
133
599
THS

This is to certify that the

thesis entitled

ON-LINE BLIND SIGNAL SEPARATION
OF SPEECH SOURCES

presented by

Walter Andres Zuluaga

has been accepted towards fulfillment
of the requirements for

Master's degree in Electrical Eng

Major professor

Date 5/12/98

ON-LINE BLIND SIGNAL SEPARATION TO SPEECH SOURCES

By

Walter Andrés Zuluaga

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Engineering

1998

ABSTRACT

ON-LINE BLIND SIGNAL SEPARATION APPLIED TO SPEECH SOURCES

By

Walter Andrés Zuluaga

Over the past decade, a great amount of literature on Independent Component Analysis (ICA) and blind signal separation has been reported. The ICA concept pertains to separating mixtures of signals that have been mixed algebraically by a constant, but unknown, matrix. There are many areas where the ICA concept may potentially be applied, e.g. separation of signals in communications, medical diagnosis, and speech processing. The objective of this thesis is to investigate and evaluate the real-time implementation of some of the algorithms found in the literature, and thus assesses the feasibility of using them in on-line applications. A DSP system is used to implement and execute the algorithms in an on-line environment

The signals to be separated are assumed to be in the human vocal bandwidth range. A pair of mixed signals is considered to be the input to the system. The

mixed signals are sampled by the DSP system and processed according to the algorithms. The generated pair of outputs is taken to be an estimate of the original unmixed signals. The implemented algorithms vary depending on the functions used to extract the high order statistics, and on the functions used to govern the learning rate of the learning rule. A performance measure to compare the performance of the various algorithms is proposed in this investigation.

To my exceptional parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Technologies allow the user to interact with electronic devices by means of his voice, either as a source of information to be transmitted or as the means to control equipment. This can be seen, for example, in the increase in use of cellular phones and the control of a computer by using the person's voice. These kind of applications require an improvement in the processing of the speech signals, by isolating the voice from the surrounding environment. It is not sufficient to reduce the background noise, but also to remove the unwanted voice signals that might share similar characteristics. The final objective is to perform the speech separation in voice controlled systems isolating the controlling signal, or to obtain a better quality in the transmission of voice.

This type of application require on-line processing, in order to perform the separation at the same time the signals are produced. As the user generates the speech signal, it must be processed without significant delay. In some other applications this may not be necessary, e.g. the separation of EEG signals, but for the present scenarios an immediate response is necessary.

The present thesis looks to implement, in a real time environment, various learning algorithms proposed in the literature [5][12] to separate blind signals, by

means of linear neural networks. Their feasibility is going to be tested, so they can be applied in the future to such mentioned devices, focusing on speech separation but not limited to this field.

The thesis is organized as follows. The first chapter covers the theoretical background, stating the problem to be addressed and developing the equations and learning rules to be applied, based on the works done by Amari *et al* [5] and by Cardoso [12]. The selection of the different parameters of the algorithm will also be studied, namely the functions applied over the input signals and the learning rate functions to be used. The second part describes the implementation of the algorithm in different environments, its design and how it works, emphasizing the problems encountered for it to work adequately in real time. A description of the DSP environment is also included. The third chapter covers the tests executed to study the performance of the different algorithms, and a measure of the intersymbol interference is provided, in order to evaluate the quality of the weight matrix obtained. Finally, the last chapter contains conclusions and recommendations drawn from the results obtained.

1 Background Theory

1.1 Blind Signal Separation

Blind signal separation consists of separating statistically independent or nearly independent signals, which have been mixed by an unknown medium from which there is no previous information. It is applied to different areas such as array processing, communications like multipath propagation, medical signal processing like isolating artifacts from heart and brain signals, and speech processing [10].

Consider a system with n statistically independent sources $s_i(t)$, which by means of a mixing medium represented by a *mixing matrix* A, produces m signals $x_j(t)$. The separation of the signals is made by calculating adaptively a *separating matrix* W(t), without any *a priori* information of the mixing matrix, and obtaining an estimate $y_i(t)$ of the input signals.

The block diagram of this system is as follows:

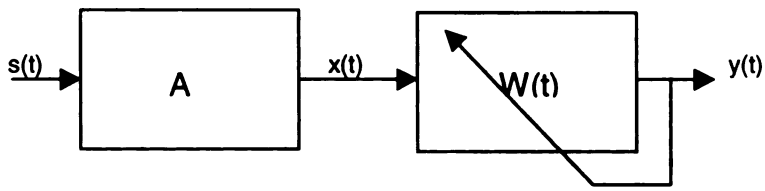


Figure 1. Model of the system.

1.2 Signal Separation.

1.2.1 Independent Component Analysis

A linear adaptive method, called Principal Component Analysis (PCA), is used to decompose patterns (for this specific case speech signals) based on the covariance among the components input signal, and thus considering only their second-order statistics. This kind of decomposition is appropriate for the separation of Gaussian signals, but since other signals have higher order statistics, the PCA approach is not suited for signal separation in more general applications [9] [11].

For example, in speech signals, the information is contained in both the amplitude and phase spectra. However, the autocorrelation of a speech signal, a second-order statistic, only carries information about the amplitude spectrum, and the rest of the information is discarded. Consequently for signal separation,

it is necessary to include higher-order statistics that carry the missing information [1].

Independent Component Analysis (ICA) may be viewed as a nonlinear version of the PCA in incorporating high-order statistics. More information is contained in these statistics, and the signals are assumed independent, stationary zero-mean stochastic processes. Many different statistics can be used, so the problem is to choose the most appropriate for the case considered.

Adaptive separation based on ICA [2] is performed by adjusting the separation matrix, using the mixed signal vector x(t), according to the equation

$$W(t+1) = W(t) - \mu(t)\, f(\, W(t), x(t)\,) \qquad (1)$$

where the term $\mu(t)$, known as adaptation rate, is used for stabilization and convergence, and f(W(t), x(t)) is the function that describes the learning algorithm. It is necessary for the algorithm to be equivariant[12], in order for the estimator to be totally independent of the mixing matrix. If the estimator *A* is considered as a mapping, the equivariance would be satisfied if

$$A(MX) = M(AX) \qquad (2)$$

with M an invertible matrix and X a data set. Once this condition is satisfied, the performance of the algorithm does not depend on the mixing matrix, but only on the present value of the separation matrix and the input data [2].

Consider the following updating algorithm:

$$W(t+1) = W(t) - \mu(t)\, H(y(t))W(t) \qquad (3)$$

If the global system is represented by C(t)=W(t)A, then left multiplying by A we obtain:

$$W(t+1)A = W(t)A - \mu(t)\, H(y(t))W(t)A$$

$$C(t+1) = C(t) - \mu(t)\, H(y(t))C(t)$$

$$C(t+1) = C(t) - \mu(t)\, H(W(t)Ax(t))C(t)$$

$$C(t+1) = C(t) - \mu(t)\, H(C(t)x(t))C(t) \qquad (4)$$

$$C(t+1) = \{ I - \mu(t)\, H(C(t)x(t))\}C(t) \qquad (5)$$

This algorithm (5) is equivariant for the global system, and therefore does not rely on the mixing matrix. This kind of algorithm is known as serial updating [12].

Based on this update law, two different classes of H(**y**(t)) can be considered, based on the form of this function [2]: symmetric and non-symmetric.

- Symmetric form. Two approaches to obtain the function are considered. The first one consists of taking a stochastic approach.

In order to find the function H(y), it is necessary to make the assumption that the sources are independent, that is, the mutual information is 0. So, the algorithm solves for the value of W with the constraint

$$E\, H(y(t))=0 \qquad (6)$$

hence obtaining a stationary point. It is necessary to minimize an objective function that according to Cardoso [12], takes the form:

$$c(B) \equiv E\phi(\mathbf{y}) = E\phi(\mathbf{Bx}) \qquad (7)$$

with ϕ a differentiable function. Expanding $\phi(\mathbf{y} + \delta\mathbf{y})$ we obtain

$$c(((I-E)B)) = c(B) + E\phi'(\mathbf{y})^{t} E\mathbf{y} + o(\delta E) \qquad (8)$$

Minimizing c(B) is accomplished by choosing E proportional to

$$\lambda E\phi'(\mathbf{y})^{t} E\mathbf{y} \qquad (9)$$

Finally, the function is obtained when the expectation in equation (9) is dropped. Hence, the obtained equation uses the function

$$H(\mathbf{y}(t)) = \phi'(\mathbf{y})\mathbf{y}^t \tag{10}$$

The other considered possibility comes from a statistical consideration. Assuming a differentiable probability distribution p_i for each of the signals s_i, the function can be formed as

$$H(\mathbf{y}(t))_{ij} = \psi_i(y_i)y_j - \delta_{ij} \tag{11}$$

Where

$$\psi_i = {}^{-p_i'}\!/_{p_i} \tag{12}$$

This kind of function satisfies the maximum likelihood estimate in applying the previous update rule.

- Non symmetric functions. These functions originate from using high order statistics and optimizing them, since they contain information related to the independence between the signals. Even as high order statistics are difficult and computationally expensive to calculate, a high order statistic quantity like the kurtosis (a fourth order cumulant), is easily implemented and yields a good performance in signal separation. This process is composed usually of two stages, the whitening stage and the orthogonal stage [11] [12] [4].

