University of Massachusetts Dartmouth

Department of Electrical and Computer Engineering

A FRAMEWORK
FOR
MULTIPLE ALGORITHM SOURCE SEPARATION

A Dissertation in

Electrical Engineering

by

Keith D. Gilbert

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

January 2019

We approve the dissertation of Keith D. Gilbert

Date of Signature

Karen L. Payton Professor, Department of Electrical and Computer Engineering Dissertation Advisor	
John R. Buck Professor, Department of Electrical and Computer Engineering Dissertation Committee	
Antonio H. Costa Chairperson and Professor, Department of Electrical and Computer Engineering Dissertation Committee	
Richard S. Goldhor CEO, Grapevine Software Dissertation Committee	
Joel M. MacAuslan CEO, Speech Technology and Applied Research Corporation Dissertation Committee	
Liudong Xing Graduate Program Director, Department of Electrical and Computer Engineering	
Jean VanderGheynst Dean, College of Engineering	
Tesfay Meressi Associate Provost for Graduate Studies	

ABSTRACT

A Framework for Multiple Algorithm Source Separation

by Keith D. Gilbert

The blind source separation (BSS) problem begins when multiple source signals are mixed together and then observed. The fundamental goal of BSS is to de-mix and estimate those signals using only the observations, without any knowledge of how the signals were combined and without any knowledge of the underlying signals. The field of BSS is over thirty years old and, although there exist a multitude of BSS methods that aim to solve the problem, each method is limited by the assumptions made about how the signals were combined or about the signals themselves. The current work proposes to use multiple BSS algorithms running in parallel to enhance the BSS solution and make it more robust without requiring knowledge of the signals or the mixing system.

In this work, the proposed multiple algorithm source separation (MASS) approach is outlined and is implemented in a software framework. Results demonstrate that MASS provides as good or better source separation than any of the single BSS methods considered under a variety of conditions. Although BSS is the goal, the more general (not just blind) source separation problem is the basis for the framework. This allows the study/use of both blind and supervised methods, as well as source extraction methods (methods that isolate a single signal) and allows them to interact.

The MASS framework presented here also offers capabilities to simplify future BSS research, allowing researchers to compare, communicate and reproduce results. These features also are generalizable to encompass other, non-BSS, multiple algorithm signal processing domains.

ACKNOWLEDGMENTS

I would like to thank the National Institutes of Health. The work presented here was partially supported by grant R44-DC011668.

I would also like to thank my committee for the various and enlightening conversations and advice they have provided to me over the years which have allowed me to complete this work.

I would like to thank all of my family, friends, and colleagues who have provided various types of support during this work.

I especially would like to thank my wife, Jenny, who has not only weathered the doctoral process alongside me, but has been both a sounding board and an invaluable source of advice during this research process.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ADS	Algorithm Demixing Solution
BSC	Blind Self-Competition
BSE	Blind Source Extraction
BSS	Blind Source Separation
CDS	Common, or Composite, Demixing Solution
CDSE	Common Demixing Solution Estimator
CDO	Common Demixing Operation, or Operator
CSE	Composite Source Estimate
DA	Data Acquisition
DO	Data Output
DS	Demixing Solution
FAS	Filtering Ambiguity Solution
MASS	Multiple Algorithm Source Separation
PAS	Permutation Ambiguity Solution
SASS	Single Algorithm Source Separation
SE	Source Extraction
SEA	Source Estimate Analysis
SEE	Source Estimate Evaluation
SEG	Source Estimate Grouping
SEM	Source Estimate Management
SEP	Source Estimate Production
SETA	Source Estimate Time-Alignment
SIR	Source to Interference Ratio
SS	Source Separation
SSC	Supervised Self-Competition
SSS	Supervised Source Separation

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

The blind source separation (BSS) problem has been studied for over three decades, and the ongoing research continues to yield promising results. That said, many areas of BSS research lack satisfactory solutions, with adaptive methods applied to time-varying convolutive scenes presenting the most significant challenges. The nature and uncertainties of the problem lead to a multitude of solution approaches, yet as of this paper, there is no "one size fits all" method of performing source separation (SS). The work presented here does not propose to solve any particular SS challenge, and instead, we propose to use multiple separation methods running in parallel to achieve a common goal: to progress the field of source separation. The purpose of this work is twofold; provide SS researchers with a platform to study SS issues and give SS practitioners a versatile framework for SS application development. In this work, we study multiple source separation algorithms running in parallel and the various ways of using information gleaned from a diverse set of algorithms. As such, this research is concerned with two distinct fields; SS and collaborative learning.

Beginning a general description of the multichannel SS problem is problematic because the very nature of a "source" is hard to define without a contextual domain. One could argue that any "source" is actually a source of information, but that in turn, raises the question, "What is information?" Before we turn to an information theorist's definition of information, we will, for now, think in a domain that most humans can relate to: the air acoustic domain, i.e. sounds. Consider a sound source, i.e. a method of producing waves of low-amplitude air pressure differentials, that imparts time and frequency dependent energy into an air environment, e.g. one of the left-most graphics in Figure 1a). Now consider a sensor in that environment, e.g. a microphone, an ear, etc., that observes the fluctuations in air pressure due to the source, as well as the various waves that have traveled from the source along reflective paths that impart time delays and spectral shaping. The observation that the sensor made is a perception, or *image,* of the source from that sensor's point of view, i.e. a location in an environment [1], [2]. Now consider multiple sound sources at various locations in the environment,

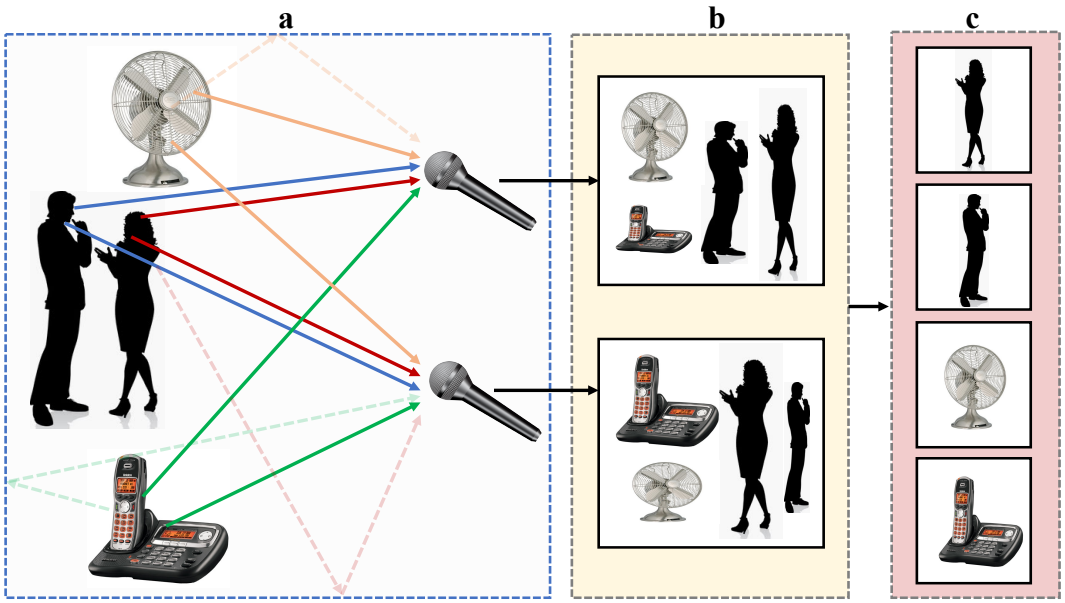

Figure 1. Source Separation Problem Overview. Multiple sources impinge upon a sensor array in a) where direct paths are given by solid lines and a few reflections are shown as dashed lines. Each sensor perceives each source within a mixture of the sources in b). Source separation aims to isolate each source from the mixtures in c).

respectively. The sensor "perceives" each source's image simultaneously and in combination, i.e. the acoustic pressure waves emitted by each source arrive at the sensor at one moment in time, *regardless of when they originated*. That is, the sources' images are mixed together, as shown in Fig. 1b). Now consider multiple sensors, each observing their respective mixtures of sources' images. As shown in Fig. 1c), the goal of SS is to use the multiple sensors' observations to estimate each of the individual sources without any contamination by the other sources.

If we define a *scene* as the multiple observations of multiple sources within an environment, then one way of classifying approaches to the source separation problem is in terms of their "blindness". When we know nothing about the scene, the act of producing renditions of the individual sources uncontaminated by any other source is known as *blind* source separation (BSS). Figure 2 shows the general signal processing model for the BSS problem where multiple unknown sources are combined together by some unknown mixing system to produce a set of observations, where each observation is a mixture of the sources. The goal of BSS is to estimate the sources, typically via an estimated demixing system which combines the mixtures to produce the source estimates.

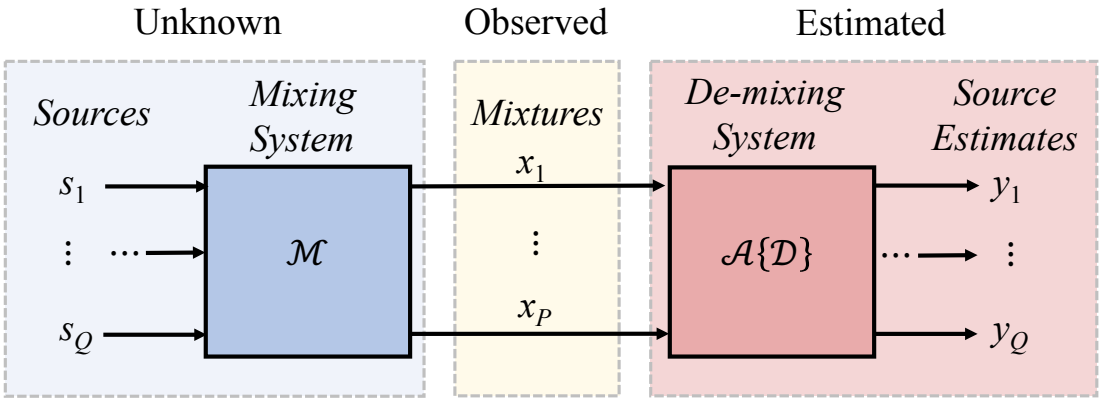

Figure 2. Blind Source Separation Model. A group of unknown sources are mixed together to produce a set of observed mixture signals. A source separation algorithm (and its associated demixing operation) estimates the sources.

