f operations. Creating the moment matrix $m_{zw}(\tau_3 - \tau_2)$ has a complexity of $f^2 \cdot n$ operations. The two dimensional DFT has a complexity of f^4 operations. The convolution with the matrix is not necessary as the cross power spectra only contains one value in the dimensions, so the matrix is scaled. These operations must be performed three times. The last step is to do the spectral subtraction in equation 11.8 this operation has a complexity of 6 operations performed f^3 times. The total complexity can for this step be determined as: $3 \cdot (n + f + f^2 + f^4) + 6f^3$
5. Smoothing the spectra by convolving with a window the equation (which is not listed in the simulations) for doing a three dimensional convolution is as follows:

$$C^S_{xyzw}(\omega_1, \omega_2, \omega_3) = \sum_{\tau_a=-L}^{L} \sum_{\tau_b=-L}^{L} \sum_{\tau_c=-L}^{L} C_{xyz}(\omega_1, \omega_2, \omega_3) \cdot h_s(\omega_1 - \tau_a, \omega_2 - \tau_b, \omega_3 - \tau_c) \quad (16.3)$$

   Where:
   h_s is a three dimensional window function with size $(2L + 1)$ times $(2L + 1)$. C^S_{xyzw} is the smoothed cumulant spectra. The complexity for doing this smoothing operation is: $(2L + 1)^3 \cdot f^3$ additions and $(2L + 1)^3 \cdot f^3$ multiplications.

Total complexity for direct method comes to: $3f^4 + 11f^3 + 2f^3(2L+1)^3 + 7f^2 + 3f + 11n$. Because of the cumulant sequence calculations in step 5 the relations becomes $f = 2n$. so the complexity is $48n^4 + 88n^3 + 16n^3(2L + 1)^3 + 28n^2 + 14n$

16.4.2 Indirect Method for calculating the Trispectrum

The complexity for each step in estimating the trispectrum listed in this subsection.

1. Removing the mean from four input vectors are n operations for the subtraction of the mean and n operations for the calculations of the mean. So the total complexity is $8n$ for the first step in procedure.
2. Calculating the cumulants is n operations for calculating one specific cumulants, where 4 multiply operations are performed, and this must repeated k times k times k. So the complexity is $4 \cdot k^3 \cdot n$

3. Applying the window to the cumulants, this step has a complexity of k^3 with one multiply operation, the complexity for constructing the window is not added as this is a one time occurrence.

4. Zero padding this is more or less adding zeros to the array so that the size of the cumulants matches the wanted size Fourier transform. This is minor and is omitted from the final calculations, and also this step can be omitted from the algorithm by making the initial array larger and doing correct indexing in the previous steps.

5. The moment spectrum to the cumulant spectrum transformation, for this up to 6 second order moment spectrum's needs to be calculated. The complexity for this is operation is $k \cdot n$. The equation in 11.1 gives an complexity of 6 operations performed k^3 times.

6. The three dimensional DFT, gives a complexity of k^3 for calculating a point in the DFT. This has to be repeated f^3 times for the entire spectrum, giving a total complexity of $k^3 \cdot f^3$

As it is know that the length Fourier transform would around the same length of the lags the following holds true $k = f$. From this the total complexity for doing the indirect method comes to: $k^6 + 4n \cdot k^3 + 7k^3 + 6kn + 8n$

Comparing the two methods the complexity comes to:
Direct method: $48n^4 + (88 + 16L^3)n^3 + 28n^2 + 17n$
Indirect method: $k^6 + (4n + 7)k^3 + 6kn + 8n$

The complexity is plotted as a function of k and n where L is set to ± 20 in figure 16.4 and L is set to ± 10 in figure 16.5.

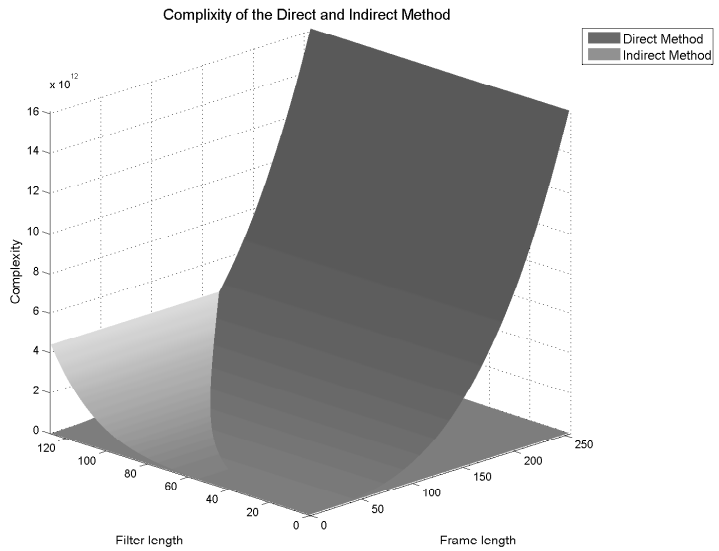


Figure 16.4: Complexity of the direct and indirect method for estimation of the bispectrum where the smoothing L in the direct method is set to ± 20

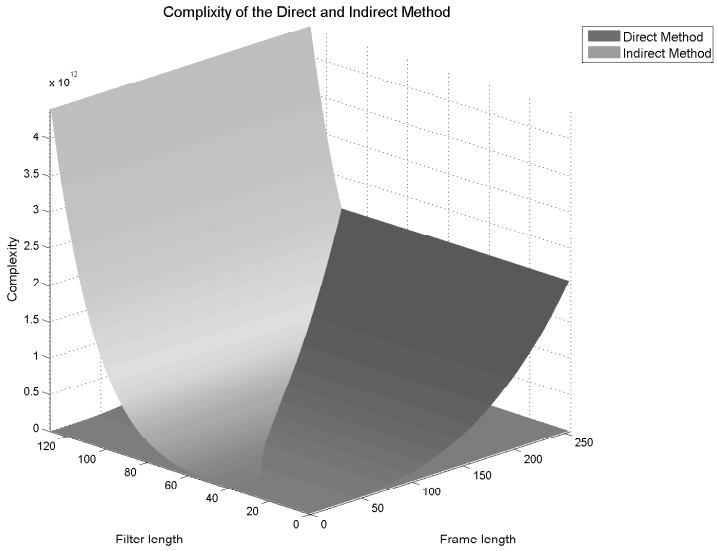


Figure 16.5: Complexity of the direct and indirect method for estimation of the bispectrum where the smoothing L in the direct method is set to ±10

The amount of smoothing performed in the direct method has a large impact on the complexity. If the smoothing parameter is set to ±20, it would only make sense to use the direct method if the frame length and the assumed size of the filter k are very close. At a smoothing parameter of ±10 this also holds true, though the filter length goes down to half the length of a frame. These observations does not take into account how precise the estimations are. The simulations results however gave the same results long filter compared to the frame the direct method is the best, short filter the indirect method is the best. So in order to make a general model where no assumption is made about a short filter the direct method is preferred over the indirect method.

The final step in the non-minimum phase filter estimation is to calculate the third order moment spectra for the two filters as described by equation 11.12 and 11.13 om page 67. Which consists of 2 subtractions 2 multiplications and one division for each entry in the bispectrum and for each estimation 4 trispectrum estimation are needed the complexity for the non-minimum phase estimation comes to:
$2 \cdot 5 \cdot (2n)^2 + 8 \cdot (48n^4 + (88 + 16(2L + 1)^3)n^3 + 28n^2 + 17n)$ Which can be reduced to :
$384 \cdot n^4 + (704 + 128(2L + 1)^3)n^3 + 264 \cdot n^2 + 136 \cdot n$

In the non-minimum phase filter estimation all the complexity lies in estimating the eight trispectrum's, so this should be the primary focus if this part are to be optimized.

16.5 Complexity for Calculating H(0) estimation

The implementation for estimating $\bar{\bar{H}}(0)$ can be found in section 12 on page 69. The implementation averaged over 100 frames. This is omitted from the complexity evaluation in order to compare it to the other parts, so only one frame is used for estimating $\bar{\bar{H}}(0)$. The complexity for

each step is as follows

1. The complexity for creating the $\bar{\bar{C}}_{x^2}(0)$ matrix is $2n^2$ for the two Fourier transforms and 4 multiplications $4 + 2n^2$ the 4 is omitted as it is a constant.
2. Making an eigenvalue decomposition has a complexity of $\frac{4}{3}n^3 + n^2$ operations [2, p. 89]. where n is the number of rows and columns in matrix, as this does not change, the complexity is omitted.
3. Creating the Transformation matrix $\bar{\bar{T}}$ is also of constant size so it is also omitted
4. Creating $\bar{y}(t)$ from $\bar{\bar{T}}$ and $\bar{x}(t)$ has a complexity of 2 additions and 4 multiplication for each entry in the frame so the complexity comes to $6n$
5. Calculate the averaged trispectrum matrix, this results in the need for 8 trispectrum matrices to be created, which each has a complexity (for the direct method) of $8 \cdot (48n^4 + (88 + 16(2L + 1)^3)n^3 + 28n^2 + 17n)$ operations which was established earlier. But as only one point is needed in the trispectrum it could be a better choice to use the indirect method here as k would be rather small compared with the frame length. However this needs to be investigated further.
6. Making an eigenvalue composition of $\hat{\bar{\bar{C}}}y(0, 0, 0)$ has a constant complexity and is omitted
7. Calculating the scaling of $\bar{G}_1(0)$ and $\bar{G}_2(0)$ is also a constant and is omitted
8. Calculating the $\bar{\bar{H}}(0)$ matrix is also omitted

The total complexity for estimating $\bar{\bar{H}}(0)$ comes to $384n^4 + (704 + 128(2L + 1)^3)n^3 + 226n^2 + 142n$ all the complexity lies in the eight trispectrum estimation which are performed in order to make the averaged trispectrum matrix. Optimizing the trispectrum estimation would also improve this part significantly.

16.6 Complexity of the Blind Source Separation

Now that the complexity for all the parts in the BSS algorithm have been examined the most complex part can be identified. The complexity for the individual parts found in the previous sections are:

Inverse filtering	$4n^2 + 17n$
Reverse third order moment spectrum	$20n^3 + 18n^2 + 4n$
Non-minimum phase filter estimation	$384n^4 + (704 + 128(2L + 1)^3)n^3 + 264n^2 + 136n$
$\bar{\bar{H}}(0)$ estimation	$384n^4 + (704 + 128(2L + 1)^3)n^3 + 226n^2 + 142n$

As mentioned in chapter 15 the target application for the BSS is a scenario with speech signals. The inverse filtering is the only part with a hard real time exection demand. With a frame length of 128 samples, a sample rate of 8 kHz and a frame overlap of 50 % there are 125 frames

per second. This gives $8.5 \cdot 10^6$ arithmetic operations per second for doing the inverse filtering.

For the filter update of the BSS method, it was assumed that a filter update rate of 25 times per second would be acceptable. When a frame length of 128 samples is used and a smoothing parameter L is set to ± 10 the number of arithmetic operations amounts to $5.2 \cdot 10^{12}$ per update or $130 \cdot 10^{12}$ operations per second with 25 updates per second. Comparing the inverse filtering it is easy to see that the complexity of this is insignificant to the rest of the BSS method. The target platform used has a maximum processing capability of $1 \cdot 10^{12}$ floating points per seconds (FLOPS). If it is possible to fully utilize the GPU and the BSS algorithm is not modified, a speedup of at least 130 times would be needed to be able to run the BSS in real time on this platform. This is without taking into consideration the simplifications made to the complexity calculations meaning that a practical number will be even larger larger.

The most complex parts are the non-minimum phase filter estimation and the $\bar{\bar{H}}(0)$ estimation. In both methods most of the complexity comes from the trispectra estimates. If the trispectra estimates are removed from the above complexity for the BSS method, the resulting complexity is stated in the following table:

Reverse third order moment spectrum	$20n^3+18n^2+ 4n$
Non-minimum phase filter estimation	$40n^2$
$\bar{\bar{H}}(0)$ estimation	$2n^2+ 6n$
Total complexity	$20n^3+60n^2+10n$

The complexity of BSS method without the trispectra estimates is around $1.1 \cdot 10^9$ arithmetic operations per second using the target application. If the same assumptions about the platform are made it would be possible to update the filters 900 times faster then it needs to. Because of this it is obvious that the best course of action is to focus on reducing the complexity of the trispectrum estimation, in order to evaluate if real time execution can be made possible.

As mentioned, the trispectra estimates are used in the non-minimum phase filter estimation and the $\bar{\bar{H}}(0)$ estimation. Both need eight trispectra estimates so the complexity in both are the same. However, as only the center slices of the trispectra are used for the non-minimum phase filter estimation, there is already a potential complexity reduction in the trispectra estimates. Consequently it has been chosen to focus on reducing the complexity in the computation of the center slice of the trispectra and subsequently implement the reduced version.

Chapter 17

Complexity reductions

From the complexity calculation in section 16, it was shown that the biggest contribution of arithmetic complexity was from the trispectrum estimation. Due to this it was chosen to examine the trispectrum estimation to see if it is possible to reduce this complexity. In the non-minimum phase filter estimation only the center slice of the trispectrum is used. This in it self gives rise to reduction in complexity and also only four unique trispectra are used in the estimation, which was not accounted for in the previous section.

Due to the smoothing of the trispectrum it is, however, still necessary to compute both the center slice and as many neighbouring slices as required for the smoothing window. Figure 17.1 illustrates this process for estimating the center slice in the trispectrum.

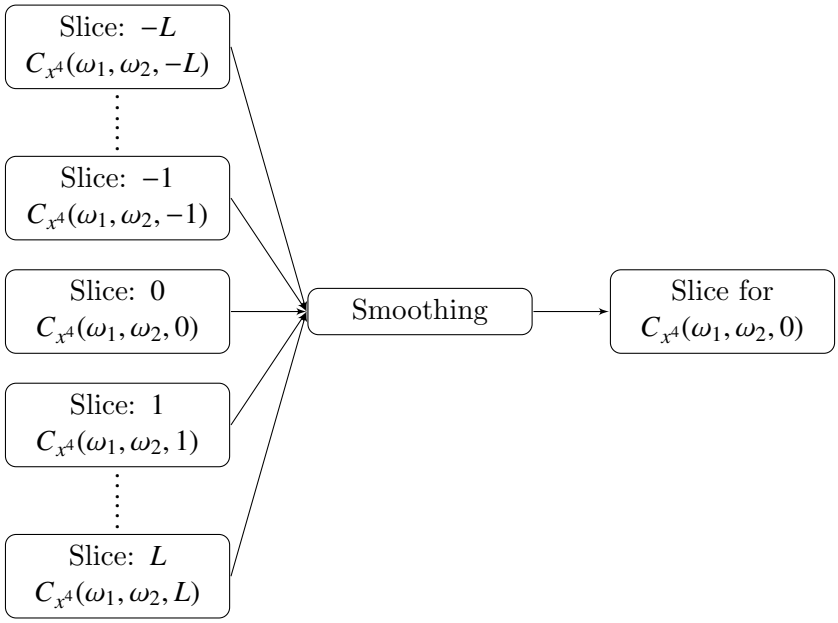


Figure 17.1: Computation of the center trispectrum slice used for the non-minimum phase filter estimation. L is the number of neighbouring slices computed.

Regardless of the value of L the center slice always needs to be computed. As such it is chosen

to examine if it is possible to optimized the computation for this slice without the smoothing of the neighbouring slices.

17.1 Project specific preconditions

In the simulation and the complexity analaysis of the BSS method no other assumptions are made about trispectrum estimation, other than the the mean value of the input signals must be zero to be able to compute the cumulants. For the implementation, however, it is possible to introduce a number of simplifications due to the way the trispectra estimates are used in the BSS method. By examining only the center slice it is also possible to introduce a simplification. These preconditions are listed below:

1. $\omega_3 = 0$ due to examining only the center slice.
2. The input signals are zero-mean or their mean values are subtracted to make them zero-mean.
3. The input signals are speech signals and thus real-valued.
4. For the general case $x(t) \neq y(t) \neq z(t) \neq w(t)$, but due to the way the trispectrum estimates are used $x(t) = y(t) = z(t) \neq w(t)$.

17.2 Block diagram of the trispectrum estimation

The MATLAB implementation to estimate the trispectrum can be divided into a number of blocks as shown in figure 17.2.

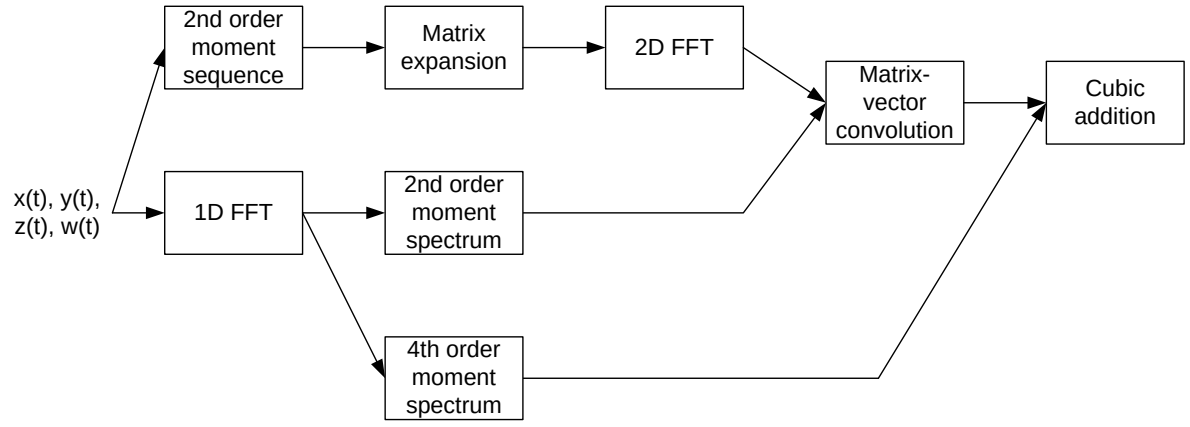


Figure 17.2: Block diagram of the MATLAB implementation to estimate the trispectrum.

As a consequence of the preconditions the calculations in each of the blocks can be reduced. The reductions can not always be confined to one block at a time, as reductions in one block may affect another block. As such the blocks will not be presented in chronological order, but

in the order that eases understanding for the reader.

As given in the preconditions only the autotrispectrum and cross-trispectrum with $x(t) = y(t) = z(t) \neq w(t)$ will be estimated. These will be referred to as the autospectrum and cross-spectrum in the following. Unless otherwise stated the reductions apply to both autospectrum and cross-spectrum calculations.

17.3 MATLAB variables

Symbol	Variable	Description
τ	**T**	Lag.
ω	**W**	Frequency.
$x(t)$	$\mathbf{x}(\cdot)$	Input signal $x(t)$ in the time domain. Similar variable name for $y(t)$ etc.
$X(\omega)$	$\mathbf{X}(\cdot)$	Input signal $x(t)$ in the frequency domain. Similar variable name for $Y(\omega)$ etc.
N	**N**	Length of the input signals in samples.
-	**fftlength**	Length of the Fourier transform output.
$m(\tau)$	$\mathbf{m}(\cdot)$	Moment sequence. The symbol is subscripted with the signal names used to calculate the sequence. For instance $m_{xy}(t)$ for the second order cross-moment sequence or $m_{x^4}(t)$ for the fourth order automoment sequence. The corresponding variable names are suffixed by the order and signal names, for instance **m2xy**.
$c(\tau)$	$\mathbf{c}(\cdot)$	Cumulant sequence. The symbol and variable names are subscripted and suffixed in the same manner as the moment sequence.
$M(\omega)$	$\mathbf{M}(\cdot)$	Moment spectrum. The symbol and variable names are subscripted and suffixed in the same manner as the moment sequence.
$C(\omega)$	$\mathbf{C}(\cdot)$	Cumulant spectrum. The symbol and variable names are subscripted and suffixed in the same manner as the moment sequence.
g	**g**	Matrix expansion of 2nd order moment sequence. The symbol and variable names are subscripted and suffixed in the same manner as the moment sequence.
G	**G**	Fourier transform of the matrix expansion. The symbol and variable names are subscripted and suffixed in the same manner as the moment sequence.

17.4 Fourth order moment spectrum

The 4th order moment cross-spectrum with arbitrary input signals is given by:

$$M_{xyzw}(\omega_1, \omega_2, \omega_3) = X^*(\omega_1 + \omega_2 + \omega_3) \cdot Y(\omega_1) \cdot Z(\omega_2) \cdot W(\omega_3)$$

The corresponding MATLAB code:

```
for W1 = 0:fftlength-1
    for W2 = 0:fftlength-1
        for W3 = 0:fftlength-1
            M4xyzw(W1+1,W2+1,W3+1) = conj(X(W1+W2+W3+1)) * Y(W1+1) * Z(W2+1) * W(W3
                +1)/N;
        end
    end
end
```

Using precondition 1 the innermost loop is unnecessary and **W3** can be set equal to 0 instead. This means that only the first value of $\mathbf{W}(\cdot)$ will be indexed, i.e. the DC component of $W(\omega)$. Because of precondition 2 the DC component will always be 0 and as this is multiplied to all values in the slice, the entire slice will be 0. This means the block can be removed from the implementation, however this would not be valid for any other slice.

17.5 Matrix-vector convolution

The matrix-vector convolutions are given by 11.8:

$$M_{xy}(\omega_1) * \mathscr{F}\left[m_{zw}(\tau_3 - \tau_2)\right]$$
$$M_{xz}(\omega_2) * \mathscr{F}\left[m_{yw}(\tau_3 - \tau_1)\right]$$
$$M_{xw}(\omega_3) * \mathscr{F}\left[m_{yz}(\tau_2 - \tau_1)\right]$$

The corresponding MATLAB code:

```
for W1 = 0:fftlength-1
    for W2 = 0:fftlength-1
        for W3 = 0:fftlength-1
            Gxyzw(W1+1,W2+1,W3+1) = M2xy(W1+1) * Gzw(W2+1,W3+1);
            Gxzyw(W1+1,W2+1,W3+1) = M2xz(W2+1) * Gyw(W1+1,W3+1);
            Gxwyz(W1+1,W2+1,W3+1) = M2xw(W3+1) * Gyz(W1+1,W2+1);
        end
    end
end
```

Due to the way the convolution is performed the output is a 3-dimensional array. However, using the same arguments as in the 4th order moment spectrum block the innermost loop can be removed and the output will be reduced to one slice of the array. Using the same arguments it is also possible to show $\mathbf{Gxwyz}(\cdot) = 0$ for all values of **W1** and **W2** and can thus be removed. Furthermore it is worth noting that only the first column of the matrices $\mathbf{Gzw}(\cdot)$ and $\mathbf{Gyw}(\cdot)$ will be accessed when $\mathbf{W3} = 0$ and $\mathbf{M2xy}(\cdot) = \mathbf{M2xz}(\cdot)$ due to precondition 4, $x(t) = y(t) = z(t)$.

Inspecting the definitions of $\mathbf{Gzw}(\cdot)$ and $\mathbf{Gyw}(\cdot)$ and remembering $y(t) = z(t)$ it is seen that $\mathbf{Gzw}(\cdot)$ and $\mathbf{Gyw}(\cdot)$ are the Fourier transforms of the same 2nd order moment sequence expanded into a matrix. As a result of this $\mathbf{Gzw}(\cdot) = \mathbf{Gyw}(\cdot)$ and only one of them needs to be calculated.

For the trispectrum there are several symmetry regions which is also the case for the slices. Although the output of the matrix-vector convolution block is not the trispectrum the symmetry regions still make it possible to prune a number of the values in the slice.

In the cross-spectrum four symmetry regions exist and two of them are mirrored around the diagonal as shown in figure 17.3 (b). This makes it possible to calculate only the upper or lower triangle of the matrix. If the first row and column of the matrix are ignored the remaining values are conjugated and mirrored along the anti diagonal as shown in figure 17.3 (b). Using these

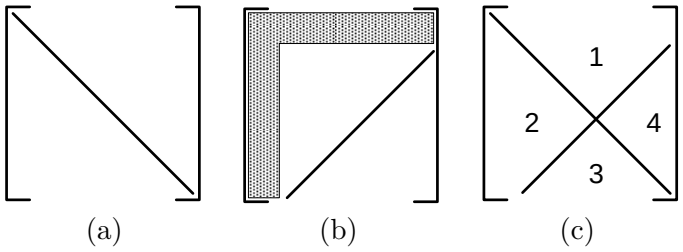


Figure 17.3: Symmetry regions of the cross-spectrum slice. (a) The values are mirrored around the diagonal. (b) If the first row and column are ignored the remaining values are conjugated and mirrored along the antidiagonal. (c) Using the symmetry regions from (a) and (b) it is only neccessary to calculate the values in region 1 or 2 while the remaining values can be obtained by mirroring and conjugating.

symmetry regions it is possible to divide the slice into four regions. To obtain all the values it is necessary to calculate either region 1 or 2 while the remaining values can be obtained through mirroring and conjugating. Due to indexing it may be better to calculate either region 1 and 4 or 2 and 3.

The same symmetry regions apply to the autospectrum.

17.6 Second order moment spectrum

The 2nd order moment cross-spectrum with arbitrary input signals is given by:

$$M_{xy}(\omega) = X^*(\omega) \cdot Y(\omega)$$

The corresponding MATLAB code:

```
1 M2xy = conj(X) .* Y/N;
2 M2xz = conj(X) .* Z/N;
3 M2xw = conj(X) .* W/N;
```

In the review of the matrix-vector convolution it was found that the cross-moment spectrum **M2xw(·)** is not needed and that **M2xy(·)** = **M2xz(·)** due to precondition 4. As such it is only neccessary to calculate one of the cross-moment spectra. Furthermore as a results of preconditions 4 and 3 **M2xy(·)** will be an automoment spectrum and half the values will be mirrored and can thus be pruned.

17.7 1D FFT

The 1-dimensional DFT is given by:

$$X(k) = \sum_{n=0}^{N-1} e^{-\frac{2\pi j}{N}kn} x(n)$$

In MATLAB the FFT is used and the corresponding code becomes:

```
X = fft(x, fftlength);
Y = fft(y, fftlength);
Z = fft(z, fftlength);
W = fft(w, fftlength);
```

Due to precondition 4 and the review of the 2nd order moment spectrum stating that is is only necessary to calculate one automoment spectrum, it is not necessary to calculate the Fourier transform of more than $x(t)$. Furthermore precondition 3 states that $x(t)$ is real and as a result half the values of the Fourier will be mirrored and can thus be pruned.

17.8 2D FFT

The 2-dimensional DFT is given by:

$$X(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \left(e^{-\frac{2\pi j}{N_1}k_1 n_1} \sum_{n_2=0}^{N_2-1} e^{-\frac{2\pi j}{N_2}k_2 n_2} x(n_1, n_2) \right)$$

In MATLAB the 2D FFT is used and the corresponding code becomes:

```
Gzw = fftn(gzw);
Gyw = fftn(gyw);
Gyz = fftn(gyz);
```

In the review of the matrix-vector convolution it was shown that $\mathbf{Gyz}(\cdot)$ is not needed and $\mathbf{Gzw}(\cdot) = \mathbf{Gyw}(\cdot)$ as well as only the first column of either $\mathbf{Gzw}(\cdot)$ or $\mathbf{Gyw}(\cdot)$ is needed. If this is taken into consideration the 2-dimensional DFT can be reduced to:

$$X(k_1, 0) = \sum_{n_1=0}^{N_1-1} \left(e^{-\frac{2\pi j}{N_1}k_1 n_1} \sum_{n_2=0}^{N_2-1} 1 \cdot x(n_1, n_2) \right)$$

Or in other words; the 2-dimensional FFT can be reduced to a 1-dimensional FFT of a column vector containing the sum of each row of $\mathbf{gzw}(\cdot)$ or $\mathbf{gyw}(\cdot)$.

17.9 Matrix expansion

The matrix expansion is given by:

$$gxy(n_1, n_2) = \begin{cases} m2xy(n_2 - n_1 + N - 1), & 0 \leq n_2 - n_1 + N - 1 \leq 2N - 2 \\ 0, & \text{otherwise} \end{cases}, \quad 0 \leq n_1, n_2 \leq N$$

After the expansion the matrix is shifted as shown in figure 17.4 to reorder the elements correctly. The corresponding MATLAB code:

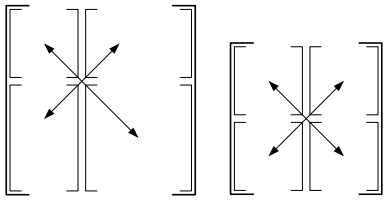


Figure 17.4: Illustration of how the elements of the matrices are shifted in the matrix expansion. (a) Odd size of the matrix. (b) Even size of the matrix.

```
for n1=0:2*N-2
    for n2=0:2*N-2
        if 0<=sum(n2-n1+N-1) & sum(n2-n1+N-1)<=2*N-2
            gzw(n1+1,n2+1)=m2zw(n2-n1+N);
            gyw(n1+1,n2+1)=m2yw(n2-n1+N);
            gyz(n1+1,n2+1)=m2yz(n2-n1+N);
        end
    end
end

gzw=ifftshift(gzw);
gyw=ifftshift(gyw);
gyz=ifftshift(gyz);
```

As shown in the review of the 2-dimensional FFT, it is not necessary to perform the matrix expansion only calculate the sum of each row in the matrix. The shifting does not affect the row sums, only the order of the values in the resulting column vector. If the shifting is performed on the column vector of row sums the result will be equal to the rows sums calculated from the unchanged MATLAB implementation. As such the column vector of row sums will be calculated directly and then shifted. Furthermore in the review of the matrix-vector convolution it is shown that $\mathbf{Gzw}(\cdot) = \mathbf{Gyw}(\cdot)$ and $\mathbf{Gyz}(\cdot)$ is redundant. This translates into only $\mathbf{gzw}(\cdot)$ or $\mathbf{gyw}(\cdot)$ being needed for further calculations.