- Whitening. This stage preprocesses the signals in order to obtain new signals with a covariance matrix approaching the identity. The corresponding update law, see Cardoso [12], is developed as follows:

The objective function to be minimized, with $R_z=Ezz^t$ as the covariance, is:

$$\Psi(S) \equiv \mathrm{Trace}(\mathrm{R}_z) - \log\det(\mathrm{R}_z) - n \tag{13}$$

provides the following expansion:

$$\Psi((I+E)S) = \Psi(S) + 2(\mathrm{E}\mathbf{z}\mathbf{z}^t - I)E + o(E) \tag{14}$$

Subsequently the update rule for S (the whitening or sphering matrix) is

$$S(t+1) = [I - \lambda(\mathbf{z}(t)\mathbf{z}(t)^t - I)]S(t) \tag{15}$$

- Orthogonalization. This stage optimizes the fourth order cumulant. It minimizes the equation

$$\Phi \equiv \sum_{i=1,n} \mathrm{E}|y_i|^4 \tag{16}$$

given that the input signals' covariance matrix is equal to the identity, which is already satisfied by the whitening stage. The update law for the *orthogonalization matrix* U, is then computed as follows:

Let c(U) be the objective function:

$$c(U) \equiv E\phi(\mathbf{y}) = E\sum |y_i|^4 \tag{17}$$

Applying the same expansion as in (8), one gets

$$c((I-E)U) = c(U) + \mathrm{E}\phi'(\mathbf{y})' E\mathbf{y} + o(E) \tag{18}$$

This equation has to be changed slightly, in order to satisfy the unitary constraint on U (i.e. $UU^t=I$), so it is necessary for E to be skew-symmetric (i.e. $E=-E^t$).

Choosing E proportional to

$$\mathrm{E}\left[\mathbf{y}\mathbf{y}' - I + \phi'(\mathbf{y})\mathbf{y}' - \mathbf{y}\phi'(\mathbf{y})'\right] \tag{19}$$

the resulting function is

$$H(\mathbf{y}(t)) = \phi'(\mathbf{y})\mathbf{y}' - \mathbf{y}\phi'(\mathbf{y})' \tag{20}$$

and its corresponding update rule is

$$U(t+1) = \left\{I - \lambda\left[\phi'(\mathbf{y})\mathbf{y}' - \mathbf{y}\phi'(\mathbf{y})'\right]\right\}U(t) \tag{21}$$

Combining both update rules, (15) and (21), into one:

$$W(t) = U(t)S(t) \tag{22}$$

$$S(t+1) = \left[I - \lambda\left(\mathbf{z}(t)\mathbf{z}(t)' - I\right)\right]S(t) \tag{23}$$

$$U(t+1)=\{I-\lambda[\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t]\}U(t) \tag{24}$$

$$W(t+1)=U(t+1)S(t+1)$$
$$W(t+1)=\{I-\lambda[\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t]\}U(t)[I-\lambda(\mathbf{z}(t)\mathbf{z}(t)^t-I)]S(t) \tag{25}$$

$$W(t+1)=\{I-\lambda[\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t]\}U(t)-U(t)\lambda(\mathbf{z}\mathbf{z}^t-I)]S(t) \tag{26}$$

$$W(t+1)=\{U-\lambda\{U(\mathbf{z}\mathbf{z}^t-I)+[\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t]U\}+\lambda^2[\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t](\mathbf{z}\mathbf{z}^t-I)U\}S \tag{27}$$

Disregarding the term with λ^2:

$$W(t+1)=\{U-\lambda\{(U\mathbf{z}\mathbf{z}^t-U)+[\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t]U\}\}S \tag{28}$$

Since $U^tU=I$:

$$W(t+1)=\{U-\lambda\{(U\mathbf{z}\mathbf{z}^t(U^tU)-U)+[\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t]U\}\}S \tag{29}$$

$$W(t+1)=\{US-\lambda\{((U\mathbf{z})(\mathbf{z}^tU^t)US-US)+[\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t]US\}\} \tag{30}$$

With $\mathbf{y}=U\mathbf{z}$, $\mathbf{y}^t=\mathbf{z}^tU^t$ and $W(t)=U(t)S(t)$

$$W(t+1)=\{W(t)-\lambda\{(\mathbf{y}\mathbf{y}^t-I)W(t)+[\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t]W(t)\}\} \tag{31}$$

$$W(t+1)=\{I-\lambda\{(\mathbf{y}\mathbf{y}^t-I)+[\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t]\}\}W(t) \tag{32}$$

The update law is of the form

$$W(t+1)=W(t)-\mu(t)H(\mathbf{y}(t))W(t) \tag{33}$$

where

$$H(\mathbf{y}(t))=(\mathbf{y}\mathbf{y}^t-I+\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t) \tag{34}$$

1.2.2 Algorithms and learning rules.

The following learning algorithms are for a linear feed-forward neural network, represented by the weight matrix W[t]. The input speech signals are assumed to be statistically independent.

From the previous section, the update algorithm to be used is

$$W(t+1)=W(t)-\mu(t)H(\mathbf{y}(t))W(t) \tag{33}$$

with different options for the function H(**y**(t)):

- Stochastic approach

$$H(\mathbf{y}(t))=\phi'(\mathbf{y})\mathbf{y}^t \tag{35}$$

- Maximum likelihood estimate

$$H(\mathbf{y}(t))_{ij}=\psi_i(y_i)'y_j-\delta_{ij} \quad , \text{ with } \quad \psi_i=-p_i'/p_i \tag{36}$$

- Optimization of a contrast function

$$H(\mathbf{y}(t))=\left(\mathbf{y}\mathbf{y}^t-I+\phi'(\mathbf{y})\mathbf{y}^t-\mathbf{y}\phi'(\mathbf{y})^t\right) \tag{37}$$

Both teams of Cardoso and Amari, use the serial update rule, but the choice of the function H(y) and the learning rate function differ in both cases, along with the measure of the performance.

1.2.2.1 An algorithm by Cardoso

Cardoso [12] uses the adaptive algorithm described in the previous section:

$$\mathbf{W}[k+1] = \mathbf{W}[k] + \mu[k]\{\mathbf{H}(\mathbf{y}[k])\}\mathbf{W}[k]$$

The simulation results obtained by Cardoso use a symmetric function with $\phi(y)=|y_i|^2 y_i$, and a constant value for $\mu(t)$ [12]. The performance is measured by the intersymbol interference: by multiplying the resulting separating matrix W with the mixing matrix A, a matrix close to the identity should be obtained. The energy of the signal in each of the estimates is obtained by $|WA_{ij}|^2$. Hence, the relative power of the j^{th} source in the l^{th} signal estimate is $|WA_{ij}|^2$.

1.2.2.2 An algorithm by Amari

The learning rule proposed by Amari *et al* is the following [5]:

$$\frac{dw_{ij}(t)}{dt} = \mu_{ij}(t)\left\{ w_{ij}(t) - f_i[y_i(t)]\sum_{p=1}^{n} w_{pj}(t)\, y_p(t) \right\} \qquad (38)$$

Expressed in vector form becomes

$$\frac{d\mathbf{w}(t)}{dt} = \mu_j(t)\left\{\mathbf{w}(t) - \mathbf{f}\left[\mathbf{y}(t)\right]\mathbf{y}^T\mathbf{w}(t)\right\} \tag{39}$$

The last equation in discrete time is given as

$$\mathbf{W}[k+1] = \mathbf{W}[k] + \mu[k]\left\{\mathbf{I} - \mathbf{f}(\mathbf{y})\mathbf{y}^{\mathrm{T}}\right\}\mathbf{W}[k] \tag{40}$$

the serial update algorithm is obtained.

Amari *et al* [5] proposes for f(y) (H(y) in this text) to be typically of the form $f(y)=y^{2k+1}$, clearly a non-symmetric function, and therefore a stochastic approach solution. The learning rates used by Amari *et al* [5] are of two types:

Standard: The learning rate is a predefined function, for example a decaying exponential, of the form

$$\mu(t) = \mu_0 e^{-t/\tau} \tag{41}$$

With this kind of function, the search is executed initially with big steps to find rapidly an energy minimum and, as the learning rate is reduced, the search becomes finer.

Adaptive: The value of the learning rate is established by a set of differential equations. This set allows the learning rate to increase in case one parameter of the update rule is driven high. This might happen when the input signals change,

after the algorithm already has converged. In this case, the learning rate will have a large value and the search algorithm will start again with coarse steps.

The proposed set of differential equations for updating this learning rate is:

$$\tau_1 \frac{dv_{ij}(t)}{dt} = -v_{ij}(t) + g_{ij}(t)$$
$$\tau_2 \frac{d\mu_{ij}(t)}{dt} = -\mu_{ij}(t) + \alpha\left|v_{ij}(t)\right| \qquad (41)$$

where $\tau_1,\tau_2>0$, $\alpha>0$ and

$$g_{ij}(t) = \left\{ w_{ij}(t) - f_i[y_i(t)] \sum_{p=1}^{n} w_{pj}(t) y_p(t) \right\} \qquad (42)$$

In discrete time, one has

$$v_{ij}[k+1] = (v_{ij}[k](\tau_1 - 1) + g_{ij}[k])/\tau_1$$
$$\mu_{ij}[k+1] = (\mu_{ij}[k](\tau_2 - 1) + \alpha\left|v_{ij}[k]\right|)/\tau_2 \qquad (43)$$

When these equations are discretized, the numerical integration method used may yield some numeric imprecision in the learning algorithm. The next chapter discusses this area.