This type of source separation is denoted as "blind" since the source estimation process only has access to the observations of the mixed sources and does not have any information about the sources or the mixing system used to produce the observations. In practice however, BSS methods make some assumptions about either the sources or the mixing system, or both, in order to gain traction on the problem. Furthermore, when we have partial or peripheral knowledge about a scene we perform *semi-blind* source separation (SBSS). When we know all of the sources, or all of the scene-specific impulse responses, we denote this as *supervised* source separation (SSS) and note that separating the sources is relatively trivial in this case. In both the SBSS and SSS cases, the separation problem is aided by outside knowledge, or side-information.

In this work, we investigate the use of multiple SS algorithms running in parallel, and an overview is given in Fig. 3. The observed set of mixtures is input into multiple SS algorithms, and each algorithm estimates a set of sources. A composite demixing system (CDS) combines all of the individual algorithms' source estimates to produce a set of composite source estimates (CSEs). The overview of MASS presented in Fig. 3 can be seen as a simple substitution of the observed mixtures with multiple sets of source estimates within the SS task. Although this is true conceptually, the framework we present here is much more versatile and includes this simple interpretation as an application. Furthermore, the framework we present in Ch. 3 does not even claim to

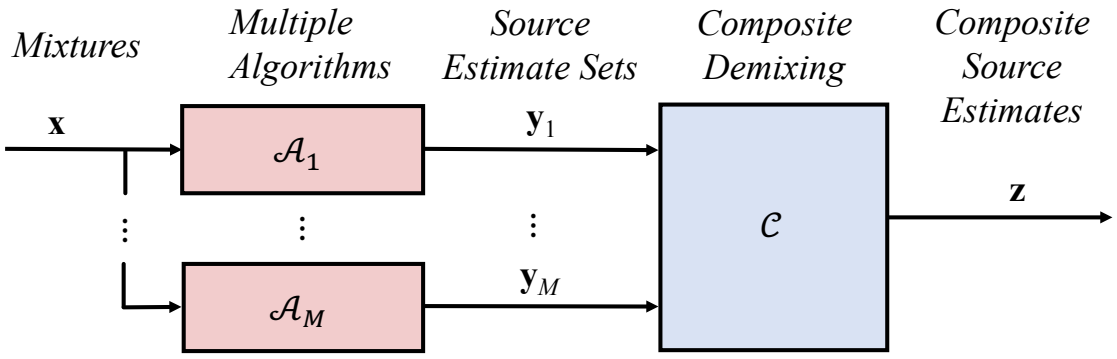

Figure 3. Conceptual Overview of Multiple Algorithm Source Separation (MASS). A set of observed mixtures is processed by multiple source separation algorithms, and each algorithm produces a set of source estimates. All source estimates produced by all algorithms are combined via a composite demixing system to produce a set of composite source estimates.

perform SS, but more fundamentally, we provide a platform for researchers to better understand the SS problem and for developers to design practical SS systems. As we shall see, the problem of combining multiple, arbitrary SS methods in a meaningful way is not a straightforward task.

In fact, the overview in Fig. 3 points out several contexts in which to discuss SS within the MASS framework. Obviously, we can talk about the SS task at the level of an *individual algorithm* which runs in parallel with other algorithms, and we can talk about SS at the *CDS* level. Each of these levels can further be discussed within the context of a general, *theoretical framework* or as a *specific realization* of a SS method or CDS. For example, when we talk about an individual source separation algorithm within the MASS framework, we are more concerned with the behavior and characteristics of source separation methods generally, since for the sake of diversity, we wish to design a framework that is compatible with as many different methods as possible. Thus, we use the term "source separation" (SS) method, so as to include BSS, SBSS and SSS methods. When we create a realization of the framework, then we need to implement particular source separation methods used in the system. We denote particular realizations of any system within the MASS framework that are provided here as a *reference component*. Although we can give an analogous example for the CDS, we note that the CDS has a more complicated interpretation (discussed further in Ch. 2), and we are sufficed to observe that the CDS can also be discussed as either a theoretical or realized SS system.

Generally speaking, we will discuss the framework in chapters 2 and 3 and the reference components in chapters 4 and 5 and Appendix A.

The impetus for the work presented here stems from two observations in the BSS landscape; all BSS methods make assumptions about the problem (sources, mixing, etc.) and adaptive BSS methods can separate out some sources quicker than other sources. With these observations in mind, we were then inspired at the conceptual level by the multiple algorithm work produced in the signal processing problems of adaptive prediction, e.g. [3]–[9], adaptive filtering, e.g. [10]–[19], and adaptive beamforming, e.g. [20]–[22], as well as the work on sequential compound decisions, e.g. [23]–[28]. The MASS problem is fundamentally different from the multiple algorithm prediction and filtering problems, and results from these two domains can rarely be directly applied to MASS, but we recognize their conceptual influence here.

That said, the notion of collaboration in the SS problem is not new. In [29], Pham showed that the combination of multiple BSS source criteria (score, gradient) functions produced an optimal solution in a maximum likelihood context under certain regularity conditions. In [30], cumulant blending was employed, and in [31], Amari used convex combinations of score-dual functions (expansion of BSS criteria functions). For this work, the most relevant example of competition in BSS was given in [32], where Comon showed that any linear combination of proper BSS cost functions results in a proper BSS cost function. All of the aforementioned approaches are limited, however, because they are simply cost function blending methods. The MASS system is much more ambitious, in that it enables individual SS methods to explicitly compete, i.e. operate independently, or explicitly cooperate, i.e. share demixing information. Moreover, the MASS system allows multiple algorithms to simultaneously and individually compete or cooperate, allowing for a greater diversity of SS solutions. The only known works on the MASS topic are given in [33], [34], and the diversity of approaches indicates the complexity of the problem. The work presented here constitutes a description of the problems involved with attempting MASS, along with a signal processing framework that deals with all known MASS problems.

In the next sections, we will quickly overview the notation used throughout this work, and then overview some background concepts used throughout this work. After that we will give a semi-technical historical perspective of the machine learning BSS

approach in the following section and end this chapter by looking at the convolutive multichannel source separation problem. Chapter 2 will outline the issues involved with approaching MASS, Chapter 3 will detail a software framework for MASS, and Chapters 4 and 5 will provide example methods and results from current MASS research.

1.1 Notation

We define the notation that will be used throughout this work in this section. At times, we will need to break with these notational conventions for presentation clarity, and we will note any deviations as they occur. Table 1 provides the general notation used in this work, and Table 2 gives a set of reserved symbols used in this work.

Table 1. General notation used throughout this work.

Example Notation	Description
$s, T, \epsilon, \mathrm{M}$	Scalar quantities will be upper or lower case, Roman or Greek.
$\boldsymbol{s}, \boldsymbol{t}, \boldsymbol{\epsilon}, \boldsymbol{\mu}$	Column vector quantities will use lower case, bold Roman or Greek.
$\boldsymbol{S}, \boldsymbol{T}, \mathbf{E}, \mathbf{M}$	Matrix quantities will use upper case, bold Roman or Greek.
$s(\cdot), t(\cdot), \boldsymbol{\epsilon}(\cdot), \mathbf{M}(\cdot)$	Functions are immediately followed by parentheses enclosing the function's arguments.
$\mathcal{S}\{\cdot\}, \mathcal{T}\{\cdot\}$	Transforms and operators will use upper case script immediately followed by curly brackets enclosing the arguments.
$\mathbb{S}, \mathbb{X}$	Sets will use upper case Double-Struck.
$s^*, \boldsymbol{t}^*, \mathbf{E}^*$	Superscript asterisk denotes complex conjugate.
$\boldsymbol{s}^{\boldsymbol{T}}, \boldsymbol{T}^{\boldsymbol{T}}$	Superscript T denotes transpose.
$\boldsymbol{s}^{\boldsymbol{H}}, \boldsymbol{T}^{\boldsymbol{H}}$	Superscript H denotes conjugate transpose (Hermite), i.e. $\boldsymbol{S}^H = (\boldsymbol{S}^*)^{\boldsymbol{T}}$.

Table 2. Summary of reserved notation used in this work.

Notation	Description
$\mathcal{E}\{\cdot\}$	Expectation operator.
$\mathcal{F}\{\cdot\}$	Fourier transform. Specifically, the DTFT, unless noted otherwise.
$\mathcal{H}\{\cdot\}$	Differential entropy, defined in section 1.2.2.
$\mathcal{J}(\cdot)$	Denotes the various cost functions used in this work.
$\mathcal{K}\{\cdot\}$	Kullback-Leibler divergence operator, given in (1) below.
$Id\{\cdot\}$	System identification operator, defined in Sect. 1.2.3.
$Im\{\cdot\}$	Signal imaging operator, defined in Sect. 1.2.3.
$Vec\{\cdot\}$	Vectorization operator, defined in (2) and (3) below.

1.1.1 Notation Exception

Random variables (RVs) and probability density functions (PDFs): In this work we will overload the notation used to denote a random variable (RV) and a realization of an RV, and we will let context disambiguate the usage. For example, if we let s be an RV with PDF f, then $s \sim f(s)$ denotes the RV s is distributed according to f. However, in the expectation of s, $\mathcal{E}\{s\} = \int sf(s)ds$, s is an RV on the left, and a realization on the right. Whenever the context is ambiguous, we will clarify the usage.

1.1.2 Kullback-Leibler Divergence

The Kullback-Leibler divergence (KLD) is an asymmetric measure of the similarity of two PDFs. Given two PDFs, g and h, with a common support space, $\mathbb{S}$, the KLD is defined as,

$$\mathcal{K}\{g(s); h(s)\} \triangleq \int_{\mathbb{S}} g(s) \log \frac{g(s)}{h(s)} ds \,. \tag{1}$$

The KLD is nonnegative, is zero when $g = h$, and in general, $\mathcal{K}\{g(s); h(s)\} \neq \mathcal{K}\{h(s); g(s)\}$.