17.10 Second order moment sequence

The second order cross moment sequence is given by:

$$m_{xy}(\tau_1) = \frac{1}{N} \sum_{t=-(N-1)}^{N-1} x(t) \cdot y(t+\tau_1)$$

The corresponding MATLAB code:

```
ytmp=[zeros(1,N) y zeros(1,N)];
ztmp=[zeros(1,N) y zeros(1,N)];
wtmp=[zeros(1,N) w zeros(1,N)];

```

```
for T1 = -(N-1):N-1
    m2yz(T1+M) = sum(ytmp(1+N+T1:2*N+T1) .* ztmp(1+N+T1:2*N+T1))/N;
    m2zw(T1+M) = sum(ztmp(1+N+T1:2*N+T1) .* wtmp(1+N+T1:2*N+T1))/N;
    m2yw(T1+M) = sum(ytmp(1+N+T1:2*N+T1) .* wtmp(1+N+T1:2*N+T1))/N;
end
```

In the review of the matrix-vector convolution it is shown that $\mathbf{Gzw}(\cdot) = \mathbf{Gyw}(\cdot)$ and $\mathbf{Gyz}(\cdot)$ is redundant. This translates into only $\mathbf{m2zw}(\cdot)$ or $\mathbf{m2yw}(\cdot)$ being needed for further calculations.

17.11 Cubic addition

The cube terms added are given in:

$$\begin{aligned} C_{xyzw}(\omega_1,\omega_2,\omega_3) &= M_{xyzw}(\omega_1,\omega_2,\omega_3) \\ &\quad -M_{xy}(\omega_1) * \mathscr{F}\left[m_{zw}(\tau_3-\tau_2)\right] \\ &\quad -M_{xz}(\omega_2) * \mathscr{F}\left[m_{yw}(\tau_3-\tau_1)\right] \\ &\quad -M_{xw}(\omega_3) * \mathscr{F}\left[m_{yz}(\tau_2-\tau_1)\right] \end{aligned} \tag{17.1}$$

The corresponding MATLAB code:

```
C4xyzw = M4xyzw + Gxyzw + Gxzyw + Gxwyz;
```

Using the preconditions to perform a number of reductions in the previous blocks it can be shown that the cubic addition reduces to a matrix addition. If the symmetry regions are used the matrix addition requires only half or a quarter of the elements depending on indexing and complex conjugates. Furthermore it has been shown that $M_{xyzw}(\omega_1,\omega_2,\omega_3)$ and $M_{xw}(\omega_3) * \mathscr{F}\left[m_{yz}(\tau_2-\tau_1)\right]$ are equal to 0 for all values of ω_1, ω_2 and ω_3 and can thus be removed from the addition.

17.12 Reduced block diagram and MATLAB code

After the reduction of the code the block diagram for the center slice reduces to the diagram shown in figure 17.5. And the MATLAB code reduces down to:

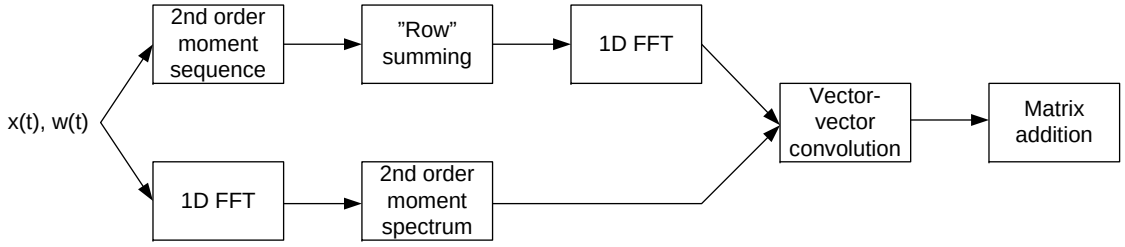


Figure 17.5: Block diagram of the reduced trispectrum MATLAB implementation.

```
% 1D FFT
X = fft(x, fftlength);

% Second order moment sequence
m2xw = zeros(1,K);
wtmp=[zeros(1,N-1) w zeros(1,N-1)];
for T1 = -(N-1):N-1
    m2xw(T1+M) = x*wtmp(N+T1:2*N+T1-1)'/N; % Sum by vector-vector mul
end

% Row summing
g=zeros(1,K);
m2xwtmp=[zeros(1,N-1) m2xw zeros(1,N-1)];
for k=1:K
    g(k)=sum(m2xwtmp(2*N-k:2*N+K-k-1));
end

pad = fftlength - K;
bpad = ceil(pad/2);
fpad = floor(pad/2);
g = ifftshift(padarray(padarray(g',fpad,'post'),bpad,'pre'));

% 1D FFT
G = fft(g);

% Second order moment spectrum
M2xx = X .* conj(X)/N;

% Vector-vector convolution
for W1 = 0:fftlength-1
    for W2 = 0:fftlength-1
        Gxyzw(W1+1,W2+1) = M2xx(W1+1) * G(W2+1);
        Gxzyw(W1+1,W2+1) = M2xx(W2+1) * G(W1+1);
    end
end

% Matrix addition
C4xyzw =-Gxyzw - Gxzyw;
```

17.13 New complexity estimation for the trispectra

The new complexity for each step in estimating the trispectrum is listed in this section. The added complexity for calculating indexes is not evaluated. The steps corresponds to the step in figure 17.5. Also the DFT used initially for the Fourier transform is replaced with a fast Fourier transform (FFT) and symmetry regions are not accounted for.

1. The complexity for removing the mean is $4n$ operations. n for computing the mean and n for subtracting the mean from all elements and times two for both x and w. Although not a part of the actual algorithm it is still a necessary step.

2. The 1-dimensional Fourier transform has a complexity of $f \cdot \log(f)$ operations. As it is used twice the total complexity for the 1D FFT is $2f \cdot \log(f)$ operations.

3. Only one second order moment spectrum is calculated with a complexity of $2f$ operations.

4. The second order moment sequence has a complexity of $4n^2$ operations.

5. The row summing has a complexity of $4n^2$

6. The vector-vector convolution has a complexity of f^2 operations. As two vector vector convolutions are computed the total complexity is $2f^2$.

7. The matrix addition has a complexity of $2f^2$ operations due to inverting the signs of both matrices.

Remembering that the number frequency bins are twice the size of the frame length the total complexity for estimating one trispectrum equates to: $(24n^2 + 4n \cdot \log(2n) + 9n)$.

The original complexity for calculating a trispectrum which contains $2n$ slices was in chapter 16 found to be:

$$48n^4 + 88n^3 + 16n^3(2L + 1)^3 + 28n^2 + 17n \tag{17.2}$$

Assuming that no smoothing is performed in the trispectra and the frame length is 128 samples, the number of arithmetic operations for computing the center slice is $398 \cdot 10^3$.It was, however, shown that the smoothing must be performed and thus the neighbouring $2L$ slices must also be computed. In order to find the complexity with smoothing, an estimate of the complexity for calculating the neighbouring slices is found using the steps in figure 17.2:

1. The operations needed for removing the mean have been done for the center slice and thus does not need to be performed again.

2. The 1-dimensional FFT of w is not computed for the center slice, but is needed for neighbouring slices in the fourth order moment spectrum. w is the same for all neighbouring slices and the complexity of the step is then $f \cdot \log(f)$.

3. For the neighbouring slices the fourth order moment spectrum is not zero. It is, however, only necessary to compute $2L$ slices of the fourth order moment spectrum each having a complexity of $16n^2$ operations giving a total complexity of $2L16n^2$.

4. The second order moment cross-spectrum between x and w are needed for the matrix vector convolution and must be computed. Like the Fourier transform this is the same for all the neighbouring slices and the added complexity is then $2f$ operations.

5. The second order auto-moment sequence of x is needed for the matrix expansion and must be computed. As the Fourier transform of w, only one is needed for all the neighbouring slices and the added complexity is then $4n^2$.

6. The matrix expansion is necessary for both the moment sequences as the following FFT can not be reduced from 2-dimension to 1-dimensional. The matrix expansion does, however, not require any arithmetic operations, only memory moves/copies and the complexity is then 0.

7. Instead of using one column of the Fourier transform of the matrix expansions, $2L$ columns are needed for the matrix-vector convolution. These columns can be computed by a 2-dimensional DFT or the full Fourier transform can be computed by a 2-dimension FFT. The latter is assumed giving a complexity for each Fourier transform of $f^2 \cdot \log(f^2)$ operations. The total complexity for the step is then $2f^2 \cdot \log(f^2)$ operations.

8. The matrix-vector convolution must be performed for all $2L$ neighbouring slices and three convolutions for each slice. Each slice has a complexity of f^2 operations giving a total complexity of $2L3f^2$.

9. The cubic addition also contains $2L3f^2$ operations for adding the matrix-vector convolutions and the fourth order moment spectrum.

With the center and neighbouring slices computed the smoothing can be done. For each output value the smoothing is performed over a cube of values with x-, y- and z-dimensions $= 2L + 1$. The cube is multiplied elementwise with the window and all the products summed. For each output value this gives a complexity of $2 \cdot (2L + 1)^3$ operations and a total complexity for the smoothing of $2 \cdot (2L + 1)^3 \cdot f^2$ operations.

The added complexity for computing the center slice with smoothing is then:

$$(80L + 8 \cdot (2L + 1)^3 + 4) \cdot n^2 + 8n^2 \log 4n^2 + 4n + 2n \log 2n$$

With $L = 10$ the total number of operations for computing the center slice with smoothing is then $1.23 \cdot 10^9 + 398 \cdot 10^3 \approx 1.23 \cdot 10^9$ arithmetic operations.

Using equation (17.2) for the original trispectrum estimation the number of arithmetic operations using same L and N is $323 \cdot 10^9$. Comparing this to the reduced complexity the difference is a factor of 263.

The complexity for the filter update in the BSS method can be now calculated. The operations required for the inverse filtering, reverse third order moment spectrum and $\bar{\bar{H}}(0)$ estimation are unchanged from the previous chapter while the operations for the non-minimum phase filter estimation can be calculated from the estimated complexity for computing one trispectrum estimation. Four unique trispectra estimates are needed for the non-minimum phase filter estimation. This gives the total number of operations for one filter update as shown in table 17.1. The non-minimum phase filter estimation intially had the same complexity as the $\bar{\bar{H}}(0)$ estima-

Inverse filtering	$8.5 \cdot 10^6$ operations
Reverse third order moment spectrum	$42 \cdot 10^6$ operations
Non-minimum phase filter estimation	$4.92 \cdot 10^9$ operations
$\bar{\bar{H}}(0)$ estimation	$2.6 \cdot 10^{12}$ operations
Total complexity	$2.6 \cdot 10^{12}$ operations

Table 17.1: Number of operations for computing one filter update in the TITO model after the complexity of the non-minimum phase estimation has been reduced.

tion, but a reduction of the complexity of around 500 have been achieved. If it is assumed that a similar reduction in complexity can be achived for the $\bar{\bar{H}}(0)$ estimation, the overall reduction for

a filter update will also be around 500 times or a complexity of about $10 \cdot 10^9$. In the previous chapter it was stated that a speed up of at least 130 had to be achieved, in order for the algorithm to run in real time on the platform. Using the reduced complexity it should be possible to run the algorithm four times faster than needed. However, there are other practical concerns with regard to the implementation that can change this number; added complexity from using complex numbers, logarithm functions taking more than one operation, memory moves etc. To get a more accurate estimate how fast a practical application can be executed, the computation of the center slice without smoothing is implemented on the platform.

Chapter 18

CUDA architecture

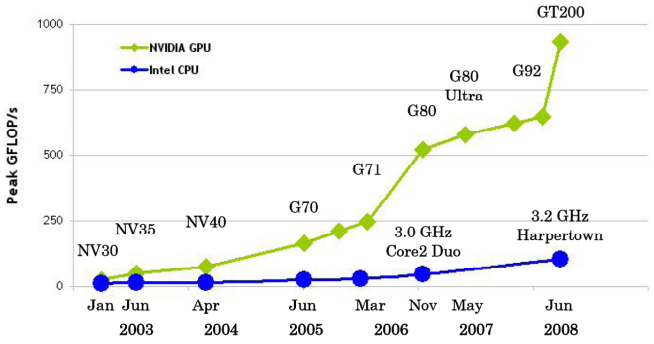
In this chapter the hardware platform, the NVIDIA GPU, is described to give an understanding of how an implementation of the BSS algorithm can be done efficiently. The programming of the platform is described in appendix B.

In the recent years using GPUs as general purpose processors has become an increasingly popular topic in the scientific community. The reason behind this is the sheer amount of computational power and memory bandwidth GPUs can deliver, in fact, GPUs surpassed Central Processing Units (CPUs) in these fields several years ago, as can be seen in figures 18.1 (a) and (b).

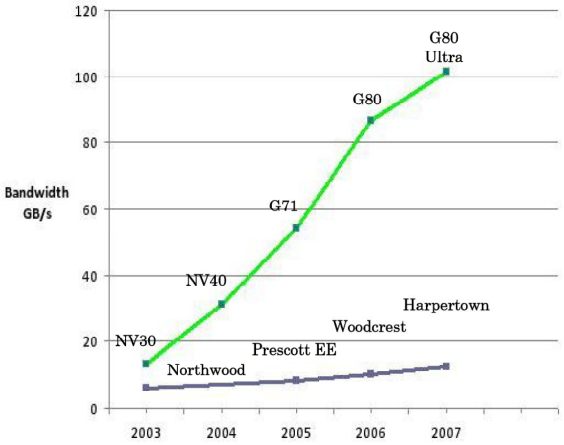
Why have GPUs not been used for general purpose processors previously? There has not been a lot of focus on GPUs outside video games and as such GPU manufacturers such as NVIDIA and ATi have not developed their GPUs for this market. Nor have most scientists seen GPUs as an alternative to the common solutions such as DSPs or FPGAs and thus there have been no need for GPUs as general purpose processors. Due to this the programming language and GPUs have not been ideal for general purpose applications, but this did not hold back the first attempts to use GPUs for scientific purposes. Since then GPUs as general purpose processors have received more attention and CUDA is a result of that.

CUDA is a programming framework developed by NVIDIA which allows programmers to easily use NVIDIA GPUs for general purpose applications. The CUDA programming language is an extension to the well known C language and hides the hardware behind an extra layer of abstraction. This simplifies the code because programmers do not need to worry about thread management and also makes the code platform independent as only the underlying layers need to be changed for different GPUs.

While GPUs offer tremendous computational power they have other downsides more common solutions do not have. GPUs have to be installed in a fully fledged computer meaning they'll take up substantially more space and in most cases the GPU alone uses more power then other common solutions resulting in an increased need for cooling. Most of the peripherals in a computer may also be redundant for scientific purposes, while in some cases it can be an advantage to have the these for I/O. Overall GPUs can provide a very large increase in computational power, but at a cost.



(a)



(b)

Figure 18.1: (a) Increase in FLOPS performed by NVIDIA GPUs and Intel CPUs since 2003. (b) Increase in available memory bandwidth of NVIDIA GPUs and Intel CPUs since 2003. [7, p. 2]

18.1 Hardware layer

The reason GPUs have such tremendous computational power compared to CPUs is the application they are designed for. 3D applications require very high data throughput for the thousands of pixel values to be computed. The instructions for the different pixels are, however, usually the same and conditional branches are less common. As such a large amount of the transistors are assigned to ALUs and very few to flow control on GPUs compared to CPUs. This is illustrated in figure 18.2. From the figure it can also be seen that very few transistors are allocated for cache on GPUs. While cache could be of use for GPUs, the memory latency introduced by using external memory is hidden behind the data throughput [7, p. 3]. The small amount of transistors allocated to flow control means that the programming of the GPUs is somewhat limited which is explained in further detail in appendix B.

Looking further into the architecture figure 18.3 shows a simplified version of the basic blocks of current CUDA capable GPUs. The device is the GPU while device memory is on the same PCB also knows as the graphics card. The GPU itself is built around N streaming multiprocessors (SMs) containing M scalar processors (SPs), two special units for transcendentals, an instruction unit and four types of memory available to the SPs; registers, read only constant

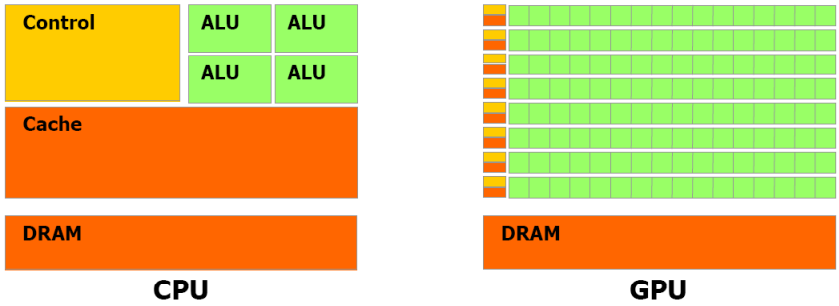


Figure 18.2: Transistor allocation on GPUs compared to CPUs. [7, p. 3]

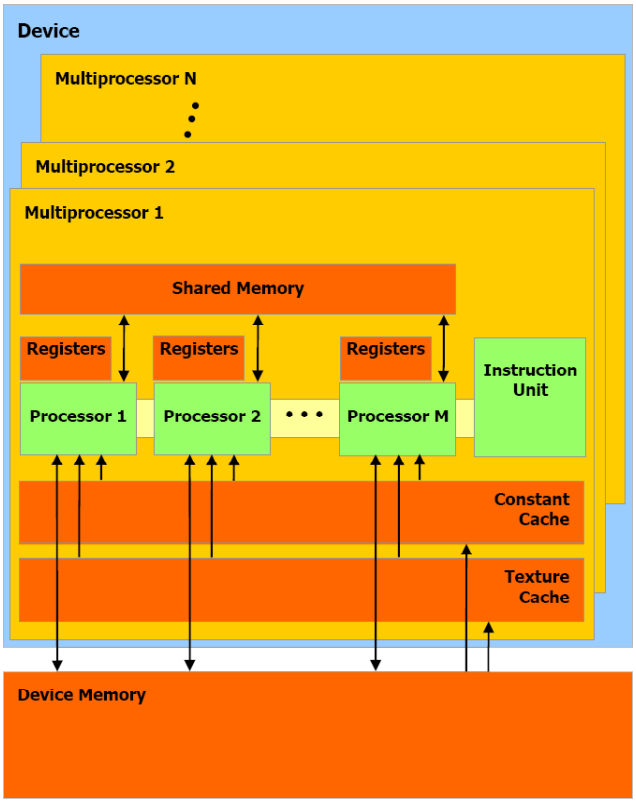


Figure 18.3: Hardware architecture of current NVIDIA GPUs capable of running CUDA code. The block diagram is simplified by leaving out blocks that are unimportant in the understanding of how CUDA is implemented on the GPU. [7, p. 17]

memory, read only texture memory and shared read-write memory. The current generation of NVIDIA GPUs feature up to 30 SMs with 8 SPs each for a total of 240 SPs per GPU. For the current generation each of the SMs can handle up to 1,024 active threads or 128 per SP, while the maximum number of threads being processed concurrently is equal to the number of SPs.

18.1.1 Registers

Registers are on-chip and the fastest type of memory available to the SPs. For the current generation there are 16,384 registers available per SM. These are assigned dynamically to the

SPs depending on the code being executed.

18.1.2 Shared memory

For each SM there is 16 KB shared memory available. The shared memory is on-chip and organized into 16 banks. All 16 banks can be accessed concurrently and will in the case with no memory bank conflicts be just as fast as registers.

18.1.3 Constant memory

Constant memory is read only and placed off-chip, but cached. The total amount of constant memory is 64 KB while 8 KB are cached for each SM. If all threads running on one SM reads the same address the constant memory is just as fast as the registers due to the caching. If reading different addresses the accesses will be serialized resulting in a slowdown that scales with the number of accesses performed.

18.1.4 Texture memory

Texture memory is read only and placed off-chip, but cached. For each SM between 6 and 8 KB of texture memory is cached. Due to the 2D nature of textures highest performance is obtained with similar data arrays and can be just as fast as registers.

18.1.5 Non-CUDA specific blocks

Figure 18.3 only shows the small part of the blocks in a GPU that is important to CUDA. Several other blocks exist, but are transparent to a CUDA programmer. Some of these are the PCI-Express interface, memory controller for device memory and so on.

To understand how a program is mapped onto this architecture is it is necessary to go to a higher abstraction level that is closely tied to the CUDA extensions to C.

18.2 Compute capability

As new GPUs are developed their functionality is increased. To specify what each GPU is capable of it is given a revision number, called the compute capability, consisting of a minor and a major number. GPUs having the same major revision number have the same core architecture while the minor revision number specify smaller improvements to the architecture such as new function support or increased amount of registers/memory.

A list of all current GPUs and their compute capability as well as changes for each revision can be found in appendix A in [7].

18.3 Software layer

One of the reasons CUDA is capable of scaling over all CUDA enabled GPUs, regardless of how many SMs they feature, is the way threads are handled.

18.3.1 Thread hierarchy

Unlike normal C functions that are executed one time when called, a function defined using the CUDA C extensions, also called a kernel, can be executed an arbitrary number of times when called. For each of these times a thread is created by the GPU to handle the operations inside the function. These threads are lightweight and may exist for as short as one clock cycle of the SPs, but are fully fledged threads in the sense that they each have their own program counter, stack and assigned memory.

The threads are arranged in thread blocks of up to 3 dimensions so that each thread in a block can be analog to an element in a vector, matrix or field. A block can at most contain 512 threads and the maximum x-, y- and z-dimensions are 512, 512 and 64, respectively. While this may not be enough threads for many applications, several block can be grouped together to form a grid of thread blocks. The grid can be up to 2-dimensions and have a maximum size of 65,535 blocks for each dimension. This gives a total of more than 2,000 billion threads per kernel. An illustration of the thread hierarchy can be seen in figure 18.4.

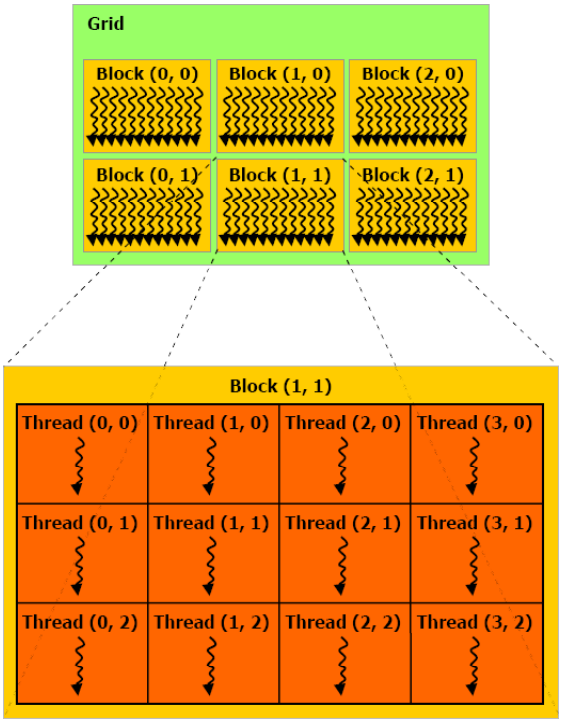


Figure 18.4: Hierarchy of threads created by a CUDA function running on a GPU. The threads are arranged in blocks of up to 3 dimensions and each block is part of a grid of blocks of up to 2 dimensions. Note that only 2 dimensional blocks are illustrated on the figure. [7, p. 10]

18.3.2 Block and thread assignment to processors

When a kernel has been defined it is called with two parameters. These define the size of the blocks and the size of the grid and are the only thread related parameters the programmers are be exposed to. Any underlying management of the blocks is handled by the GPU and threads by the SMs.

When a kernel is called the blocks are assigned to the SMs. A block can not be split between several SMs, but one SM can handle several blocks. The number of blocks assigned to an SM depends on the number of blocks in the grid and the number of registers and shared memory necessary to execute one block. If the SMs do not have enough registers and shared memory to execute at least one block, the kernel will fail. When the blocks have been assigned to the SMs, these will schedule threads in bundles, called warps, containing 32 threads. Each thread within a block has a unique ID that can be found using the thread index of the block. The threads are always ordered with increasing and consecutive IDs, starting with thread 0 in the first warp. If all threads of a warp follows the same execution path, the threads will be executed in parallel. If the threads diverge due to data dependent branches, the SM will serialize the execution of the paths until they all converge.

The ability of CUDA applications to scale from low end GPUs with few SMs to high end GPUs with tens of SMs is a consequence of the way kernels are split into blocks. Any CUDA enabled GPU can assign the blocks to an arbitrary number of SMs and let the SMs handle the threads. It is, however, up to the programmer to determine appropriate sizes of the grid and blocks to allow applications to scale for future GPUs as well as obtain the best performance. These considerations are addressed in appendix B.

Chapter 19

CUDA implementation

In this chapter it is described how a CUDA program to estimate the center slice of the trispectrum estimate is implemented. For users not familiar with the CUDA framework it is recommended to read appendix B before reading this chapter.

The second order moment sequence kernel is used as an example to display both basic programming principles as well as optimization principles in chapter 20. To give the reader a deeper understanding of both the kernel and the calling process both are described using abstract code followed by the CUDA implementation.

For figures showing how a kernel is designed, memory arrays are illustrated as squares with a number inside. The number is the address of the array. Threads are illustrated with circles where the number inside is the thread number.

19.1 Baseline implementation

The implementation described in this chapter is a baseline implementation. This means several issues that may decrease performance will not be addressed as the functionality of the implementation is first priority. Issues that may arise include shared memory bank conflicts, complex and fast versus simple but slow implementations, **__syncthreads()** related slowdowns and loop unrolling. These issues will be addressed in chapter 20.

19.2 CUDA variables

The MATLAB variable names introduced in chapter 17 are reused in the CUDA implementation as well as the extensions. Pointers to any variables are prefixed with a **p** while most important

variables and functions will be suffixed by either **_d** or **_h** to determine if they are located in host or device memory. To simplify indexing in the code four variables are introduced:

Variable	Description
tx	x-index of thread in block.
ty	y-index of thread in block.
bx	x-index of block in grid.
by	y-index of block in grid.

19.3 The cufft and cuComplex types

The NVIDIA complex types are based on the **complex** type defined in the C99 standard. **cuComplex** is basically a **typedef** of the **complex** type, but adapted to suit the CUDA syntax. Depending on compilation settings **cuComplex** will be compiled with either **float** or **double** precision. A **cuComplex** array consists of interleaved real and imaginary values where the real part is accessed by using **variablename**[·]**.x** and imaginary part by using **variablename**[·]**.y**.

For the cufft, the **cufftComplex** is simply a **typedef** of **cuComplex** while **cufftReal** is a typedef of **float**.

As the **cufft** and **cuComplex** types are both **typedef**s of **float**/**double** and the implementation is done with single precision, **float1** and **float2** will be used for real and complex values, respectively, instead. For functions requiring either a **cufft** or **cuComplex** type as input a typecast will be used.

19.4 Block overview

The implementation has been done based on the reduced block diagram shown in figure 17.5. However, a few of the blocks have been split up or combined as described below.

In section 17.9 it was described that the matrix expansion was not necessary and replaceable by a block calculating the sum of each row in the matrix. This block should also shifted the row sums before the 1D FFT, but for the sake of simplicity the shifting has been implemented in a kernel of its own. Furthermore in a practical application it may be desired to use an FFT length that is larger than the size of the input vector. If this is the case the input must be zero-padded to obtain the wanted FFT length. This is implemented in the same block as the shifting.

The last two blocks in the trispectrum estimation, vector-vector convolution and matrix addition, have also been merged into one block. This is due to the fact that the matrix output of the vector-vector convolution can become quite large, dependant on the input signal length and the resultant memory moves between the two blocks can be completely avoided by merging them into one.

With that in mind the block diagram for the CUDA implementation is illustrated in figure 19.1.

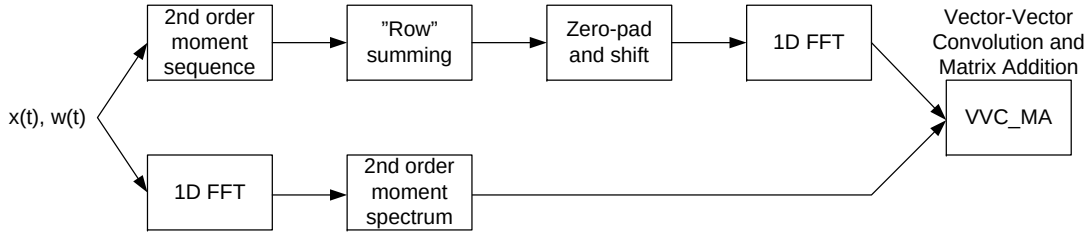


Figure 19.1: Block diagram of the CUDA implementation.

19.5 Second order moment sequence

Kernel name	Input	Input type	Output	Output type
momSeq2_d	**x_d**[·] **w_d**[·]	**float2** **float1**	**m2xw_d**[·]	**float1**

19.5.1 Description

The second order moment sequence can be seen as calculating the inner product of several vectors, i.e. multiply and accumulate for different values of **T** and lastly divide by **N**. It can also be written as the matrix-vector multiplication shown in equation (19.1) which will make it easier to the understand block partitioning of the kernel.

$$\mathbf{m2xw\_d} = \begin{bmatrix} 0 & \dots & 0 & \mathbf{w\_d[0]} \\ \vdots & \iddots & \mathbf{w\_d[0]} & \mathbf{w\_d[1]} \\ 0 & \iddots & \iddots & \vdots \\ \mathbf{w\_d[0]} & \iddots & \iddots & \mathbf{w\_d[N\text{-}1]} \\ \vdots & \iddots & \iddots & 0 \\ \vdots & \iddots & \iddots & \vdots \\ \mathbf{w\_d[N\text{-}1]} & 0 & \dots & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x\_d[0]} \\ \mathbf{x\_d[1]} \\ \vdots \\ \mathbf{x\_d[N\text{-}1]} \end{bmatrix} \cdot \frac{1}{\mathbf{N}} \tag{19.1}$$

Each row in the matrix corresponds to a value of **T**, starting with **T**=-(**N**-1) and ending with **T**=**N**-1 for a total ot 2·**N**-1 values of **T**.