2 Implementation

The design of the algorithm assumed that two different mixed signals were going to be sampled. The program was developed with this feature in mind, but with the possibility to increase the number of mixed signals if it becomes necessary. It was also assumed that for n source signals, there are going to be n corresponding channels, meaning that a square matrix would represent the separation matrix.

The signal separation algorithm was implemented in two different environments. The first was an off-line algorithm, used to check the algorithm's behavior and feasibility, allowing a fast prototyping of the program, since it is easier to modify it under controlled conditions. After certain tests were performed with predetermined speech signals and the parameters were chosen, the program was migrated to an on-line process of the DSP system

.

2.1 Off-line

The program was initially designed on a Sun SPARC workstation, to run under the Matlab 5.0 environment. Speech samples available on Matlab were used, and the performance of the algorithm in separating the signals was tested. Note that no speed performance on how fast the algorithm was achieving this separation was analyzed, since the final objective of the work was the on-line implementation. It is enough to say that the separation took about 6 times the size of the sample. After the parameters were adjusted to obtain a good output for certain random mixing matrices, the program was implemented in ANSI C in the same workstation.

This next step is logical, since the C compiler for the DSP was going to be used to generate the assembler program. It would also be a program of a lower level than the one made in Matlab, allowing better control over the low-level functions and optimizations, and therefore increasing the speed of execution. Still, Matlab was being used as the output interface of the program. The same tests executed on Matlab were executed in the C program, obtaining the same results both in the weight matrix and in the appearance of the output signals. As expected, the

time performance was improved in almost half the execution time, being appropriate for an off-line process, where immediate results are not needed.

Note that the program was made with a static data structure, since the test signals to be used were finite and had a predetermined length, read at the beginning of the program in one stage.

The following example figures show the behavior of the algorithm initially implemented on the PC. Two speech signals were applied, each of 42028 samples. The corresponding waveforms are shown in Figure 2. These two signals were mixed through a random matrix, obtaining the mixed signals to be processed and separated by the algorithm. The resulting signals are shown in Figure 3. Note the dominance of signal s1(t) over the other signal s2(t). The algorithm was applied to the signals and the corresponding signal estimates were obtained, as depicted in Figure 4 and Figure 5, where each of the estimates (blue) is accompanied by the original signal (green). The estimate of signal s1(t) resembles accurately the original signal, but the other estimate still presents some crossover. This is due to the attenuation of this signal in certain periods of time.

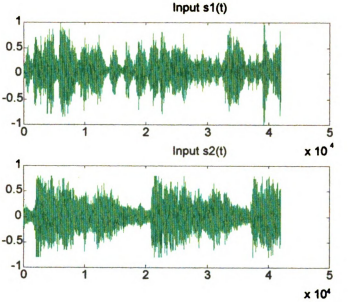


Figure 2. Input Signals

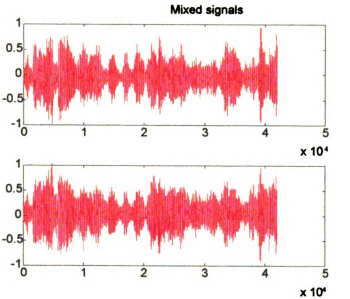


Figure 3. Mixed Signals

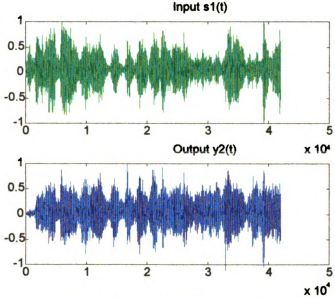


Figure 4. Estimated Signal for s1(t)

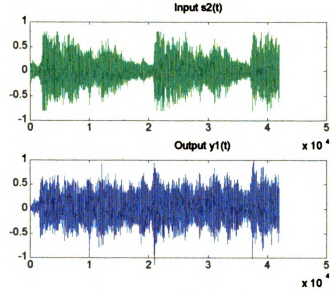


Figure 5. Estimated Signal for s2(t)

Since the learning rate is adaptive, the presence of such periods effects the value of the parameter, producing big changes in the weight matrix and therefore altering the output. Even if the separation matrix has the ideal values, such a disturbance will alter the coefficients; when the strength of the signal rises again, the values will converge to the previous value.

This can be seen in Figure 6. There, the signal s2(t) (green) is illustrated with the estimated signal (blue) and the behavior of the individual weights in time. When the strength of the signal decreases (region A), a variation of the learning rate is reflected in the weights and therefore the estimates are affected. The influence is higher in the first portion, since the main envelope of the learning rate is higher there. As time progresses, the learning rate will decrease even if disturbances are present, and therefore the influence of these "silent periods" will diminish with time. If the final weight matrix were used without allowing any other change over the subsequent samples, a very good separation will be obtained, given that the mixing matrix would not change after that point. Although the signals may not look similar in some regions, the sound produced by the estimates is close to that of the original unmixed sounds, previous to the mixing.

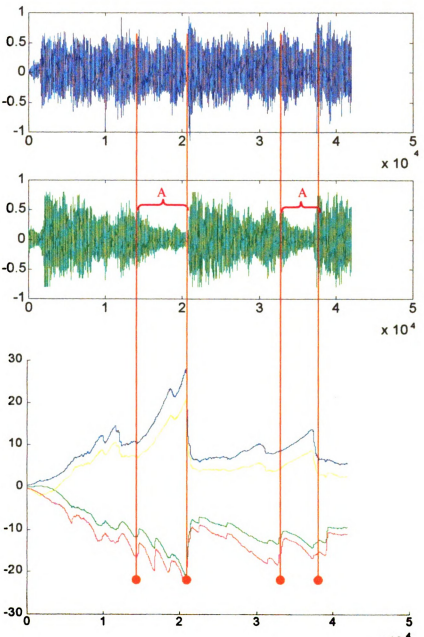


Figure 6. Influence of silence periods

2.2 Real Time

After the program was tested on the workstation, it was modified to satisfy the requirements of the DSP environment. Even as the algorithm is the same, several modifications had to be made to optimize the algorithm to fit the DSP structure and change the source of the input signals. In addition, a PC program was prepared to interface with the DSP board. It was also necessary to change the static data structure already mentioned, since the signals were not going to arrive all at the same time and be stored, but had to be processed one by one, as they were sampled.

As a first step, the program was migrated to the DSP and the input signals were sent from the PC, being the same ones used in the workstation. After the program was running smoothly, the source of the signals was changed to the Daughter Modules where analog mixed signals were sampled and processed directly.

2.2.1 DSP Environment

The DSP environment where the application was implemented consists of a development module based on a TMS320C32 DSP from Texas Instruments [13].

It uses a PC as a host with a bank of DPRAM for fast data exchange. It also has two banks of SRAM for data storage and program execution. In addition, the PC/C32 interfaces with an on-board LSI daughter module with dual channel ADC/DAC ports. Following is the description of each of the parts [15] [16] [17].

2.2.1.1 DSP

The TMS320C32 is a 60MHz digital signal processor, with 32 bit floating point capabilities and a peak performance of 60 MFLOPS.

The C32 has a single cycle floating-point/integer multiplier that can operate in parallel with the ALU. The ALU also performs single cycle operations, and the mentioned parallelism is accomplished by the use of two CPU buses that carry operands from memory and two register buses, carrying the data from the register file. The CPU also has two auxiliary register arithmetic units, which generate addresses.

The register file consists on a total of 28 registers. Of these, 8 are extended precision registers, supporting 32 bit integer numbers and 40 bit floating point numbers. There are 8 auxiliary registers, used for indirect addressing, loop

counters or 32-bit general purpose registers. The other registers are for specific purposes like data-page pointer, stack pointer, indexing, etc.

The memory organization for the C32 model consists on two RAM banks, each of 256x32 bits. These banks can be accessed twice on a single cycle, taking advantage of two address buses. It also has a cache memory, storing 64x32 bit instructions, reducing the number of off-chip accesses.

The system has 256kbytes of extended memory, with certain restricted to memory positions, reserved for the system.

2.2.1.2 Interface

The communication between the PC and the DSP is made through two types of interfaces: a memory mapped interface and an I/O mapped interface.
The memory mapped interface is done through a 2k x 16 DPRAM. This memory allows interchanging information between the DSP and the PC, without interrupting the process in either of the devices. Since any sector of the DPRAM can be accessed at the same time, a semaphore logic is provided, allowing to control the resource and avoiding data corruption.

The I/O mapped interface consists of five 16 bit registers in the PC I/O map as a control interface providing access to software configurable functions and other facilities. The five registers are a control register, a status register, interrupt register, DPRAM base address register and semaphore register.

2.2.1.3 Signal Acquisition

The signal acquisition is done by using the LSI AM/D16SA daughter module. This is an on-board module that provides two Analog/Digital Converters allowing two input signals to be sampled, and two Digital/Analog Converters for the outputs. Each channel uses 16 bits-based and the conversion, made by successive approximations, can attain a maximum sample rate of 200kHz. There are also input and output filters, which can be changed according to the application. The configuration and data registers are mapped in the memory of the DSP, and allow a large range of operations.

2.2.2 Program and Algorithms

The program implemented on the system is composed of two different parts, one running on the PC where the DSP board is installed and the other running on the DSP. Both programs are synchronized by means of a software semaphore

implemented on the system, preventing data corruption in the DPRAM, as it might be changed by any of the programs at any time.