1.1.3 Vectorization operator

If we consider the matrix $\boldsymbol{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2, \dots, \boldsymbol{a}_K]$, where $\boldsymbol{a}_k$ is the k^{th} column of $\boldsymbol{A}$ for $k = 1, \dots, K$, we define a vectorization operator as,

$$Vec\{\boldsymbol{A}\} \triangleq [\boldsymbol{a}_1^T, \boldsymbol{a}_2^T, \dots, \boldsymbol{a}_K^T]^T \tag{2}$$

More generally, if we consider the set of K column vectors $\mathbb{A} = \{\boldsymbol{a}_1, \boldsymbol{a}_2, \dots, \boldsymbol{a}_K\}$ whose lengths are not necessarily equal, the vectorization operator is defined as,

$$Vec\{\mathbb{A}\} \triangleq Vec\{\boldsymbol{a}_1, \boldsymbol{a}_2, \dots, \boldsymbol{a}_K\} \triangleq [\boldsymbol{a}_1^T, \boldsymbol{a}_2^T, \dots, \boldsymbol{a}_K^T]^T. \tag{3}$$

1.2 Background Concepts

In this section, we provide a rudimentary overview of some of the background concepts useful in understanding the source separation problem and the MASS framework.

1.2.1 Correlation and Coherence

The *auto-correlation* of the discrete-time stochastic process, $u(n)$, is defined as,

$$r_u(n, \kappa) = \mathcal{E}\{u(n)u^*(n + \kappa)\}. \tag{4}$$

where $\mathcal{E}$ is the expectation operator, * denotes complex conjugation, and n is a discrete time index. When the process $u(n)$ has a constant mean and its auto-correlation is independent of the time-index, n, then the auto-correlation is only dependent on the lag, κ, i.e.,

$$r_u(\kappa) = \mathcal{E}\{u(n)u^*(n + \kappa)\}, \ \ \forall n, \tag{5}$$

and the process is termed *wide-sense stationary* (WSS) [35, p. 376,388]. The *auto-power spectrum* (PS) of this discrete-time WSS process is defined as the discrete-time Fourier transform of the process's auto-correlation function, given by [35, p. 408],

$$R_u(\omega) = \mathcal{F}\{r_u(\kappa)\}, \tag{6}$$

where $\mathcal{F}\{\cdot\}$ is the discrete-time Fourier transform. In many interesting source separation problems, the assumption of source stationarity is violated, e.g. speech, heartbeats, neuro-transmissions, etc., but we can make adjustments in the signal processing, e.g. analysis window lengths, so that any data being analyzed can be roughly treated as WSS. As we shall see, whether this assumption makes sense or not is entirely up to the MASS user.

Considering two stochastic processes, $u(n)$ and $v(n)$, the *cross-correlation* is defined as,

$$r_{uv}(n,\kappa) = \mathcal{E}\{u(n)v^*(n+\kappa)\}. \tag{7}$$

If both $u(n)$ and $v(n)$ are WSS processes, then the cross-correlation is only dependent on the lag, i.e.,

$$r_{uv}(\kappa) = \mathcal{E}\{u(n)v^*(n+\kappa)\}, \tag{8}$$

and we note that the auto-correlation is a special case of the cross-correlation for $u(n) = v(n)$. Similar to the PS above in (6), we can define the *cross-power spectrum* (XPS) as,

$$R_{uv}(\omega) = \mathcal{F}\{r_{uv}(\kappa)\}. \tag{9}$$

The *coherence spectrum*, $C_{uv}(\omega)$, between two stochastic processes, $u(n)$ and $v(n)$, is defined as,

$$C_{uv}(\omega) = \frac{|R_{uv}(\omega)|^2}{R_u(\omega)R_v(\omega)}, \tag{10}$$

and we note that $0 \leq C_{uv}(\omega) \leq 1$.

1.2.2 Information-Theoretic Concepts

Although the focus of this work involves inclusion of diverse methods of SS, the statistical notion of information is useful as both an exemplary and practical treatment of the MASS subject, so we will run through some elementary information-theoretic material in this section. The most relevant concept to this work is *entropy*, which is both a statement about the amount of information a source contains, and about the uncertainty in describing a source of information when we only have a statistical model of that source. Entropy is intertwined with most information-theoretic concepts, but here we will focus on two specific quantities; a special case of relative entropy, *mutual information*, and another type of "relative" entropy, *negentropy*. Relative entropy is the Kullback-Leibler divergence (KLD) between two probability density functions (PDFs), and mutual information (MI) is the special case of the KLD between a joint PDF and the product of its marginal PDFs. Thus, MI provides a measure of statistical independence. Negentropy is the difference between the entropy of a random variable (RV) and the entropy of a Gaussian PDF which has the same variance as the RV, and it is a measure of how Gaussian an RVis. Mutual information is widely used as a cost function in the blind

source separation (BSS) problem, and negentropy has applications in the blind source extraction (BSE) problem.

For a continuous RV, u, with a PDF, $f_u(u)$, i.e.,

$$u \sim f_u(u), \tag{11}$$

the *differential entropy* of u is defined as,

$$\mathcal{H}(u) \triangleq -\int f_u(u) \log f_u(u)\, du\,, \tag{12}$$

and we note that all integrals given in this section, including (12), come with the caveat "if it exists". The entropy is the expected value of the *information* contained in u, where information is the quantity,

$$-\log f_u(u)\,, \tag{13}$$

and entropy denotes the amount of uncertainty in describing u, statistically. For example, when the logarithm in (12) is base 2, then entropy gives the number of binary digits (bits) needed to completely convey u, e.g. in a digital communications channel, a file on a computer, etc. Now consider the *N*-variate continuous random variable, $\boldsymbol{u} = [u_1, u_2, \dots, u_N]^T$, with a joint PDF, $f_{\boldsymbol{u}}(\boldsymbol{u})$. We can then define the joint differential entropy as,

$$\mathcal{H}(\boldsymbol{u}) \triangleq -\int f_{\boldsymbol{u}}(\boldsymbol{u}) \log f_{\boldsymbol{u}}(\boldsymbol{u})\, d\boldsymbol{u}\,, \tag{14}$$

where the integration is with respect to the individual RVs in $\boldsymbol{u}$, i.e. $\int \cdot\, d\boldsymbol{u} = \int \cdots \int \cdot\, du_1 \cdots du_N$. When the components of $\boldsymbol{u}$ are mutually, statistically independent, i.e. $f_{\boldsymbol{u}}(\boldsymbol{u}) = \prod_{i=1}^{N} f_i(u_i)$, where $f_i(u_i)$ is the marginal pdf of the i^{th} RV for $i = 1, \dots, N$, then,

$$\mathcal{H}(\boldsymbol{u}) = \sum_{i=1}^{N} \mathcal{H}(u_i)\,, \tag{15}$$

so that the joint entropy is completely defined by the marginal entropies [36, pp. 243, 249, 253].

Confining our attention to the random vector $\boldsymbol{u}$, and considering two index sets $\mathbb{K}$ and $\mathbb{L}$ that are equal to or are subsets of $\{1, \dots, N\}$, respectively, we define two random vectors as $\boldsymbol{w} = Vec\{\{u_k : k \in \mathbb{K}\}\}$ and $\boldsymbol{v} = Vec\{\{u_l : l \in \mathbb{L}\}\}$, where the set ordering is

irrelevant. A distribution of the random vector $\boldsymbol{w}$ conditioned on the random vector $\boldsymbol{v}$, $f_{\boldsymbol{w}|\boldsymbol{v}}(\boldsymbol{w}|\boldsymbol{v})$, can be generally defined by,

$$f_{\boldsymbol{w}|\boldsymbol{v}}(\boldsymbol{w}|\boldsymbol{v}) = \frac{f_{\boldsymbol{w}}(\boldsymbol{w})}{f_{\boldsymbol{v}}(\boldsymbol{v})}. \tag{16}$$

Consequently, we now define the joint entropy of $\boldsymbol{w}$ conditioned on $\boldsymbol{v}$, i.e. the conditional joint entropy, as,

$$\mathcal{H}(\boldsymbol{w}|\boldsymbol{v}) \triangleq -\int f_{\boldsymbol{w}}(\boldsymbol{w}) \log f_{\boldsymbol{w}|\boldsymbol{v}}(\boldsymbol{w}|\boldsymbol{v})\, d\boldsymbol{v}\,, \tag{17}$$

where we break with notation, since in general, $\mathcal{H}(\boldsymbol{w}|\boldsymbol{v})$ is not equal to the definition of entropy in (12). Using the definitions of joint entropy (14) and conditional joint entropy (17), a chain rule appears by equating the two as,

$$\mathcal{H}(\boldsymbol{u}) = \sum_{i=1}^{N} \mathcal{H}(u_i|u_1, u_2, \dots, u_{i-1})\,, \tag{18}$$

which gives us a more general decomposition of the joint entropy [36, pp. 249, 253].

Mutual information (MI) is typically defined in the bivariate, i.e. N=2, case, but a general N-variate definition of mutual information is given by,

$$\mathcal{J}_{MI}(\boldsymbol{u}) \triangleq \int f_{\boldsymbol{u}}(\boldsymbol{u}) \log \frac{f_{\boldsymbol{u}}(\boldsymbol{u})}{\prod_{i=1}^{N} f_i(u_i)}\, d\boldsymbol{u}\,. \tag{19}$$

Recognizing (19) as the KLD between $f_{\boldsymbol{u}}(\boldsymbol{u})$ and $\prod_{i=1}^{N} f_i(u_i)$, we note that,

$$\mathcal{J}_{MI}(\boldsymbol{u}) \geq 0. \tag{20}$$

If we expand the logarithm term in (19), we can use the definitions of the marginal (12) and joint (14) entropies to give an equivalent form of mutual information as,

$$\mathcal{J}_{MI}(\boldsymbol{u}) = -\mathcal{H}(\boldsymbol{u}) + \sum_{i=1}^{N} \mathcal{H}(u_i)\,. \tag{21}$$

Given that $\mathcal{J}_{MI}$ is nonnegative, i.e. (20), we can substitute the definition of joint entropy (18) into the definition of $\mathcal{J}_{MI}$ in (21) to yield,

$$\sum_{i=1}^{N} \mathcal{H}(u_i) \geq \sum_{i=1}^{N} \mathcal{H}(u_i|u_1, u_2, \dots, u_{i-1})\,. \tag{22}$$

Recognizing the left-hand side of (22) as the joint entropy when all elements of $\boldsymbol{u}$ are statistically independent, then (22) relays a fundamental tenet of information theory; besides no effect, conditioning can only reduce entropy. Only when all elements of $\boldsymbol{u}$ are

statistically independent does (22) become a strict equality, and therefore MI is zero [36, pp. 251–253].