The the kernel is designed such that the multiplications will be performed first followed by adding all of the products together.

The fastest way, in terms of parallelism, to calculate a sum of values is using a tree structure as shown in figure 19.2. If the input number of values to be added together is not a power of two the width of the tree will become odd at some point during the loops. To avoid this

problem it is chosen to extend the tree with one element containing a zero each time a loop contains an odd width.

On a block level each thread loads one element from **x_d** and one from **w_d**, multiplies

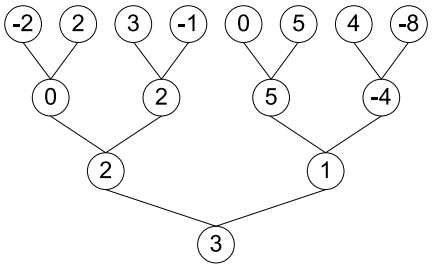


Figure 19.2: The figure shows the fastest way, in terms of parallelism, to calculate the sum of a number of values.

them and stores the product in shared memory. Half the threads will then each add together two products from the multiplication and store the result back in shared memory. When all elements have been summed the result will be divided by **N** and stored in global memory as shown in figure 19.3. Depending on which row/column is accessed, the kernel should load zero to shared memory instead of performing the multiplication of **x_d** and **w_d**. On a grid level it is

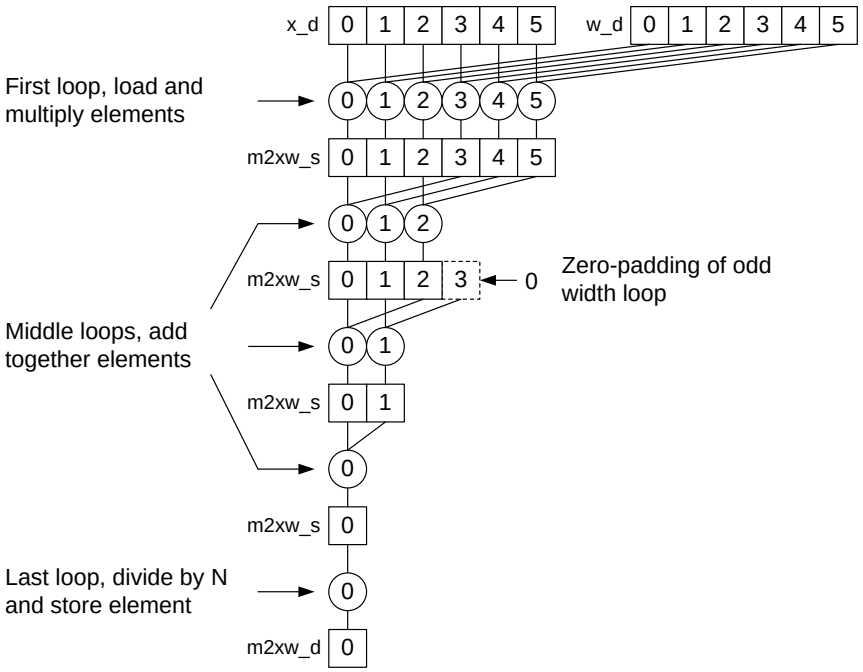


Figure 19.3: The block level operations of the **momSeq2_d** kernel.

necessary to partition the maxtrix-vector multiplication into several blocks due to the fact that the input vectors may be longer than the maximum block size. If the y-dimension of the block is limited to 1 the resulting y-dimension of the grid will be 2·**N**-1. This introduces redundancy as **x_d** will be loaded separately by each block, but indexing is simplified.

With a block y-dimension of 1 it is, however, still not always possible for one block to handle one row of the matrix-vector multiplication. As a result of this each row must be split into

a number of blocks calculating intermediate sums that need to be added together to obtain the final result. This requires a synchronization of the blocks which can only be achieved if the kernel calls are used as synchronization points, otherwise dependency issues may become a problem. The kernel is designed to be able to calculate both intermediate values and add these together depending on which parameters it is called with. Depending on block and input vector size it may be necessary to perform several calls or iterations of the kernel.

The grid level design also affects the block level design as it is only necessary to load and multiply elements from **x_d** and **w_d** in the first iteration while the remaining iterations only load intermediate results and add these together. Furthermore it is not necessary to divide by **N** in each iteration, only the last iteration. The block partitioning is illustrated in figure 19.4.

If the x-dimension of the block times the x-dimension of the grid exceeds **N**, it is chosen to zero-pad the input array and have all threads do the same work rather than constructing **if** statements to handle both cases, as this would result in divergent warps.

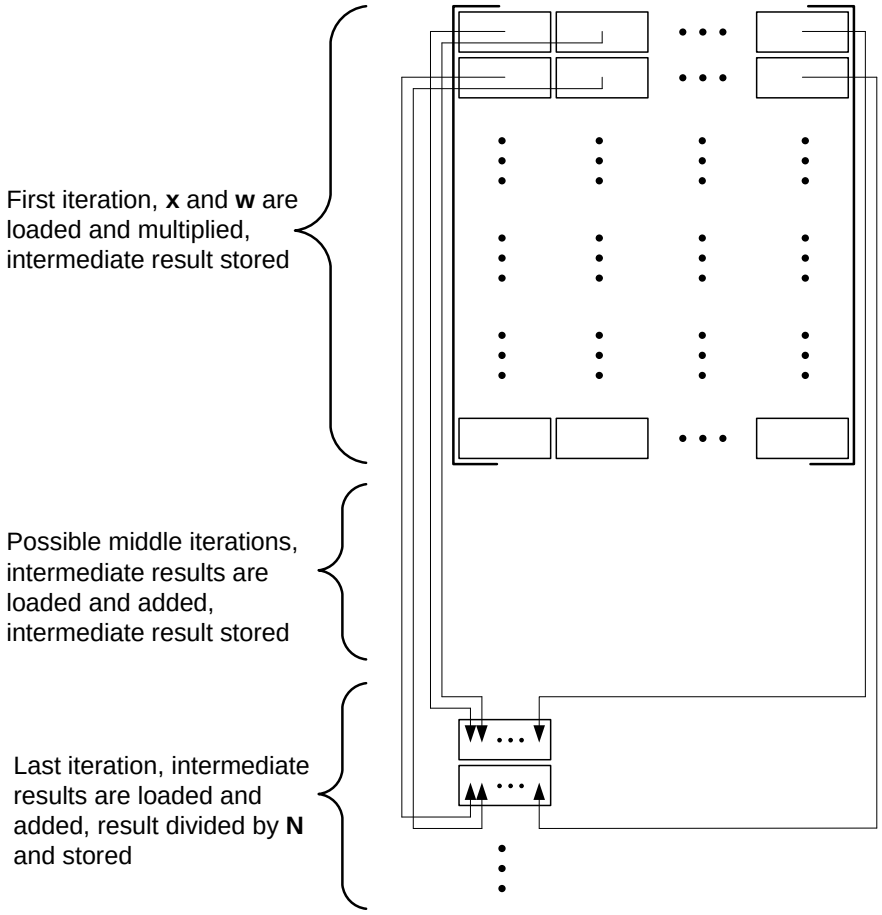


Figure 19.4: The grid level operations of the **momSeq2_d** kernel.

19.5.2 Abstract and CUDA code

The code in the following sections are in most cases self-explanatory in the sense of the work they do, but many of the lines have been explained in further detailing following the code snippets. This is to give the reader an insight into some of the concerns in the implementation.

Kernel abstract code

```
1  momSeq2 kernel(in x, in w, out m2xw, in iteration, in number of elements){
2      if (first iteration){
3          if (thread index corresponds to nonzero entry of w matrix){
4              load x and w from global memory;
5              multiply x and w and store result in shared memory
6          }
7          else {
8              store 0 in shared memory;
9          }
10     }
11     else {
12         if (thread index lower than number of intermediate results){
13             load intermediate result from global memory and store in shared memory;
14         }
15         else {
16             store 0 in shared memory
17         }
18     }
19
20     synchronize threads;
21
22     while (number of elements > 1){
23         if (number of elements is odd){
24             zeropad after last element;
25             increment number of elements by 1;
26         }
27
28         if (thread index < number of elements/2){
29             add from shared memory using indexes: thread index and thread index +
                   number of elements/2 and store result in shared memory;
30         }
31
32         synchronize threads;
33         number of elements = number of elements/2;
34     }
35
36     if (last iteration) {
37         divide final result from shared memory by N and store in global memory;
38     }
39     else {
40         store intermediate result from shared memory in global memory;
41     }
42 }
```

Data load, line 2-18:
To take advantage of the fast on-chip memory data is loaded to shared memory from global

memory. There are two cases, in the case of the first iteration **x_d** and **w_d** are loaded and multiplied together before storing in shared memory and in the case of the remaining iterations the intermediate are loaded directly to shared memory.

Data load control during first iteration, line 3 and 7:
The **if-then-else** statement presented in line 3 and 7 can be split up into three cases. Looking at the **w_d** matrix in equation (19.1) zeros need to be loaded if the index: Is in the upper left corner of the matrix, lower right corner of the matrix or outside the matrix. The thread index will only be outside the matrix in the x-dimension due to choosing the y-dimension of the blocks equal to 1, i.e. the number of blocks needed to cover all values in the y-dimension is always an integer number. If this is not possible for the chosen x-dimension of the blocks the last block will have thread indexes that lie outside the matrix and thus need to load zeros for those threads.

Data load control during remaining iterations, line 12 and 15:
For the **if-then-else** statement in line 12 and 15 there is only one case. As all the intermediate results are needed they must all be loaded, but due to the block dimensions as described above the thread index may exceed the number of intermediate results and thus need to load zeros for those threads.

Thread synchronization, line 20:
Before starting the tree structure addition of the elements it must be ensured that all elements are loaded. This is done using the thread synchronization function. If the thread synchronization was omitted it would be possible for threads to perform the addition before data was available in shared memory and thus yielding an incorrect result.

Tree structure addition, line 22-34:
The tree structure can be implemented in several ways using **while** or for loops. Regardless of how the implementation is done every loop must check that there is an even number of elements and zero pad if this is not the case (line 24). As the number of elements are needed for indexing it is important to increment the number of elements variable when a zero-padding is performed (line 25).

When there is an even number of elements the addition can be performed. Only half as many threads as there are elements are needed as each thread will load and add two elements from shared memory and store them back as shown in figure 19.3 (line 29).

When one loop of the tree is finished it is necessary to synchronize the threads again. This serves the purpose of avoiding one thread advancing to a new step in the tree, before all threads have finished the previous stop. If omitted the same problem with threads performing additions on data that is not yet available may arise.

Data store, line 36-41:
The last step in the kernel is to store the result of the tree structure addition. If it is the last iteration the result will be divided by **N** otherwise it will be stored in global memory directly.

Kernel CUDA code

```
1  __global__ void momSeq2_d(float2 *px_d, float1 *pw_d, float1 *pm2xw_d, int iteration,
       int n, float invN){
2      // Register variables
3      int s;
4
5      // Shared variables
6      extern __shared__ char data[];
7      float1 *m2xw_s = (float1*)data;
8
9      // Block index
10     int bx = blockIdx.x;
11     int by = blockIdx.y;
12
13     // Thread index
14     int tx = threadIdx.x;
15
16     // Load x*w or intermediate results from global memory to shared memory
17     if (iteration == 0){
18         if (tx+bx*(int)blockDim.x>=n-1-by && tx+bx*blockDim.x<2*n-1-by && tx+bx*
               blockDim.x<n)
19             m2xw_s[tx].x = px_d[tx+bx*blockDim.x].x*pw_d[tx+bx*blockDim.x+by-(n-1)].x;
20         else m2xw_s[tx].x = 0;
21     }
22     else {
23         if (tx+bx*blockDim.x < n) m2xw_s[tx].x = pm2xw_d[tx+bx*blockDim.x+by*n].x;
24         else m2xw_s[tx].x = 0;
25     }
26
27     __syncthreads();
28
29     // Perform tree structure addition
30     for (s=blockDim.x; s>1; s>>=1){
31         // Pad with one zero if array is odd size
32         if ((s&1) == 1){
33             if (tx==(s>>1)) m2xw_s[s].x=0;
34             s++;
35         }
36         if (tx<(s>>1)) m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+(s>>1)].x;
37         __syncthreads();
38     }
39
40     // Store m2xw or intermediate results in global memory
41     if (tx==0 && gridDim.x>1) pm2xw_d[bx+by*gridDim.x].x = m2xw_s[tx].x;
42     else if (tx==0 && gridDim.x==1) pm2xw_d[bx+by*gridDim.x].x = m2xw_s[tx].x*invN;
43 }
```

Function parameters, line 1:
The **x_d** array is declared as a **float2** although the imaginary part is always 0. This is because **x_d** is also used as input for the 1D FFT block which requires both real and imaginary parts. **n** the number of elements to add together and **iteration** is used to identify which iteration is the first. The variable **invN** will be explained below.

Data load control during first iteration, line 3 and 7:
As mentioned above there are three cases for the first iteration: **tx+bx*(int)blockDim.x>=n-1-by** makes sure zeros are loaded in the upper left corner of the **w_d** matrix. **blockDim.x** is typecast to **int** from **unsigned int** to make sure a comparison to the integer value **n-1-by** is

possible as the latter will become negative dependent on **by**. **tx+bx*blockDim.x<2*n-1-by** makes sure zeros are loaded in the lower right corner and **tx+bx*blockDim.x<n** makes sure zeros are loaded if the thread index exceeds the matrix.

Data load control during remaining iteraions, line 12 and 15:
For the remaining iterations **tx+bx*blockDim.x<n** makes sure zeroes are loaded if the thread index exceeds the number of elements.

Tree structure addition, line 30-38:
The tree structure is implemented as a **for** loop, but uses the x-dimension of the block rather than **n** as the starting value for the loop variable, **s**. This is due to the block partitioning meaning that only as many elements as the x-dimension of the block will be added together in one block, rather than **n** elements. This also means **s** will only become odd if the x-dimension of the block is not a power of two rather than **n** not being a power of two. The **for** loop will run as long as **s** is greater than one, i.e. there is a minimum of two elements to add together.

To determine if **s** is odd a bitwise comparison of the LSB is performed (line 32). If this evaluates to true a zero will be padded at the **s**'th index and **s** incremented by one. Only one thread will perform the zero-padding as the padding will be visible to all threads through shared memory (line 33). By using the same thread that adds the **s**'th element to zero-pad there will be no dependency issues and no need to synchronize the threads. All threads will, however, need to increment **s** as this is a register variable and thus not shared among the threads in the block.

As opposed to the abstract code all divisions by two have been replaced by the equivalent bitwise operation to improve speed.

Data store, line 41-42:
Like the zero-padding only one thread needs to store the result from the tree structure addition in global memory. As the result is always store in the 0'th index in shared memory the thread with index 0 is chosen. To determine whether or not the kernel call is the last of the iterations the x-dimension of the grid is compared to one. If the x-dimension is larger there are more intermediate values to add together and one or more kernel calls will be needed. If the x-dimension is one it is the last iteration and the result should be divided by **N** before storing in global memory. Rather than dividing by **N** the inverse of **N**, **invN**, is multiplied to improve speed.

Calling process abstract code

```
1  allocate memory space for x, w and m2xw on device;
2  copy x and w to device;
3
4  set start value of n = N;
5  set start value of iteration = 0;
6
7  set block size;
8  set grid size (y-dimension);
9  while (n>1) {
10     calc grid size (x-dimension);
11
```

```
12      call kernel;
13
14      increment iteration by 1;
15      calc new n;
16  }
17
18  free allocated memory on device;
```

No data is copied back from the device as the output of the kernel will only be used as input for the following block.

Memory allocation, line 1:
Allocating memory for **x_d** and **w_d** is straightforward as both vectors have length **N** and a size dependent on their type. For **m2xw_d** it is necessary to allocate memory for both the output vector and intermediate values.

Kernel loop, line 9-16:
The x-dimension of the grid size is calculated using the value of **n** and the x-dimension of the blocks. There must be enough blocks to cover all **n** elements.

For the kernel call the block and grid sizes are input as well as the amount of dynamically allocated shared memory used by each block.

A new **n** value must be calculated after each loop. The number of elements is always reduced by a factor equal to the x-dimension of the blocks rounding up.

Calling process CUDA code

```
1  // Allocate memory on device
2  cudaMalloc((void**)&px_d, N*sizeof(float2));
3  cudaMalloc((void**)&pw_d, N*sizeof(float1));
4  cudaMalloc((void**)&pm2xw_d, ceil((float)N/64.0)*(2*N-1)*sizeof(float1));
5
6  // Copy data to device
7  cudaMemcpy(px_d, px_h, N*sizeof(float2), cudaMemcpyHostToDevice);
8  cudaMemcpy(pw_d, pw_h, N*sizeof(float1), cudaMemcpyHostToDevice);
9
10 // Second order moment sequence
11 iteration = 0;
12 dimBlock.x = 64;        dimBlock.y = 1;
13                         dimGrid.y = 2*N-1;
14 for (n=N; n>1; n=(int)ceil((float)n/(float)dimBlock.x)){
15     dimGrid.x = (int)ceil((float)n/(float)dimBlock.x);
16     momSeq2_d <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d, pm2xw_d,
           iteration, n, 1.0/(float)N);
17     iteration++;
18 }
19
20 // Free allocated memory on device
21 cudaFree(px_d);
22 cudaFree(pw_d);
23 cudaFree(pm2xw_d);
```

Memory allocation, line 1-3:
For **m2xw_d** it will be necessary to allocate space for intermediate results. For each block in the y-dimension there will be as many intermediate results as there are blocks in the x-dimension, i.e. the grid size for the first iteration. The y-dimension of the grid is 2*$\mathbf{N}$-1 while the x-dimension of the grid is $\mathbf{N}$/x-dimension of block rounded up. An x-dimension of the block equal to 64 is used in the code snippet.

Grid and block dimensions, line 12-13:
The x-dimension of the blocks is determined at compile time, in the code snippet 64 is used. The y-dimension of the blocks is chosen to one resulting in an y-dimension of the grid equal to 2*$\mathbf{N}$-1.

Kernel loop, line 14-18:
The kernel loop is implemented as a **for** loop. As described in the abstract code, the loop variable **n** is set equal to **N** in the first iteration and subsequently divided by the block dimension rounded up each iteration. The loop will keep running as long as there is more than one element left. The typecasts are necessary due to the use of the **ceil()** function

The x-dimension of the grid is determined from the number of elements and the x-dimension of the block, i.e. $\mathbf{N}$/x-dimension of block rounded up.

The kernel is called using the specified grid and block size. The shared memory used is determined to be equal to the x-dimension of the blocks. The kernel only loads as many elements into shared memory as there are threads.

19.6 Row summing

Kernel name	Input	Input type	Output	Output type
rowSum_d	**m2xw_d**[·]	**float1**	**g_d**[·]	**float1**

19.6.1 Description

Because the row sum kernel performs the same function as the **momSeq2_d** kernel without the multiplications and division by **N**, both the block level and grid level designs are almost identical. The row sum kernel still adds the values together in a tree structure, zero-pads the array on both a grid and block level and each row sum is equal to one output in **g_d**. Other than the multiplications and division by **N** the only difference is the input matrix, shown in equation

19.2, of which the rows sums are calculated.

$$\begin{bmatrix} \textbf{m2xw\_d[N-1]} & \textbf{m2xw\_d[N]} & \dots & \textbf{m2xw\_d[2*N-2]} & 0 & \dots & 0 \\ \textbf{m2xw\_d[N-2]} & \textbf{m2xw\_d[N-1]} & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \textbf{m2xw\_d[0]} & \ddots & \ddots & \ddots & \ddots & \ddots & \textbf{m2xw\_d[2*N-2]} \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & \textbf{m2xw\_d[0]} & \dots & \dots & \textbf{m2xw\_d[N-1]} \end{bmatrix} \quad (19.2)$$

This results in a different indexing and zero-loads compared to the **momSeq2_d** kernel, but has no effects on the inner loops adding together the values.

19.7 Zero-pad and shift

Host function name	Input	Input type	Output	Output type
padShift_d	**g_d[·]**	**float1**	**gpadshift_d[·]**	**float2**

19.7.1 Description

Due to the way the following 1D FFT block is implemented **gpadshift_d** needs to be complex valued.

The input array **g_d** always has a length of 2·**N**-1 values. For the real part of **gpadshift_d** the kernel shifts the upper and lower part of **g_d** and zero-pads if the wanted FFT length is larger than 2·**N**-1. The last **N** values of **g_d** are put in the beginning of **gpadshift_d** followed by FFT length - (2·**N**-1) zeroes and lastly the **N**-1 first values of **g_d**. Depending on the index of each thread it will load either a value from **g_d** in global memory or a zero and store the value in **gpadshift_d** also in global memory.

For the imaginary part each thread stores a zero on in **gpadshift_d**. This is illustrated in figure 19.5. With regard to block partitioning the kernel is straightforward. The y-dimension of both block and grid should both be one while the x-dimension of the block times the x-dimension of the grid should just be large enough to cover all values in **gpadshift_d**. If the number of threads exceed the number of values in **gpadshift_d** these threads should do no work.

19.8 1D FFT

Host function name	Input	Input type	Output	Output type
FFT1_h	**x_d[·]**	**cufftComplex**	**X_d[·]**	**cufftComplex**
	gpadshift_d[·]	**cufftComplex**	**G_d[·]**	**cufftComplex**

19.8.1 Description

Performs Fourier transform of a 1-dimensional array of values. The 1D FFT is used two times in the CUDA implementation and both times a real to complex Fourier transform is calculated. The FFT is, however, calculated as a complex to complex transform as this arranges output data to be used without changes in the following blocks. The FFT used is from NVIDIAs CUFFT library which is based on FFTW (www.fftw.org). The FFT output is ordered starting with the DC component at index 0 and the normalized π frequency at the center index. **N** can be anything below 8 million elements, but the FFT algorithm performs best when **N** is a power of 2, 4, 8 or similarly small primes.

There is no need to make any considerations about memory management or execution configuration as this is all hidden in the library and already optimized. It is, however, neccesary to zero-pad the input array if an FFT length longer than **N** is needed.

19.9 Second order moment spectrum

Host function name	Input	Input type	Output	Output type
momSpec2_d	**X_d**[·]	**float2**	**M2xx_d**[·]	**float1**

19.9.1 Description

The second order moment spectrum is calculated by multiplying each element of the output of the Fourier transform of **x_d** with its conjugate. The input array is complex valued while the output array will be real valued. With the complex input array being represented as a **float2**

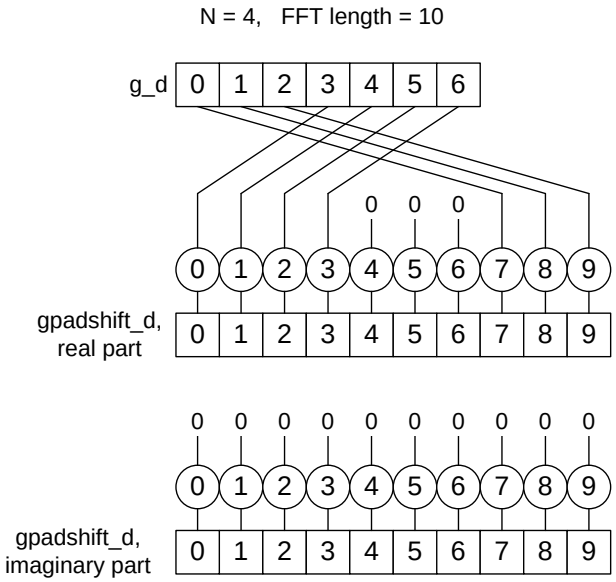


Figure 19.5: The operations of the **padShift_d** kernel.

type the output value can be calculated as the square of the real and imaginary part added together and divided by **N**. The kernel is designed such that one thread will load both real and imaginary parts of one element to shared memory, square them, add them together, divide by **N** and store the result in global memory. This is illustrated in figure 19.6. The block partitioning

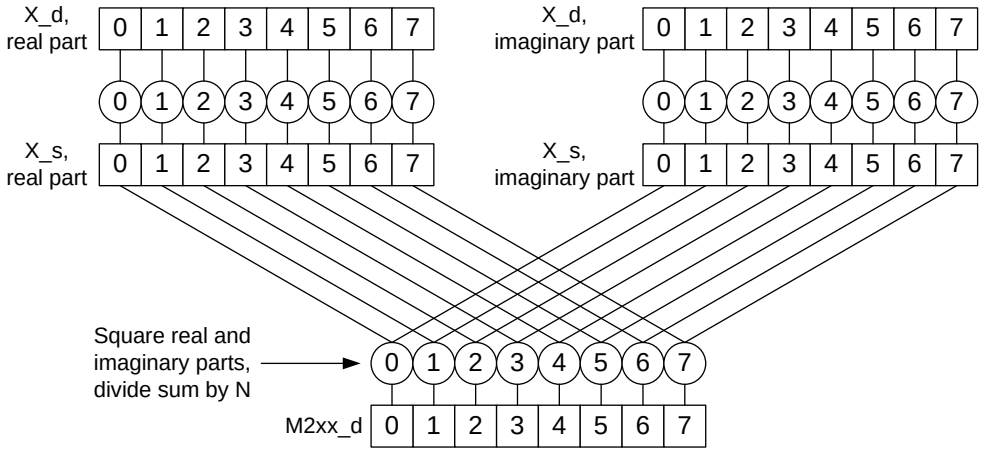


Figure 19.6: The operations of the **momSpec2_d** kernel.

is identical to the one used in the **padShift_d** kernel, i.e. grid and block y-dimension of one, x-dimension of grid times x-dimension of block should be large enough to cover all values and any threads exceeding the number of values should do no work.

19.10 Vector-vector convolution and matrix addition

Host function name	Input	Input type	Output	Output type
VVC_MA_d	**X_d**[·] **G_d**[·]	**float2** **float2**	**C4_d**[·]	**float1**

19.10.1 Description

Technically the vector-vector convolution is identical to a column vector-row vector multiplication yielding a matrix output. The entire expression for both blocks is the sum of two column vector-row vector multiplications, each one with the vectors swapped around, i.e. a*b' + b*a'. The sum is performed with the signs of both matrices inverted following the multiplications.

On a block level one thread is assigned to each element in the output matrix. Every thread performs the corresponding multiplications and adds together the products to obtain the final result in trispectrum slice, **C4_d**. The input vectors are loaded to shared memory by only the first column and row of threads in a thread block to avoid unnecessary loads.

On a grid level the dimensions of the blocks times the dimensions of the grid must cover all values of the output matrix and any threads exceeding the matrix size should do no work.

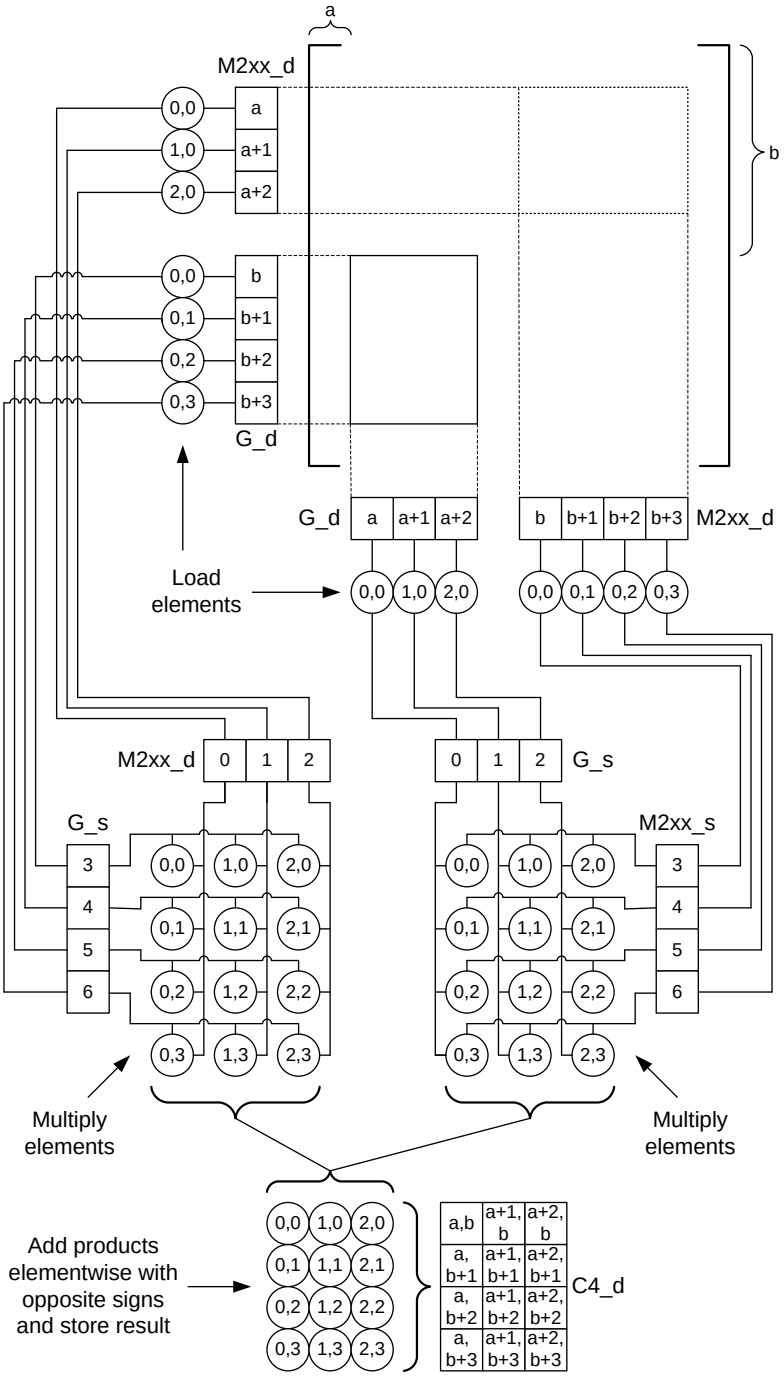


Figure 19.7: The operations of the **VVC_MA_d** kernel.