The design of the DSP program was made using a polling model, instead of interruptions, of the input and output ports. When interruptions are used, the program is "interrupted" from the normal flow; in this case, it would be interrupted every time a new sample is acquired. Since it is not desirable to interrupt in the middle of executing the update rule, which is the dominant part of the program and uses the input data for its computations, interruptions would have to be disabled for this part. Moreover, all the calculations depend on the input data and therefore there is one single program path, so the use of interruptions would unnecessarily add an overhead each time it is invoked. When polling is used, the program asks the port regularly if new data has arrived, and when it does, it proceeds to use it to perform the corresponding calculations. When the program is done, it waits for the next samples. The description of each of the programs now follows. The flow chart of both the PC and DSP programs are in Figure 7.

2.2.2.1 PC program (host)

1. The host program takes care of the initialization of the DSP board. First, the I/O address and the DPRAM position are established, and with these values the board is initialized.

2. The assembled DSP program is loaded into the DSP memory and the program is executed from that position.

3. The PC gets the semaphore and sends the following information: Initial weight matrix, initial value for the learning rate and different parameters for the update rule. It also sets the EndData Flag to 1, indicating to the DSP that it can start acquiring data from the input ports.

4. The semaphore is released, and the program waits for the input from the user that establishes the end of the process.

5. When this is done, the semaphore is requested by the PC, and after this the EndData Flag is set to 0.

6. The semaphore is released, waiting for the DPS to return the final values, and when the PC is in possession of the semaphore, these values are read, and the DSP system is closed.

2.2.2.2 DSP program

Two DSP programs were implemented during the development of the thesis. The difference between these two versions is the source signals. While in the first program the input came from the PC by means of the DPRAM, the second one obtained the training data from the input ports of the daughter module. The following explanation of the program corresponds to the latter implementation, since it is the one to be executed in real-time.

2.2.2.2.1 Program flow

The host program dictates when the DSP program starts executing.

7. As soon as the DSP program starts, it waits for the semaphore to be released, in order to read the initial parameters of the implemented algorithm. Then, the Daughter Modules are initialized, setting the clock source, the sampling rate and other configuration parameters. The DSP releases the semaphore, so the PC program can change the EndData flag. Next, the main loop starts based on the EndData flag, set by the PC. As long as this flag is 1, the loop is executed.

8. The DSP polls the input port until a new sample is obtained.

9. The main loop is composed of three stages. The first stage reads the input buffer status, and reads the data from the two input ports when the buffer is full.

10. The second stage calculates the output data based on the last value of the weight matrix and the result is sent to the output ports.

11. The last stage calculates the update rule for the weight matrix, depending on the input values.

12. Once the EndData flag is set to 0, the loop finishes, and the semaphore is requested by the DSP.

13. When it is free, the weight matrix is written to the DPRAM, and the semaphore is released again.

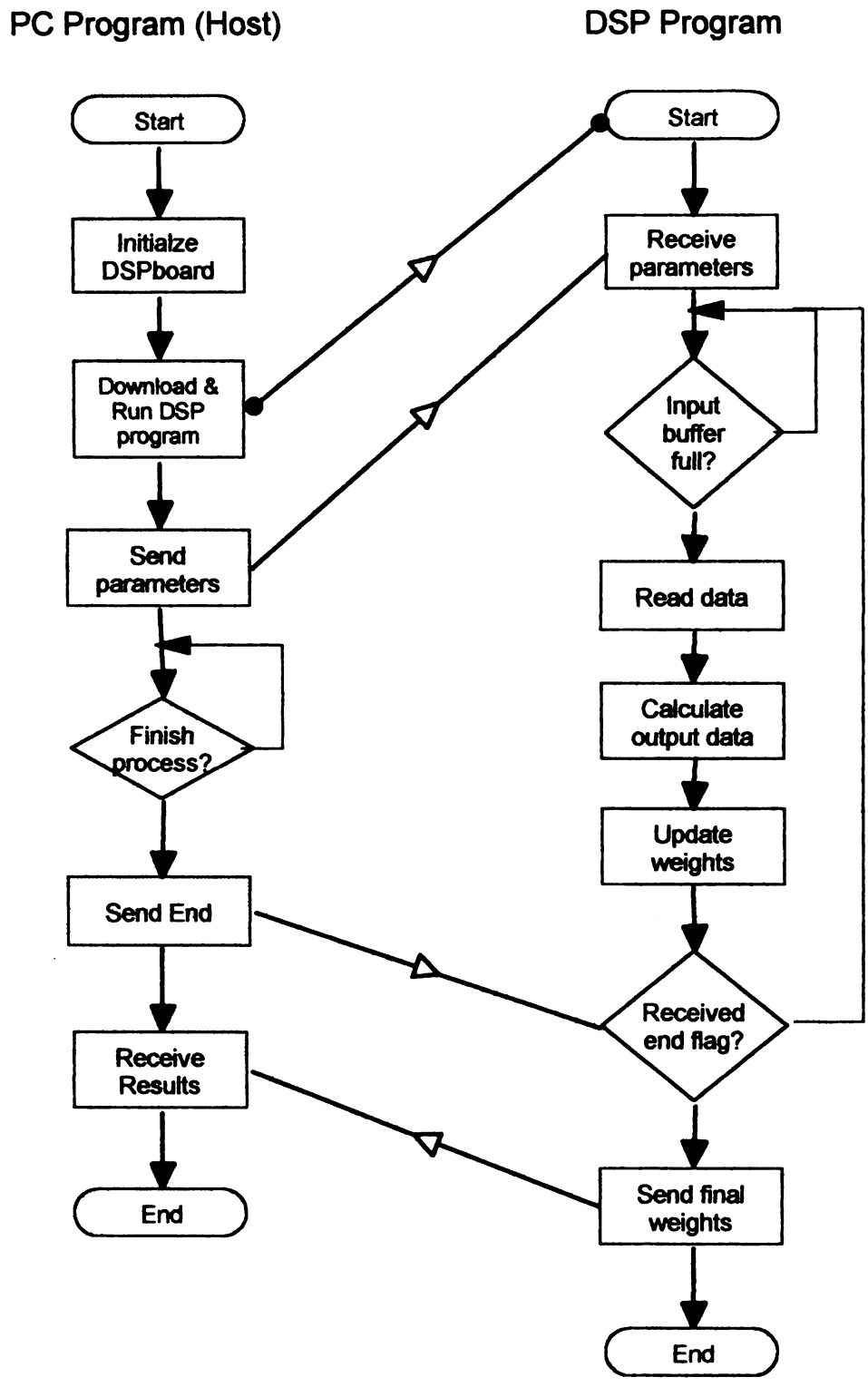


Figure 7. Program flowchart

2.2.2.2.2 Considerations in the program design

Different considerations had to be taken into account in the realization of the program. First, the numerical approximation of the update rule. The original update rule is continuous, and hence a numerical integration method must be used. In this case the selected method is the rectangular Riemann sum, which will be described in the next section. This approximation is also applied to the differential equations defining the learning rate in the adaptive rule by Amari *et al* [5].

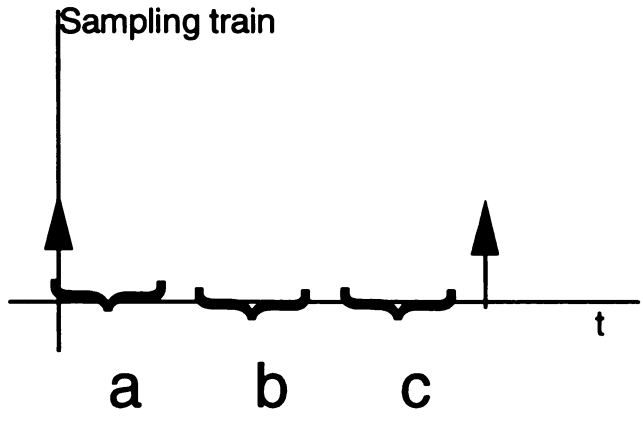
The other important factor is the sampling rate. The frequency at which the input signal is sampled has a lower limit and an upper limit. The lower limit is the Nyquist rate, which depends on the bandwidth of the signals to be processed. For a signal with bandwidth ω_m, the sampling frequency ω_s is determined by

$$\omega_s > 2\omega_m \tag{44}$$

in order to be able reconstruct the original signal.

Since this is a theoretical value, a higher value is to be used as the lower bound of the sampling frequency.

The upper limit is determined by the time required to process the data between samples. If after getting a sample the program takes a long time to execute before reading the next sample, it can lose this sample, and therefore the sampling frequency would be altered. The minimum time between samples can generally be broken in three portions (Figure 8): (a) reading the input data, (b) calculating and writing the new output data, and (c) updating the weight matrix. Of these stages, the last one consumes the longest time. Stage (c) also depends on the particular algorithm implemented where the update rule and the learning



rate may require special consideration.

Figure 8. Sampling and program length.

The frequencies used as lower and upper limits were 40 kHz and 50 kHz respectively. For the Nyquist rate, it was chosen based on the input low-pass filter, with a cut-off frequency of 18kHz. Hence, choosing this as the bandwidth of

the signal, a sampling frequency of 36kHz is obtained. The 40 kHz frequency is selected to obtain a safe region. The upper frequency (50kHz) was obtained by executing the largest algorithm, and visually checking the form of the known output signals, generated by echoing the input signals.

2.2.2.2.3 Implemented Algorithms

The implementation of the algorithms only vary in the selection of the H(y) function, depending on the optimization method selected (stochastic, maximum likelihood estimator and contrast function) and in the selection of the learning rate.