Negentropy is a measure of how Gaussian (or non-Gaussian) an RV is, and it is based on a well-known fact in information theory: for continuous RVs with a given variance, the Gaussian distribution has the highest differential entropy. So, if we have an RV, u, with a variance, σ_u^2, then the negentropy is defined as,

$$\mathcal{J}_{NE}(u) \triangleq \mathcal{H}_G(u) - \mathcal{H}(u), \tag{23}$$

where $\mathcal{H}_G(u)$ is the entropy of Gaussian RV with variance σ_u^2. We note that $\mathcal{J}_{NE}(u) \geq 0$ since the Gaussian distribution has the highest entropy of RVs with a variance of σ_u^2, thus negentropy is nonnegative and is zero only when u is distributed as a Gaussian [36, p. 255] [37, pp. 112-113,277] [38].

1.2.3 Linear Estimation

Consider the case when an observation, $y(n)$, consists of a WSS stochastic process, $u(n)$, run through a linear, time-invariant (LTI), L-length FIR filter, $\boldsymbol{a}$, i.e.,

$$y(n) = \boldsymbol{a}^T \boldsymbol{u}(n), \tag{24}$$

where $\boldsymbol{a}$ is a vector of FIR coefficients, given by,

$$\boldsymbol{a} = [h(0), h(1), \dots, h(L-1)]^T,$$

and $\boldsymbol{u}(n)$ is a vector of the past and current L samples of the input $u(n)$, given as,

$$\boldsymbol{u}(n) = [u(n), u(n-1), \dots, u(n-L+1)]^T, \tag{25}$$

In the case where we would like to estimate $\boldsymbol{a}$, as $\widehat{\boldsymbol{a}}$, a natural place to start is to look at the difference between what we observe, $y(n)$, and what we estimate, $\widehat{\boldsymbol{a}}^T\boldsymbol{u}(n)$, i.e.,

$$e(n) \triangleq y(n) - \widehat{\boldsymbol{a}}^T\boldsymbol{u}(n). \tag{26}$$

Since the observation consists of an LTI filtered WSS process, we expect that the observed signal is also a WSS process, and we define the mean squared error (MSE) cost function as,

$$\mathcal{J}_e(n) \triangleq \mathcal{E}\{|y(n) - \widehat{\boldsymbol{a}}^T\boldsymbol{u}(n)|^2\}. \tag{27}$$

We seek to find the filter, $\widehat{\boldsymbol{a}}$, that minimizes (27), i.e.

$$\widehat{\boldsymbol{a}} = \arg\min_{\boldsymbol{g}} \mathcal{E}\{|y(n) - \boldsymbol{g}^T\boldsymbol{u}(n)|^2\}. \tag{28}$$

To that end, we can take the gradient of (27) with respect to the filter coefficients, i.e.,

$$\begin{aligned}\nabla_{\hat{\boldsymbol{a}}}\mathcal{J}_e(n) &= -2\mathcal{E}\{\boldsymbol{u}(n)e(n)\} \\ &= -2\mathcal{E}\{\boldsymbol{u}(n)y(n)\} + 2\mathcal{E}\{\boldsymbol{u}(n)\boldsymbol{u}(n)^{\boldsymbol{T}}\}\hat{\boldsymbol{a}} \\ &= -2\boldsymbol{r}_{\boldsymbol{uy}} + 2\boldsymbol{R}_{\boldsymbol{uu}}\hat{\boldsymbol{a}} \ ,\end{aligned} \tag{29}$$

set the gradient equal to zero, and solve for $\hat{\boldsymbol{a}}$ to get the optimal solution for an L-length filter as,

$$\hat{\boldsymbol{a}}^{opt} \triangleq \boldsymbol{R}_{uu}^{-1}\boldsymbol{r}_{uy} \ , \tag{30}$$

where $\boldsymbol{R}_{\boldsymbol{uu}} = \mathcal{E}\{\boldsymbol{u}(n)\boldsymbol{u}(n)^{\boldsymbol{T}}\}$ is the auto-correlation matrix of $\boldsymbol{u}(n)$ with $(i,j)^{th}$ entry $[\boldsymbol{R}_{\boldsymbol{uu}}]_{ij} = r_{uu}(j-i)$ and $\boldsymbol{r}_{\boldsymbol{uy}} = \mathcal{E}\{\boldsymbol{u}(n)y(n)\}$ is the cross-correlation vector between $\boldsymbol{u}(n)$ and $v(n)$ with i^{th} entry $[\boldsymbol{r}_{\boldsymbol{uy}}]_i = r_{\boldsymbol{uy}}(-i)$. This optimal filter is known as a Wiener filter [39, pp. 100–104].

The Wiener filter has a number of applications, but the formulation given above constitutes a noiseless, single input single output (SISO), system identification problem; a single known signal is input into an unknown system which adds no noise and outputs a single signal, and we estimate the filter used by the system from the single output. This estimation can be extended to multiple input single output (MISO), single input multiple output (SIMO), and multiple input multiple output (MIMO) systems, although SIMO and MIMO systems can typically be broken up into multiple SISO and MISO problems, respectively.

The linear estimation problem can also be extended to the case where the system contains additive noise, i.e., $v(n) = \boldsymbol{a}^T\boldsymbol{u}(n) + w(n)$ where $w(n)$ is a noise process that is statistically independent of $u(n)$. For the work presented here, the noisy SISO and MISO problems are the most relevant, and we can define the MISO case with the same notation as above by letting $\boldsymbol{u}(n) = Vec\{\boldsymbol{u}_1(n), \boldsymbol{u}_2(n), \dots, \boldsymbol{u}_K(n)\}$ and $\boldsymbol{a} = Vec\{\boldsymbol{a}_1, \boldsymbol{a}_2, \dots, \boldsymbol{a}_K\}$, where $\boldsymbol{u}_k(n) = [u_k(n), u_k(n-1), \dots, u_k(n-L+1)]^T$ are the past L samples of the k^{th} input signal, and $\boldsymbol{a}_k = [a_k(0), a_k(1), \dots, a_k(L-1)]^T$ are the coefficients of the k^{th} L-length FIR filter for $k = 1, \dots, K$.

In the work presented here, a MISO system identification problem will be encountered, and although we use the Wiener filter as the solution here, we will introduce some nomenclature that allows for other estimation techniques as well. First, we define the *Wiener system identification* operator as,

$$Id_W\{\boldsymbol{u}(n), y(n); L\} \triangleq \boldsymbol{R}_{\boldsymbol{uu}}^{-1}\boldsymbol{r}_{\boldsymbol{uy}}, \tag{31}$$

we define the *Wiener imaging* operator,

$$Im_W\{\boldsymbol{u}(n), y(n); L\} \triangleq (\boldsymbol{R}_{\boldsymbol{uu}}^{-1}\boldsymbol{r}_{\boldsymbol{uy}})^{\boldsymbol{T}}\boldsymbol{y}(n). \quad (32)$$

where $\boldsymbol{x}(n) = [x_1(n), x_2(n), \dots, x_P(n)]^T$ is a vector of P time-dependent signals at time n, $y(n)$ is a time-dependent signal, $\boldsymbol{u}(n) = Vec\{\boldsymbol{x}_1(n), \boldsymbol{x}_2(n), \dots, \boldsymbol{x}_P(n)\}$, $\boldsymbol{x}_p(n) = [x_p(n), x_p(n-1), \dots, x_p(n-L+1)]^T$ for $p = 1, \dots, P$, and $\boldsymbol{y}(n) = [y(n), y(n-1), \dots, y(n-L+1)]^T$. That is, Id_W produces a PL-length filter relating one (set of) signal(s), $\boldsymbol{x}(n)$, to another signal, $y(n)$, and Im_W produces a signal formed by applying the filter from Id_W to $y(n)$.

It is worth noting here that many adaptive algorithms have been designed to either approximate or converge to the Wiener filter in (30). Adaptive methods generally work sequentially in time so that the filter estimate is time-varying and is, generally, a function of the past filter estimate and a cost function. The most notable adaptive methods that use the MSE criteria in (27) are the least mean square (LMS) , recursive least squares (RLS), and the Kalman (generalization of RLS) adaptive filters [40, pp. 104-111,163-174,198-202]. There are also many methods that use non-MSE criteria, and two methods that use information-theoretic cost functions are the minimum entropy (ME), e.g. [41], [42], and the maximum mutual information (MMI) filters [43], [44].

With these alternative methods of linear estimation in mind, we can denote the general system identification and imaging operators as $Id\{\boldsymbol{x}, y; \boldsymbol{\theta}\}$ and $Im\{\boldsymbol{x}, y; \boldsymbol{\theta}\}$, respectively, where $\boldsymbol{\theta}$ is a method-specific parameter set. Unless noted otherwise in this work, we will employ the Wiener system identification and imaging operators given by (31) and (32), respectively, i.e. the generalized notation is specified as the Wiener notation, here. The linear system identification process and operator described here is crucial to this work, since it allows us to define a scope for our problem as we will see in Chapters 2 onward, as well as giving a succinct format to describe interesting cases used in this exposition.

Another interesting problem is encountered if we let the filtered signal in the previous section be identical to the estimated signal but delayed by one sample. Thus, we are now seeking to estimate the current signal based upon past values of that signal, i.e. we wish to create a filter that *predicts* the signal. For obvious reasons, signals whose samples are uncorrelated from one sample to any other sample cannot be predicted in the manner we describe. We term these signals as "white" since their autocorrelation

functions are a delta function, the delta function has equal power across all frequencies, and "white" light has equal power across all electromagnetic frequencies visible to humans.

That said, white noise is a helpful tool to learn about prediction, since it clarifies some attributes of "predictable" signals. By a predictable signal we mean signals of the form,

$$y(n) = \boldsymbol{a}_p^T \boldsymbol{y}(n-1) + \nu(n), \tag{33}$$

where $\boldsymbol{y}(n) = \left[y(n), y(n-1), \ldots, y(n-L_p-1)\right]^T$, $y(n)$ is a sample of the "predictable" signal at time n, $\boldsymbol{a}_p = \left[a(0), a(1), \ldots, a\left(L_p-1\right)\right]^T$ is an L_p-length filter, $a(k)$ is the k^{th} "autoregressive" (prediction) filter coefficient, and $\nu(n)$ is a innovation sequence. As just mentioned, we will assume $\nu(n)$ is a zero-mean, white noise sequence, so that

$$\mathcal{E}\{\nu(i), \nu(j)\} = \begin{cases} \sigma_\nu^2, & i = j \\ 0, & i \neq j \end{cases}, \tag{34}$$

where we allow $\nu(n)$ to have a constant variance of σ_ν^2 for generality. Thus, the current sample of the signal $y(n)$ is comprised of a weighted sum of its past values plus an innovation sequence. [39, pp. 136–141].