19.11 Verification of the implementation

To verify that the implemented kernels compute the correct output they are tested. A sequence of floating point numbers are generated in C and used as input vectors to the kernels. The outputs are compared to outputs from C implementations of the kernels to see if there are any differences. The **FFT1_h** block is not tested, as it is only a wrapper function to the NVIDIA CUFFT library kernel.

Each kernel is tested using several values of **N** and different execution configurations. The results can be seen in table 19.1 For all kernels the mean error is less than 10^{-7} which can be

Kernel:	**momSeq2_d**	**rowSum_d**	**padShift_d**	**momSpec2_d**	**VVC_MA_d**
Mean error:	$< 10^{-7}$	$< 10^{-7}$	0	$< 10^{-7}$	$< 10^{-6}$
Max error:	$< 10^{-6}$	$< 10^{-3}$	0	$< 10^{-6}$	$< 10^{-6}$

Table 19.1: Results from the verification of the implementation. The mean error between the output from each kernel and equivalent CPU implementations has been calculated and the maximum error has been found.

explained by rounding mode differences in the CUDA implementation and in C. The maximum error was less than 10^{-3} and found in the **rowSum_d** kernel. The maximum error scaled with **N**. This is due to the fact that the **rowSum_d** kernel accumulates a large amount of values and thus the error accumulates equivalently. Only the **padShift_d** produced a mean error of zero due to performing no arithmetic operations.

The combined implementation was also tested and compared to results from MATLAB. The input **x** and **w** vectors were of size $\mathbf{N} = 4096$ and generated by the **randn** function with mean value subtracted. The results can be seen in table 19.2. Comparing the single precision float-

Mean error	Mean square error	Max error
9.38·10⁻6	8.90·10⁻9	0.0307

Table 19.2: Results from the verification of the implementation. The mean error, mean square error and maximum error between MATLAB output and CUDA implementation output have been recorded.

ing point representation of the CUDA implementation to the double precision of MATLAB the mean and mean square errors are low. The maximum error can, however, be quite large due to rounding errors being accumulated in several steps. For further calculations this has small effect as long large errors are limited, which is the case as shown by the mean error.

19.12 Test of execution configuration and scalability

To show what impact the execution configuration can have on the performance of a kernel, each is tested using different block sizes and thus different grid sizes. To accurately measure the execution times a for loop has been inserted in each kernel such that data loads, arithmetic operations and data stores can be repeated an arbitrary number of times. This also means kernel launch overhead becomes less significant when the number of loops is large. Results for all the kernels can be found on the enclosed CD while only selected plots are shown in this

section. The plots can be seen in figures 19.8-19.10 For the **momSeq2_d**, **rowSum_d** and

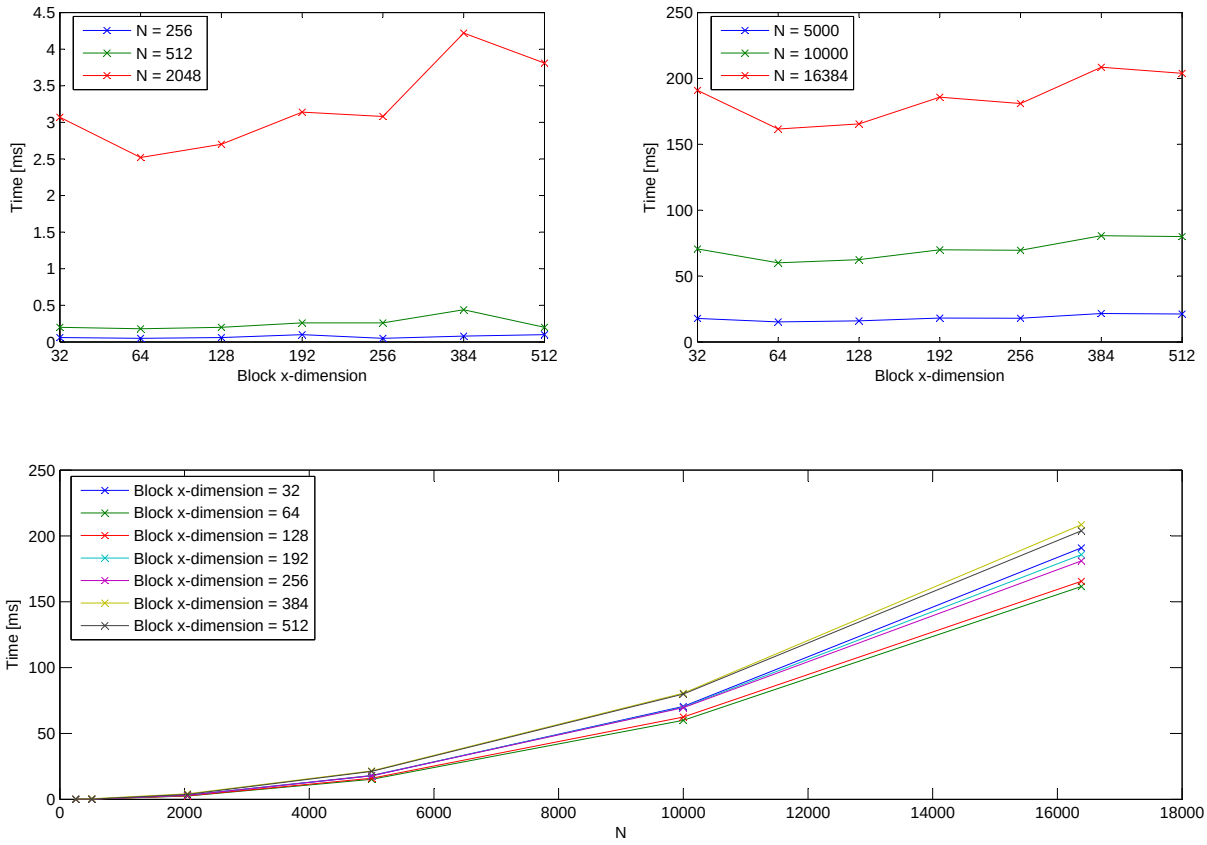


Figure 19.8: Plots of execution time as a function of block x-dimension and **N** for the **momSeq2_d** kernel.

padShift_d kernels the general tendency shows a block x-dimension of 64 to be the fastest and larger block dimensions to be slower. A block x-dimension of 32 proved to be slowest for the before mentioned kernels as well as the **momSpec2_d** kernel. This can be explained by the notion about registers performing best with block sizes that are multiples of 64 [7, p. 69]. The block sizes 192 and 384 showed worse performance than larger block sizes that would otherwise have been expected to be slower in the **momSeq2_d** and **rowSum_d** kernels. This is due to these block sizes requiring 0-insertion to maintain an even number of elements during the tree structure addition. For the **VVC_MA_d** kernel using a block size that had a small x-dimension dimension compared to the y-dimension executed considerably slower than when the opposite was true. When a smaller x-dimension is used the elements loaded from global memory are more spread out compared to using a larger x-dimension. This makes it harder for the GPU to coalesce memory accesses and as a result the execution time increases for blocks with smaller x-dimension

With regard to the scalability of the kernels it is easy to see that the **momSpec2_d** kernel scales linearly with **N**, which was also the case for the **padShift_d** kernel. For the **momSeq2_d** kernel the number of arithmetic operations are $N \cdot (2 \cdot \mathbf{N}\text{-}1) + (\mathbf{N}\text{-}1) \cdot (2 \cdot \mathbf{N}\text{-}1)$ translating into a complexity of $4 \cdot \mathbf{N}^2\text{-}4 \cdot \mathbf{N}$. It can be shown in MATLAB that the measured execution times for the **momSeq2_d** kernel are best approximated by second order polynomials, i.e. the kernel exhibits a square growth as a function of **N** as would be expected from the complexity. This behavior is also exhibited by the **rowSum_d** and **VVC_MA_d** kernels.

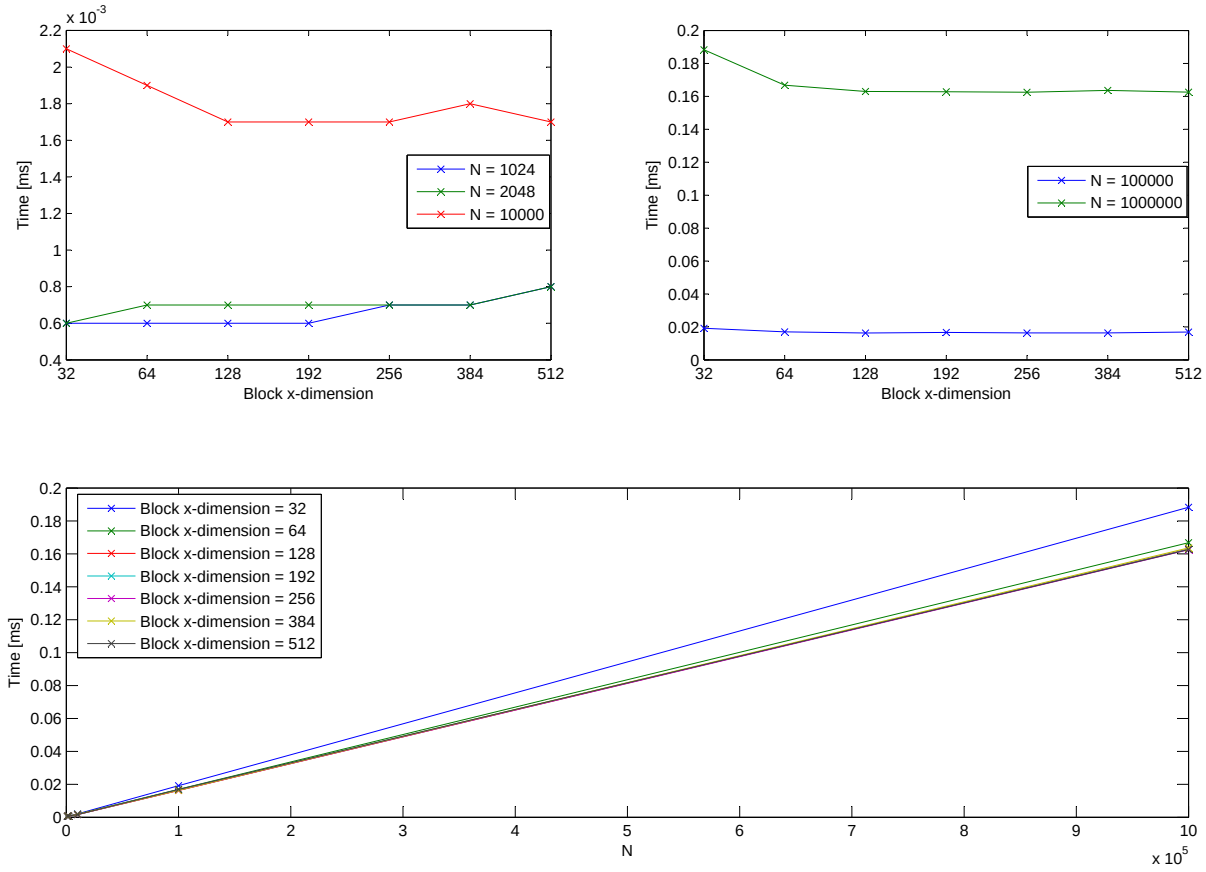


Figure 19.9: Plots of execution time as a function of block x-dimension and **N** for the **momSpec_d** kernel.

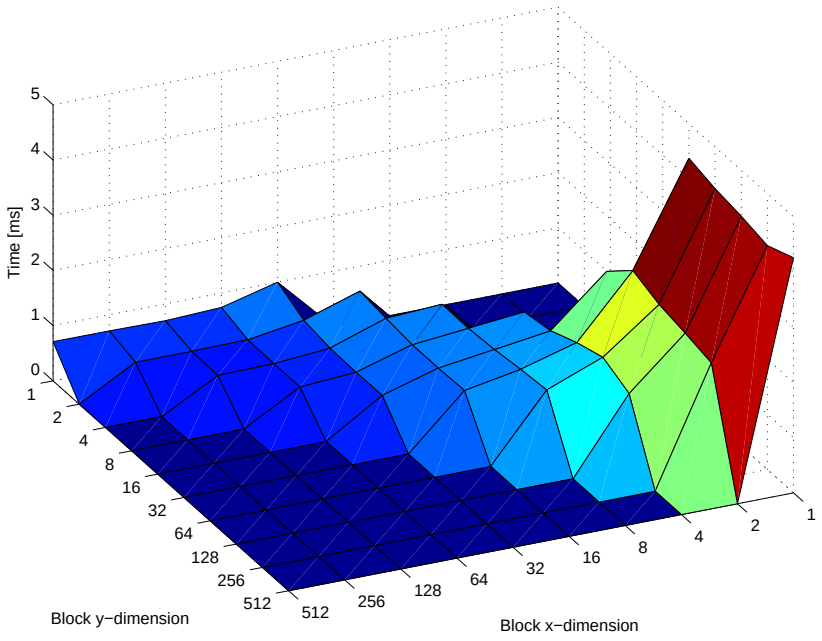


Figure 19.10: Plots of execution time as a function of block dimensions for the **momSeq2_d** kernel.

Chapter 20

Optimization of the CUDA implementation

In this chapter the **momSeq2_d** kernel described in section 19.5 is reviewed and optimized with regard to execution time. To optimize execution time there are two paramters to consider:

- FLOPS: If the kernel requires many operations per element loaded.
- Bandwidth: If the kernel requires few operations per element loaded.

For the **momSeq2_d** kernel there is **N** multiplications and **N**-1 additions per 2**N** (**x_d[n]**·**w_d[n]** + **x_d[n+1]**·**w_d[n+1]**) elements loaded. Even without including storing and loading intermediate results this indicates a low number of operations per element loaded, i.e. bandwidth is the bottleneck and should be the parameter to optimize.

After each step in the optimization the bandwidth and resulting speedup will be calculated. When calculating the bandwidth it is worth noting that storing and loading of intermediate results must be included. **N** = 16384 and 20 loops will be used for all execution time tests. The tests have been run with several block sizes to see if the speedup for each block size is significantly different.

20.1 Baseline implementation

In the baseline implementation shared memory bank conflicts are avoided as a result of two design choices. The first choice is performing the multiplication of **x_d** and **w_d** during the load in the first iteration and thus storing the **N**'th product in the **N**'th bank modulo 16. This means that each half-warp in the first loop of the tree structure will access two strides of 16 elements that will all be in different banks. The second choice is how the results of each step in the tree structure are stored. By having each half-warp read the lower half of the values added together

and store them at the same address the stride structure will be maintained and conflicts avoided. To illustrate the difference between this structure and a structure resulting in shared memory bank conflicts see figure 20.1. The bandwidth of the baseline implementation is shown in table

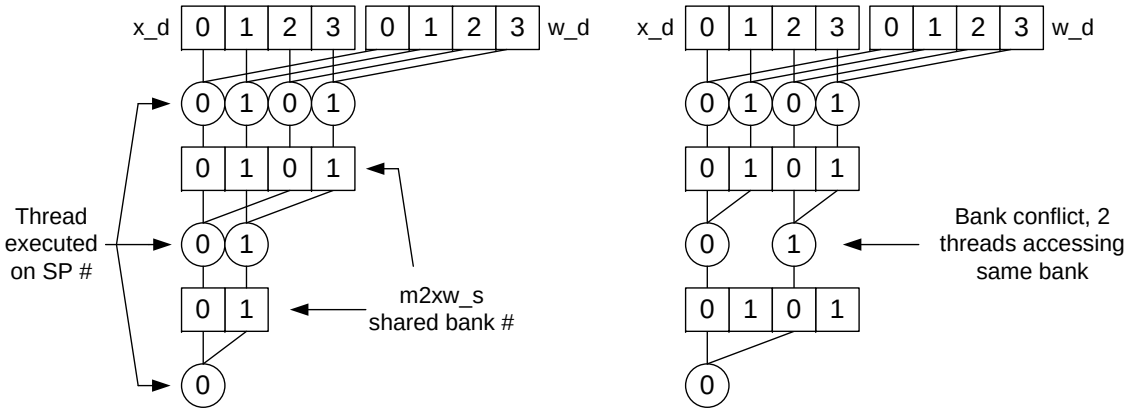


Figure 20.1: Tree structure and use of shared memory. For the shared memory the number shows which bank the element is stored in and for the threads the number shows what scalar processor the thread is executed on. For the sake of simplicity only two shared memory banks and scalar processors are assumed. Left figure shows the baseline implementation of **momSeq2_d** with no conflicts and right shows an example that would result in conflicts after the first step of the tree structure addition.

20.1.

Kernel	Block size	Time (ms)	Bandwidth (GiB/s)	Step speedup	Total speedup
Baseline	32	190.92	10.48		
	64	161.60	12.38		
	128	165.51	12.08		
	192	185.75	10.77		
	256	180.97	11.05		
	384	208.48	9.59		
	512	203.88	9.81		

Table 20.1: Bandwidth of the baseline version of the **momSeq2_d** kernel.

20.2 Unroll last warp

As explained in the design of the kernel it is required to synchronize after each loop of the tree structure addition and this was implemented in CUDA:

```
1 // Perform tree structure addition
2 for (s=blockDim.x; s>1; s>>=1){
3     // Pad with one zero if array is odd size
4     if ((s&1) == 1){
5         if (tx==(s>>1)) m2xw_s[s].x=0;
6         s++;
7     }
8     if (tx<(s>>1)) m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+(s>>1)].x;
```

```
9       __syncthreads();
10  }
```

Recalling that the SMs execute a warp at a time it is not necessary to synchronize the threads if there is less than one warp of threads doing work left. The ”synchronization” will happen automatically as the same warp of threads will be executed in each loop and thus executed in a sequential manner. This eliminates the need for an explicit thread synchronization and the loops for the last warp can also be unrolled:

```
1   // Perform tree structure addition
2   for (s=blockDim.x; s>64; s>>=1){
3       if (tx<(s>>1)) m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+(s>>1)].x;
4       __syncthreads();
5   }
6
7   if (blockDim.x>32) // Skip if N<64
8       if (tx<32)
9           m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+32].x;
10  if (tx < 32){
11      m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+16].x;
12      m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+8].x;
13      m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+4].x;
14      m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+2].x;
15      m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+1].x;
16  }
```

The zero-padding is still possible to perform in the unrolled loop, but not practical and is instead removed. This means block sizes that are not a power of two can not be used, but these block sizes have already been shown to be inferior in performance due to the zero-padding. The resulting bandwidth and speedups are shown in table 20.2.

Kernel	Block size	Time (ms)	Bandwidth (GiB/s)	Step speedup	Total speedup
Unrolled last warp	32	126.63	15.79	1.51x	1.51x
	64	92.60	21.60	1.64x	1.72x
	128	92.57	21.61	1.79x	1.79x
	256	105.35	18.98	1.72x	1.71x
	512	124.01	16.13	1.64x	1.64x

Table 20.2: Bandwidth and speedup of the **momSeq2_d** kernel after unroll of last warp in inner loop has been implemented.

20.3 Completely unrolled

Unrolling the last warp to and removing the thread synchronization yielding a speedup of as much 1.79x. Unrolling the remaining loops should decrease the execution time as well. There is, however, a problem with this approach as the loop is dependent on the block size which is not known at compile time. One way to solve this is by creating a kernel for each block size - in

C++ this is done by the use of templates:

```
template <unsigned int blockSize>
__global__ void momSeq2_d(float2 *px_d, float1 *pw_d, float1 *pm2xw_d, int iteration,
    int n, float invN)
```

```
// Perform tree structure addition
if (blockSize==512){
    if (tx < 256) m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+256].x;
    __syncthreads();
}
if (blockSize>=256){
    if (tx < 128) m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+128].x;
    __syncthreads();
}
if (blockSize>=128){
    if (tx < 64) m2xw_s[tx].x = m2xw_s[tx].x + m2xw_s[tx+64].x;
    __syncthreads();
}
if (tx < 32){
    if (blockSize>=64) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+32+ty*blockDim.x].x;
    if (blockSize>=32) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+16+ty*blockDim.x].x;
    if (blockSize>=16) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+8+ty*blockDim.x].x;
    if (blockSize>=8) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+4+ty*blockDim.x].x;
    if (blockSize>=4) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+2+ty*blockDim.x].x;
    if (blockSize>=2) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+1+ty*blockDim.x].x;}
```

All the **if** statements using the template parameter **blockSize** is then evaluated at compile time and the compiler will create a kernel for each value of the template parameter. For this to be possible the template parameter must be known at compile time and as such the calling process must be changed from:

```
// Second order moment sequence
iteration = 0;
dimBlock.x = 64;        dimBlock.y = 1;
                        dimGrid.y = 2*N-1;
for (n=N; n>1; n=(int)ceil((float)n/(float)dimBlock.x)){
    dimGrid.x = (int)ceil((float)n/(float)dimBlock.x);
    momSeq2_d <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d, pm2xw_d,
        iteration, n, 1.0/(float)N);
    iteration++;
}
```

To:

```
// Second order moment sequence
iteration = 0;
dimBlock.x = 64;        dimBlock.y = 1;
                        dimGrid.y = 2*N-1;
for (n=N; n>1; n=(int)ceil((float)n/(float)dimBlock.x)){
    dimGrid.x = (int)ceil((float)n/(float)dimBlock.x);
    if (dimBlock.x==32)
        momSeq2_d <32> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
            pm2xw_d, iteration, n, 1.0/(float)N);
```

```
9     if (dimBlock.x==64)
10        momSeq2_d <64> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
              pm2xw_d, iteration, n, 1.0/(float)N);
11    if (dimBlock.x==128)
12        momSeq2_d <128> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d,
              pw_d, pm2xw_d, iteration, n, 1.0/(float)N);
13    if (dimBlock.x==256)
14        momSeq2_d <256> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d,
              pw_d, pm2xw_d, iteration, n, 1.0/(float)N);
15    if (dimBlock.x==512)
16        momSeq2_d <512> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d,
              pw_d, pm2xw_d, iteration, n, 1.0/(float)N);
17    iteration++;
18 }
```

The resulting bandwidth and speedups are shown in table 20.3. Unlike unrolling the last warp

Kernel	Block size	Time (ms)	Bandwidth (GiB/s)	Step speedup	Total speedup
Completely unrolled	32	115.10	17.38	1.10x	1.66x
	64	88.88	22.50	1.04x	1.82x
	128	83.48	23.96	1.11x	1.98x
	256	90.49	22.10	1.16x	2.00x
	512	103.79	19.27	1.19x	1.96x

Table 20.3: Bandwidth and speedup of the **momSeq2_d** kernel after complete unroll of inner loop has been implemented.

the complete unrolling of the loop yields only a small decrease in execution time. This is due to the fact that the threads still have to be synchronized and this is the most time consuming part of the loop.

20.4 First add during load

For all the iterations following the first, one thread loads one element and one thread adds together two elements in the first step of the tree structure. This means half the threads are doing no work in the first loop! To utilize the threads a bit better the first add can be performed during the load, i.e. one thread loads two elements, adds them together and store them in shared memory. This changes the code from:

```
1  // Load x*w or intermediate results from global memory to shared memory
2  if (iteration == 0){
3      if (tx+bx*(int)blockDim.x>=n-1-by && tx+bx*blockDim.x<2*n-1-by && tx+bx*
           blockDim.x<n)
4          m2xw_s[tx].x = px_d[tx+bx*blockDim.x].x*pw_d[tx+bx*blockDim.x+by-(n-1)].x;
5      else m2xw_s[tx].x = 0;
6  }
7  else {
8      if (tx+bx*blockDim.x < n) m2xw_s[tx].x = pm2xw_d[tx+bx*blockDim.x+by*n].x;
9      else m2xw_s[tx].x = 0;
10 }
```

To:

```
// Load x*w or intermediate results from global memory to shared memory
if (iteration == 0){
    if ((int)(tx+bx*blockDim.x)>=n-1-by && tx+bx*blockDim.x<2*n-1-by && tx+bx*
        blockDim.x<n)
        m2xw_s[tx].x = px_d[tx+bx*blockDim.x].x*pw_d[tx+bx*blockDim.x+by-(n-1)].x;
    else m2xw_s[tx].x = 0;
}
else {
    if (tx+bx*blockDim.x*2+blockDim.x<n)
        m2xw_s[tx].x = pm2xw_d[tx+bx*blockDim.x*2+by*n].x + pm2xw_d[tx+bx*blockDim.x
            *2+blockDim.x+by*n].x;
    else if (tx+bx*blockDim.x*2<n)
        m2xw_s[tx].x = pm2xw_d[tx+bx*blockDim.x*2+by*n].x;
    else
        m2xw_s[tx].x = 0;
}
```

The resulting bandwidth and speedups are shown in table 20.4. While performing the first add

Kernel	Block size	Time (ms)	Bandwidth (GiB/s)	Step speedup	Total speedup
First add during load	32	109.53	18.26	1.05x	1.74x
	64	83.35	24.00	1.07x	1.92x
	128	78.57	25.46	1.06x	2.11x
	256	86.59	23.10	1.05x	2.09x
	512	100.65	19.87	1.03x	2.03x

Table 20.4: Bandwidth and speedup of the **momSeq2_d** kernel after first add during load has been implemented.

during the load does the decrease in execution time, the decrease is very small. This is due to the fact that most work is done in the first iteration and this optimization only affects the following iterations, where the number of elements to be added together has already been reduced by a factor equal to the block size.

20.5 Several adds during load

Using the block size with x-dimension equal to 128 yields a bandwidth of 24 GiB/s after the previous optimizations. This is far from the theoretical bandwidth of 148.06 GiB/s and an indication that something can be done much more efficiently.

Looking closely at the tree structure addition it becomes clear that this way of adding together the elements only proves to be the fastest solution, if all adds in each step of the tree can be performed in parallel. For the CUDA implementation this is not the case as each step is split into smaller blocks performing their own tree structure addition. This would, however, still be efficient if there was only as many blocks as there are SMs (possibly twice as many blocks as SMs to hide overhead) so they could all run in parallel. As there are 2·**N**-1 blocks in the y-dimension of the grid and several blocks in the x-dimension the number of blocks far exceeds the number of SMs resulting in an inefficient implementation. Furthermore in each block many

threads are idle due to the tree structure - only half the number of threads as there are elements left to add together can do any work.

One way to avoid these idle threads is by avoiding the tree structure until there are less elements than 2x the number of threads that can run in parallel. This can be achieved by letting all threads add together elements in a sequential manner and adding together the last elements using the tree structure as shown in figure 20.2. For the CUDA implementation this translates

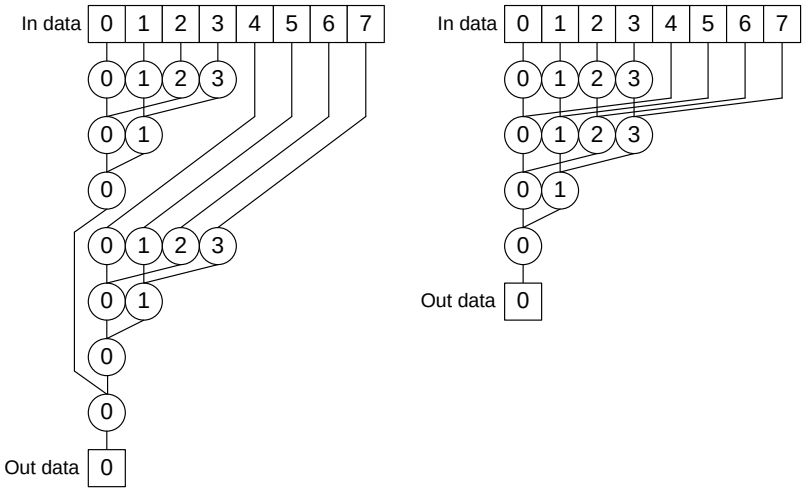


Figure 20.2: Left: Full tree structure addition using four threads. Because the eight elements have to be added together four elements at a time in the tree structure the utilization of some of the threads will be low. Right: Sequential addition of the elements until the tree structure addition can be performed resulting in a higher utilization of the threads and thus faster execution.

into using less blocks in the x-dimension, but each block performing a number of sequential adds before finishing with the tree structure addition. As there are already 2·**N**-1 blocks in the y-dimension, i.e. more than the number of SMs, the best utilization of each thread will occur with only one block in the x-dimension, such that the number of blocks is kept low. This way the number of blocks performing tree structure additions will be minimized while still having enough blocks to utilize all SMs. A side bonus to this solution is the elimination of any intermediate results and it will not be neccessary to call the kernel several times to add these together. This results in the following calling process code:

```
1  // Second order moment sequence
2  dimBlock.x = 64;          dimBlock.y = 1;
3  dimGrid.x  = 1;           dimGrid.y = 2*N-1;
4  if (dimBlock.x==32)
5      momSeq2_d <32> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
           pm2xw_d, n, 1.0/(float)N);
6  if (dimBlock.x==64)
7      momSeq2_d <64> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
           pm2xw_d, n, 1.0/(float)N);
8  if (dimBlock.x==128)
9      momSeq2_d <128> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
           pm2xw_d, n, 1.0/(float)N);
10 if (dimBlock.x==256)
11     momSeq2_d <256> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
           pm2xw_d, n, 1.0/(float)N);
```

```
12 if (dimBlock.x==512)
13     momSeq2_d <512> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
           pm2xw_d, n, 1.0/(float)N);
```

And the kernel code loading elements from global memory is replaced with:

```
1 // Load x*w from global memory and add to shared memory
2 m2xw_s[tx].x = 0;
3 for (int i=0; tx+i<n; i+=blockDim.x){
4     if (tx+i>=n-1-by && tx+i<2*n-1-by)
5         m2xw_s[tx].x += px_d[tx+i].x*pw_d[tx+i+by-(n-1)].x;
6 }
```

The resulting bandwidth and speedups are shown in table 20.5.