Of the three optimization methods, the maximum likelihood estimator was not considered, since it is based on the knowledge or assumption of the probability distribution of the input signals. The other two were considered, selecting two non-symmetric functions (stochastic) and one symmetric function (4^{th} order cumulant). These functions are:

Non-symmetric functions:

$$H(\mathbf{y}) = \mathbf{y}^3(t)\mathbf{y}(t)^T \tag{45}$$

$$H(\mathbf{y}) = \mathbf{y}^5(t)\mathbf{y}(t)^T \tag{46}$$

Symmetric function:

$$H(\mathbf{y}) = \mathbf{y}(t)\mathbf{y}(t)^T - I + \mathbf{y}^3(t)\mathbf{y}(t)^T - \mathbf{y}(t)\mathbf{y}^3(t)^T \tag{47}$$

For the learning rate, two different functions were used. The first one is a piecewise function, where the first part is a constant value μ_0, the second part is a decaying exponential and the last piece is a constant value of 0 meaning that the update is stopped (see Figure 9). The constant part allows the algorithm to take coarse steps in the search space, and after some time, finer steps are taken until they go to 0.

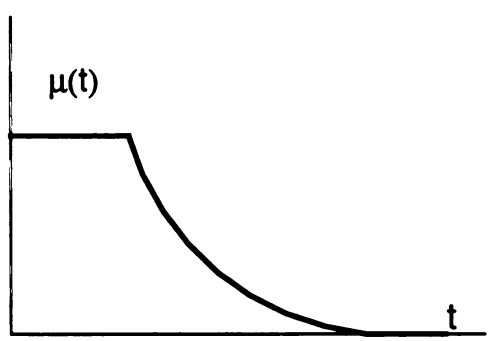


Figure 9. Learning rate

The second function is the solution of the differential equations proposed by Amari's group, where the learning rate is adaptive. This system of equations is rewritten here:

$$\nu_{ij}[k+1] = (\nu_{ij}[k](\tau_1 - 1) + g_{ij}[k])/\tau_1$$
$$\mu_{ij}[k+1] = (\mu_{ij}[k](\tau_2 - 1) + \alpha|\nu_{ij}[k]|)/\tau_2$$

For both implementations, the gradient descent update rule and the system of first order differential equations that describe the behavior of μ(t), two numerical integration methods based on the Riemman sum were used. The first and more basic is the rectangular rule (or the Euler approximation) given by the formula

$$R_n(f) = h\sum_{k=1}^{n} f(a+kh), \qquad h = (b-a)/n \tag{48}$$

$$R_n(f) = h\sum_{k=1}^{n-1} f(a+kh) + hf(b) \tag{49}$$

$$R_n(f) = R_{n-1}(f) + hf(b) \tag{50}$$

Since

$$\mathbf{w}_j(t) = \int_0^t \mu_j(t)\{H(t)\mathbf{w}(t)\}dt \tag{51}$$

using the approximation,

$$\mathbf{w}[k+1] = h\mu_j[k]\{H[k]\mathbf{w}[k]\} + \mathbf{w}[k] \tag{52}$$

the discrete update rule to be implemented is obtained by setting h=1.

The second method is the trapezoidal rule given by

$$T_n(f)=h\left[\frac{f[a]}{2}+f[a+h]+f[a+2h]+\cdots+f[a+(n-1)h]+\frac{f[b]}{2}\right], \qquad h=(b-a)/n \tag{53}$$

where by manipulating the terms we obtain

$$T_n(f)=T_{n-1}(f)+h\left[\frac{f[a+(n-1)h]}{2}+\frac{f[b]}{2}\right], \qquad h=(b-a)/n \tag{54}$$

Applying it to the gradient descent rule the update rule becomes

$$\mathbf{w}[k+1]=\mathbf{w}[k]+\{h\mu_j[k]\{H[k]\mathbf{w}[k]\}+h\mu_j[k-1]\{H[k-1]\mathbf{w}[k-1]\}\}/2 \tag{55}$$

Both rules were implemented and tested for different mixing matrices. Since no improvement in the performance was noticed, the simplest rule (the rectangular) was used. Note that the implementation of the trapezoidal rule adds more programming lines to the algorithm, affecting the length of the code and therefore limiting the frequency constraints already enumerated.

3 Tests and Results

To test the implementation of the on-line separation algorithms and the feasibility of applying them in different systems, several tests were performed. Different algorithms were programmed, depending on the approaches listed in the previous chapters. The update rule

$$\mathbf{w}_j[k] = h\mu_j[k]\{H[k]\mathbf{w}[k]\} + \mathbf{w}[k]$$

is common to all the algorithms, but variations were introduced on the function H[k] and on the function describing the learning rate μ(t). Three H[k] functions already described and two different descriptions of the learning rate yield six algorithms to be tested.

The complexity of the algorithms varies, where the simplest is the one using the fewer number of calculations, based on the function

$$H(\mathbf{y}) = \mathbf{y}^3(t)\mathbf{y}(t)^T \tag{45}$$

and the learning rate

$$\mu(t) = \begin{cases} \mu_0 & 0 \le t \le t_0 \\ \mu_0 e^{-t/\tau} & t_0 < t < t_1 \\ 0 & t_1 \le t \end{cases} \tag{56}$$

The most complex uses the function

$$H(\mathbf{y}) = \mathbf{y}(t)\mathbf{y}(t)^T - I + \mathbf{y}^3(t)\mathbf{y}(t)^T - \mathbf{y}(t)\mathbf{y}^3(t)^T \tag{47}$$

and the system of differential equations for the learning rate.

Two sources are used for test signals. One is a sine wave of frequency 10kHz and the second is a triangular wave of frequency 3kHz. The low frequency for the latter is due to the high frequency components, and given the restrictions of the input frequency, higher frequencies would be lost. In any case, the bandwidth covered is sufficient for the considered applications, pertaining speech separation.

The signal sources are mixed by an analog circuit, where the matrix coefficients are easily changed to test different conditions. Each of the coefficients can take any value between 0 and 1, and by using an inverting buffer, a negative coefficient can be obtained hence covering a long range of mixing matrices.

Two different tests were executed for each of the algorithms. The first consisted on applying the signals without mixing them, that is, the mixing matrix is the

identity. The expected weight matrix is a permutation matrix, multiplied by a diagonal matrix:

$$\mathbf{W} = \mathbf{I}_p \lambda \tag{57}$$

A weight matrix close to the following is an acceptable result:

$$\mathbf{W} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1.2 & 0 \\ 0 & 1.05 \end{bmatrix} = \begin{bmatrix} 0 & 1.05 \\ 1.2 & 0 \end{bmatrix} \tag{58}$$

Note that in this case the output signals will be switched, depending on the initial configuration. The values near to zero represent the intersymbol interference, in this case being equal to zero. The further these interference "weights" are, the less acceptable is the performance of the algorithm.

The second test performed consisted of applying a random mixing matrix, and then, observing the results as the algorithm converged. In order to analyze the weight matrix, it was necessary to multiply the resulting matrix with the mixing matrix, in order to obtain the permutation matrix.

In both cases it was necessary to reorganize the results of the permutation matrix, so that the weights obtained would be in the same position for all the

samples. The reorganization was done so that the matrix would resemble the identity matrix.

To check the performance of the algorithms, each algorithm was executed 50 times and the results were averaged to obtain a matrix to be used to compare the intersymbol interference.

An example of the obtained values for the identity mixing matrix are depicted in table 1.

Table 1. Average of weight matrices for identity mixing channel.

H(y) / Learning rate	Exponential	Adaptive (Amari)
$H(\mathbf{y})=\mathbf{y}^3(t)\mathbf{y}(t)^T$	$\begin{bmatrix}1.2470 & 0.1652\\ 0.1270 & 1.1124\end{bmatrix}$	$\begin{bmatrix}1.2509 & 0.0700\\ 0.0355 & 1.1038\end{bmatrix}$
$H(\mathbf{y})=\mathbf{y}^5(t)\mathbf{y}(t)^T$	$\begin{bmatrix}1.2447 & 0.1459\\ 0.1221 & 1.1170\end{bmatrix}$	$\begin{bmatrix}1.1568 & 00562\\ 0.0253 & 1.0360\end{bmatrix}$
$H(\mathbf{y})=\mathbf{y}(t)\mathbf{y}(t)^T-I+\mathbf{y}^3(t)\mathbf{y}(t)^T-\mathbf{y}(t)\mathbf{y}^3(t)^T$	$\begin{bmatrix}1.3562 & 0.1944\\ 0.1914 & 1.2969\end{bmatrix}$	$\begin{bmatrix}1.2325 & 0.0049\\ 0.0051 & 1.4294\end{bmatrix}$

By normalizing the terms corresponding to the estimated channel, a measure of the intersymbol interference will be obtained to evaluate the performance. After the normalization, the maximum interference ratio is selected as the performance value for the specific algorithm. This can be expressed by the formula

$$p = \max\left(\frac{W_{12}}{W_{11}}, \frac{W_{21}}{W_{22}}\right) \qquad (58)$$

where p is the performance measure of the weight matrix. Table 2 shows the performance for each case averaged.

Table 2. Algorithm performance for identity mixing matrix.