Using the same principles we used to estimate a system identification filter above, we can estimate the prediction filter by minimizing the *prediction* error given by,

$$e_p(n) \triangleq y(n) - \widehat{\boldsymbol{a}}_p^T \boldsymbol{y}(n-1), \tag{35}$$

via a mean-squared prediction error cost function, i.e.,

$$\mathcal{J}_p(n) \triangleq \mathcal{E}\left\{\left|y(n) - \widehat{\boldsymbol{a}}_p^T \boldsymbol{y}(n-1)\right|^2\right\}, \tag{36}$$

where $\widehat{\boldsymbol{a}}_p$ and $\boldsymbol{y}(n-1)$ are L-length, column-oriented vectors. Just as with the system identification problem, setting the gradient of $\mathcal{J}_p(n)$ with respect to $\widehat{\boldsymbol{a}}_p$ equal to zero, we can solve for the optimal L-length prediction filter as,

$$\widehat{\boldsymbol{a}}_p^{opt} \triangleq \boldsymbol{R}_{\boldsymbol{uu}}^{-1} \boldsymbol{r}_{\boldsymbol{u}y}, \tag{37}$$

where we let $\boldsymbol{u}(n) = \boldsymbol{y}(n-1)$, so that $\boldsymbol{R}_{\boldsymbol{uu}} = \mathcal{E}\{\boldsymbol{y}(n-1)\boldsymbol{y}^T(n-1)\}$, $\boldsymbol{r}_{\boldsymbol{u}y} = \mathcal{E}\{\boldsymbol{y}(n-1)y(n)\}$, and to minimize the prediction error, $L \geq L_p$. Thus, the prediction filter estimation problem solution can be succinctly given by the $Id_W\{\}$ operator as,

$$\widehat{\boldsymbol{a}}_p^{opt} = Id_W\{\boldsymbol{y}(n-1), y(n); L\}. \tag{38}$$

Just as in the formulation of the system identification problem, we can generalize $\boldsymbol{u}(n)$ to contain more than one signal, and we will give an example of this extension in Sects. 1.3.5 and 4.7 [39, pp. 136–142], [45], [46].

For now, we simply note that we have given a brief and profoundly understated overview of two linear estimation principles, whose shared connection was further connected to a third problem, smoothing, not mentioned here in the seminal work of Norbert Wiener, e.g. [47]. Volumes have been written on the filtering and prediction problems, as well as their respective estimation fields.

1.3 Blind Source Separation

Although there are several general approaches to the BSS problem, in this section we will confine our attention to BSS under the assumption of mutually, statistically independent sources, generally known as independent component analysis (ICA). As well as providing a historical perspective of the BSS problem, this section also gives the information-theoretical background for a standard BSS method. Beyond that, we will delve into the mechanics of convolutive mixing and demixing, we will outline some of the inherent problems involved with BSS, and we will end the chapter by briefly discussing blind source extraction.

1.3.1 Independent Component Analysis

In 1982, a paper by Bar-Ness et al. [48] roughly formulated the BSS problem within the context of electronic communications. According to [49, p. 1], independently that same year, J. Herault, C. Jutten, and B. Ans were studying the neuroscience of motion in vertebrates, and they not only formulated the BSS problem, but they began to pursue and research the problem. That said, their first published works [50]–[52] began in 1984, and they are widely credited with founding the BSS research area.

Herault et al. tackled the BSS problem by assuming that the sources are pairwise mutually statistically independent, and most accurately, they began the field of independent component analysis (ICA) from which the field of BSS emerged, once researchers realized that statistical independence was not the only criterion which could constrain the BSS estimation problem. The original solution Herault et al. provided, known as the Herault-Jutten algorithm, was limited to two mixtures of two sources, used

a feedback circuit which introduced instability issues, and was not guaranteed to converge to a valid separation solution. However, they paved the way for researchers in the various fields of signal processing, machine learning, communications, etc., to study the problem.

Efforts on the BSS problem were mainly confined to Europe until the mid 1990s. Articles appearing in widely recognized publications in the early 1990s, e.g. [53]–[55], exposed the international scientific community to the BSS problem, and BSS has been a popular area of research ever since. An exhaustive overview of the progress in the BSS arena between the mid-1990s and now is beyond the scope of this paper, and we invite the interested reader to consult [37], [49], [56]–[58] for an overview of and references to research on BSS. For reference, Comon and Jutten cite a telling statistic, "The number of papers published on the subject of BSS and ICA is enormous: in June 2009, 22,000 scientific papers are recorded by Google Scholar in Engineering, Computer Science, and Mathematics." [49, p. 2] That said, we will review the specific BSS research history that is pertinent to this work.

Until 1993, the BSS problem had only been studied in the instantaneous case, i.e. memoryless mixing systems, and this area of study is mainly applicable to the imaging and biomedical fields, e.g. [59], [60]. In the instantaneous case, an observed mixture at time-index, n, can be modeled as the weighted sum of the sources' samples at time-index n, or,

$$\boldsymbol{x}(n) = \boldsymbol{A}^T \boldsymbol{s}(n), \tag{39}$$

where $\boldsymbol{x}(n) = [x_1(n), x_2(n), \dots, x_P(n)]^T$ is a vector of observations, $x_p(n)$ is the observation at the p^{th} sensor for $p = 1, \dots, P$, $\boldsymbol{s}(n) = [s_1(n), s_2(n), \dots, s_Q(n)]^T$ is a vector of source samples, $s_q(n)$ is a sample of the q^{th} source for $q = 1, \dots, Q$, and $\boldsymbol{A}$ is a QxP mixing matrix of the form,

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdots & a_{1P} \\ \vdots & & \vdots \\ a_{Q1} & \cdots & a_{QP} \end{bmatrix}, \tag{40}$$

where a_{qp} is the scaling of the q^{th} source in the p^{th} mixture. The goal of BSS is to recover the sources from the mixtures without any knowledge of the sources or the mixing system, and a linear solution is of the form,

$$\boldsymbol{y}(n) = \boldsymbol{D}^T \boldsymbol{x}(n), \tag{41}$$

where $\boldsymbol{y}(n) = [y_1(n), y_2(n), \dots, y_Q(n)]^T$ is a vector of estimated source signals, with $y_q(n)$ being an estimate of a source for $q = 1, \dots, Q$, and $\boldsymbol{D}$ is PxQ demixing matrix. We use a slightly different notation than is mainly used in the literature, namely we use the transpose of the mixing and demixing matrices, so that the dimensions of the systems denote the number of inputs followed by the number of outputs. Where applicable, we make the necessary adjustments to accommodate our notation within the context of relevant works, and the results of extant work may look slightly different here.

The fundamental assumption in ICA is that the sources are mutually, statistically independent. Formally, we assume that each individual source, s_q, is distributed according to a respective marginal PDF, g_q, i.e.,

$$s_q \sim g_q(s_q), \qquad q = 1, \dots, Q, \tag{42}$$

and that all of the sources, $\boldsymbol{s}$, are jointly distributed according the PDF, $g_{\boldsymbol{s}}(\boldsymbol{s})$, i.e.,

$$\boldsymbol{s} \sim g_{\boldsymbol{s}}(\boldsymbol{s}), \tag{43}$$

where we drop the time-index for clarity. If the sources are mutually, statistically independent, then their joint distribution is equal to the product of the marginals, or,

$$g_{\boldsymbol{s}}(\boldsymbol{s}) = \prod_{q=1}^{Q} g_q(s_q). \tag{44}$$

From this assumption, we further assume that any estimates of the sources, i.e. (41), should also be statistically independent. Therefore, if the source estimates are marginally distributed as,

$$y_q \sim f_q(y_q), \qquad q = 1, \dots, Q, \tag{45}$$

and jointly distributed as,

$$\boldsymbol{y} \sim f_{\boldsymbol{y}}(\boldsymbol{y}), \tag{46}$$

then the ICA assumption in (44) produces the criterion that

$$f_{\boldsymbol{y}}(\boldsymbol{y}) = \prod_{q=1}^{Q} f_q(y_q). \tag{47}$$

That is, we would like to find a set of source estimates such that the product of the marginal distributions is equal to the joint distribution. Noting that this is a comparison of two joint PDFs, we can use the KLD as a measure of similarity between the source estimates' joint PDF and marginal PDFs' product as,

$$\mathcal{J}_{MI}(\boldsymbol{y}) = \mathcal{K}\left\{f_{\boldsymbol{y}}(\boldsymbol{y}); \prod\nolimits_{q=1}^{Q} f_q(y_q)\right\}, \tag{48}$$

where we further note that this is the mutual information (MI) of $\boldsymbol{y}$ as given in (19). Although there are various measures of statistical independence, Comon [54] produced the original formulation of ICA as a source estimate MI minimization problem.

If we now consider that we have not used all of the ICA assumptions made in (42)-(44), we can see a connection between ICA and the maximum likelihood (ML) approach under the source independence criteria. Specifically, the ML approach further assumes that the source estimates are distributed according to the original source PDFs. Taking into account the independence criteria, we can summarize this relation as,

$$\prod_{q=1}^{Q} f_q(y_q) = \prod_{q=1}^{Q} g_q(s_q), \tag{49}$$

where a comparison of these two joint distributions via the KLD is given by,

$$\mathcal{J} = \mathcal{K}\left\{\prod\nolimits_{q=1}^{Q} f_q(y_q); \prod_{q=1}^{Q} g_q(s_q)\right\}. \tag{50}$$

In fact, the expectation of the (normalized) log-likelihood function for the BSS of mutually, statistically independent sources is the sum of (48) and (50), a fact that Obradovic and Greco recognized in [61] and Cardoso examined in a broader context in [62]. If we consider that (48) is sufficient to separate sources in the ICA problem, the astute reader will note that the additional constraints imposed by (50) have profound implications for the BSS problem. In essence, (50) says that we wish to perfectly recreate the sources, but we are unconcerned about the order in which sources are presented as estimates. That is, even though we want the joint PDFs in (50) to be equal, we make no distinction as to the ordering of the estimates (or the original sources, for that matter). This phenomenon is known as the BSS *permutation ambiguity*, and we will cover it further in Sect. 1.3.4. Furthermore, if we note that MI is scale-invariant, then (48) says we can separate the sources at an arbitrary scaling, a constraint that is implicitly forbidden in (50). This is the BSS *scaling ambiguity* which will also be addressed in Sect. 1.3.4, but in a more general form.