Kernel	Block size	Time (ms)	Bandwidth (GiB/s)	Step speedup	Total speedup
Several adds during load	32	33.27	60.11	3.29x	5.73x
	64	29.08	68.78	2.87x	5.56x
	128	29.08	68.78	2.70x	5.69x
	256	29.13	68.66	2.97x	6.21x
	512	29.93	66.82	3.36x	6.81x

Table 20.5: Bandwidth and speedup of the **momSeq2_d** kernel after severals adds during load have been implemented.

20.6 2-dimensional block size

As described in the previous kernel the tree structure addition is ineffective if not all blocks can run in parallel - even when performing sequential adds during load. As there are 2·**N**-1 blocks in the y-dimension of the grid there will be more blocks than SMs except for very small values of **N**. The number of blocks can be reduced by allowing the y-dimension of the blocks to be larger than one. The y-dimension of the grid will then be calculated from the y-dimension of the blocks in the calling proces:

```
1  // Second order moment sequence
2  dimBlock.x = 32;   dimBlock.y = 4;
3  dimGrid.x = 1;     dimGrid.y = (int)ceil((float)N/(float)dimBlock.y);
4  if (dimBlock.x==32)
5      momSeq2_d <32> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
           pm2xw_d, n, 1.0/(float)N);
6  if (dimBlock.x==64)
7      momSeq2_d <64> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
           pm2xw_d, n, 1.0/(float)N);
8  if (dimBlock.x==128)
9      momSeq2_d <128> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
           pm2xw_d, n, 1.0/(float)N);
10 if (dimBlock.x==256)
11     momSeq2_d <256> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
           pm2xw_d, n, 1.0/(float)N);
```

```
12 if (dimBlock.x==512)
13     momSeq2_d <512> <<<dimGrid, dimBlock, dimBlock.x*sizeof(float)>>> (px_d, pw_d,
           pm2xw_d, n, 1.0/(float)N);
```

While the basic structure of the kernel can be left unchanged the indexing throughtout the kernel must be adapted to the 2-dimensional block size:

```
1  // Load x*w from global memory and add to shared memory
2  m2xw_s[tx+ty*blockDim.x].x = 0;
3  for (int i=0; tx+i<n; i+=blockDim.x){
4      if (tx+i>=n-1-ty-by*(int)blockDim.y && tx+i<2*n-1-ty-by*blockDim.y)
5          m2xw_s[tx+ty*blockDim.x].x += px_d[tx+i].x*pw_d[tx+i+ty+by*blockDim.y-(n-1)].
               x;
6  }
7
8  __syncthreads();
9
10 // Perform tree structure addition
11 if (blockSize==512){
12     if (tx < 256) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+256+ty*blockDim.x].x;
13     __syncthreads();
14 }
15 if (blockSize>=256){
16     if (tx < 128) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+128+ty*blockDim.x].x;
17     __syncthreads();
18 }
19 if (blockSize>=128){
20     if (tx < 64) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+64+ty*blockDim.x].x;
21     __syncthreads();
22 }
23 if (tx < 32){
24     if (blockSize>=64) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+32+ty*blockDim.x].x;
25     if (blockSize>=32) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+16+ty*blockDim.x].x;
26     if (blockSize>=16) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+8+ty*blockDim.x].x;
27     if (blockSize>=8) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+4+ty*blockDim.x].x;
28     if (blockSize>=4) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+2+ty*blockDim.x].x;
29     if (blockSize>=2) m2xw_s[tx+ty*blockDim.x].x += m2xw_s[tx+1+ty*blockDim.x].x;
30 }
31
32 // Store m2xw in global memory
33 if (tx==0 && ty+by*blockDim.y<2*n-1) pm2xw_d[ty+by*blockDim.y].x = m2xw_s[ty*
       blockDim.x].x*invN;
```

As there are a multitude of combinations of x- and y-dimensions of the blocks only the slowest and fastest combination are listed in the table. The full table of results can be found on the enclosed CD. The resulting bandwidth and speedups are shown in table 20.6.

20.7 Coalesced memory access

As described in appendix B memory coalescing is needed when accessing global memory to maximize memory bandwidth. For devices of compute capability 1.2 and higher memory accesses will be coalesced when the addresses accessed lie within segments of 32, 64 or 128 bytes and aligned

Kernel	Block size	Time (ms)	Bandwidth (GiB/s)	Step speedup	Total speedup
2-dimensional block size	x=16 y=2	33.41	59.86	0.87x	4.84x
	x=4 y=32	20.96	95.42	1.39x	7.71x

Table 20.6: Bandwidth and speedup of the **momSeq2_d** kernel after 2-dimension block sizes have been implemented. The step and total speedup for the chosen block sizes are calculated compared to the fastest block sizes of the previous kernel and baseline kernel, respectively.

to these segments. The alignment is automatic if the built-in variables are used and memory is allocated using one of the allocation routines from the CUDA driver API, hence the use of **float2** for **x_d** and **float1** for **w_d** and **m2xw_d**. **x_d** is packed into the **float2** type due to it being used in the 1D FFT. This is, however, not optimal as this also means **x_d** will take up twice as much memory where the imaginary values are 0. For coalesced transfers bandwidth will be wasted on loading those zeros and using the **float1** type for **x_d** will increase memory bandwidth.

When accessing the **w_d** vector there is also coalescing problems due to the way **w_d** is indexed:

```
1 m2xw_s[tx+ty*blockDim.x].x += px_d[tx+i].x*pw_d[tx+i+ty+by*blockDim.y-(n-1)].x;
```

The **by*blockDim.y-(n-1)** part of the **w_d** index means that the part of **w_d** that is accessed for each output in **m2xw_d** will be very likely to be split between two memory segments and thus can not be coalesced.

For the sake of showing how high memory bandwidth can be attained, **x_d** is packed into the **float1** type and the above mentioned part of the **w_d** index removed. This will yield incorrect results and is not used in the actual implementation.

Using a block size with x-dimension = 8 and y-dimension = 16 yielded the lowest execution time of 16.67 ms. This translates into a memory bandwidth of 119.98 GiB/s which is 81% of the maximum theoretical bandwidth.

As there are still more blocks than multiprocessors there are still a number of idle threads in each block due to the tree structure addition. If the block partitioning or indexing could be done in a smarter way to thus utilize the last idle threads, it would be possible to push the bandwidth even closer to the theoretical max.

Chapter 21

Execution time test of the CUDA implementation

In this chapter the combined CUDA implementation is tested to determine the execution time with **N** = 128. This value is derived from the target application and used to determine if the implementation is fast enough for real time execution. Other reasonable values of **N** for the target application are 160 and 256 and the test is run for each of these as well. Each block in the implementation have also been tested individually to determine which take the longest to execute. The execution configuration yielding the lowest execution time is used for each block in the test and the optimized **momSeq2_d** kernel has been used in the test.

21.1 Results

The results from the test are listed in table 21.1 and plottet in figure 21.1. For **N** = 128 the

Block\ N	128	160	256
momSeq2_d (ms)	0.0034	0.0045	0.0107
rowSum_d (ms)	0.0213	0.0337	0.0799
padShift_d (ms)	0.0009	0.0009	0.0009
FFT1_h x2 (ms)	2·0.0122	2·0.0320	2·0.0154
momSpec2_d (ms)	0.0006	0.0006	0.0006
VVC_MA_d (ms)	0.0104	0.0144	0.0368
Sum (ms)	0.0610	0.1181	0.1597
Combined (ms)	0.0786	0.1699	0.2382

Table 21.1: Execution time results for both combined and block test of the CUDA implementation. **N** = 128, 160 and 256 has been used. The sum value is for the blocks tested one by one while the combined value is with overhead from kernel launch.

combined execution time is measured to 0.0786 ms and the overhead compared to running the blocks separately is 0.0176 ms. For higher **N** the overhead increases.

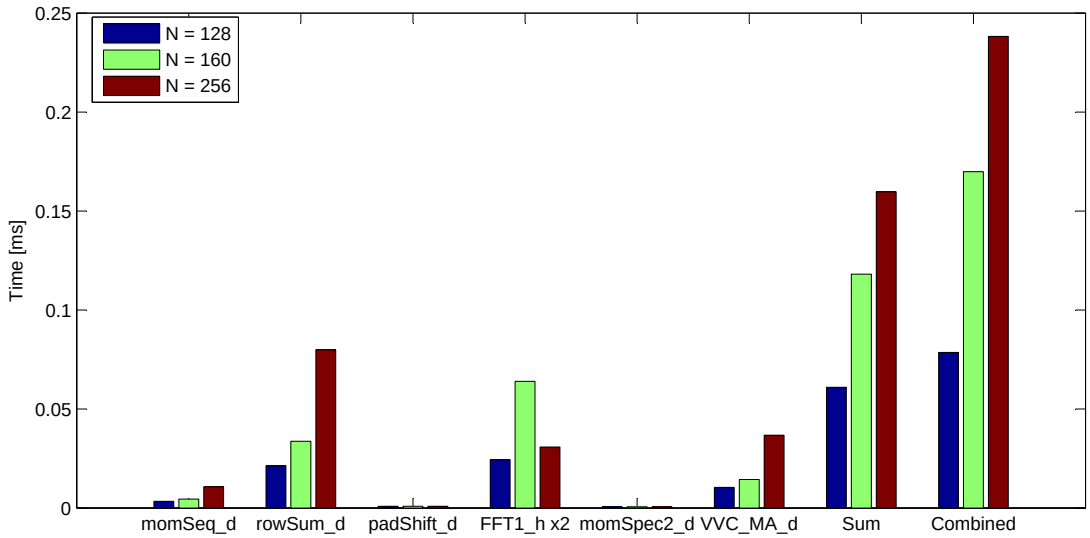


Figure 21.1: Plot of the results from combined and block test of the CUDA implementation.. The sum value is for the blocks tested one by one while the combined value is with overhead from kernel launch.

The most time consuming blocks are **rowSum_d**, **FFT1_h** and **VVC_MA_d** while the optimized **momSeq2_d** has been reduced to be less significant. The execution time for the **padShift_d** and **momSpec2_d** blocks are insignificant for all used values of **N**. For the **FFT1_h** block it is also worth nothing that it performs significantly worse for $\mathbf{N} = 160$ as this is not a power of two.

In section 17.13 it was shown that the number of arithmetic operations for calculating the center slice is $398 \cdot 10^3$ when $\mathbf{N} = 128$. With the combined execution time this translates into 5.06 GFLOPS.

Chapter 22

Conclusion

The intial complexity analysis gave a complexity of $5.2 \cdot 10^{12}$ arithmetric operations per filter update. If this is used for the reference application it would result in $130 \cdot 10^{12}$ arithmetric operations per second. The largest complexity was caused by the trispectra estimations which accounted for more then 99 % of the total complexity. If the application was implemented on the target platform without any modifications, a speedup of at least 130 times would be needed to run the application with a filter update rate of 25 in real time. In a practical case this number would be even higher, as the simplifications made when calculating the complexity have not been taken into account.

To reduce the complexity the part with the highest complexity, i.e. the trispectrum estimation, was examined. The total complexity for estimating one trispectrum using the direct method was $323 \cdot 10^9$ arithmetric operations. This included a smoothing over the entire spectrum. An optimization was performed on the center slice of the trispectrum, which is the only slice of the trispectrum that is used in the non-minimum phase filter estimation. In the optimization the smoothing was not intially included and the complexity could thus be reduced to $398 \cdot 10^3$ operations. However, as it is known that the smoothing is necessary, the complexity for calculating the neighbouring slices and performing the smoothing was found. The complexity for this was estimated to be $1.23 \cdot 10^9$ arithmetic operations or a reduction of 263 times compared to the unmodified algorithm.

In order to evaluate what number of FLOPS can be achieved in a practical case, the calculation of the center slice without smoothing was implemented. The calculation was split up into smaller blocks and a baseline implementation of each was done in the CUDA programming language. The overall structure of each block was described while the functionality of the block calculating the second order moment sequence was explained in detail through abstract code followed by an explanation of the CUDA code.

A verification of the blocks showed that rounding errors between the CUDA implementation and equivalent C implementations were less than 10^{-6} on average and the maximum error was less than 10^{-3}. Compared to MATLAB the error in the calculated center slice was smaller than 10^{-5} and 10^{-8} for mean and mean square error, respectively. The maximum error was 0.0307 which was acceptable due to the low mean and mean square error. From these numbers it was

concluded that the implementation works as expected.

As the implementation was a baseline implementation, several issues that impact performance was not considered. The block calculating the second order moment sequence was optimized to handle some of these issues and tested after each optimization. The final version showed a speedup up 7.71 times compared to the baseline implementation. It was also shown that a memory bandwidth usage of 81 % could be achieved.

After the optimization the implementation was tested to find the overall execution time for computing the center slice. When using parameters derived from the target application the measured execution time was 0.0768 ms. It was also shown that after optimization the second order moment sequence block did not take a significant amount of time to execute compared to three other blocks. As one of these blocks is an FFT function implemented by NVIDIA it can not be optimized, but if it is assumed that the two other blocks can be sped up by the same factor of 7.71 through optimization, the execution time would be lowered to 0.0334 ms.

With the estimated execution time and the number of arithmetic operations for computing the center slice without smoothing, the number of FLOPS is 11.916 GFLOPS. If it is assumed that the trispectra estimates in the $\bar{\bar{H}}(0)$ estimation can be reduced similarily in complexity, the total number of arithmetic operations for estimating the filters would be $10 \cdot 10^9$. This results in a filter update rate of around 1.19 per second.

This concludes the implementation of the algorithm.

Part V

Conclusion and Appendices

Chapter 23

Conclusion

This report is an investigation of blind source separation (BSS) on two signals mixed using a two input two output (TITO) mixing model and a study of real time implementation aspects of the BSS. The target application for the implementation is the cocktail party scenario, where multiple people are talking at the same time and the objective is to separate the voices of these people. This application is reduced to fit the TITO model so that only two people are talking and two sensors (microphones) are used. The BSS method works by estimating the two filters in the TITO model and inverting the model. An algorithm was developed based on previous work and the complexity evaluated. The part of the algorithm with the highest complexity was modified to lower the complexity and implemented on a platform using a graphics processing unit (GPU).

The theory behind the separation is based on the method presented in [8] for solving the separation problem. The method uses higher order statistics (HOS) which have a high computational complexity and thus poses a problem for a real time implementation. The method for the BSS only gives estimated bispectra of the filters in the TITO model. As such the filters had to be recovered from the bispectra. In [8] it is suggested to use the method presented in [4] to recover the filters and this was also done for this project. The BSS method presented in [8] focuses mostly on estimating the filters in the TITO model, while the inversion of the model is developed by the project group. If the filters in the TITO model are non-minimum phase the inverse model will be unstable. The developed method handles this by introducing delay filters in the inverse model to make the system stable.

To verify the BSS method it was split into blocks and an initial implementation and simulation of these blocks were made in MATLAB. A combined simulation of all the blocks showed that the BSS method worked and could achieve up to 10 dB signal to interference ratio (SIR) for the separated sources compared to the mixed signal with -5 dB SIR. This was comparable to results seen in [8]. The BSS method requires the DC-gain of the filters, $\bar{\bar{H}}(0)$, in the TITO model to be known and a method for doing this is also presented in [8]. However, simulations of this method showed that it required longer time to be estimated compared to the other blocks in the BSS. It was shown that if a better estimator can be found the achieved SIR will be up to 3 dB higher. Another observation in the simulations was that adding a positive offset to the correct $\bar{\bar{H}}(0)$ would produce better results. It was argued that a possible explanation to this problem was that the trispectra estimates can be improved. However, the results achieved are

relatively good compared to the results produced in most published methods for BSS, which show improvements in SIR between 3.1 and 21 dB.

The HOS part in the BSS consists of estimation of the trispectra which is used for both the $\bar{\bar{H}}(0)$ estimation and for the estimation of the bispectra of the filters. Two methods are described for computing the trispectrum estimation; the direct method and the indirect method. If the number of filter coefficients is close to the frame length used for the trispectrum, the direct method would be the preferred one to use. If the number of filter coefficient is small compared to the frame length then the indirect method should be used instead.

As the simulations of the BSS proved that the method produced acceptable results, it was suitable for implementation on a platform. A complexity analysis of the BSS method used for the simulation was conducted to identify the parts with the highest complexity. The analysis showed that more then 99 % of the arithmetic operations were used in estimating the trispectra, which supported the intial assumption of HOS being the part with the highest complexity.

With the target application in mind, the complexity analysis showed that target platform would be able to update the filters in the TITO model every 5.2 seconds if fully utilized. For the target application a filter update rate of 25 times per second is assumed and as such the complexity of the method was far too high for a real time implementation without any modifications.

In order to reduce the complexity of the trispectrum estimation the way the estimates are used in the algorithm were analyzed. The trispectrum estimation is used in both the $\bar{\bar{H}}(0)$ estimation and the filter estimation, but only the way that it is used in the filter estimation was analyzed. For the center trispectrum slice it was shown that the complexity could be reduced to $398 \cdot 10^3$ without smoothing, while it was estimated that the complexity could be reduced to $1.23 \cdot 10^9$ or by a factor of 263 with smoothing. For the implementation on the target platform it was chosen to implement only the estimation of the center slice of the trispectrum using no smoothing.

The GPU target platform used was an NVIDIA Geforce GTX 285. For high performance computation NVIDIA provides a framework called Compute Unified Device Architecture (CUDA) which is an extension to the C programming language. Through a description of the underlying hardware architecture it was shown that the the GPU is capable of massive parallel execution of threads.

A baseline implementation was written in CUDA and it was verified that each kernel computed the correct results with rounding errors compared to MATLAB and C. Following the verification one of the kernels was selected for optimization. Two optimizations principles gave the largest memory bandwidth speedups for the chosen kernel: Increasing the thread utilization and unrolling of the last warp. The step speedups achieved was at most a factor of 3.36 and 1.79, respectively. Compared to the baseline implementation the final optimized kernel yielded a memory bandwidth speedup factor of 7.71. With this speedup the memory bandwidth achieved was 65 % of the theoretical limit. It was also shown that 81 % could be achieved with a change to the indexing and data arrangement, however, this was not practical for the kernel implementation.

A final test of the combined implementation was conducted with the target application in mind

and the execution time for computing the center slice is measured to be 0.0786 ms. If it was assumed that a similar speedup could be achieved for the remaining kernels the execution time would be around 0.0334 ms, which equates to 11.916 GFLOPS. As the algorithm was estimated to require around $10 \cdot 10^9$ arithmetic operations for one filter update, the expected update rate would be 1.19 updates per second.

The intial goal was to investigate if it is possible to make a BSS on speech signals using HOS that can run in real time on an NVIDIA GPU. With the used platform platform and implementation it is not feasible to run the target application in real time, as the required update rate of the filters of 25 times per second can not be reached unless the implementation can be sped up by a factor of 21. However, the BSS method can still be used in other application areas where the requirements for the update rate of the filters are lower.

Chapter 24

Future work

Several topics discussed in this project can be developed or explored further. A few of these are summarized below:

The first thing would be to implement more parts of the BSS algorithm to further substantiate the claim that a speed up of around 18 times would indeed allow for real time execution on the platform. Also the requirement for a filter update rate of 25 per second should be further investigated. The requirement is established on what the project group found to be acceptable, but for a practical application the enviroment will have a large impact on the update rate. As the target application is a scenario involving speech signals, an acceptable degree of separation between the signals may also vary significantly from one listener to another. As such a more realistic estimate could be found, based on a number of tests with both different listeners and in different enviroments requiring different update rates, for instance a conference where people are mostly stationary for longer periods of time compared to the cocktail party scenario.

Another topic that is important for a practical application is the averaging over multiple estimates of the filters as well as $H(0)$. The simulations showed that the averaging was necessary to get good results and thus it must be implemented in the BSS model. This averaging can, for instance, be done by using a recursive filter. However, as the filters and $H(0)$ depend on the enviroment and how this changes, on could argue that it is possible to make a qualified guess on how the filters and $H(0)$ change. As such, it could be better to use a Kalman filter to predict the estimates based on previous values rather than the recursive filter. A proposed model of a system using Kalman filters can be seen in figure 24.1. As most system are not limited to just two sources it is relevent to extended the system to allow for more sources. This would result in a drop in performance and problems with uniquely identifying different sources would arise, but it would be necessary step to make the model useful for more application.

If the processing speed was drastically increased, a scenario where the filters are updated faster than it takes for a new frame of sampled data to arrive to the system would be possible. In this case it would be interesting to allow the filter and $H(0)$ estimation to use overlapping frames and examine how fast the estimates would converge to the correct values.

Another interesting result was the simulations that showed that the correct value of $H(0)$ does

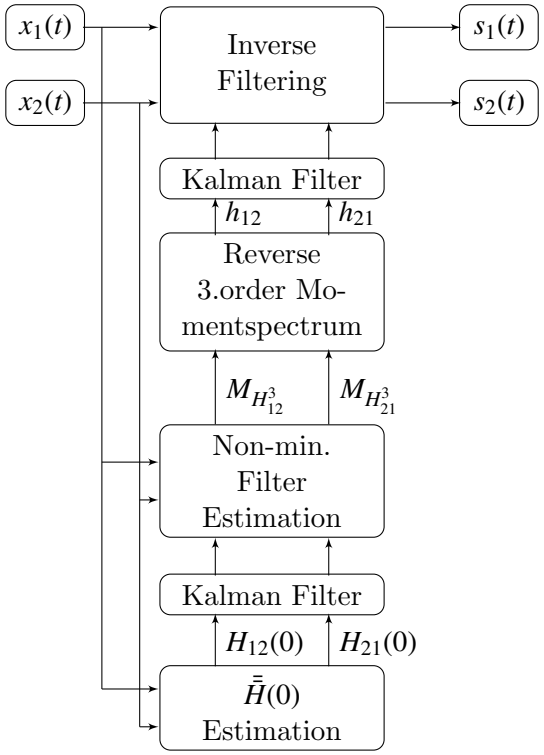


Figure 24.1: Illustration of a practical implementation of the BSS using Kalman filter to predict the estimates of the filters in the TITO model and $H(0)$ rather than averaging over several estimates.

not necessarily give the best SIR. Another way of finding $H(0)$ could be a system that iterates over $H(0)$ to find the value that yields the highest decorrelation between the sources. This method could also be used on other parts of the algorithm to improve the results, for instance the problem with shifts from the reverse third order moment spectrum estimation could be solved, if the decorrelation is calculated for several shift values and then picking the one that yields the best result.

Also other methods for doing the smoothing in the trispectrum estimation would be worth looking at. As the problem that the smoothing fixes is localized around DC, one could argue that only a smoothing around DC would be necessary. Improving this could also help bring down complexity for the trispectra estimation.

In general wireless system using time multiplexing in the frequency band might benefit from using BSS, as it would allow sources to communicate while occupying the same time slots and frequency bands. This would increase the throughput of the entire system as all users could communicate at full speed. Also in wireless communication BSS could be used to improve the SIR in a channel where the interference could be other sources or reflections of the signal. For instance sources with a slow communication type that are more robust towards interference, could be changed to a faster communication form as the SIR improves.

Bibliography

[1] Chrysostomos L. Nikias, Athina P. Petropulu [February 2005], *Higher-order spectra analysis: a nonlinear signal processing framework*, PTR Prentice Hall.

[2] David R. Martinez, Robert A. Bond, M. Michael Vai Bilge E. S. Akgul [2008], *High Performance Embedded Computing Handbook*, 1 edn, CRC Press.

[3] et al., Lenore Blum ... [1998], *Complexity and real computation*, 1 edn, Springer.

[4] Maria Rangousse, Georgios B. Giannakis [March 1991], *FIR Modeling Using Log-Bispectra: Weighted Least-Squares Algorithms and Performance Analysis*, IEEE.

[5] Michael Syskind Pedersen, Jan Larsen, Ulrik Kjems, and Lucas C. Parra [????], *A Survey of Convolutive Blind Source Seperation Methods*, 1 edn, Springer.

[6] Nikias, Chrysostomos L. and Jerry M. Mendel [1993], *Signal Processing with Higher-Order Spectra*, 1 edn, IEEE.

[7] NVIDIA Corporation [2008], *NVIDIA CUDA Programming Guide*, 2.1 edn, NVIDIA Corporation.

[8] S. Shamsunder, G. B. Giannakis [November 1997], *Multichannel Blind Signal Separation and Reconstruction*, IEEE.

[9] Shao, Jun [1999], *Mathematical Statistics*, Springer.

[10] Yellin, Daniel and Benjamin Friedlander [1996], *Blind Multi-Channel System Identification and Deconvolution: Performance Bounds*, 1 edn, IEEE.

Appendix A

Higher Order Statistics

This appendix contains a description Higher Order Statistics (HOS), the main area of interest are higher order spectra's, or polyspectra, which they are also known as. There is in general two different kinds of spectra when working with HOS:

Cumulant spectra: which are commonly used to analyze stochastic signals.
Moment spectra: which are commonly used to analyze deterministic signals.

The moment spectra's are mostly used when deterministic (known) signals are used, as moment spectra's maintain all information about the signal, so it is possible to reconstruct the original signal. If the signals are stochastic in nature normally a specific characteristic of the PDF is interest, like the mean or the variance e.g. Here the cumulant spectra is normally used as it only provide information about a specific characteristic in the PDF of the stochastic signal. There are two important features of HOS [1]:

- HOS conserves the true phase of the system as opposed to minimum phase in normal statistics.
- Gaussian noise is eliminated in HOS.

Using the normal Gaussian distribution as point of reference the polyspectra gives an indication of how much a given stochastic variable diverts from this distribution. In the following tchapters the theory behind HOS is presented as well as some examples about how it can be used in different scenarios. There are allot of different areas where HOS can be applied, but the main focus of this appendix is to clarify the mathematics behind HOS and give a deep enough insight into HOS to use in the practice, and understand the HOS theory used in the main report.

A.1 Moments and Cumulants

This section describes moments and cumulants, that are the main components in HOS. The moments and the cumulants are mainly used for describing the PDF of stochastic signals, they

can also be used for determining dependencies between stochastic signals by determining cross-moments or cross-cumulants. First the moments are defined for the continuous case as well as for the discreet case. Then the cumulants are presented for the continuous and the discreet case as well. The first step for defining the moments one must first look at the characteristic function for defining a PDF.

A.1.1 The Characteristic Function

Equation A.1 describes how to calculate the probability of some given outcomes from a stochastic variable x by using the cumulative density function (CDF) of this stochastic variable.

$$\Pr(a \leq x \leq b) = \int_a^b F(x) \tag{A.1}$$

Where:
$F(x)$ is the CDF of x
a, b is the interval of the outcome which probability is calculated.

If the derivative of the CDF exists then this can also be expressed as a function of the PDF of the stochastic variable x. The relation between the PDF and the CDF is described by equation A.2.

$$f(x) = \frac{d}{dx}F(x) \tag{A.2}$$

Where:
$f(x)$ is the PDF of the stochastic variable x.

Because of the relation between the CDF and PDF equation A.1 can rewritten as a function of the PDF as described by equation A.3.

$$\Pr(a \leq x \leq b) = \int_a^b f(x)\,\mathrm{d}\,x \tag{A.3}$$

The expectation of a function ($g(x)$) of x is defined as:

$$\mathrm{E}\left[g(x)\right] = \int_{-\infty}^{\infty} g(x) \cdot f(x)\,\mathrm{d}\,x \tag{A.4}$$

If $\varphi(g(x))$ is the characteristic function of $g(x)$ with a distribution function $f(x)$ defined as real numbers, then the expected value can be defined as:

$$\varphi(g(x)) = \mathrm{E}\left[e^{j \cdot g(x) \cdot x}\right] \tag{A.5}$$

For easier notation and because it is used as standard notation for indexes in the characteristic function $g(x)$ in equation A.5 is replaces by ω:

$$\varphi(\omega) = \mathrm{E}\left[e^{j \cdot \omega \cdot x}\right] = \int_{-\infty}^{\infty} e^{j \cdot \omega \cdot x} \cdot f(x)\,\mathrm{d}\,x \tag{A.6}$$

If equation A.6 is expanded into a Taylor series something interesting can be seen:

$$\begin{aligned}
\varphi(\omega) &= \int_{-\infty}^{\infty} f(x)\,\mathrm{d}\,x && \text{(A.7)}\\
&\quad + j\cdot\omega\int_{-\infty}^{\infty} x\cdot f(x)\,\mathrm{d}\,x && \text{(A.8)}\\
&\quad + \frac{1}{2}(j\cdot\omega)^2\cdot\int_{-\infty}^{\infty} x^2\cdot f(x)\,\mathrm{d}\,x && \text{(A.9)}\\
&\quad + \frac{1}{6}(j\cdot\omega)^3\ldots && \text{(A.10)}\\
&= \sum_{k=0}^{n}\frac{(j\cdot\omega)^k}{k!}\cdot\int_{-\infty}^{\infty} x^k\cdot f(x)\,\mathrm{d}\,x && \text{(A.11)}
\end{aligned}$$

The only things that changes in equation A.11 is the $\int_{-\infty}^{\infty} x^k \cdot f(x)\,\mathrm{d}\,x$ term, as it is dependent on the PDF of the random variable. The rest of the equation $\frac{(j\cdot\omega)^k}{k!}$ is constant regardless of which random variable is examinde.

If the first term in the Taylor series A.7 is examined, it would give a one. The second term A.8 in contains that mean of the PDF multiplied with $j\cdot\omega$. The third term A.9 contains the variance of the PDF multiplied with $\frac{1}{2}(j\cdot\omega)^2$. Which all (except the first term) gives a description of the PDF of random variable x.