H(y) / Learning rate	Exponential	Adaptive (Amari)
$H(\mathbf{y}) = \mathbf{y}^3(t)\mathbf{y}(t)^T$	0.1325	0.0560
$H(\mathbf{y}) = \mathbf{y}^5(t)\mathbf{y}(t)^T$	0.1172	0.0486
$H(\mathbf{y}) = \mathbf{y}(t)\mathbf{y}(t)^T - I + \mathbf{y}^3(t)\mathbf{y}(t)^T - \mathbf{y}(t)\mathbf{y}^3(t)$	0.1476	0.0040

Proceeding in the same way when the mixing matrix A has arbitrarily selected values the process was executed. Note that in all executions the algorithm converged. The analogous results are summarized in tables 3 & 4

$$A = \begin{bmatrix} 0.44975 & .40647 \\ 0.66568 & .80043 \end{bmatrix}$$

Table 3. Average of weight matrices for an arbitrary mixing channel.

H(y) / Learning rate	Exponential	Adaptive (Amari)
$H(\mathbf{y})=\mathbf{y}^3(t)\mathbf{y}(t)^T$	$\begin{bmatrix}0.9720 & 0.2321\\ 0.2166 & 0.9842\end{bmatrix}$	$\begin{bmatrix}0.9635 & 0.2252\\ 0.2218 & 0.9437\end{bmatrix}$
$H(\mathbf{y})=\mathbf{y}^5(t)\mathbf{y}(t)^T$	$\begin{bmatrix}0.9857 & 0.2441\\ 0.2259 & 0.9078\end{bmatrix}$	$\begin{bmatrix}0.8655 & 0.2004\\ 0.2042 & 0.8968\end{bmatrix}$
$H(\mathbf{y})=\mathbf{y}(t)\mathbf{y}(t)^T-I+\mathbf{y}^3(t)\mathbf{y}(t)^T-\mathbf{y}(t)\mathbf{y}^3(t)^T$	$\begin{bmatrix}1.1310 & 0.2840\\ 0.2545 & 1.0558\end{bmatrix}$	$\begin{bmatrix}1.1405 & 0.2687\\ 0.2651 & 1.0586\end{bmatrix}$

Using the same measure of performance:

Table 4. Algorithm performance for mixing matrix A.

H(y) / Learning rate	Exponential	Adaptive (Amari)
$H(\mathbf{y})=\mathbf{y}^3(t)\mathbf{y}(t)^T$	0.2388	0.2337
$H(\mathbf{y})=\mathbf{y}^5(t)\mathbf{y}(t)^T$	0.2315	0.2488
$H(\mathbf{y})=\mathbf{y}(t)\mathbf{y}(t)^T-I+\mathbf{y}^3(t)\mathbf{y}(t)^T-\mathbf{y}(t)\mathbf{y}^3(t)$	0.2419	0.2511

As it can be seen in the case of he identity mixing matrix, the best results were obtained from the algorithm implemented with an adaptive learning rate and with the symmetric function. However, the performance is almost the same for the other experiments. Because of this, it is recommended to use the mentioned

algorithm with the best performance in the first test. Note that this algorithm is more complex, and therefore takes the longest time to execute.

Following are some printouts of the input (Figure 10), mixed (Figure 11) and estimated signals (Figure 12).

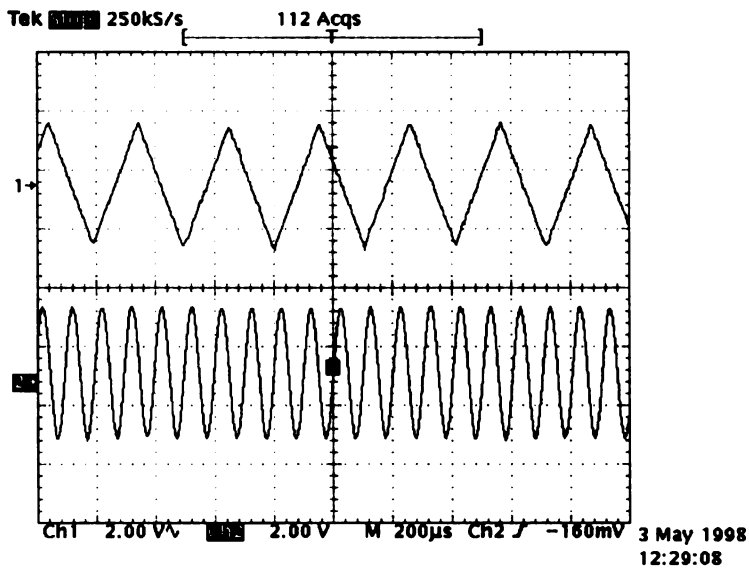


Figure 10. Original signals

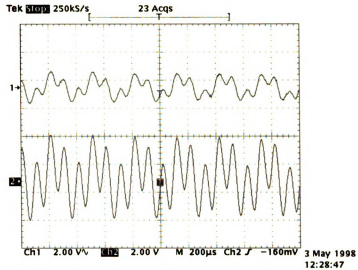


Figure 11. Mixed signals

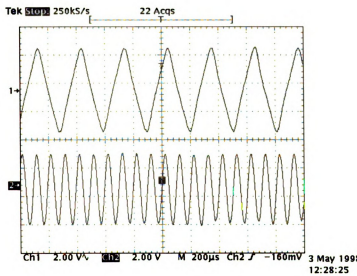


Figure 12. Estimated signals

4 Conclusions and Recommendations

The on-line implementation of the algorithms shows that the application to real-time systems is potentially feasible, taking into account the restrictions imposed by the length of input signals, which will affect the frequency ranges and the computational time. The performance might be improved by rewriting the code completely in assembly language, but this improvement might not be considerable, given that the optimizer provided for the C compiler does a good job. This optimization may not be necessary if the program is to be used in an off-line environment, and the only frequency limit would be imposed by the sampling rate, since no immediate response is necessary.

A problem that can be seen in the implementation of the algorithms, either on-line or off-line, is channel switching. This happens when the source signal in channel 1 appears as the output in channel 2 and *vice versa*. Given that it is not possible to know the structure of the mixing matrix, no change within the algorithm has been found yet. To solve this permutation it is necessary to implement a post-processing stage so the algorithm can be used in real on-line applications. Since the off-line implementation allows the user to choose the

desired output, it may not be an issue in this case, but the importance of this separation in on-line algorithms is crucial. To design this stage, it may be necessary to use *a priori* information of the input signals, depending on the application. In speech, for example, some features can be used to select the desired separated output from the others, based on the energy spectrum, waveform characteristics, etc. Another *a priori* information that may also be used is the probability distribution of the signals, since this can state the nature of the desired function H(y). The presence of some special noise can be filtered out with this knowledge.

Another problem to address is the use of a dynamic mixing matrix. Assuming that the mixing medium varies slowly in time (slow enough for the algorithm to converge) the initial solution found may not be valid some time after the update process has finished. To solve this problem, the learning rate must be taken into account. If it has a constant value, the update rule will adapt to a changing environment , but it might not produce an accurate result, since it will either make coarse steps in the search space when large values are used, or may take a long time to converge when the search is fine, with the possibility that this time is going to be longer than the rate at which the mixing matrix changes.

This can be somehow overcome by the use of an exponential instead of a constant. A decaying exponential, as the one implemented in the algorithms, will pose problems after the separation matrix has converged, if the system changes. This is because the final value of the learning rate is 0, and therefore no additional change in the weights will be produced. By selecting some measure of the change in the variation of the weights, compared to some established value, assuming a constant learning rate (that is, before multiplying by $\mu(t)$), the exponential can be applied repeatedly over and over, initiating the process every time the condition is met, hence allowing the system to change.

Finally, this problem is addressed by Amari by using the system of two differential equations to describe an adaptive learning rate, where it depends both on its previous values and the current value of the weight matrix. The problem lies in that the selection of the parameters is difficult and have to be fine tuned for each application. An improvement may be obtained by using a previous stage where these parameters can be set or "learned" based on the information of the signals to be treated. For example, the decaying rate of the exponential would depend on the frequency of the signals and the "change sensitivity" would be done by knowing on how fast the medium may change.

The final problem to be considered, and maybe the most important one, is the non-invertibility of the mixing matrix. If the system is not invertible or has a very small determinant, the algorithm will not converge. New alternatives must be looked for, maybe by applying some kind of pre-processing on the signals, using some knowledge about their nature and modifying one of the inputs in such a way that one of the mixed signals will be affected.

The next step to be done in a future work is to apply actual speech signals and process them in real time, to verify the performance of the implementation. Since the algorithm works well with a controlled mixing medium (it is simulated by adding the signals), the mixing medium to be used next should be the environment, and the input signals should come from microphones. Since in this case the mixing matrix is unknown, the evaluation of performance can only be done by listening to the actual output. A precise comparison of the waveforms can be done, but the delays produced between the input and the output, along with the acquisition of the original signals may generate some problems for adequate test environment. Enough time and patience may produce good results, but the evaluation obtained by listening to the estimated signals should be satisfactory in some cases.

Also, as it was said before, the switching problem should be addressed in order to think of an actual application in real devices, along with the problem of invertibility. A proposed solution can be thought of by changing the mixing medium parameters. If a non-invertible medium is detected (the algorithm does not converge in a reasonable time) somehow it might be possible to alter the position of an input microphone. By providing another input located in a different position and switching to it, the characteristics of the environment would be different, and the signals may be separated. This solution should be considered for real applications, until a new training algorithm is developed to solve the problem in an easier way.