Even if we combine (48) and (50) as the separation criteria, we should note that we will only be able to separate a set of sources which contains, at most, one Gaussian

distributed source, since the combination of two or more Gaussians will produce a Gaussian. In the instantaneous case, separation of multiple independent, identically distributed (IID) Gaussians is impossible. In the instantaneous case, when the set of Gaussians exhibit pairwise unique (beyond scaling) time-structure, i.e. unique covariance matrices, unique AR processes, etc., then methods exist to separate these sources, e.g. [49, pp. 227–279]. In the convolutive case (discussed later), even multiple IID Gaussians can be separated if they have unique spatial locations (time offsets).

Having discussed the separation criteria necessary for ICA, we now look at how to use these criteria in an *algorithm* to solve the BSS problem. In this work, we typically use the term "BSS algorithm" in a very general sense to denote any method used to solve the BSS problem. Here, however, we will use the term BSS algorithm to denote a set of source separation criteria coupled with an optimization method. Specifically, we will now look at the MI separation criteria under the gradient descent optimization technique, given generally as,

$$\widehat{\boldsymbol{D}}(n+1) = \widehat{\boldsymbol{D}}(n) + \mu \nabla_{\boldsymbol{D}} \mathcal{J}_{MI}(\boldsymbol{y}) \,,$$

where $\widehat{\boldsymbol{D}}(n)$ is an estimated demixing matrix at time n, μ is a scalar step size, and $\nabla_{\boldsymbol{D}}$ is the gradient operation with respect to $\boldsymbol{D}$.

To begin, we look at the gradient of $\mathcal{J}_{MI}(\boldsymbol{y})$ with respect to the demixing matrix when the mixtures are critically-determined, given here as,

$$\nabla_{\boldsymbol{D}^T} \mathcal{J}_{MI}(\boldsymbol{y}) = \boldsymbol{D}^{-1} - \boldsymbol{\phi}(\boldsymbol{y})\boldsymbol{x}^T, \tag{51}$$

where $\boldsymbol{\phi}(\boldsymbol{y})$ is a vector of *score function* values whose q^{th} entry is $[\boldsymbol{\phi}(\boldsymbol{y})]_q = \phi_q(y_q)$ given by,

$$\phi_q(y_q) = -\frac{d}{dy_q} \log g_q(y_q) \,, \tag{52}$$

and a full derivation is given in [55]. Approaching the BSS problem via gradient descent, i.e. using (51) as a demixing solution update mechanism, has several problems: it converges slowly, typically, it is not guaranteed to converge to a global minimum, the demixing matrix needs to be inverted at every iteration, and the solution is dependent upon the, possibly ill-conditioned, mixtures.

From independent perspectives, Amari [63] and Cardoso and Laheld [64] produced solutions that are identical when applied to BSS. Cardoso and Laheld observed that the truly problematic term in the gradient of mutual information was the dependence

upon the observations, therefore they transformed the mixtures into the outputs of a learning system, and then proceeded to study the area near an optimal solution in probabilistic terms. Amari, on the other hand, treated the gradient of mutual information as a curved, Riemannian, space, and used geodesics to point to the true direction of minimum mutual information. Amari's *natural* gradient, and Cardoso and Laheld's *relative* gradient are synonymous within the context of BSS, and they allow faster convergence to a global minimum of mutual information. Given in terms of the gradient of a general BSS cost function, $\mathcal{J}$, the natural/relative gradient is defined as,

$$\nabla_{NG}\mathcal{J}(\boldsymbol{y}) \triangleq [\nabla_{\boldsymbol{D}^T}\mathcal{J}(\boldsymbol{y})]\boldsymbol{D}\boldsymbol{D}^T\,, \tag{53}$$

so that the natural/relative gradient of (51) becomes,

$$\nabla_{NG}\mathcal{J}_{MI}(\boldsymbol{y}) \triangleq [\boldsymbol{I} - \boldsymbol{\phi}(\boldsymbol{y})\boldsymbol{y}^T]\boldsymbol{D}^T, \tag{54}$$

where $\boldsymbol{I}$ is the QxQ identity matrix. Besides the trivial solution, $\boldsymbol{D} = \boldsymbol{0}$, we see the gradient is zero when $\boldsymbol{\phi}(\boldsymbol{y})\boldsymbol{y}^T = \boldsymbol{I}$, i.e. when all off-diagonal terms of $\boldsymbol{\phi}(\boldsymbol{y})\boldsymbol{y}^T$ are zero and the diagonal terms are one.

At this point, we have implicitly assumed that the number of sources is known, i.e. via the dimension of the demixing matrix, and that the sources are statistically stationary. Although methods exist for estimating the number of sources, e.g. [65], and the problem of non-stationary sources has been studied, e.g. [66], we will briefly note the work of Amari et al. [67] which gives a solution to both of these issues when there are, at least, as many sensors as there are sources, i.e. the critically- and over-determined cases. In studying BSS of non-stationary sources with an adaptive, feed-forward system, Amari et al. recognized that constraining the diagonal terms of $\boldsymbol{\phi}(\boldsymbol{y})\boldsymbol{y}^T$ to be equal to one at every time-step in a learning process was unrealistic, since the statistics of the *nonstationary* sources were *necessarily time-varying*. Thus, they proposed the learning rule,

$$\widehat{\boldsymbol{D}}(n+1) = \widehat{\boldsymbol{D}}(n) + \mu[\boldsymbol{\Lambda}(n) - \boldsymbol{\phi}(\boldsymbol{y}(n))\boldsymbol{y}(n)^T]\widehat{\boldsymbol{D}}^T(n)\,, \tag{55}$$

where the so-called *non-holonomic* constraint is represented by $\boldsymbol{\Lambda}(n)$, i.e. the matrix formed by setting all off-diagonal elements of $\boldsymbol{\phi}(\boldsymbol{y}(n))\boldsymbol{y}(n)^T$ to zero. We recognize this as the gradient descent method, using the natural gradient of MI, i.e. (54), with $\boldsymbol{\Lambda}(n)$ substituted in place of the identity matrix. We note that the diagonal elements of $\boldsymbol{\Lambda}(n) - \boldsymbol{\phi}(\boldsymbol{y}(n))\boldsymbol{y}(n)^T$ are always zero, so that the only task for the learning method is to

minimize the mutual dependencies of the outputs. We also note that the $\boldsymbol{\Lambda}(n)$ term does not suppose that its diagonal elements are non-zero, thus the non-holonomic constraint can estimate a number of sources that is less than or equal to the number of mixtures. For completeness, we note that the Cichocki-Unbehauen algorithm [68]–[70], derived in the early 1990s, has a striking resemblance to Amari et al.'s [67] natural gradient descent of mutual information under the non-holonomic constraint just outlined.

In this work, we are mainly concerned with the convolutive SS problem, which has a compact representation, similar to (39) above, but in the frequency domain, and given by,

$$\boldsymbol{x}(\omega) = \boldsymbol{A}^T(\omega)\boldsymbol{s}(\omega), \tag{56}$$

where ω is a normalized frequency, $\boldsymbol{x}(\omega) = [X_1(\omega), X_2(\omega), \dots, X_P(\omega)]^T$ is a vector of observation spectra, $X_p(\omega)$ is the Fourier transform of $x_p(n)$ for $p = 1, \dots, P$, $\boldsymbol{s}(\omega) = [S_1(\omega), S_2(\omega), \dots, S_Q(\omega)]^T$ is a vector of source spectra, $S_q(\omega)$ is the Fourier transform of $s_q(n)$ for $q = 1, \dots, Q$, and $\boldsymbol{A}(\omega)$ is a QxP frequency-dependent mixing matrix of the form,

$$\boldsymbol{A}(\omega) = \begin{bmatrix} A_{11}(\omega) & \cdots & A_{1P}(\omega) \\ \vdots & & \vdots \\ A_{Q1}(\omega) & \cdots & A_{QP}(\omega) \end{bmatrix}, \tag{57}$$

where $A_{qp}(\omega)$ is the frequency response of a filter describing the channel from the q^{th} source to the p^{th} mixture. Although Comon first mentioned possible approaches to the convolutive BSS problem in 1990 [71], the first treatment of the subject occurred in 1993 by Weinstein et al. [72], and Yellin and Weinstein soon provided very strong conceptual and practical treatments of the problem with their collaborative works in 1994 [73] and 1996 [74]. As a testament to their work, the observations Yellin and Weinstein made about the criteria for convolutive BSS are still employed today.

Buchner et al. [75], [76] generalized to the convolutive domain the problem of ICA under the natural gradient of mutual information with a non-holonomic constraint and signal processing suitable for non-white, non-stationary, and non-Gaussian sources with their TRINICON algorithm, which stands for TRIple-N Ica for CONvolutive mixtures. In parallel work, they focused on a second-order statistics approach, which can be viewed as a sub-class of TRINICON when the source distributions are considered to be Gaussian [77].

1.3.2 Convolutive Mixtures

In addition to the frequency-domain convolutive mixture representation given in (56), we now consider alternative representations in the time-domain that will be useful in this work. Consider the case where Q source signals impinge upon a set of P sensors. In many situations, we can model this interaction as a linear combination of the sources. If we define an L_m-length vector of samples from the q^{th} source at time-index n as,

$$\boldsymbol{s}_q(n) = \left[s_q(n), s_q(n-1), \ldots, s_q(n-L_m+1)\right]^T, \qquad q = 1, \ldots, Q \tag{58}$$

where $s_q(n)$ is a sample of the q^{th} source at time-index n, then we can model the observation at sensor p, at time-index n, as,

$$x_p(n) = \sum_{q=1}^{Q} \boldsymbol{a}_{qp}^T \boldsymbol{s}_q(n) \ , \tag{59}$$

where $\boldsymbol{a}_{qp}$ is an L_m-length vector containing the coefficients of the finite impulse response (FIR) filter describing the channel between the q^{th} source and the p^{th} sensor, i.e.,

$$\boldsymbol{a}_{qp} = \left[a_{qp}(0), a_{qp}(1), \ldots, a_{qp}(L_m-1)\right]^T. \tag{60}$$

In general, $\boldsymbol{a}_{qp}$ can be time-varying, i.e. $\boldsymbol{a}_{qp}(n)$, due to moving sources or sensors, but for now, we will just consider time-invariant mixing. Furthermore, we have modeled the mixing filter in (60) as a constant *length* FIR filter, but in reality, the channel filter length might be more realistically modeled as time-varying. Although this is an interesting thought, generally, here we will assume that all filter lengths are equal to the maximum filter's length, i.e. shorter filters are padded with zeros.