What has been done is actually a kind of Fourier transform of the PDF, where information about the mean variance etc. has been extracted from a function of x. So by taking the inverse Fourier transform of $\varphi(\omega)$ the PDF is obtained as:

$$f(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-j\cdot\omega\cdot x}\cdot\varphi(\omega)\,\mathrm{d}\,\omega \tag{A.12}$$

To summerize what can be derived for the characteristic function in equation A.11. The following properties of the characteristic function hold true [9, 27-29].

1. If $\mathrm{E}[x]$ is finite, then $\left.\frac{\partial\varphi(\omega)}{\partial\omega}\right|_{\omega=0} = -j\,\mathrm{E}[x]$
2. If $\mathrm{Var}[x]$ is finite, then $\left.\frac{\partial^2\varphi(\omega)}{\partial^2\omega}\right|_{\omega=0} = -\,\mathrm{E}[x^2]$
3. In general $\mathrm{E}[x^p]$ is finite for a positive integer, then $\left.\frac{\partial^p\varphi(\omega)}{\partial^p\omega}\right|_{\omega=0} = (-1)^{\frac{p}{2}}\,\mathrm{E}[x^p]$

A.1.2 Moments

In the previous section it was show that the PDF can be approximated by the characteristic function equation A.11. The moments are defined from this characteristic function. The definition for the nth order moment of x is:

$$\mu'_{x^n} = (-j)^n\left.\frac{\partial^n\varphi_\omega}{\partial^n\omega}\right|_{\omega=0} \tag{A.13}$$

From equation A.13 it can be concluded that the moments are related to the characteristic function by the characteristic function real derivatives. So from the properties (3) for the characteristic function equation A.14 holds true for the moments.

$$\mu'_{x^n} = (-j)^n \left.\frac{\partial^n \varphi_\omega}{\partial^n \omega}\right|_{\omega=0} = \mathrm{E}[x^n] \tag{A.14}$$

The give an example the first order moment of the random variable x is calculated in equations A.15. Which should result in the mean of x.

$$\begin{aligned}
\mu'_{x^1} &= (-j) \left.\frac{\partial \varphi_\omega}{\partial \omega}\right|_{\omega=0} \\
&= (-j) \left.\frac{\partial \int_{-\infty}^{\infty} \left(e^{j\cdot\omega\cdot x}\right) \cdot f(x)\,dx}{\partial \omega}\right|_{\omega=0} \\
&= (-j) \int_{-\infty}^{\infty} \left.\frac{\partial \left(e^{j\cdot\omega\cdot x}\right)}{\partial \omega}\right|_{\omega=0} \cdot f(x)\,dx \\
&= (-j) \int_{-\infty}^{\infty} j\cdot x\cdot e^{j\cdot\omega\cdot x}\Big|_{\omega=0} \cdot f(x)\,dx \\
&= (-j) \int_{-\infty}^{\infty} j\cdot x\cdot f(x)\,dx \\
&= \int_{-\infty}^{\infty} x\cdot f(x)\,dx = \mathrm{E}[x]
\end{aligned} \tag{A.15}$$

Which is in accordance with equation A.14. These calculations for the first order moment can be extended to the n-th order moment giving the general function as:

$$\mu'_{x^n} = \int_{-\infty}^{\infty} x^n \cdot f(x)\,dx = \mathrm{E}\left[x^n\right] \tag{A.16}$$

Equation A.16 describes all the moments op to order n. But it should be noted that it is not always necessary to have infinitely high order to describe a PDF. The maximum order of moments needed to describe a random variable is given by equation A.17.

$$\mathrm{E}\left[|x|^n\right] = \int_{-\infty}^{\infty} |x^n| \cdot f(x)\,dx \tag{A.17}$$

If equation A.17 converges to a value then the moment at that given order exists and all the moment of a lesser order then n also exist. On the other hand if it converges to infinite the moment of that given order does not exist, and all moments higher the n does not exist either as they would include lower order moments.

Equation A.6 is normally rewritten so it does not contain the imaginary part, this gives equation A.18, which is referred to as the moment generating function (MGF) for a random variable x and is normally used to derive the moments of x.

$$M_{x^n}(\omega) = \int_{-\infty}^{\infty} e^{x\cdot\omega} \cdot f(x)\,dx \tag{A.18}$$

This can be proven by a Taylor expansion of the MGF function, with the same procedure used to get to equation A.11:

$$M_{x^n}(\omega) = \int_{-\infty}^{\infty} \left(1 + x\cdot\omega + \frac{x^2\,\omega^2}{2!} + \cdots + \frac{x^n\,\omega^n}{n!}\right) f(x)\,dx \tag{A.19}$$

By expanding the integral using equation A.16 for the moments the following can be obtained:

$$M_{x^n}(\omega) = 1 + \mu_1' \cdot \omega + \frac{\mu_2' \cdot \omega^2}{2!} + \cdots = 1 + \sum_{n=1}^{\infty} \frac{\mu_n' \omega^n}{n!} \tag{A.20}$$

The moment generating function contains all the moments from 1 to n. If the moment generating function is differentiated once with regards to ω and setting ω to zero the result would be the first moment, differentiating a second time would give the second moment, etc. Which was the same as for the characteristic function but the complex part is removed from the equations.

A.1.3 Central Moment

The central moment is mainly defined as it makes the computations of the cumulants easier. The idea is to remove the mean from the random variable x, before the moments are calculated.

The central moment is defined as:

$$\mu_{x^n} = \mathrm{E}\left[(x - \mu_{x^1}')^n\right] \tag{A.21}$$

The first order central moment is always zero as:

$$\begin{aligned} \mu_{x^1} &= \mathrm{E}\left[(x - \mu_{x^1}')^1\right] \\ &= \mathrm{E}[x] - \mathrm{E}\left[\mu_{x^1}'\right] \\ &= \mu_{x^1}' - \mu_{x^1}' = 0 \end{aligned} \tag{A.22}$$

The second order central moment is the same as the variance of the random variable x:

$$\begin{aligned} \mu_{x^2} &= \mathrm{E}\left[(x - \mu_{x^1}')^2\right] \\ &= \mathrm{E}\left[x^2 + {\mu_{x^1}'}^2 - 2x\mu_{x^1}'\right] \\ &= \mathrm{E}\left[x^2\right] + {\mu_{x^1}'}^2 \end{aligned} \tag{A.23}$$

The third order central moment gives information about the ”skewness” of the PDF of x, if this moment is zero the pdf would have a normal Gaussian distribution. Therefore the third order central moment gives information about the skewness deviation from a normal distribution as illustrated in figure A.1

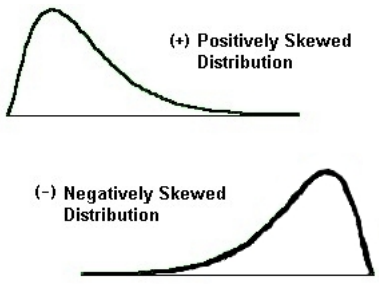


Figure A.1: Illustration of a PDF with positive skewness and negative skewness

It should be noted here that every distribution that is symmetrical e.g uniform distribution would have an identically skewness as a normal Gaussian distribution.

The fourth order central moment describes the "Kurtosis" of the PDF for x, this a measure for how spiky or flatter a distribution is compared to a normal distribution. There are a disagreement in the literature whether the Kurtosis is calculated via the central moments or whether is calculated via cumulants, the difference between these two calculations are 3, which means that a normal Gaussian distribution with a variance of one, would have a kurtosis of 3 if calculated via central moments, or it would have a kurtosis of 0 if it was calculated via its cumulants. There are also some terms connected with the kurtosis if the it is lower then the normal distribution. It is referred to as having sub Gaussian kurtosis. If it has a higher kurtosis, it is referred to as super Gaussian. Figure A.2 illustrates how the deviation is from super and sub Gaussian distributions [1].

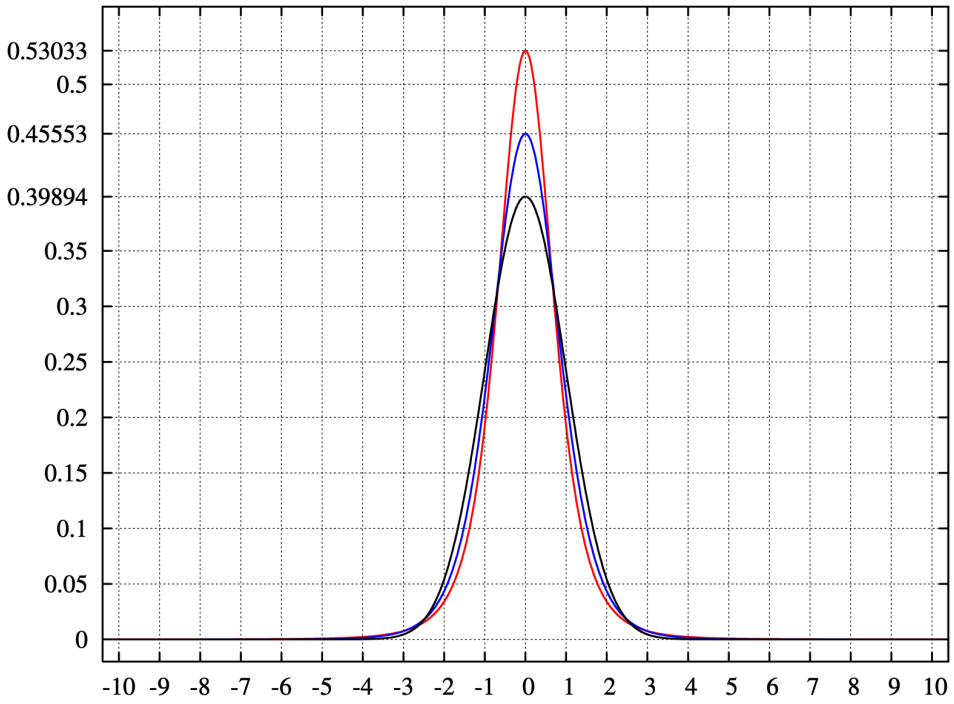


Figure A.2: Illustration of a super Gaussian and a sub Gaussian PDF

A.1.4 Joint Moments

Until now the focus has been on the PDF from one random variable, but is it also interesting to look on the joint PDF between two random variables. The previous theory also holds for cross moments to describe this joint PDF. The joint moment between the second order moment of two random variables, actually gives the cross correlation between these two random variables. If given a two random variables x and y. Their joint moment for an order of n would be given by:

$$\mu'_{x^c\,y^z} = \mathrm{E}\left[x^c \cdot y^z\right] = \int_{-\infty}^{+\infty} x^c \cdot y^z \cdot f(x) \cdot f(y)\ \mathrm{d}\,x\ \mathrm{d}\,y \tag{A.24}$$

where: $n = c + z$

The joint moment can also be written as a function of the MGF.

$$\begin{aligned}\mu'_{x^c y^z} &= \left.\frac{\partial^c M_x(\omega_1)\,\partial^z M_y(\omega_2)}{\partial^c \omega_1\,\partial^z \omega_2}\right|_{\omega_1=\omega_2=0} \\ &= \left.\frac{\partial^n \mathrm{E}\left[e^{\omega_1\cdot x+\omega_2\cdot y}\right]}{\partial^c \omega_1\,\partial^z \omega_2}\right|_{\omega_1=\omega_2=0}\end{aligned} \tag{A.25}$$

This can be extended to k random variables $\{x_1, x_2, ..., x_k\}$ of order $n = c_1 + c_2 + \ldots + c_k$

$$\mu'_{x_1^{c_1}\,x_2^{c_2}\,\ldots\,x_k^{c_k}} = \left.\frac{\partial^n \mathrm{E}\left[e^{\omega_1\cdot x_1+\omega_2\cdot x_2+\cdots+\omega_k\cdot x_k}\right]}{\partial^{c_1}\omega_1\,\partial^{c_2}\omega_2\,\cdots\,\partial^{c_k}\omega_k}\right|_{\omega_1=\omega_2=\ldots=\omega_k=0} \tag{A.26}$$

A.1.5 Discrete Moments

As the most DSP system cannot work with continuous system it necessary to work with discrete systems instead. The moments therefore needs to be defined for the discrete case as well in order to use HOS on a DSP system. If $x(t)$ is observations at time indexes $t = 0, \pm 1, \pm 2 \cdots$ from a real stationary random process x with moments up to the order of n, then equation A.16 would only depend on the time difference τ

$$\mu'_{x(t)\,x(t+\tau_1)\,\ldots\,x(t+\tau_{n-1})} = \mathrm{E}\left[x(t)\cdot x(t+\tau_1)\cdots x(t+\tau_{n-1})\right] \tag{A.27}$$

It is therefore possible to write the moments of a random stationary process in the discreet domain as:

$$m'_{x^n}(\tau_1,\ldots,\tau_{n-1}) = \mathrm{E}\left[x(t)\cdot x(t+\tau_1)\cdots x(t+\tau_{n-1})\right] \tag{A.28}$$

The n'th order discreet moment of a real signal $x(t)$ can from A.16 be written as.

$$m'_{x^n}(\tau_1,\ldots,\tau_{n-1}) = \sum_{t=-\infty}^{\infty} x(t)\cdot x(t+\tau_1)\cdots x(t+\tau_{n-1}) \tag{A.29}$$

If the mean is subtracted from $x(t)$ (central moment), then the second order central moment becomes the discrete autocorrelation function of x.

$$m_{x^2}(\tau) = \sum_{t=-\infty}^{\infty} x(t)\cdot x(t+\tau) \tag{A.30}$$

In general it is not practical to sum from minus infinity to infinity. However if the random variable x is ergodic in the most general form and all moments can be determined from a single observation sequence $x(t)$, then the expected value in equation A.28 can be replaced by a time averaging.

$$m'_{x^n}(\tau_1,\cdots,\tau_{n-1}) = \lim_{M\to\inf}\frac{1}{(M-(-M))+1}\sum_{t=-M}^{M} x(t)\cdot x(t+\tau_1)\cdots x(t\tau_{n-1}) \tag{A.31}$$

If a sample size of N is available from $x(t)$ then an biased estimate of equation A.31 can be made as:

$$\begin{aligned}m'_{x^n}(\tau_1,\cdots,\tau_{n-1}) &= \frac{1}{(N-(1))+1}\sum_{t=1}^{N} x(t)\cdot x(t+\tau_1)\cdots x(t\tau_{n-1}) \\ &= \frac{1}{N}\sum_{t=1}^{N} x(t)\cdot x(t+\tau_1)\cdots x(t+\tau_{n-1})\end{aligned} \tag{A.32}$$

A.1.6 Discrete Joint Moments

The discrete joint moments between random variables can be made in the same way as for the discrete moments. The same conditions that the random variables must be ergodic also applies. The n'th order joint central moments of the real signals $x(t)$ and $y(t)$ is defined as:

$$m_{x^c y^z}(\tau_1, \ldots, \tau_{n-1}) = \sum_{t=-\infty}^{\infty} x(t) \cdots x(t+\tau_{c-1}) \cdot y(t+\tau_c) \cdots y(t+\tau_{n-1}) \tag{A.33}$$

where:
$n = c + z$

As in the discrete moments it also possible to make a biased estimation of the moment from a sample set of size N:

$$m_{x^c y^z}(\tau_1, \cdots, \tau_{n-1}) = \frac{1}{N} \sum_{t=0}^{N} x(t) \cdots x(t+\tau_{c-1}) \cdot y(t+\tau_c) \cdots y(t+\tau_{n-1}) \tag{A.34}$$

A.1.7 Cumulants

At a normal (Gaussian) distribution all the information about it is contained in the first two moments. But this does not mean that the higher moments are zero all though they don't contains any new information. This is because that higher order moments actually contains information about the lower order moments as well. In order to remove this dependencies with lower order moments the cumulants are introduced. From this a cumulant of order n is a sum moments from order of n to zero. So if a cumulant generating function (CGF) is defined as for the moments each cumulant would be a sum of lower order moments. This would lead to a exponential relation between the MGF and the CGF, do from this the following relations between the MGF and CGF is defined as:

$$G_{x^n}(\omega) = \ln\left(M_{x^n}(\omega)\right) \tag{A.35}$$

The CGF is defined in the same way as the MGF, as a sum of cumulants at different order up to the order n:

$$G_{x^n}(\omega) = \sum_{n=1}^{\infty} \kappa_n \frac{\omega^n}{n!} \tag{A.36}$$

where:
κ_n is the cumulant of order n

Equation A.35 can from equation A.36 can be rewritten as:

$$M_{x^n}(\omega) = \mathrm{e}^{G(\omega)} = 1 + \sum_{n=1}^{\infty} \frac{\mu'_n \omega^n}{n!} = \mathrm{e}^{\sum_{n=1}^{\infty} \frac{\kappa_n \omega^n}{n!}} \tag{A.37}$$

To find the relations between moments and cumulants the last two terms in equation A.37 is explored by making a Taylor expansion of the last term.

$$\begin{aligned}
1+\sum_{n=1}^{\infty}\frac{\mu'_n\omega^n}{n!} &= e^{\sum_{n=1}^{\infty}\frac{\kappa_n\omega^n}{n!}} \\
&= \sum_{i=0}^{\infty}\frac{\left(\sum_{n=1}^{\infty}\frac{\kappa_n\omega^n}{n!}\right)^i}{i!} \\
&= 1+\sum_{n=1}^{\infty}\frac{\kappa_n\omega^n}{n!}+\left(\frac{1}{2}\sum_{n=1}^{\infty}\frac{\kappa_n\omega^n}{n!}\right)^2+\left(\frac{1}{6}\sum_{n=1}^{\infty}\frac{\kappa_n\omega^n}{n!}\right)^3+\dots \\
&= 1+\left(\kappa_1\cdot\omega+\kappa_2\cdot\frac{\omega^2}{2}+\kappa_3\cdot\frac{\omega^3}{6}+\dots\right) \\
&\quad +\frac{1}{2}\left(\kappa_1\cdot\omega+\kappa_2\cdot\frac{\omega^2}{2}+\kappa_3\cdot\frac{\omega^3}{6}+\dots\right)^2 \\
&\quad +\frac{1}{6}\left(\kappa_1\cdot\omega+\kappa_2\cdot\frac{\omega^2}{2}+\kappa_3\cdot\frac{\omega^3}{6}+\dots\right)^3+\dots
\end{aligned} \tag{A.38}$$

To find the first order moment the MGF is differentiated once with regard to ω

$$\mu'_1=\left.\frac{\partial M_{x^1}(\omega)}{\partial\omega}\right|_{\omega=0}=\left.\frac{\partial 1+\sum_{n=1}^{\infty}\frac{\mu'_n\omega^n}{n!}}{\partial\omega}\right|_{\omega=0} \tag{A.39}$$

If equation A.38 is differentiated once as in A.39 the only terms that survies is:

$$\mu'_1 = \kappa_1 \tag{A.40}$$

If equation A.38 is differentiated two, three and four times gives the following relations between the moments and the cumulants.

$$\mu'_2 = \kappa_2+\kappa_1^2 \tag{A.41}$$
$$\mu'_3 = \kappa_3+3\kappa_2\kappa_1+\kappa_1^3 \tag{A.42}$$
$$\mu'_4 = \kappa_4+4\kappa_3\kappa_1+3\kappa_2^2+6\kappa_2\kappa_1^2+\kappa_1^4 \tag{A.43}$$

From equations A.40 to A.43 it is possible to create equation A.44, which describes the relation between cumulants and moments.

$$\kappa_n=\mu'_n-\sum_{k=1}^{n-1}\binom{n-1}{k-1}\kappa_k\mu'_{n-k} \tag{A.44}$$

Replacing the moment in equations A.40 to A.43 with the central moment reduces the equations to the following

$$\mu_1 = 0 \tag{A.45}$$
$$\mu_2 = \kappa_2 \tag{A.46}$$
$$\mu_3 = \kappa_3 \tag{A.47}$$
$$\mu_4 = \kappa_4+3\cdot\kappa_2^2 \tag{A.48}$$

From this it can be concluded that κ_1 is the mean value, κ_2 and κ_3 is the variance and the "Skewness" respectively, as was the same for the central moments in section A.1.3 The fourth order cumulant minus the variance this is sometimes also referred to as the normalized kurtosis, but there are some disagreement in the literature whether the kurtosis is the fourth order cumulant or the fourth order central moment.

Discrete Cumulants

The relations between the moments and the cumulants described in equation A.40 to A.43 is used to calculate the discrete cumulants from the discrete moments.

For this project only cumulants up to the 4th order are of interest therefore they are the only ones described here. And because it makes the calculations easier the central moments are used as described in the relation in equations A.45 to A.48

The first order cumulants are the same as the first order moment so:

$$c_x = m'_x \tag{A.49}$$

As described before the cumulants are calculated from the central moment therefore the cumulants can be written in the following way using the expected operator.

$$\begin{aligned} c_{x^2}(\tau_1) &= \mathrm{E}\left[(x(t) - m'_x) \cdot (x(t-\tau_1) - m'_x)\right] \\ &= \mathrm{E}\left[x(t) \cdot x(t-\tau_1)\right] - \mathrm{E}\left[x(t)\right] \cdot m'_x - \mathrm{E}\left[x(t-\tau_1)\right] \cdot m'_x + m'_x \cdot m'_x \\ &= m'_{x^2} - (m'_x)^2 \\ &= m_{x^2}(\tau_1) \end{aligned} \tag{A.50}$$

The third order discreet cumulants can from the same procedure as equation A.50 be established as:

$$\begin{aligned} c_{x^3}(\tau_1, \tau_2) &= \mathrm{E}\left[(x(t) - m'_x) \cdot (x(t-\tau_1) - m'_x) \cdot (x(t-\tau_2) - m'_x)\right] \\ &= \mathrm{E}\left[x(t) \cdot x(t-\tau_1) \cdot x(t-\tau_2)\right] - \mathrm{E}\left[x(t) \cdot x(t-\tau_1)\right] \cdot m'_x - \\ &\quad \mathrm{E}\left[x(t) \cdot x(t-\tau_2)\right] \cdot m'_x + \mathrm{E}\left[x(t)\right] \cdot {m'_x}^2 + \mathrm{E}\left[x(t-\tau_1)\right] {m'_x}^2 + \\ &\quad \mathrm{E}\left[x(t-\tau_2)\right] \cdot {m'_x}^2 + {m'_x}^3 \\ &= m'_{x^3}(\tau_1, \tau_2) - m'_x \cdot \left(m'_{x^2}(\tau_1) + m'_{x^2}(\tau_2) + m'_{x^2}(\tau_2 - \tau_1)\right) + 2 \cdot (m'_x)^3 \\ &= m_{x^3}(\tau_1, \tau_2) \end{aligned} \tag{A.51}$$

The last conversion of discreet moments to cumulants of interest is the fourth order cumulants, using the same procedure as in the previous examples the following result can be obtained.

$$
\begin{aligned}
c_{x^4}(\tau_1,\tau_2,\tau_3) &= \mathrm{E}\left[(x(t)-m'_x)\cdot(x(t-\tau_1)-m'_x)\cdot(x(t-\tau_2)-m'_x)\cdot(x(t-\tau_3)-m'_x)\right] \\
&\Downarrow \\
&= m'_{x^4}(\tau_1,\tau_2,\tau_3) - m'_{x^2}(\tau_1)\cdot m'_{x^2}(\tau_3-\tau_2) - m'_{x^2}(\tau_2)\cdot m'_{x^2}(\tau_3-\tau_1) \\
&\quad -m'_{x^2}(\tau_3)\cdot m'_{x^2}(\tau_2-\tau_1) - m'_x\cdot m'_{x^3}(\tau_2-\tau_1,\tau_3-\tau_1) - m'_x\cdot m'_{x^3}(\tau_2,\tau_1) \\
&\quad -m'_x\cdot m'_{x^3}(\tau_1,\tau_2) - m'_x\cdot m'_{x^3}(\tau_2,\tau_3) + {m'_x}^2\cdot m'_{x^2}(\tau_1) \\
&\quad +{m'_x}^2\cdot m'_{x^2}(\tau_2) + {m'_x}^2\cdot m'_{x^2}(\tau_3) + {m'_x}^2\cdot m'_{x^2}(\tau_3-\tau_1) \\
&\quad +{m'_x}^2\cdot m'_{x^2}(\tau_3-\tau_2) + {m'_x}^2\cdot m'_{x^2}(\tau_2-\tau_1) + 6\cdot {m'_x}^4 \\
&= m_{x^4}(\tau_1,\tau_2,\tau_3) - m_{x^2}(\tau_1)\cdot m_{x^2}(\tau_3-\tau_2) \\
&\quad -m_{x^2}(\tau_2)\cdot m_{x^2}(\tau_3-\tau_1) - m_{x^2}(\tau_3)\cdot m_{x^2}(\tau_2-\tau_1)
\end{aligned} \tag{A.52}
$$

This concludes how to transform the discreet moments into the discreet cumulants the next step is to do the same for joint cumulants between two random variables.

A.1.8 Joint Cumulants

This is more less the function of the Joint MGF where the logarithm is applied in order to arrive to the Joint CGF. So the equation for the joint cumulants becomes the following.

$$
C_{x_1^{c_1},x_2^{c_2},\ldots,x_k^{c_k}} = \left.\frac{ln\left(\partial^n \mathrm{E}\left[e^{\omega_1\cdot x_1+\omega_2\cdot x_2+\ldots+\omega_k\cdot x_k}\right]\right)}{\partial\omega_1^{c_1}\,\partial\omega_2^{c_2}\,\ldots,\partial\omega_k^{c_k}}\right|_{\omega_1=\omega_2=\ldots=\omega_k=0} \tag{A.53}
$$

Where: $n = c_1 + c_2 + \ldots + c_k$

Discrete Estimator Joint Cumulants

The discreet joint cumulants are more or less the same as for the if we have zero mean on the output from the random variables. This is the same theory as in section A.1.7 but the random variable x is replaced by x, y, z, w to get the joint cumulants. As the derivations are the same, they are not repeated here, but the results are as follows, for the joint moments to joint cumulants:

$$
\begin{aligned}
c_{xy}(\tau_1) &= m_{xy}(\tau_1) - m_x \cdot m_y \\
&= m_{x^2}(\tau_1) && \text{(A.54)} \\
c_{xyz}(\tau_1, \tau_2) &= m_{xyz}(\tau_1, \tau_2) - m_x \cdot m_{yz}(\tau_1) - m_y \cdot m_{xz}(\tau_2) \\
&\quad -m_z \cdot m_{xy}(\tau_2 - \tau_1) + 2 \cdot m_x \cdot m_y \cdot m_z \\
&= m_{xyz}(\tau_1, \tau_2) && \text{(A.55)} \\
c_{xyzw}(\tau_1, \tau_2, \tau_3) &= m_{xyzw}(\tau_1, \tau_2, \tau_3) - m_{xy}(\tau_1) \cdot m_{zw}(\tau_1) \\
&\quad -m_{xz}(\tau_1) \cdot m_{yw}(\tau_1) - m_{xw}(\tau_1) \cdot m_{yz}(\tau_1) \\
&\quad -m_x \cdot m_{yzw}(\tau_1, \tau_2) - m_y \cdot m_{xzw}(\tau_1, \tau_2) \\
&\quad -m_z \cdot m_{xyw}(\tau_1, \tau_2) - m_w \cdot m_{xyz}(\tau_1, \tau_2) \\
&\quad +2 \cdot m_{xy}(\tau_1) \cdot m_z \cdot m_w + 2 \cdot m_{xz}(\tau_1) \cdot m_y \cdot m_w \\
&\quad +2 \cdot m_{xw}(\tau_1) \cdot m_y \cdot m_z + 2 \cdot m_{yw}(\tau_1) \cdot m_x \cdot m_z \\
&\quad +2 \cdot m_{zw}(\tau_1) \cdot m_x \cdot m_y + 2 \cdot m_{yz}(\tau_1) \cdot m_x \cdot m_w \\
&\quad -6 \cdot m_x \cdot m_y \cdot m_z \cdot m_w \\
&= m_{xyzw}(\tau_1, \tau_2, \tau_3) - m_{xy}(\tau_1) \cdot m_{zw}(\tau_3 - \tau_2) \\
&\quad -m_{xz}(\tau_2) \cdot m_{yw}(\tau_3 - \tau_1) - m_{xw}(\tau_3) \cdot m_{yz}(\tau_2 - \tau_1) && \text{(A.56)}
\end{aligned}
$$

A.2 Properties of Moments and Cumulants

There are some interesting properties of moments and cumulants, the properties present in this section are based on the properties from [1, pp. 12-14] and [6, p. 17]

- Scaling by a constant:
  $c_{a \cdot x} = a \cdot c_x$ and $m_{a \cdot x} = a \cdot m_x$
  this applies for the continuous case as well as the discreet case.
- Symmetric functions:
  Moments and cumulants are symmetric functions in their arguments
  $c_{x^3}(\tau_1, \tau_2, \tau_3) = c_{x^3}(\tau_3, \tau_2, \tau_1) = \ldots$ and $c_{x^3}(\tau_1, \tau_2, \tau_3) = c_{x^3}(\tau_2, \tau_1, \tau_3) = \ldots$
- Independent random variables:
  If the random variable x is independent of the random variable y then:
  $c_{x+y} = c_x + c_y$ this does in general hold for moments as:
  $m_{x+y} = \mathrm{E}\left[(x + y)\right] \neq m_x + m_y$
- Gaussian distribution:
  If a set of or one random variable has a Gaussian distribution then all the information about the distribution is contained in the moments of order lower then 3 so higher order moments do not give additional information (higher order moments also contains information about lower order). This then leads to the fact that all cumulants for a Gaussian distribution of higher order then 2 are zero. Therefore one could argue that the cumulants higher then order 2 gives an measure of the non-Gaussian nature of the distribution.