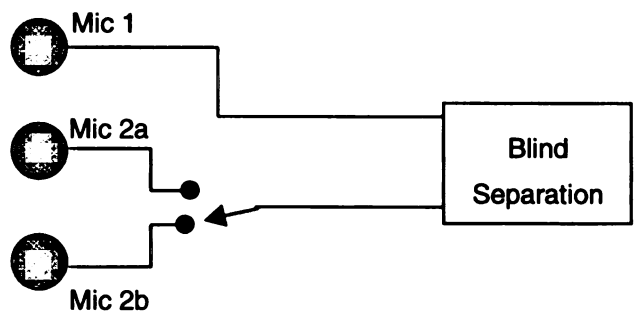


Figure 13. Additional input to solve the invertibility problem

APPENDIX

APPENDIX A

PC Program

```
#define True  1
#define False 0
#define DSP_SEM_LOCATION      0xc00040
#define DSP_BUFFER_START      0xc00004
#define DSP_BUFFER_END        0xc001fd
#define DSP_BUFFER_OUT_START  0xc00200
#define DSP_BUFFER_OUT_END    0xc003fd

#include "stdlib.h"
#include "stdio.h"
#include "time.h"
#include "d:\user\zuluaga\tic32nt.h"

#pragma hdrstop

void main()
{
  char      * filename;
  int       c,i;
  HANDLE    hProcessor_Data ;
  ULONG     PC,port,port_out;
  float     dataout2,t2,t1,alpha,mu,w;
  char      deviceName[100];
  FILE      *FID, *FID2;


  ((PLSI_WIN95_PCC32)deviceName) -> baseAddress  = 0x290;
  ((PLSI_WIN95_PCC32)deviceName) -> DPRAMAddress = 0xd0000;
  c=Open_System();
  hProcessor_Data = Open_Processor_ID(deviceName);
  c=Global_Reset(hProcessor_Data);
  filename="d:/user/zuluaga/amari.out";
  c=Load_Object_File(filename, hProcessor_Data);
  Set_Processor_Data_Type_Size_32(hProcessor_Data);
  PC=Get_Entry_PC(hProcessor_Data);
  Run_From(PC,hProcessor_Data);

       // Initialize parameters.

  Request_Semaphore(hProcessor_Data);
  while (Read_Semaphore(hProcessor_Data))
  {
      Request_Semaphore(hProcessor_Data);
  }

  port=DSP_SEM_LOCATION;
  c=Put_DPRAM_Float_32(port, 1, hProcessor_Data);
  srand( (unsigned)time( NULL ) );
  t1=10000; /*1000,1000,.01  */
```

```
  t2=10000;
  alpha=0.1;
  mu=.001;

  port=DSP_BUFFER_START;
  port_out=DSP_BUFFER_OUT_START;

  //t2, t1, alpha, mu
  c=Put_DPRAM_Float_32(port++, t2, hProcessor_Data);
  c=Put_DPRAM_Float_32(port++, t1, hProcessor_Data);
  c=Put_DPRAM_Float_32(port++, alpha, hProcessor_Data);
  c=Put_DPRAM_Float_32(port++, mu, hProcessor_Data);

  //W
  w=2*((float)rand()/RAND_MAX)-1;
  c=Put_DPRAM_Float_32(port++, w, hProcessor_Data);
  w=2*((float)rand()/RAND_MAX)-1;
  c=Put_DPRAM_Float_32(port++, w, hProcessor_Data);
  w=2*((float)rand()/RAND_MAX)-1;
  c=Put_DPRAM_Float_32(port++, w, hProcessor_Data);
  w=2*((float)rand()/RAND_MAX)-1;
  c=Put_DPRAM_Float_32(port++, w, hProcessor_Data);
        // DSP reads data
   Release_Semaphore(hProcessor_Data);
  // Stop process
   scanf(": \n");
  Request_Semaphore(hProcessor_Data);
  while (Read_Semaphore(hProcessor_Data))
  {
  Request_Semaphore(hProcessor_Data);
  }
 port=DSP_SEM_LOCATION;
  c=Put_DPRAM_Float_32(port, 0, hProcessor_Data);
  Release_Semaphore(hProcessor_Data);
  FID=fopen("w.txt","a");
  port_out=DSP_BUFFER_OUT_START+2;
  c=Get_DPRAM_Float_32(port_out++, &dataout2, hProcessor_Data);
  printf("%f \t",dataout2);
  fprintf(FID,"%f \t",dataout2);
  c=Get_DPRAM_Float_32(port_out++, &dataout2, hProcessor_Data);
  printf("%f \n",dataout2);
  fprintf(FID,"%f \t",dataout2);
  c=Get_DPRAM_Float_32(port_out++, &dataout2, hProcessor_Data);
  printf("%f \t",dataout2);
  fprintf(FID,"%f \t",dataout2);
  c=Get_DPRAM_Float_32(port_out++, &dataout2, hProcessor_Data);
  printf("%f \n",dataout2);
  fprintf(FID,"%f \n",dataout2);
  Close_Processor_ID(deviceName);
  Close_System();
  fclose(FID);
}
```

DSP Programs

a) Symmetric function, adaptive learning rate.

```
# include "D:/tool_dir/math.h"

#define DSP_SEM_LOCATION    0xc00040
#define DSP_BUFFER_START    0xc00004
#define DSP_BUFFER_END      0xc001fd
#define DSP_BUFFER_OUT_START    0xc00200
#define DSP_BUFFER_OUT_END      0xc003fd
#define SEMAPHORE             0x820000

#define TIMER1          0x81A005
#define UCR             0x81A008
#define ACR             0x81A00A
#define CONFIG          0x81A00F
#define IMR             0x81A00B
#define DCR             0x81A00C
#define CH0             0x81A002
#define CH1             0x81A006

#define IMR_DEF         0x000010000
#define DCR_DEF         0x0001e0000
#define TIMER1_DEF      0x0Fecb0000
#define UCR_DEF         0x0A4000000
#define ACR_DEF         0x000F20000
#define CONFIG_DEF      0x08dff0000

#define CLEARINT        0x0FFFFFFFE
#define ITTP            0x06000000

typedef unsigned long UINT32;

void main(void)
{
const int scale =(-0x4fFFFFFF);

float             * dport,* dport_out,a,b;
float             mu_i, t1, t2, alpha;
float             t1_temp,t2_temp,alpha_temp;
volatile unsigned int   * sem;
float             * end_data;
UINT32            * confdm;
register float    y[2],g,g2,y3[2],ytemp[2],ytemp2[2];
register float    W[2][2],mu[2][2],v[2][2];
register short    * dataDM0,* dataDM1;
signed int        k,j,i;

confdm = (UINT32 *)DCR;
* confdm = DCR_DEF;
confdm = (UINT32 *)UCR;
* confdm = UCR_DEF;
confdm = (UINT32 *)ACR;
* confdm = ACR_DEF;
confdm = (UINT32 *)TIMER1;
```

```
* confdm = TIMER1_DEF;
confdm = (UINT32 *)CONFIG;
* confdm = CONFIG_DEF;
confdm = (UINT32 *)IMR;
* confdm = IMR_DEF;

sem=(unsigned int *)SEMAPHORE;
* sem = 1;
      dataDM0 = (short *)CH0;
      dataDM1 = (short *)CH1;

// Wait for parameters.
      dport = (float *)DSP_BUFFER_START;
      dport_out = (float *)DSP_BUFFER_OUT_START;
// Initialize parameters.
      * sem=0;
      while (* sem)
      {
            * sem=0;
      }
      t2= * dport++;
      t1= * dport++;
      alpha= * dport++;
      mu_i=* dport++;
      W[0][0]=* dport++;
      W[0][1]=* dport++;
      W[1][0]=* dport++;
      W[1][1]=* dport++;
      t1_temp=(t1-1)/(t1);
      t2_temp=(t2-1)/(t2);
      alpha_temp=alpha/(t2);
      * sem = 1;
      i=0;
      v[0][0]=0;
      v[0][1]=0;
      v[1][1]=0;
      v[1][0]=0;
      mu[0][0]=mu_i;
      mu[0][1]=mu_i;
      mu[1][0]=mu_i;
      mu[1][1]=mu_i;
//Receive Data
      end_data=(float *)DSP_SEM_LOCATION;
      while  (* end_data)
      {
      while (!((* confdm) & (0x00010000)))
      {}
      a=(float)(* dataDM0)/scale;
      b=(float)(* dataDM1)/scale;
      y[0]=(W[0][0]*a+W[0][1]*b);
      y[1]=(W[1][0]*a+W[1][1]*b);


      * dataDM0=scale*y[0];
      * dataDM1=scale*y[1];
      ytemp[0]=y[0]*W[0][0]+y[1]*W[1][0];
```

```
        ytemp[1]=y[0]*W[0][1]+y[1]*W[1][1];
        y3[0]=y[0]*y[0]*y[0];
        y3[1]=y[1]*y[1]*y[1];
        ytemp2[0]=y3[0]*W[0][0]+y3[1]*W[1][0];
        ytemp2[1]=y3[0]*W[0][1]+y3[1]*W[1][1];

        for (k=0;k<2;k++)
        {
              for (j=0;j<2;j++)
              {
              g=W[k][j]-(y3[k]*ytemp[j]-y[k]*ytemp2[j]+y[k]*ytemp[j]);
              W[k][j]=W[k][j]+mu[k][j]*g;
              v[k][j]=v[k][j]*(t1_temp)+g/(t1);
              mu[k][j]=(mu[k][j]*(t2_temp))+(alpha_temp*v[k][j]*v[k][j]);
              }
        }
        }

        * sem=0;
        while (* sem)
        {
              * sem=0;
        }

        dport_out = (float *)(DSP_BUFFER_OUT_START+2);
        * dport_out++=W[0][0];
        * dport_out++=W[0][1];
        * dport_out++=W[1][0];
        * dport_out++=W[1][1];
        * sem = 1;
}
```