An alternative form of convolutive mixing is given if we define the *image* of source q in the p^{th} mixture as,

$$\tilde{s}_{qp}(n) \triangleq \boldsymbol{a}_{qp}^T \boldsymbol{s}_q(n) \ , \tag{61}$$

so that,

$$x_p(n) = \sum_{q=1}^{Q} \tilde{s}_{qp}(n) \ , \tag{62}$$

and we see that a convolutive mixture is a simple sum of all source images for that sensor. This is just an instantaneous mixture of sensor-specific, filtered sources.

Conceptually, the form in (62) is very useful, but the next form for convolutive mixing provides notational simplicity. Using the vectorization operator, $Vec\{\cdot\}$, defined in (3), we define the "stacked" vector of source samples at time-index n as,

$$\boldsymbol{s}(n) = Vec\{\boldsymbol{s}_1(n), \boldsymbol{s}_2(n), \dots, \boldsymbol{s}_Q(n)\}, \tag{63}$$

and the stacked vector of filter coefficients for the p^{th} mixture as,

$$\boldsymbol{a}_p = Vec\{\boldsymbol{a}_{1p}, \boldsymbol{a}_{2p}, \dots, \boldsymbol{a}_{Qp}\}, \tag{64}$$

so that the p^{th} mixture can be expressed as,

$$x_p(n) = \boldsymbol{a}_p^T \boldsymbol{s}(n). \tag{65}$$

Furthermore, we can define the matrix $\boldsymbol{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2, \dots, \boldsymbol{a}_P]$, so that the vector of all observations at time-index n is given by,

$$\boldsymbol{x}(n) = \boldsymbol{A}^T \boldsymbol{s}(n), \tag{66}$$

where $\boldsymbol{x}(n) = [x_1(n), x_2(n), \dots, x_P(n)]^T$.

1.3.3 Linear Demixing

From a general standpoint, a source separation algorithm, $\mathcal{A}$, uses a demixing operation, $\mathcal{D}_{\mathcal{A}}$, on a set of observations, $\boldsymbol{x}(n)$, to produce a set of source estimates, $\boldsymbol{y}_{\mathcal{A}}(n)$, as,

$$\boldsymbol{y}_{\mathcal{A}}(n) = \mathcal{D}_{\mathcal{A}}\{\boldsymbol{x}(n); \mathbb{P}_{\mathcal{A}}(n)\}, \tag{67}$$

where $\mathbb{P}_{\mathcal{A}}(n)$ is a set of (potentially time-varying) parameters used by the demixing operation. Various algorithms have various demixing operators, but for exemplary purposes we will define a linear demixing operator, $\mathcal{D}_l$. Working in a similar way to obtain the linear mixtures in the previous section, we define a vector of stacked demixing filter coefficients, $\boldsymbol{d}_q$, as,

$$\boldsymbol{d}_q = Vec\{\boldsymbol{d}_{1q}, \boldsymbol{d}_{2q}, \dots, \boldsymbol{d}_{Pq}\}, \qquad q = 1, \dots, Q, \tag{68}$$

where $\boldsymbol{d}_{pq} = \left[d_{pq}(0), d_{pq}(1), \dots, d_{pq}(L_d - 1)\right]^T$ is a vector of L_d-length FIR coefficients describing the demixing filter applied to the p^{th} mixture used to recover the q^{th} source. If we let $\boldsymbol{x}(n) = Vec\{\boldsymbol{x}_1(n), \boldsymbol{x}_2(n), \dots, \boldsymbol{x}_P(n)\}$ where,

$$\boldsymbol{x}_p(n) = \left[x_p(n), x_p(n-1), \dots, x_p(n - L_d + 1)\right]^T, \qquad p = 1, \dots P, \tag{69}$$

then we can produce the q^{th} source estimate at time-index n as,

$$y_q(n) = \boldsymbol{d}_q^T \boldsymbol{x}(n). \tag{70}$$

We can define the matrix of coefficients used to estimate all sources as,

$$\boldsymbol{D}_l = [\boldsymbol{d}_1, \boldsymbol{d}_2, \dots, \boldsymbol{d}_Q] \tag{71}$$

and the linear demixing operation is defined as,

$$\mathcal{D}_l\{\boldsymbol{x}(n);\ \boldsymbol{D}_l\} \triangleq \boldsymbol{D}_l^T \boldsymbol{x}(n), \tag{72}$$

where the demixing operator is the matrix-vector product operation, and the parameter set is the demixing matrix, i.e. $\mathbb{P}_l = \{\boldsymbol{D}_l\}$. The structure of $\boldsymbol{D}_l$ points out some potential challenges in the source separation problem.

1.3.4 Issues in the Blind Source Separation Problem

If we note that the dimensions of $\boldsymbol{D}_l$ are $[PL_a \text{x} Q']$, then we first see that Q' is the number of sources that are estimated via $\mathcal{D}_l$. In the case where we do not know the number of sources, Q, then a particular algorithm, $\mathcal{A}$, needs to explicitly or implicitly estimate the number of sources as $\hat{Q}$ in order to construct the matrix $\boldsymbol{D}_l$. There are many works on source enumeration, e.g. [65], [78]–[80], but the simplest source number assumption is $\hat{Q} = P$, so that the mixing system is assumed to be critically-determined. When $\hat{Q} < P$, i.e. the over-determined case, a general strategy is to reduce the dimensions to a critically-determined problem by using $\hat{Q}$ mixtures as inputs to a critically-determined SS algorithm. Amari's non-holonomic constraint is a robust technique that uses all P mixtures, but allows outputs to be zero, so that it handles both over- and critically-determined mixing systems without needing an explicit estimate of Q. In the under-determined case, i.e. $\hat{Q} > P$, there are either an infinite number of algebraic solutions or no algebraic solution exists, in the time-domain. Typical solutions to this problem involve transforming the observations to a domain in which the problem is either critically- or over-determined, e.g. [21], [81]–[85], but we also note that the very broad sensor array-processing fields of beam-forming and beam-nulling, e.g. see [86], have suitable solutions for the general convolutive BSS problem under suitable sensor arrangements.

The structure of $\boldsymbol{D}_l$ is also useful for describing the *permutation ambiguity* in the SS problem. Let's first consider the case in which $\boldsymbol{D}_l$ is a perfect demixing solution so that the sources are perfectly recovered, i.e. $\boldsymbol{s}(n) = \boldsymbol{D}_l^T \boldsymbol{x}(n)$. Now consider the demixing matrix, $\boldsymbol{G}$, which contains the same coefficients as $\boldsymbol{D}$, but the columns have been

rearranged, e.g. $\boldsymbol{G} = \boldsymbol{D}\boldsymbol{P}$ where $\boldsymbol{P}$ is a QxQ permutation matrix. The columns of $\boldsymbol{D}$ determine the outputs, so $\boldsymbol{G}_l^T \boldsymbol{x}(n)$ still produces perfect estimates of the original sources, but the sources are reordered. Both $\boldsymbol{D}$ and $\boldsymbol{G}$ are valid solutions to the SS problem, since they perfectly separate the sources, but they differ in the ordering of the sources. At first glance, this is an inconsequential result that stems from our labeling the sources in a particular order. However, in SS algorithms that process data sequentially in time, a reordering of the source estimates at each time-step is devastating, and any method susceptible to this permutation ambiguity must correct for this issue. In the MASS problem, the permutation ambiguity is relevant.

In the blind scenario, even the critically-determined SS problem admits an infinite number of solutions. To see this, we recall the frequency domain representation of mixing given in (56), where the mixing system, $\boldsymbol{A}(\omega)$, is a square matrix due to the critically-determined assumption, and for simplicity we will consider the $P = Q = 2$ case so that $\boldsymbol{A}(\omega)$ is given as,

$$\boldsymbol{A}(\omega) = \begin{bmatrix} A_{11}(\omega) & A_{12}(\omega) \\ A_{21}(\omega) & A_{22}(\omega) \end{bmatrix},$$

If it exists, $\boldsymbol{A}^{-1}(\omega)$ is, clearly, a solution that produces the separated sources from the mixtures. In fact, $\boldsymbol{A}^{-1}(\omega)$ represents a *deconvolution* solution in that it produces, *exactly*, the sources presented at the input to the mixing system. If we look at a representation of the inverse of $\boldsymbol{A}(\omega)$ in terms of the determinant and adjoint, i.e.

$$\boldsymbol{A}^{-1}(\omega) = \frac{1}{\det\big(\boldsymbol{A}(\omega)\big)} \mathrm{adj}\big(\boldsymbol{A}(\omega)\big) \tag{73}$$

where the adjoint is given as,

$$\mathrm{adj}\big(\boldsymbol{A}(\omega)\big) = \begin{bmatrix} A_{22}(\omega) & -A_{12}(\omega) \\ -A_{21}(\omega) & A_{11}(\omega) \end{bmatrix},$$

we see that $\mathrm{adj}\big(\boldsymbol{A}(\omega)\big)$ is also a solution that separates the sources. Furthermore, if we note that the reciprocal of the determinant in (73) is just a constant scalar, then we can define a general demixing solution as, $\boldsymbol{D}(\omega) = \alpha(\omega)\mathrm{adj}\big(\boldsymbol{A}(\omega)\big)$, where $\boldsymbol{D}(\omega)$ is the frequency response of the demixing system, and $\alpha(\omega)$ is an arbitrary, but constant, frequency response. That is, sources can be separated up to some arbitrary filtering, which is known as the *filtering ambiguity*. We can combine the permutation ambiguity and the filtering ambiguity, to give a general demixing system response as,

$$\boldsymbol{D}(\omega) = \boldsymbol{P}(\omega)\alpha(\omega)\mathrm{adj}\big(\boldsymbol{A}(\omega)\big), \tag{74}$$

where $\boldsymbol{P}(\omega)$ is the permutation matrix described previously. We described the permutation ambiguity generally and its effect on time-domain BSS earlier, but (74) points out a particularly devastating effect for BSS in the frequency domain if the permutation matrix is frequency-dependent. If $\alpha(\omega)\mathrm{adj}\big(\boldsymbol{A}(\omega)\big)$ applied to the mixtures produces a valid separation solution at every frequency, a frequency-dependent permutation matrix has the potential to undo that separation and remix the sources.