- Cumulants are additive in their arguments eg.
  $c_{x+y,z^2}(\tau_1,\tau_2,\tau_3) = c_{x,z^2}(\tau_1,\tau_2,\tau_3) + c_{y,z^2}(\tau_1,\tau_2,\tau_3)$
- Cumulants are blind to additive constants. This is because lower order moments are removed from the cumulant sequence
  $c_{k\cdot x,z^2}(\tau_1,\tau_2,\tau_3) = c_{x,z^2}(\tau_1,\tau_2,\tau_3)$
- The random variable x is independent from the rest of the random variables then.
  $c_{x,z^2}(\tau_1,\tau_2,\tau_3) = 0$

A.3 Moment spectra

The section contains the theory behind the moment spectra which is a useful tool for the analysis of time series in the Fourier domain. This section is mainly inspired from [1, pp. 71-121]. It is defined as the Fourier transform of the moments, which was defined earlier.

For the n-th order moment spectra the following discreet Fourier transform is performed on the n-th order moment.

$$M_{x^n}(\omega_1,\cdots,\omega_{n-1}) = \sum_{\tau_1=-\infty}^{\infty} \cdots \sum_{\tau_{n-1}=-\infty}^{\infty} m'_{x^n}(\tau_1,\ldots,\tau_{n-1}) \cdot e^{-j(\omega_1\tau_1+\cdots+\omega_{n-1}\tau_{n-1})} \tag{A.57}$$

Inserting the expression for the moment given in equation A.32 and rewriting yields:

$$\begin{aligned} M_{x^n}(\omega_1,\cdots,\omega_{n-1}) &= \sum_{\tau_1=-\infty}^{\infty} \cdots \sum_{\tau_{n-1}=-\infty}^{\infty} \sum_{t=-\infty}^{\infty} x(t)\cdot x(t+\tau_1)\cdots x(t+\tau_{n-1}) \cdot e^{-j(\omega_1\tau_1+\cdots+\omega_{n-1}\tau_{n-1})} \\ &= \sum_{t=-\infty}^{\infty} x(t) \cdot \prod_{i=1}^{n-1} \sum_{\tau_i=-\infty}^{\infty} x(t+\tau_i) e^{-j\omega_i\tau_i} \end{aligned} \tag{A.58}$$

The last term in equation A.58 can be rewritten into:

$$\begin{aligned} \sum_{\tau_i=-\infty}^{\infty} x(t+\tau_i)e^{-j\omega_i\tau_i} &= \sum_{\tau_i=-\infty}^{\infty} x(t+\tau_i)e^{-j\omega_i\tau_i}\cdot e^{-j\omega_i t}e^{j\omega_i t} \\ &= \sum_{\tau_i=-\infty}^{\infty} x(t+\tau_i)e^{-j\omega_i(t+\tau_i)}\cdot e^{j\omega_i t} \\ &= X(\omega_i)\cdot e^{j\omega_i t} \end{aligned} \tag{A.59}$$

Inserting this in equation A.58 yields:

$$M_{x^n}(\omega_1,\ldots,\omega_{n-1}) = \sum_{t=-\infty}^{\infty} x(t) \prod_{i=1}^{n-1} X(\omega_i)\cdot e^{j\omega_i t} \tag{A.60}$$

Which can be rewritten to:

$$M_{x^n}(\omega_1,\ldots,\omega_{n-1}) = X^*(\omega_1+\ldots+\omega_{n-1}) \cdot \prod_{i=1}^{n-1} X(\omega_i) \tag{A.61}$$

Equation A.61 shows that it is possible to calculate the moment spectra in the frequency domain as the Fourier transform of the time sequence $x(t)$, as there is two possible ways to calculate the moment spectra either with A.58 or A.60. Figure A.3 summarizes this relation between the time domain and the Fourier domain.

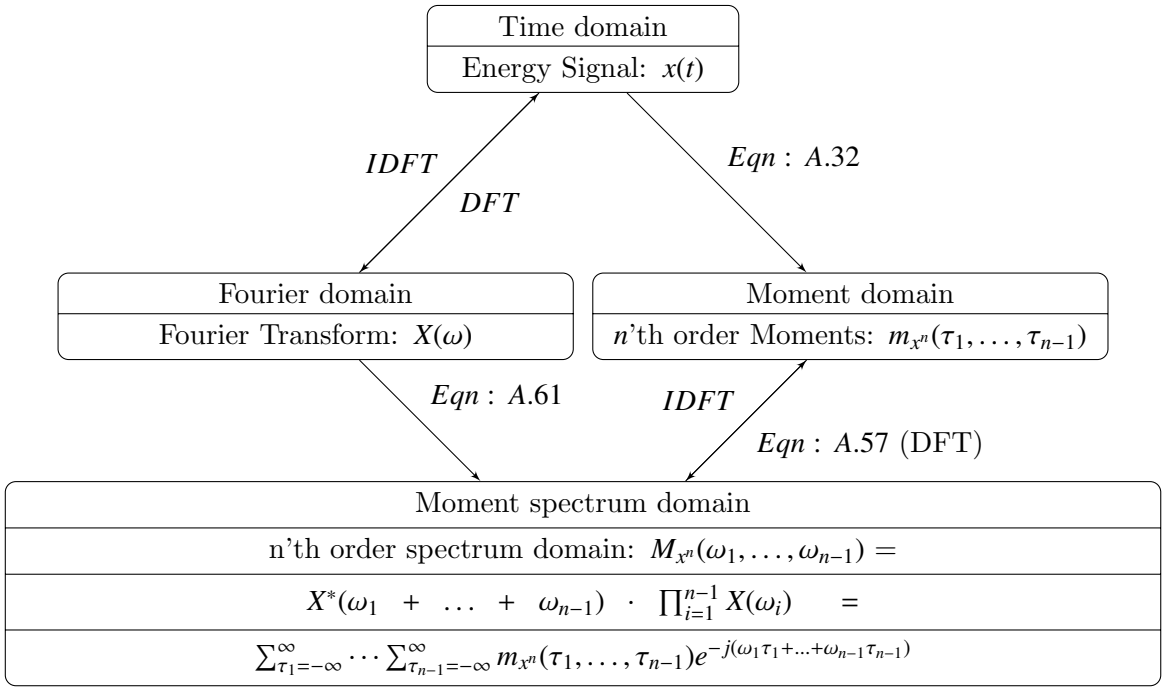


Figure A.3: Illustration of the relations between the time domain the frequency domain for the moment spectras

A.4 Moment Cross Spectra

Similarly as in the moments and cross moments, it is also possible to create a cross moment spectra. An example with four random variable is used here, for the fourth order cross moment spectra between x, y, z and w.

$$M_{xyzw}(\omega_1, \omega_2, \omega_3) = \sum_{\tau_1=-\infty}^{\infty} \sum_{\tau_2=-\infty}^{\infty} \sum_{\tau_3=-\infty}^{\infty} m'_{x^4}(\tau_1, \tau_2, \tau_3) \cdot e^{-j(\omega_1\tau_1+\omega_2\tau_2+\omega_3\tau_3)} \tag{A.62}$$

Inserting the expression for the fourth order cross moment and rewriting yields:

$$\begin{aligned} M_{xyzw}(\omega_1, \omega_2, \omega_3) \quad = \quad & \sum_{\tau_1=-\infty}^{\infty} \sum_{\tau_2=-\infty}^{\infty} \sum_{\tau_3=-\infty}^{\infty} \sum_{t=-\infty}^{\infty} x(t) \cdot y(t+\tau_1) \\ & \cdot z(t+\tau_2) \cdot w(t+\tau_3) \cdot e^{-j(\omega_1\tau_1+\omega_2\tau_2+\omega_3\tau_3)} \end{aligned} \tag{A.63}$$

By redoing the same steps as in equation A.59 and A.60 gives:

$$M_{xyzw}(\omega_1, \omega_2, \omega_3) = X^*(\omega_1 + \omega_2 + \omega_3) \cdot Y(\omega_1) \cdot Z(\omega_2) \cdot W(\omega_3) \tag{A.64}$$

It should be noted if discreet moments are used then a factor of $\frac{1}{N}$ should be multiplied with the spectra, where N is the size of the sample data. This concludes the moment spectras now the cumulants sectras needs be defined.

A.5 Cumulant Spectra

The reason for using cumulant spectra compared to moment spectra, is that the "*cumulant provides suitable means to detect statistical dependencies in time series*". Furthermore the sum of two random, nonzero mean independent processes equal the sum of their cumulant spectra which does not hold for their moment spectra making it easier to work with combined spectra's cumulant spectra's. The procedure for calculating the cumulants spectra's is the same as for the moment spectra take the n-1 dimensional FFT of the cumulants.

The n'th order cumulant spectra is defined as a DFT:

$$C_{x^n}(\omega_1, \omega_2, \ldots, \omega_{n-1}) = \sum_{\tau_1=-\infty}^{\infty} \cdots \sum_{\tau_{n-1}-\infty}^{\infty} c_{x^n}(\tau_1, \ldots, \tau_{n-1}) \cdot e^{-j(\omega_1\tau_1+\omega_2\tau_2+\ldots+\omega_{n-1}\tau_{n-1})} \tag{A.65}$$

The cumulant spectra for $n = 2$ of a random variable (x) is normally know as the power spectrum of x with x having zero mean. The equation is as follows:

$$C_{x^2}(\omega) = \sum_{\tau=-\infty}^{\infty} c_{x^2}(\tau) \cdot e^{-j(\omega\tau)} \tag{A.66}$$

Where:
$|\omega| \leq \pi$

The following symmetry conditions apply for the time domain :

$$c_{x^2}(\tau) = c_{x^2}(-\tau) \tag{A.67}$$

Therefore the condition in equation A.68 must also hold.

$$C_{x^2}(\omega) = C_{x^2}(-\omega) \tag{A.68}$$

And because equation A.61 also can be used for the power spectrum if x is zero mean (see equation A.54) then the following two conditions must also be true for the powerspectrum.

$$C_{x^2}(\omega) \geq 0 \tag{A.69}$$
$$C_{x^2}(\omega) \subset \mathfrak{R} \tag{A.70}$$

From this it can be concluded that the spectrum is mirrored around $\omega = 0$ if x only contains real signals.

Another important spectrum is the bispectrum for $n = 3$. Equations A.71 is the bispetrum for the random variable x

$$C_{x^3}(\omega_1, \omega_2) = \sum_{\tau_1=-\infty}^{\infty} \sum_{\tau_2=-\infty}^{\infty} c_{x^3}(\tau_1, \tau_2) \cdot e^{-j(\omega_1\cdot\tau_1+\omega_2\cdot\tau_2)} \tag{A.71}$$

Where:
$|\omega_1| \leq \pi$
$|\omega_2| \leq \pi$

$|\omega_1 + \omega_2| \leq \pi$

If the random variable x has zero mean the bispectrum can also be calculated using A.61. From this equation it can be seen why the bispectrum contains information about the phase whereas the powerspectrum does not contain any information about the phase. Any power spectrum of a complex signal would become absolute value of these to reconstruct the real and imaginary parts from the power spectrum, one must assume minimum or maximum phase. But as the bispectrum still contains information about the phase it is possible to reconstruct the correct phase from the signal.

The following symmetry conditions apply for the third order cumulants:

$$\begin{aligned} c_{x^3}(\tau_1, \tau_2) &= c_{x^3}(\tau_2, \tau_1) \\ &= c_{x^3}(-\tau_2, \tau_1 - \tau_2) \\ &= c_{x^3}(\tau_2 - \tau_1, -\tau_1) \\ &= c_{x^3}(t_\tau - \tau_2, -\tau_2) \\ &= c_{x^3}(-\tau_1, \tau_2 - \tau_1) \end{aligned} \tag{A.72}$$

This gives rise to 6 symmetry regions for the third order cumulants. The symmetry regions would therefore transfer onto the the bispectrum resulting in 12 symmetry regions for real signals ($x \subset \Re$) and 6 symmetry regions for complex signals.

$$\begin{aligned} C_{x^3}(\omega_1, \omega_2) &= C_{x^3}(\omega_2, \omega_1) \\ &= C^*_{x^3}(-\omega_2, -\omega_1) \\ &= C_{x^3}(-\omega_1 - \omega_2, \omega_2) \\ &= C_{x^3}(\omega_1, -\omega_1 - \omega_2) \\ &= C_{x^3}(-\omega_1 - \omega_2, \omega_1) \\ &= C_{x^3}(\omega_2, -\omega_1 - \omega_2) \end{aligned} \tag{A.73}$$

These symmetry regions for the bispectrum is illustrated in figure A.4.

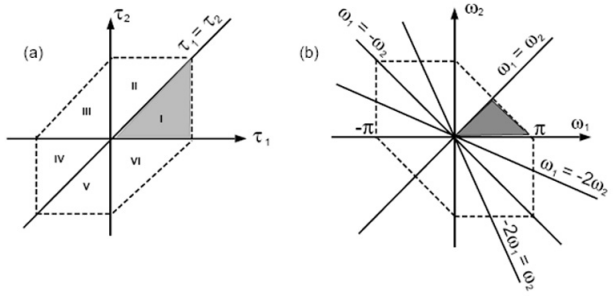


Figure A.4: Symmetry regions for the moment sequence and the moment spectra.

The last spectrum of interest is the trispectrum ($n = 4$), the equation for the trispectrum is as follows:

$$C_{x^4}(\omega_1, \omega_2, \omega_3) = \sum_{\tau_1=-\infty}^{\infty} \sum_{\tau_2=-\infty}^{\infty} \sum_{\tau_3=-\infty}^{\infty} c_{x^3}\tau_1, \tau_2, \tau_3) \cdot e^{-j(\omega_1 \cdot \tau_1 + \omega_2 \cdot \tau_2 + \omega_3 \cdot \tau_3)} \tag{A.74}$$

Where:
$|\omega_1| \leq \pi$
$|\omega_2| \leq \pi$
$|\omega_3| \leq \pi$
$|\omega_1 + \omega_2 + \omega_3| \leq \pi$

As for the bispectrum and the power spectrum the trispectrum also have allot of symmetry regions:

$$C_{x^4}(\omega_1, \omega_2, \omega_3) = C_{x^4}(\omega_2, \omega_1, \omega_3) = C_{x^4}(\omega_1, \omega_3, \omega_2) = C_{x^4}(\omega_2, \omega_3, \omega_1) = etc. \tag{A.75}$$

The total amount of symmetry regions are by [1, p. 23] given as 96 for real signals. It is not possible to illustrate all these regions, mainly because they are in a 3 dimensional plot, but it is worth noticing that there is a lot of redundant regions in the plots.

As for the moment spectra it is possible to make a representation of the relations between the Fourier domain and the time domain, this is more or less the same figure as in A.3 but here the cumulants are used instead of the moments. Now that the spectra's for both the cumulants

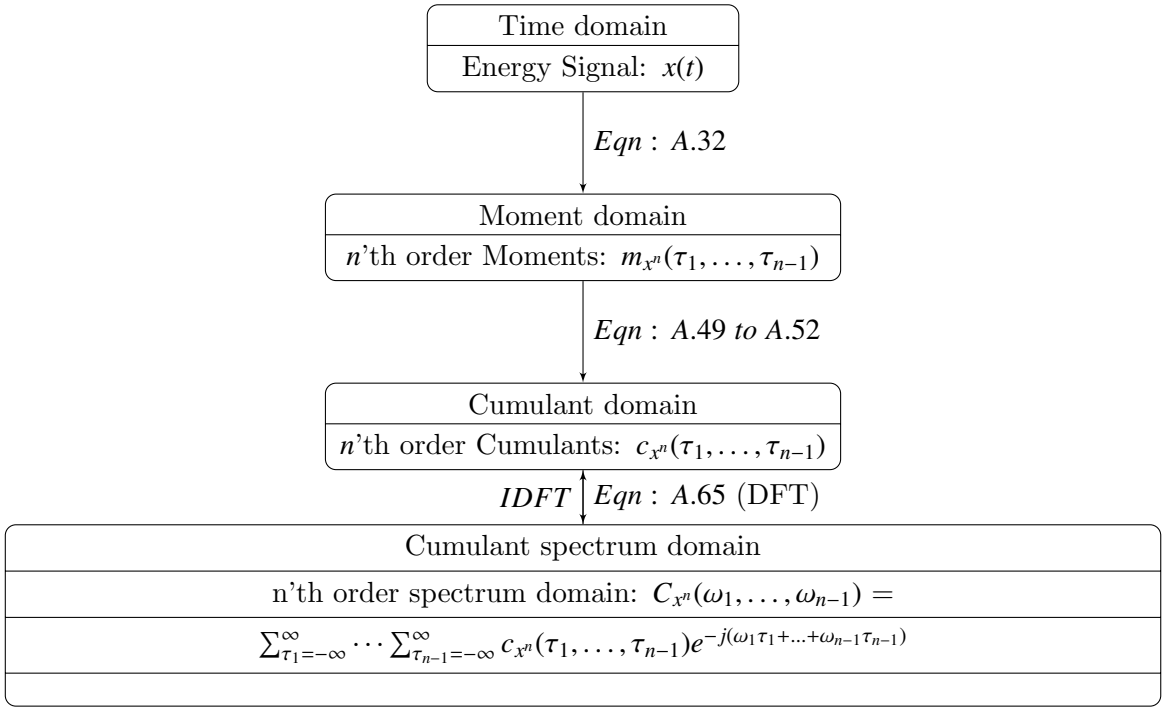


Figure A.5: Illustration of the relations between the time domain the frequency domain for the cumulant spectra's, please note that the that the moment to cumulant transformation is only defined up to the fourth order

and moments are defined, the next part is windowing of these functions

A.6 Windowing of the Fourier transform

This section contains details about the windowing of the moment or cumulant sequence. Normally a window function would be used to reduce spectral leakage for the same reason a window

is added for estimation of the power spectra

The window is normally applied to the n-dimensional cumulant sequences before the Fourier transformation, therefore the window also needs to be n-dimensional. Function A.76 describes how to construct a n-dimensional window from a 1-dimensional window [1, p.126].

$$w_n(\tau_1, \ldots, \tau_{n-1}) = w_1(\tau_1 + \ldots + \tau_{n-1}) \cdot w_1(\tau_1) \cdots w_1(\tau_{n-1}) \tag{A.76}$$

Where:
w_n is the n-dimensional window
w_1 is a normal one dimensional window

There are several window types that are available to reduce the spectral leakage two good choices for windows are the optimum window presented in equation A.78 and a Parzen window present in equation A.77 [6, p.21].

$$w_1(\tau) = \begin{cases} 1 - 6\left(\frac{|\tau|}{L}\right)^2 + 6\left(\frac{|\tau|}{L}\right)^3 & , \quad |\tau| \leq \frac{L}{2} \\ 2(1 - \left(\frac{|\tau|}{L}\right)^3) & , \quad \frac{L}{2} < |\tau| \leq L \\ 0 & , \quad 1 < |\tau| \end{cases} \tag{A.77}$$

Where:
L is the length of the window.

$$w_1(\tau) = \begin{cases} \frac{1}{\pi}\left|\sin\left(\frac{\pi\tau}{L}\right)\right| + \left(1 - \frac{|\tau|}{L}\right) \cdot \cos\left(\frac{\pi\tau}{L}\right) & , \quad |\tau| \leq L \\ 0 & , \quad L < |\tau| \end{cases} \tag{A.78}$$

A.7 Properties of Cumulant Spectra's

This chapter contains some general properties of cumulants spectra's as well as some examples on how to use them.

A.7.1 Multilinearity of cumulant spectra's and cross cumulant spectra's

This section gives an example of the multilinearity properties of cumulant spectra's, the example is based on how multiliniarity is used in the main report.
Consider a signal z in the time domain which contains the variables x and the random variable y convolved with a filter h x and y have zero mean and are independent of each other

$$z = x + h * y \tag{A.79}$$

The third order moment of z can be expressed as:

$$\begin{aligned} m_{z^3}(\tau_1,\tau_2) &= \mathrm{E}\left[z(t)\cdot z(t-\tau_1)\cdot z(t-\tau_2)\right] \\ &= \mathrm{E}\left[\,(x(t)+h*y(t))\cdot(x(t-\tau_1)+h*y(t-\tau_1))\cdot(x(t-\tau_2)+h*y(t-\tau_2))\,\right] \\ &= \mathrm{E}\left[\,x(t)\cdot x(t-\tau_1)\cdot x(t-\tau_2)+h*y(t)\cdot h*y(t-\tau_1)\cdot h*y(t-\tau_2)+\right. \\ &\quad x(t)\cdot h*y(t-\tau_1)\cdot x(t-\tau_2)+h*y(t)\cdot x(t-\tau_1)\cdot x(t-\tau_2)+ \\ &\quad \left. x(t)\cdot h*y(t-\tau_1)\cdot h*y(t-\tau_2)+h*y(t)\cdot x(t-\tau_1)\cdot h*y(t-\tau_2)\,\right] \end{aligned} \tag{A.80}$$

If x and y are uncorrelated then the last four terms in equation A.80 can be removed

$$\begin{aligned} m_{z^3}(\tau_1,\tau_2) &= \mathrm{E}\left[x(t)\cdot x(t-\tau_1)\cdot x(t-\tau_2)\right]+\mathrm{E}\left[h*y(t)\cdot h*y(t-\tau_1)\cdot h*y(t-\tau_2)\right] \\ &= m_{x^3}(\tau_1,\tau_2)+m_{(h*y)^3}(\tau_1,\tau_2) \end{aligned} \tag{A.81}$$

So the moment spectra of z is actually the moment spectra of y and x added together if they are uncorrelated. Making the same calculations for higher order gives the same results so in general the following equation apply.

$$m_{z^n}(\tau_1,\tau_2) = m_{x^n}(\tau_1,\tau_2)+m_{(h*y)^n}(\tau_1,\tau_2) \tag{A.82}$$

This can be extended to cumulants as well as it only removes the lower order moments contribution so in general for cumulants the following also holds.

$$c_{z^n}(\tau_1,\tau_2) = c_{x^n}(\tau_1,\tau_2)+c_{(h*y)^n}(\tau_1,\tau_2) \tag{A.83}$$

The condition is of course that the n-th order cumulant spectra exist for either x or y.

Looking closer on the filter h convolved with the random variable y, the convolution can be written as.

$$h*y=\sum_{\tau_h=-\infty}^{\infty}h(\tau_h-t)\cdot y(t) \tag{A.84}$$

If equation A.84 is applied to third order moment of the convolution gives the following.

$$\begin{aligned} m_{(h*y)^3}(\tau_1,\tau_2) &= \mathrm{E}\left[\sum_{\tau_{h1}=-\infty}^{\infty}h(\tau_{h1}-t)\cdot y(t)\cdot\sum_{\tau_{h2}=-\infty}^{\infty}h(\tau_{h2}-t-\tau_1)\cdot y(t-\tau_1)\right. \\ &\quad \left.\cdot\sum_{\tau_{h3}=-\infty}^{\infty}h(\tau_{h3}-t-\tau_2)\cdot y(t-\tau_2)\right] \\ &= \mathrm{E}\left[\sum_{\tau_{h1}=-\infty}^{\infty}\sum_{\tau_{h2}=-\infty}^{\infty}\sum_{\tau_{h3}=-\infty}^{\infty}h(\tau_{h1}-t)\cdot h(\tau_{h2}-t-\tau_1)\cdot h(\tau_{h3}-t-\tau_2)\cdot\right. \\ &\quad \left. y(t)\cdot y(t-\tau_1)\cdot y(t-\tau_2)\right] \\ &= \mathrm{E}\left[(h(t)\cdot h(t-\tau_1)\cdot h(t-\tau_2))*(y(t)\cdot y(t-\tau_1)\cdot y(t-\tau_2))\right] \\ &= \mathrm{E}\left[(h(t)\cdot h(t-\tau_1)\cdot h(t-\tau_2))\right]*\mathrm{E}\left[(y(t)\cdot y(t-\tau_1)\cdot y(t-\tau_2))\right] \\ &= m_{h^3}(\tau_1,\tau_2)*m_{y^3}(\tau_1,\tau_2) \end{aligned} \tag{A.85}$$

If the random process y is uncorrelated from one time instants to the next, it would normally be referred to as a random white process, and would manifest it self by its power spectrum being flat. If a spectrum of order n is flat. Then the random variable would be referred to as being n'th-order white. If both random variables in equation A.85 are third order white noise process. Their spectrum's would be flat e.i. it would have the same value all over. So if equation A.85 is Fourier transformed the spectrum would be a function of the filter as the random variables would be constant γ which is also known as the skewness for the third order cumulants [1, pp. 37-40].

$$C_{(h*y)^3}(\tau_1, \tau_2) = M_{h^3}(\tau_1, \tau_2) \cdot \gamma_{y^3} \tag{A.86}$$

This can also be extended into n-th order case. If the random variables x and y are n-th order white noise, then equation A.83 can be written as:

$$C_{z^n}(\omega_1, \omega_2) = \gamma_{x^n} + M_{h^n}(\omega_1, \omega_2) \cdot \gamma_{y^n} \tag{A.87}$$

A.8 Example of system identification using HOS

This example is thought up, but gives a good example how HOS can be used for system identification, the example here is based on an example given by [1, pp. 53-56]. Figure A.6 show a system where the filters $A(z)$ $B(z)$ and $C(z)$ are to be determined.

The following things are known about the system:

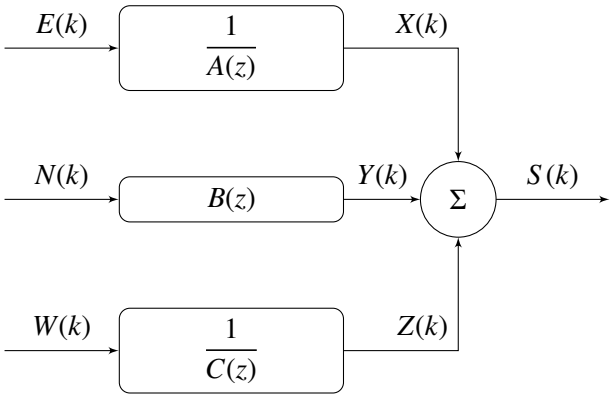


Figure A.6: Example of system where the filters A, B and C are to be identified.

- The sources $E(k)$, $N(k)$ and $W(k)$ are independent white noise processes.
- $E(k)$ is a zero mean, white Gaussian Process with a variance larger then zero.
- $N(k)$ is a zero mean, white non-Gaussian Process with a variance, skewness and kurtosis larger then zero.
- $W(k)$ is a zero mean, white non-Gaussian Process with a variance and kurtosis larger then zero. But the skewness is zero.

- $X(k)$ is an Gaussian AR process generated from:

$$X(k) = -\sum_{i=1}^{p_i} a_i \cdot X(k-i) + E(k) \tag{A.88}$$

- $Y(k)$ is a non-Gaussian MA process generated from:

$$Y(k) = \sum_{i=1}^{q_i} b_i \cdot N(k-i) \tag{A.89}$$

- $Z(k)$ is a non-Gaussian AR process generated from:

$$Z(k) = -\sum_{i=1}^{p_i} c_i \cdot Z(k-i) + W(k) \tag{A.90}$$

If the powerspectrum of $S(k)$ is examined, then because the sources $E(k)$, $N(k)$ and $W(k)$ are independent then it can be expressed as a sum of the power spectra's of $X(k)$, $Y(k)$ and $Z(k)$.

$$C_{S^2}(\omega) = C_{X^2}(\omega) + C_{Y^2}(\omega) + C_{Z^2}(\omega) \tag{A.91}$$

As the sources are white their cumulant spectra's becomes a function of the filter they are convolved with so:

$$C_{X^2}(\omega) = \gamma_{E^2} \frac{1}{A(\omega) \cdot A^*(\omega)} \tag{A.92}$$

$$C_{Y^2}(\omega) = \gamma_{N^2} B(\omega) \cdot B^*(\omega) \tag{A.93}$$

$$C_{Z^2}(\omega) = \gamma_{W^2} \frac{1}{C(\omega) \cdot C^*(\omega)} \tag{A.94}$$

If the bispectrum of $S(k)$ is examined, then it can be expressed as the bispectrum of $Y(k)$. The bispectrum of $X(k)$ and $Z(k)$ are zero as their sources have a skewness of zero so:

$$C_{S^3}(\omega_1, \omega_2) = C_{Y^3}(\omega_1, \omega_2) \tag{A.95}$$

The bispectra of $Y(k)$ can be expressed as:

$$C_{Y^3}(\omega_1, \omega_2) = \gamma_{N^2} B(\omega_1) \cdot B(\omega_2) \cdot B^*(\omega_1 + \omega_2) \tag{A.96}$$

From this it is possible to estimate the filter B with some unknown scale factor

The last thing to examine is the trispectrum of $S(k)$. The trispectrum can be expressed as a function of the trispectrum of $Y(k)$ and $Z(k)$, as the source signal for $X(k)$ is Gaussian its kurtosis is zero this means that trispectrum of $X(k)$ is also zero.

$$C_{S^4}(\omega_1, \omega_2, \omega_3) = C_{Y^4}(\omega_1, \omega_2, \omega_3) + C_{Z^4}(\omega_1, \omega_2, \omega_3) \tag{A.97}$$

The trispectra of $Y(k)$ and $Z(k)$ can be expressed as a function of their source signal multiplied with their respective filters.

$$C_{Y^3}(\omega_1, \omega_2, \omega_3) = \gamma_{N^2} B(\omega_1) \cdot B(\omega_2) \cdot B(\omega_3) \cdot B^*(\omega_1 + \omega_2 + \omega_3) \tag{A.98}$$

$$C_{Z^3}(\omega_1, \omega_2, \omega_3) = \gamma_{W^2} C(\omega_1) \cdot C(\omega_2) \cdot C(\omega_3) \cdot C^*(\omega_1 + \omega_2 + \omega_3) \tag{A.99}$$

As the filter $\boldsymbol{B}$ could be determined in bispectrum (with some scale error) it should be possible to estimate the filters C from the trispectrum and finally the filter A from the powerspectrum. This here was short example on how higher order statistics could be used for system identification. This concludes the appendix on higher order statistics.

1

Appendix B

CUDA programming

In this appendix the programming of a Compute Unified Device Architecture (CUDA) capable GPU is described. The appendix is divided into two sections; the first focusing on the C aspect of the CUDA programming language and the second focusing on how to achive the best performance. The reader is expected to have knowledge about the underlying hardware architechture and related basic terms such as threads, blocks or compute capability. The appendix is based on [7].