b) Non-symmetric function, exponential learning rate.

```
# include "D:/tool_dir/math.h"
#define DSP_SEM_LOCATION     0xc00040
#define DSP_BUFFER_START     0xc00004
#define DSP_BUFFER_END       0xc001fd
#define DSP_BUFFER_OUT_START     0xc00200
#define DSP_BUFFER_OUT_END       0xc003fd
#define SEMAPHORE              0x820000

#define TIMER1          0x81A005
#define UCR             0x81A008
#define ACR             0x81A00A
#define CONFIG          0x81A00F
#define IMR             0x81A00B
#define DCR             0x81A00C
#define CH0             0x81A002
#define CH1             0x81A006

#define IMR_DEF         0x000010000
#define DCR_DEF         0x0001e0000
#define TIMER1_DEF      0x0Ff100000
#define UCR_DEF         0x0A4000000
#define ACR_DEF         0x000F20000
#define CONFIG_DEF      0x08dff0000

#define CLEARINT        0x0FFFFFFFE
#define ITTP            0x06000000

typedef unsigned long UINT32;

void main(void)
{
const int scale =(-0x4fFFFFFF);

float             * dport,* dport_out,a,b;
float             mu,mu_i, t1, t2, alpha;
volatile unsigned int   * sem;
float             * end_data;
UINT32            * confdm;
register float    y[2],g,g2,y3[2],ytemp[2],W[2][2];
register short    * dataDM0,* dataDM1;
signed int  k,j,i;

confdm = (UINT32 *)DCR;
* confdm = DCR_DEF;
confdm = (UINT32 *)UCR;
* confdm = UCR_DEF;
confdm = (UINT32 *)ACR;
* confdm = ACR_DEF;
confdm = (UINT32 *)TIMER1;
* confdm = TIMER1_DEF;
confdm = (UINT32 *)CONFIG;
* confdm = CONFIG_DEF;
```

```
confdm = (UINT32 *)IMR;
* confdm = IMR_DEF;

sem=(unsigned int *)SEMAPHORE;
dataDM0 = (short *)CH0;
dataDM1 = (short *)CH1;

// Wait for parameters.
      dport = (float *)DSP_BUFFER_START;
      dport_out = (float *)DSP_BUFFER_OUT_START;

// Initialize parameters.
      * sem=0;
      while (* sem)
      {
      * sem=0;
      }
      t2= * dport++;
      t1= * dport++;
      alpha= * dport++;
      mu_i=* dport++;
      W[0][0]=* dport++;
      W[0][1]=* dport++;
      W[1][0]=* dport++;
      W[1][1]=* dport++;
      t1_temp=(t1-1)/(t1);
      t2_temp=(t2-1)/(t2);
      alpha_temp=alpha/(t2);
      * sem = 1;
      i=0;
      mu=mu_i;
//Receive Data
      end_data=(float *)DSP_SEM_LOCATION;

      while  (* end_data)
      {
            while (!((* confdm) & (0x00010000)))
            {}
            a=(float)(* dataDM0)/scale;
            b=(float)(* dataDM1)/scale;
            y[0]=(W[0][0]*a+W[0][1]*b);
            y[1]=(W[1][0]*a+W[1][1]*b);

            * dataDM0=scale*y[0];
            * dataDM1=scale*y[1];
            ytemp[0]=y[0]*W[0][0]+y[1]*W[1][0];
            ytemp[1]=y[0]*W[0][1]+y[1]*W[1][1];
            y3[0]=y[0]*y[0]*y[0];
            y3[1]=y[1]*y[1]*y[1];

            if (i<30000)
            {
                  for (k=0;k<2;k++)
                  {
                  for (j=0;j<2;j++)
                   {
```

```
                g=W[k][j]-(y3[k]*ytemp[j]);
                W[k][j]=W[k][j]+mu_i*g;
                }
               }
               i++;
        }
        else
        {
        if (i<40000)
        {
               for (k=0;k<2;k++)
               {
               for (j=0;j<2;j++)
                {
                g=W[k][j]-(y3[k]*ytemp[j]);
                W[k][j]=W[k][j]+mu*g;
                }
               }
               mu=mu_i*exp(-.0005*(i-30000));
               i++;
        }
        else
        {
        g=W[0][0]-(y3[0]*ytemp[0]);
        g2=g*g;
        i=0;
        mu=mu_i;
        if (g2>100)
        {
        dport=(float *)DSP_BUFFER_START+4;
        W[0][0]=* (dport++);
        W[0][1]=* (dport++);
        W[1][0]=* (dport++);
        W[1][1]=* (dport);
        }
        }
        }
    }
        * sem=0;
        while (* sem)
        {
        * sem=0;
        }
    dport_out = (float *)(DSP_BUFFER_OUT_START+2);
    * dport_out++=W[0][0];
    * dport_out++=W[0][1];
    * dport_out++=W[1][0];
    * dport_out++=W[1][1];
    * sem = 1;
}
```

REFERENCES

REFERENCES

[1] Bell A.J. & Sejnowski T.J. 1996. "Learning the higher-order structure of a natural sound". [Online] Available ftp://ftp.cnl.salk.edu/pub/tony/stir4.ps, April 29, 1998

[2] Cardoso, J.. "Performance and implementation of invariant source separation algorithms". Proceedings ISCAS'96, 1996. [Online] Available ftp://sig.enst.fr/pub/jfc/Papers/iscas96_invar.ps.gz, April 29, 1998.

[3] Cardoso, J. and Comon, P.. "Independent component analysis, a survey of some algebraic methods". Proceedings ISCAS'96, vol.2, pp. 93-96, 1996. [Online] Available ftp://sig.enst.fr/pub/jfc/Papers/iscas96_algebra.ps.gz, April 29, 1998.

[4] Cardoso, J. and Souloumiac, A. "An efficient technique for blind separation of complex sources". Proceedings IEEE SP Workshop on Higher-Order Stat., Lake Tahoe, USA, pg 275-279, 1993. [Online] Available ftp://sig.enst.fr/pub/jfc/Papers/hos93.ps.gz, April 29, 1998.

[5] Cichocki A., Amari S., Adachi M. and Kasprzak W. "Self-Adaptive Neural Networks for Blind Separation of Sources". 1996 IEEE International Symposium on Circuits and Systems, ISCAS'96, Vol. 2, IEEE, Piscataway, NJ, 1996, 157-160. [Online] Available http://www.bip.riken.go.jp/absl/kas/PSPAP/iscas96.ps.gz, April 29, 1998.

[6] Comon, P. "Independent Component Analysis, A New Concept?". Signal Processing, vol. 36, pp 287-314, 1994. Elsevier.

[7] Davis, Philip and Rabinowitz, Philip. Methods of Numerical Integration. Academic Press, Inc. 1984.

[8] De Lathauwer, L., Comon, P. , De Moor, B. and Vandewalle, J. "Higher-Order Power Method - Application in Independent Component Analysis". [Online] Available http://www.bip.riken.go.jp/absl/nolta/delathauwer.ps.gz.

[9] Haykin, S. Neural Networks: A comprehensive foundation. Macmillan Publishing Company, 1994.

[10] Karhunen, J., Hyvärinen, Aapo, Vigário, R., Hurri, J. and Oja, E. "Applications of Neural Blind Separation to Signal and Image Processing". Proc. ICASSP 1997.

[11] Karhunen, J., Hyvärinen, Aapo, Vigário, R., Hurri, J. and Oja, E. "A Class of Neural Networks for Independent Component Analysis". IEEE Transactions on Neural Networks. May 1997.

[12] Laheld B. and Cardoso, J.. "Adaptive source separation with uniform performance". Proceedings EUSIPCO, pages 183-186, Edinburgh, September 1994. [Online] Available ftp://sig.enst.fr/pub/jfc/Papers/eusipco94_PFS.ps.gz, April 29, 1998.

[13] TMS320C3x, User's Guide. Texas Instruments. 1997.

[14] "Generating Efficient Code with TMS320 DSPs: Style Guidelines." Application Report SPRA366. Texas Instruments. July 25, 1997. [Online] Available http://www-s.ti.com/sc/psheets/spra366/spra366.pdf, April 29, 1998.

[15] PC/C32 Board, Technical Reference Manual. Loughborough Sound Images. August 1996.

[16] AM/D16SA Burr-Brown ADC/DAC Daughter Module, User Manual. Loughborough Sound Images. March 1996.

[17] PC/C32 Win32 Support Package, User Guide. Loughborough Sound Images. June 1997.

General References:

Smaragdis, P. "Paris' Independent Component Analysis & Blind Source Separation page". [Online] Available http://sound.media.mit.edu/~paris/ica.html, April 29, 1998.

Lee, T. WWW Page for Blind Source Separation. [Online] Available http://www.cnl.salk.edu/~tewon/Blind/blind.html, April 29, 1998.

Kasprzak, W. Basic Blind Source Separation. [Online] Available http://www.bip.riken.go.jp/absl/kas/researchBS.html, April 29, 1998.