B.1 C for CUDA

The CUDA programming language is designed to allow programmers to easily write code for NVIDIA GPUs, provided they are familiar with the C programming language. As such CUDA is merely a set of extensions to C with similar syntax.

The following only describes the basics of CUDA programming. For more in depth information it is recommended to read [7].

B.1.1 Extensions

The CUDA extensions to C can be divided into five subcategories: Function type qualifiers, variable type qualifiers, a new directive to specify how kernels are executed on the GPU, four built-in variables that specify the dimensions of blocks and grid and indices of threads and blocks and a set of built-in vector types.

Function type qualifiers

There are three types of function qualifiers that describes where functions are executed and how they are called[7, p. 20]:

- **__global__**: Executed on the device and called by the host. This is the kernel function.
- **__device__**: Executed on the device and called by the device. Can only be used as a subroutine in a kernel function.
- **__host__**: Executed on the host and called by the host. This is equivalent to a standard C function. The qualifier can be omitted or used in combination with **__device__** to compile the function for both host and device.

The programming of the GPUs is somewhat limited due to the small amount of flow control logic. This results in the following restrictions to the functions (full list in [7, pp. 20-21]):

- **__device__** and **__global__** functions do not support recursion.
- **__device__** and **__global__** functions cannot declare static variables inside their body.
- **__global__** functions must have void return type.
- A call to a **__global__** function is asynchronous, meaning it returns before the device has completed its execution.

Variable type qualifiers

There are three types of variable qualifier that describes where the variable resides in memory, the lifetime and accessibility.

- **__device__**: Resides in device global memory, has the lifetime of an application and can be accessed from all threads in the grid as well as the host.
- **__constant__**: Resides in constant memory space, has the lifetime of an applicatio and can be accessed from all threads in the grid as well as the host.
- **__shared__**: Resides in shared memory of a thread block, has the lifetime of the block and can be accessed from all threads within the block.

Like the function type qualifiers there are also a number of restrictions to variables, the full list can be found in [7, p. 22]:

- Neither of the qualifiers are allowed on struct and union members, on formal parameters and on local variables within a function that executes on the host.
- **__device__** and **__constant__** variables are only allowed at file scope.
- The address obtained by taking the address of a **__device__**, **__shared__** or **__constant__** variable can only be used in device code. The address of a **__device__** or **__constant__** variable obtained through **cudaGetSymbolAddress()** can only be used in host code.
- **__constant__** variables cannot be assigned to from the device, only from the host through host runtime functions.

- Pointers in code that is executed on the device are supported as long as the compiler is able to resolve whether they point to either the shared memory space or the global memory space, otherwise they are restricted to only point to memory allocated or declared in the global memory space.

Built-in vector types

Like threads can be ordered in vectors, matrices or fields inside blocks, so can most numerical values by using the built-in vector types [7, p. 25]:

char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, float4, double2.

The built-in vector types are derived from the corresponding basic C types. All of the vector types are defined as structs containing up to four members depending on the number in the type name, specifying the number of dimension. The members can be accessed using the member names **x**, **y**, **z** and **w** for the 1st, 2nd, 3rd and 4th dimension, respectively.

The built-in vector types uses special constructors of the form **make_<type name>**, for instance:

```
1 int2 make_int2(int x, int y);
```

The **dim3** type is a special vector type based on the **unit3** type where any unspecified members is initialized to 1. The **dim3** is used to specify the dimensions of the grid and thread blocks of a kernel.

Built-in variables

Instead of using **for** or **while** to index large arrays CUDA uses the blocks and threads for the indexing. Regular loops can still be used, but this does not create a thread for each loop and thus lowers performance. For this purpose there are five built-in variables [7, pp. 23-24]:

- **gridDim**: This variable is of type **dim3** and contains the dimensions of the grid.
- **blockIdx**: This variable is of type **uint3** and contains the block index within the grid.
- **blockDim**: This variable is of type **dim3** and contains the dimensions ofthe block.
- **threadIdx**: This variable is of type **uint3** and contains the thread index within the block.
- **warpSize**: This variable is of type **int** and contains the warp size in threads.

A few restrictions are related to the built-in variables [7, p. 24]:

- It is not allowed to take the address of any of the built-in variables.
- It is not allowed to assign values to any of the built-in variables.

Execution configuration

When a **__global__** function is called the execution configuration must be specificied. The execution configuration is a set of four parameters that are put in special <<<...>>> brackets and inserted between the function name and argument paranthesis. The four paramters are [7, p. 23]:

- **Dg**: Is of type **dim3** and specifies the dimension of the grid. The **x** and **y** dimensions can be specified while the z dimension must be equal to 1.
- **Db**: Is of type **dim3** and specifies the dimension of the block.
- **Ns**: Is of type **size_t** and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory. Defaults to 0 and can be omitted.
- **S**: Is of type **cudaStream_t** and specifies the associated stream. Defaults to 0 and can be omitted.

For instance, a function declared as:

```
1 __global__ void function(int* paramter){}
```

Must be called as:

```
1 function<<< Dg, Db, Ns, S >>>(parameter);
```

B.1.2 Thread synchronization

While all scheduling of the threads are handled by the SMs the programmers can synchronize threads using the intrinsic function **__syncthreads()**. Threads are synchronized on kernel level meaning all threads in all blocks execute up to the same point in the code before any further execution. **__syncthreads()** is typically used to synchronize memory transfers between blocks.

B.1.3 Example of matrix-matrix addition

The following example has been included to show basic principles of CUDA programming. Both kernel and calling code has been included. Explanatory comments have been included in the code:

The block partitioning is done so each block handles a small part of the matrix addition and shared memory equal to 3 times the number of elements per block is allocated dynamically.

```
1  // Matrix–matrix addition of matrices A and B = C
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <cuda.h>
6
7  #define h 128       // Rows in the matrix
8  #define w 256       // Columns in the matrix
9
10 // Forward declarations
11 __global__ void MatrixAdd(int *Ap, int *Bp, int *Cp);
12
13 // Main function
14 int main(int argc, char** argv){
15     int m, n, size, A[h][w], B[h][w], C[h][w];
16     int *Ap,*Bp,*Cp; // Pointers to the matrices
17     float error=0;
18
19     // Insert numbers in A and B
20     for (m=0; m<h; m++){
21         for (n=0; n<w; n++){
22             A[m][n]=n+m*w;
23             B[m][n]=h*w-(n+m*w);
24         }
25     }
26
27     // Allocate memory for A, B and C on device
28     size=h*w*sizeof(int);
29     cudaMalloc((void**)&Ap, size);
30     cudaMalloc((void**)&Bp, size);
31     cudaMalloc((void**)&Cp, size);
32
33     // Copy A and B to device
34     cudaMemcpy(Ap, A, size, cudaMemcpyHostToDevice);
35     cudaMemcpy(Bp, B, size, cudaMemcpyHostToDevice);
36
37     // Set block and grid size
38     dim3 dimBlock(8,8);
39     dim3 dimGrid(w/dimBlock.x,h/dimBlock.y);
40
41     // Call MatrixAdd kernel to calculate C=A+B, dynamic allocation of shared
           memory
42     MatrixAdd<<<dimGrid, dimBlock, dimBlock.x*dimBlock.y*sizeof(int)>>>(Ap, Bp, Cp);
43
44     // Copy C from device
45     cudaMemcpy(C, Cp, size, cudaMemcpyDeviceToHost);
46
47     // Free allocated memory on device
48     cudaFree(Ap);
49     cudaFree(Bp);
50     cudaFree(Cp);
51
52     // Compare values of C calculated by CPU and GPU and sum the error
53     for (m=0; m<h; m++){
54         for (n=0; n<w; n++){
```

```
55              error+=C[m][n]-(A[m][n]+B[m][n]);
56          }
57      }
58      error=error/(float)(m*n); // Divide by number of elements to obtain mean error
59
60      // Print error
61      printf("Mean error between CPU and GPU result: %10.4f",error);
62
63      return(0);
64  }
65
66  // Kernel functions
67  __global__ void MatrixAdd(int *Ap, int *Bp, int *Cp){
68      // Shared variables, dynamic allocation allocated
69      extern __shared__ int shared[];
70      int *As = (int*)shared;
71      int *Bs = (int*)&shared[blockDim.x*blockDim.y];
72      int *Cs = (int*)&shared[2*blockDim.x*blockDim.y];
73
74      // Block index
75      int bx = blockIdx.x;
76      int by = blockIdx.y;
77
78      // Thread index
79      int tx = threadIdx.x;
80      int ty = threadIdx.y;
81
82      // Global memory index
83      int idx = tx + bx*blockDim.x + ty*blockDim.x*gridDim.x + by*blockDim.x*gridDim.x*
            blockDim.y;
84
85      // Load elements for corresponding block from global memory to shared memory
86      As[tx+ty*blockDim.x] = Ap[idx];
87      Bs[tx+ty*blockDim.x] = Bp[idx];
88
89      // Calculated C = A + B
90      Cs[tx+ty*blockDim.x] = As[tx+ty*blockDim.x]+Bs[tx+ty*blockDim.x];
91
92      // Store elements for corresponding block from shared memory to global memory
93      Cp[idx] = Cs[tx+ty*blockDim.x];
94  }
```

B.2 Performance guidelines

Optimizing CUDA code for execution on the GPUs can roughly be divided into three subcategories: Choice of instructions, memory management and execution configuration.

B.2.1 Instruction performance

Arithmetic instructions

Most of the considerations with regard to arithmetic instruction performance is a consideration of speed versus precision. Many of the math functions are implemented as two functions where the faster, but less precise, version uses the same function name prefixed with __, for instance **sinf(x)** and **__sinf(x)**. A number of the fast functions also have a suffix inserted between the function name and argument paranthesis specifying the rounding mode, for instance **__fadd_rn(x,y)** and **__fadd_rz(x,y)**. There are four different rounding modes [7, p. 90]:

- Functions suffixed with **_rn** operate using the round-to-nearest-even rounding mode.
- Functions suffixed with **_rz** operate using the round-towards-zero rounding mode.
- Functions suffixed with **_ru** operate using the round-up (to positive infinity) rounding mode.
- Functions suffixed with **_rz** operate using the round-down (to negative infinity) rounding mode.

A full list of the fast functions and their error bounds can be found in appendix B of [7].

Like most architechtures division and modulo operations are very costly and should be avoided if possible. If not the programmer should try and replace these operations with their respective bitwise operations, for instance right shifting if dividing with a number that is a power of 2.

Control flow instructions

Threads in a warp that diverges due to control flow instructions, **for/while/do/if/switch**, will be executed in serial rather than in parallel. This will significantly impact performance and should be avoided. It is, however, possible to use control flow instructions without serializing the execution of threads, if the conditions are constructed with the warp size in mind. A simple example of this could be using **(threadIdx / warpSize)** as the condition. In this example the condition would be aligned with the warp size resulting in whole warps diverging rather than threads of a warp. If a whole warp diverges the threads in it will still be executed in parallel without any impact on performance.

Special attention must be paid to loops because the compiler may unroll these to increase performance. This only happens for small loops with a known trip count, but can be controlled using the **#pragma unroll <num>** directive where **<num>** determines how many times the loop should be unrolled. The directive is inserted directly before the loop and only affects the following loop, for instance:

```
1 #pragma unroll 5
2 for (int n = 0; n < N; n++){}
```

This will force the compiler to unroll the loop five times regardless of the value of **N**. In some cases this behavior may affect the correctness of the program and it is up to the programmer to ensure it does not. If no number is specified the loop will always be unrolled if the trip count is constant and never unrolled if it is variable. **#pragma unroll 1** will prevent the compiler from unrolling the loop regardless of the trip count.

B.2.2 Memory management

Memory management is an important issue while optimizing CUDA applications because there are very large differences in access time depending on which type of memory is accessed. Access to uncached device memory takes as long as 400 to 600 clock cycles compared to less than 10 clock cycles for on-chip memory. Transfers between host and device memory is even slower and can thus impact performance significantly.

Global memory

Global memory resides in device memory and thus off-chip. Global memory is not cached and as such one of the slowest types of memory the GPU can access. This means the number of accesses to global memory should be reduced, which can be done in two ways.

Firstly, data can be arranged in memory such that it is read into registers in a single load instruction. To do this the data must be aligned with the memory addresses, such that the addresses of the elements are multiples of the size of their type in bytes. Furthermore the size of the type must be 4, 8 or 16 bytes. The alignment can be achieved automatically if the built-in vector types are used. For structures a special alignment specifier should be used as described in [7, p. 56].

Secondly, memory accesses of threads of a half-warp can be coalesced into a single memory transaction if a number of requirements are fulfilled. Devices of compute capability of 1.2 or higher have much looser requirements than devices of lower compute capability. Only coalescing for devices of compute capability 1.2 or higher is described in this report while coalescing for devices of lower compute capability can be found in [7, p. 57].

For devices of compute capability of 1.2 or higher the global memory is partitioned into segments of 32, 64 or 128 bytes. Any memory access of a half-warp is coalesced into a single transaction if all of the requested addresses lie within the same segment, regardless of multiple threads accessing the same address or the threads accessing the addresses in a random pattern. If a half-warp accesses adresses in n segments n transactions will be performed. The GPU reads a whole segment at a time, meaning unused addresses will be wasting bandwidth. To avoid this the GPU will automatically choose the smallest segment size containing all of the addresses accessed by the half-warp. This is illustrated in figure B.1.

Local memory

Local memory resides in device memory and is not cached, but always coalesced. Variables are only placed in local memory by choice of the compiler and can be avoided by using any of the variable type qualifiers. To identify if a kernel uses local memory the total local memory usage can be reported by compiling with the –**ptxas-options=-v** option.

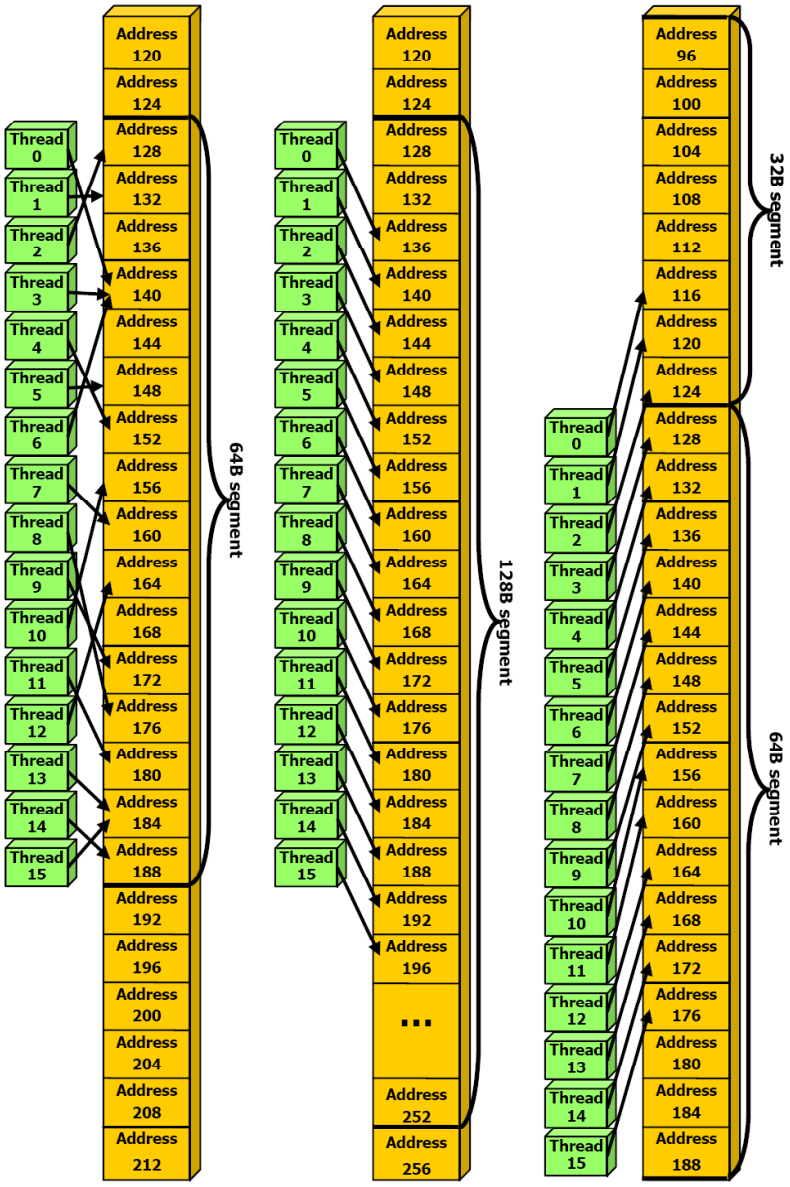


Figure B.1: Left: Random float memory access within a 64B segment, resulting in one memory transaction. Center: Misaligned float memory access, resulting in one transaction. Right: Misaligned float memory access, resulting in two transactions. [7, p. 62]

Constant memory

Constant memory resides in device memory, but is cached and will only read from device memory on a cache miss. If all threads of a half-warp reads from the same address the access will be as fast as access to a register. If several addresses are accessed the cost scales linearly with the number of different addresses accessed. For future generations it is recommended to have all threads of a warp read from the same address rather than only a half-warp.

Texture memory

Constant memory resides in device memory, but is cached and will only read from device memory on a cache miss. Accesses to the texture cache achieve best performance if the threads of the half-warp access data with similar localization as texture, i.e. 2D spatial locality. Using texture memory is not as straight forward as using constant or global memory, but provides several benefits over both. For more info the reader is referred to pages 26, 29, 72 and appendix D of [7].

Shared memory

Shared memory resides on-chip and is divided into n banks. For current generation GPUs $n = 16$ which is very convenient as this is also the size of a half-warp. All threads of a half-warp can read from shared memory simultaneously if no bank conflicts occur. Bank conflicts occur when two or more threads of a half-warp access adresses in the same bank. If this is the case, the SM will split the memory accesses causing bank conflicts into as many separate conflict free accesses as neccessary. If the number of separate accesses is called m, the memory access is said to have caused an m-way bank conflict and the resulting bandwidth reduction is proportional to m. If no bank conflicts occur shared memory access is as fast as accessing registers. To avoid bank conflicts it is neccessary to understand how data is arranged in shared memory.

When data is assigned to shared memory consecutive 32 bit words are stored in successive banks, i.e. the first 32 bit word goes into bank 0, the second 32 bit word goes into bank 1 and so forth. When a 32 bit word has been stored in the 16th bank the pattern is repeated and the following 32 bit goes into bank 0. Special attention must be paid to data types that do not have a size of 32 bits. For instance an array of 8 bit **chars** will have the first four consecutive **chars** stored in bank 0, the next four consecutive **chars** in bank 1 and so forth. The programmer must either rearrange the **chars** before storing them in shared memory or avoid accessing consecutive **chars** belonging to the same bank. **doubles** by nature cause a 2-way bank conflict due to their size of 64 bits. One way to avoid bank conflicts with **doubles** is to split them into a high and a low part and store them separately. This may not always improve performance and will be slower on future architectures. Structs also require special attention as described in [7, p. 65]. Figure B.2 shows examples of shared memory access both with and without bank conflicts. Having several threads read from the same adress does not always cause a bank conflict. This is due to a feature called the broadcast mechanism. This allows one bank to broadcast to several threads while other threads can read from other banks simultaneously. If for instance all threads of a half-warp read the same address this can be broadcast to all threads rather than causing a

16-way bank conflict. Two examples using the broadcast mechanism are shown in figure B.3.

Registers

Registers require 0 extra clock cycles to access per instruction, but may introduce delays due to read-after-write dependencies and bank conflicts. To delays introduced by read-after-write dependencies can be ignored if there are at least 192 active threads per SM and bank conflicts are minimized if the block size is a multiple of 64. Bank conflicts are minimized at compile and thread scheduling time and can not be modified by any other means than the above mentioned.

Host memory

Transferring data between the host memory and device memory is by far the most time consuming memory operation. As such any transfers between host memory and device memory should be avoided where possible, even if it means running code that is not suitable for parallel execution on the GPU. When needed transfers between host memory and device memory should also be grouped into as few large transfers as possible.

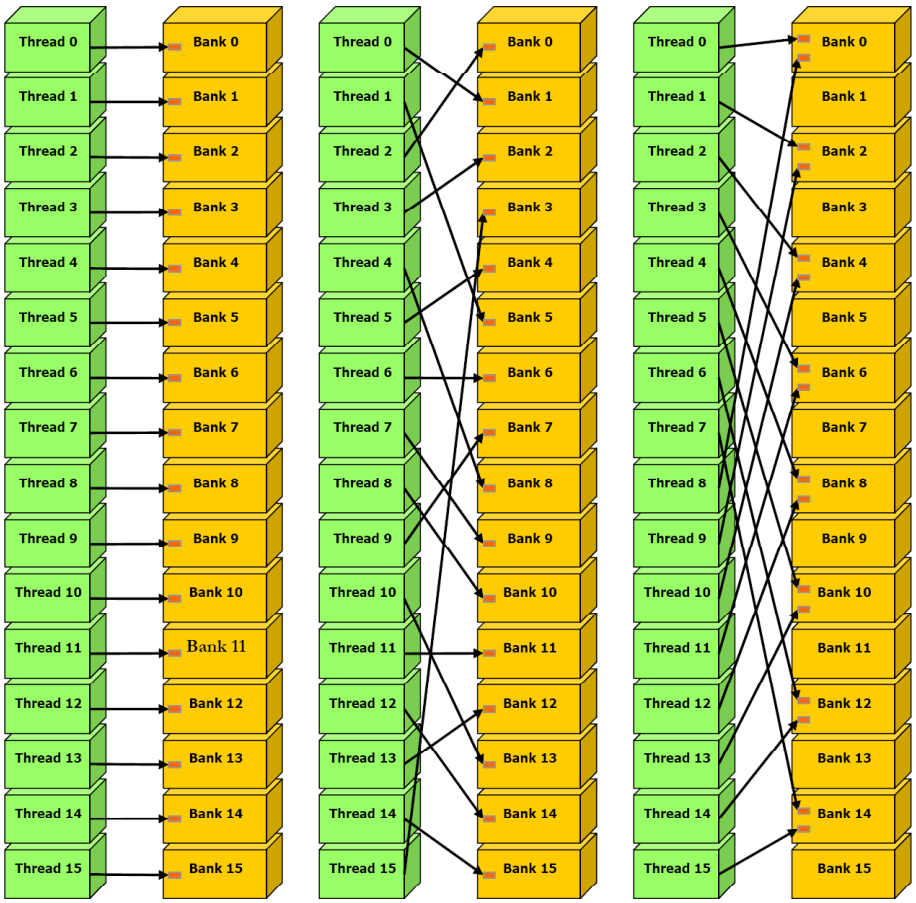


Figure B.2: Left: Consecutive threads accessing consecutive banks causing no bank conflicts.. Center: Random memory access causing no bank conflicts. Right: Memory access causing a 2-way bank conflict. [7, pp. 68-69]

B.2.3 Execution configuration

In the execution configuration two parameters are of very big importance to the performance of a kernel: The block size, **Db**, and the grid size, **Dg**. Which values of block and grid size maximizes the performance is usually dependant on the kernel, but a number of general rules can be followed.

Before considering how to maximize performance by tuning the execution configuration the programmer must ensure that the kernel can be launched. For this to be true one block must be able to execute on an SM without exceeding the constraints: Number of threads per block, available registers and available shared memory. The maximum number of threads per block and maximum number of registers and shared memory per SM is given in appendix A of [7]. The number of threads per block is only dependant on the execution configuration while the number of registers for a block can be calculated as:

$$ceil\left(R \cdot ceil(T, 32), \frac{R_{max}}{32}\right) \tag{B.1}$$

Where:
R is the number of registers required for the kernel.

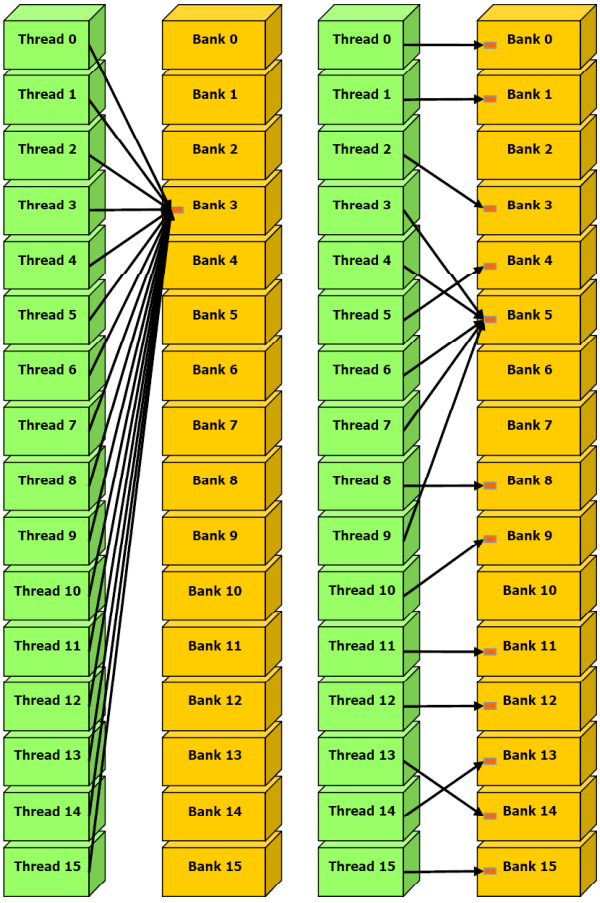


Figure B.3: Left: All threads of a half-warp accessing the same bank. Using broadcast no bank conflicts occur. Right: Severals threads accessing bank 5 and simultaneous accesses to other banks. If bank 5 is broadcast no bank conflicts occur. [7, pp. 70]

R_{max} is the total number of registers per SM.
T is the number of threads per block.
$ceil(x, y)$ is equal to x rounded up to the nearest multiple of y.

The shared memory for a block is equal to the sum of the dynamically and statically allocated shared memory and the shared memory used to pass arguments to the kernel. The number of registers, R, as well as local, shared and constant memory used by a kernel can be reported by the compiler by using the option **−ptxas-options=-v**. When these constraints are followed the kernel can be launched and the execution configuration can be optimized.

The grid should at least contain as many blocks as there are SMs on the GPU or ideally several blocks per SM. If not an SM may run idle due to having no blocks assigned or synchronization issues. If there are several blocks per SM, it will be possible to schedule another block to be executed on the SM during idle time. Blocks are also executed in a pipeline fashion meaning that several blocks being executed on one SM will not only reduce the idle time, but also overhead. Increasing the number of blocks in the grid does, however, not ensure this is the case, if the blocks assigned to an SM are not all active. For several block to be active there must be enough registers and shared memory available on the SM for all of the blocks. If performance on future generations of GPUs is a concern, the grid size should be in the proximity of 100 blocks to scale to future devices or 1,000 to scale over several generations. Due to the scheduling and execution of a warp at a time the block size should be chosen such that the number of threads is a multiple of the warp size. This will ensure warps are not underpopulated and the SPs will not run idle. If possible the block size should be chosen as a multiple of 64 due to register memory bank issues. Larger block size will be more efficient due to time slicing, however. The maximum block size must not exceed the above mentioned limit, but should be chosen such that at least two blocks can be active on one SM. The limiting factor is still the available registers and shared memory, but there is also a maximum number of active threads per SM given in appendix A of [7]. This sets another limit to how large the blocks can be while still allowing several blocks to be active. As such the minimum size of a block should be 64 threads while the maximum size is dependant on available registers and shared memory as well as the maximum number of active threads per SM.

Overall the programmer must consider several tradeoffs between grid size, block size, active threads and blocks, register usage and shared memory usage. To sum up the following statements are recommended for optimizing performance:

- The grid size should allow for at least 2 blocks per SM, ideally several. 100 or 1,000 to scale to future devices and generations, respectively.
- The block size should be multiple of the warp size or ideally a multiple of 64. Larger than 64 to achieve better time slicing.
- There should be at least 2 active blocks per SM, ideally several - not to be confused with grid size. For several blocks to be active the following limits must not be exceeded:
  - The total number of threads in the active blocks must not exceed the maximum number of active threads on an SM.
  - The total number of registers used by the active blocks must not exceed the number of registers available to an SM.

- The total amount of shared memory used by the active blocks must not exceed the amount of shared memory available to an SM.

In other words the programmer should try and make many lightweight threads. If the threads are lightweight they use few resources and the block size can be chosen large to obtain better time slicing. The number of threads must, however, not be so high that the number of active blocks is reduced, be it due to the maximum number of active threads or available registers and shared memory. The data sets in the application should also be large enough for a large grid size allowing several blocks per SM. Using current GPUs with up to 30 SMs and the minimum values for block size and active blocks per SM, 64 and 2 respectively, there will be 3,840 threads. This makes it quite clear that best performance is obtained when the data sets are large and also easier for the programmer to choose a suitable execution configuration.

Appendix C

Notation

Hermetisk transposed: A^H
Matrix transposed: A^T
Matrix conjugated (without transposed): A^*

Frequency Domain: H(w), uppercase
Time Domain: h(t), lowercase
(w) can (t) be omitted.

Matrix: $\bar{\bar{A}}$
Vector: $\bar{A}$

Subscript: Indexing or naming.
Superscripts: Operator.

j = imaginary unit.

Appendix D

Abbreviations

SIR : Signal to Interference Ratio
SNR : Signal to Noise Ratio
TITO : Two Input Two Output
GPU : Graphic Processing Unit
MATLAB : MATrix LABoratory
FLOP : FLoating point OPerations
FLOPS : FLoating point Operations Per Second
BSS : Blind Source Separation
SOS : Second Order Statistics
HOS : Higher Order Statistics
PDF : Probability Density Function
DFT : Discreet Fourier Transform
FFT : Fast Fourier Transform
SM : Streaming Multiprocessors
SP : Scalar Processor
CUDA : Cumpute Unified Device Architechture