1.3.5 Blind Source Extraction

Here, we will briefly consider the problem of extracting a subset of the sources contained in a set of observations, or blind source extraction (BSE). The approaches to this problem are at least as varied as the approaches to the BSS problem, since any BSS method can be turned into a BSE method, simply by omitting or selecting a subset of the estimated sources. Although a comprehensive overview of BSE methods is well beyond the scope of this document, we note that almost every volume dedicated to the BSS task, e.g. [37], [49], [56]–[58], contains methods completely devoted to the BSE task (that is, methods that are not a simple selection of outputs from a BSS task).

In the work presented here, we will utilize two BSE methods. The first method, given in [45], [46], was originally intended as a single source dereverberation technique, but under certain source and sensor geometries, this method is able to extract and isolate certain sources. The second method has been developed with the aid of the MASS framework presented here, and the method is capable of extracting sources with certain time-structures and is explained in detail in Sect. 4.8 and Appendix A. Thus, the two BSE methods we use here exhibit a diversity in the assumptions made about the observations, where one method is geometry-dependent, and the other is time-dependent.

We simply note the BSE topic here since the MASS framework is designed to allow as many diverse source estimation algorithms as possible, and BSE is a valid approach to source estimation. Analogous to the BSS/SBSS/SSS classifications in SS, we will refer to this topic as source extraction (SE) when speaking generally about extraction techniques or SS when talking of general source separation.

1.3.6 Separation Analysis: The Signal to Interference Ratio (SIR)

For each signal produced by a SS technique, the goal of the method is to isolate a single source in that signal, and before concluding this chapter we now turn our attention to analyzing whether the method has achieved that goal or not. The signal to interference ratio (SIR) is a supervised method (sources are known) that can be used to evaluate the performance of a particular SS output in terms of the power of a particular source relative to the total power of the interfering sources.

Letting $y_i(n)$ be the i^{th} output of a SS technique, we will model $y_i(n)$ as a linear mixture of source images, e.g. (62) in Sect. 1.3.2, and we assume that all sources are zero-mean, and mutually uncorrelated. Defining an L_m-length vector of samples of the q^{th} source at time-index n as,

$$\boldsymbol{s}_q(n) = \left[s_q(n), s_q(n-1), \dots, s_q(n-L_m+1)\right]^T, \qquad q = 1, \dots, Q \tag{75}$$

where $s_q(n)$ is a sample of the q^{th} source at time-index n, we model $y_i(n)$ as,

$$y_i(n) = \sum_{q=1}^{Q} \tilde{s}_{qi}(n)\,, \tag{76}$$

where,

$$\tilde{s}_{qi}(n) = \boldsymbol{g}_{qi}^T \boldsymbol{s}_q(n) \tag{77}$$

is the image of the q^{th} source in $y_i(n)$, $\boldsymbol{g}_{qi}$ is an general L_m-length filter applied to the q^{th} source in $y_i(n)$ and given by,

$$\boldsymbol{g}_{qi} = \left[g_{qi}(0), g_{qi}(1), \dots, g_{qi}(L_m-1)\right]^T. \tag{78}$$

Letting $\tilde{\sigma}_{qi}^2(n) = \mathcal{E}\{\tilde{s}_{qi}^2(n)\}$ be the variance of the q^{th} source image in $y_i(n)$, then we can define the SIR of the q^{th} source in $y_i(n)$ as,

$$SIR_{qi}(n) \triangleq \frac{\tilde{\sigma}_{qi}^2(n)}{\sum_{\substack{k=1 \\ k\neq q}}^{Q} \tilde{\sigma}_{ki}^2(n)}. \tag{79}$$

That is, the SIR of the q^{th} source in $y_i(n)$ is the power of the q^{th} source's image divided by the total power of all other source images in $y_i(n)$. [87]–[89]

In general, we do not know the SIR for a particular source in a given signal, and thus we need to estimate the SIR. Clearly, we need to estimate the images of all the sources in the signal, therefore we require the underlying, pristine source signals, i.e.

$s_q(n)$ for $q = 1, \dots, Q$. With these source signals we can estimate the images via the imaging operator, $Im\{\}$, as,

$$\tilde{z}_{qi}(n) = Im\{\boldsymbol{s}_q(n), y_i(n); L, D\}, \qquad q = 1, \dots, Q, \tag{80}$$

where $\tilde{z}_{qi}(n)$ is an estimate of the q^{th} source in $y_i(n)$, L is a filter length with $L \geq L_m$, and D is a delay applied to $\boldsymbol{s}_q(n)$ to ensure a causal solution. With these image estimates in hand, we can calculate the individual variances and use (79) to calculate SIR. [2]

CHAPTER 2: MULTIPLE ALGORITHM SOURCE SEPARATION (MASS)

Multiple algorithm source separation (MASS) aims to utilize the source estimates produced by multiple source separation (SS) algorithms running in parallel to produce a set of composite source estimates (CSEs) that estimate the underlying sources contained in a set of observation data. The MASS problem represents a collaborative-learning take on the SS problem *domain* and can been viewed as either completely or partially encompassing the SS domain. In this work, we have chosen to tackle and describe the MASS problem in terms of a signal processing software *framework*, and well-developed frameworks are domain-specific and require the collaboration of domain experts with software engineering experts [90], [91]. This chapter deals with MASS domain-specific issues, whereas Chapter 3 deals with the software definition of a specific MASS framework resulting from our knowledge of the MASS domain.

The motivation behind MASS stems from two observations within the SS landscape; all SS algorithms make at least one assumption about the sources, sensors, environment, etc. and are thus suited to *particular scenes*, and in the realm of adaptive SS systems, one can observe that for a given algorithm, the individual sources are typically separated out at different rates. From these observations, we assert that in order to impose the fewest limitations possible, i.e. increase "blindness", we should actually make as many assumptions as we can by using a diverse set of SS algorithms running in parallel and take advantage of differing rates and levels of individual source separation.

The inspiration behind MASS comes from the multiple algorithm work done in other signal processing domains, e.g. model selection, adaptive filtering, linear prediction, adaptive beamforming, etc. Although these fundamental problems are not directly extensible to the general SS problem, some of the general methods used to solve these multiple-algorithm problems influence thoughts on the MASS problem, as we shall see in sects. 2.2 and 2.3.

The software framework in the next chapter both facilitates and enhances SS research and development, but the generality of the framework ensures that the set of CSEs produced cannot be *guaranteed* to be "as good as" any of the source estimate sets produced by any individual algorithm. Indeed, the current framework is purely designed to study and implement SS methods on a very general level within a MASS context, and

the knowledge and expertise of the framework user and the plugin implementations available to the user have a large influence on the MASS results.

However, the MASS framework is versatile in the sheer number of possible configurations even with a small number of plugin implementations, and the MASS framework's main utilities manifest as a sandbox for researchers to develop SS methods and a tool for SS practitioners to implement SS applications. In the hands of an experienced SS researcher, the MASS framework is an invaluable tool to better understand, teach, and manage the intricacies of the SS problem.

In the rest of this chapter we will cover the main concepts encountered while tackling the MASS problem generally, along with some assumptions and strategies that help formulate a MASS framework realization. In Sect. 2.1 we will give an overview of the MASS problem, as well as a summarized version of the main components contained in the MASS framework we present in this work. In Sect. 2.2 we will cover the issues involved with using multiple source estimates from multiple SS methods, and we will outline the fundamental assumptions we make in this work to produce a usable MASS system. We will briefly overview the issues involved with evaluating multiple source estimates from a diverse set of SS algorithms in Sect. 2.3, as well as reviewing some simple strategies for constructing CSEs.

2.1 An Overview of a General, Block-Processing MASS Framework

In general, the SS method can be decomposed into a source enumeration problem, a set of criteria that determines how sources can be identified and separated, and a method for dealing with the permutation ambiguity. From a top-level view, MASS is just another method to perform SS, so one can easily conclude that MASS must also address the source enumeration, separation criteria, permutation ambiguity, and filtering ambiguity issues. In fact, this SS problem decomposition forms the starting point to create a usable MASS framework. Figure 4 shows all of the basic components of the MASS signal processing framework where the SS decomposition components are given in pink (salmon), and the MASS-specific components are shown in blue. Even though the approach we give in this section is general, we note that our approach is only one possibility. We will now give a brief overview of the system before exploring some of the key ideas, components, and capabilities of this general MASS framework.

The components given in pink in Figure 4 deal with the SS-specific issues: Source Estimate Production system (SEPs), Source Enumeration (SNUM), Source Estimate Evaluation (SEE), Permutation Ambiguity Solution (PAS), and Filtering Ambiguity Solution (FAS). SEPs is composed of multiple source estimate production (SEP) plugins, where each SEP plugin is composed of an SS method that produces source estimates, thus the SEPs produces multiple sets of source estimates. The SEE plugin, fundamentally, evaluates the SEPs' source estimates under a set of SS criteria in order to create a set of composite sources estimates (CSEs). The SNUM plugin allows the user to specify a method that explicitly estimates the number of sources given the input data, the PAS plugin allows a user to specify a method to solve the permutation problem on the CSEs, and the FAS plugin is provided as a "stitching" mechanism so that source estimates are consistent in a block-processing context. The SNUM, SEE, PAS, and FAS plugins represent a decomposition of the SS problem, and the individual SEP plugins represent full-formed SS methods. The SEP and SEE plugins present conceptual challenges to the MASS system, and we will discuss these components later in this chapter. The SNUM, PAS, and FAS plugins represent important areas of research, however their insertion into the MASS framework is fairly straightforward, and we will only give their respective fundamental concepts a cursory treatment in this work.

The blue elements in Figure 4 are components that deal with issues that arise when designing a MASS framework. The Source Estimate Grouping (SEG) plugin sorts the source estimates from the SEPs into groups, where each group contains estimates of one particular source. The Source Estimate Time-Alignment (SETA) plugin time-aligns the source estimates from either the SEPs or the SEG plugins. Both the SEG and SETA plugins' functionality arise from various SEE methods that require grouped and/or time-aligned source estimates. We supply the SEG and SETA plugins for generality, but beyond the plugin definitions given in Ch. 4 and some basic examples given in Appendix A, we will not treat the SEG and SETA goals in in any meaningful manner in this work.

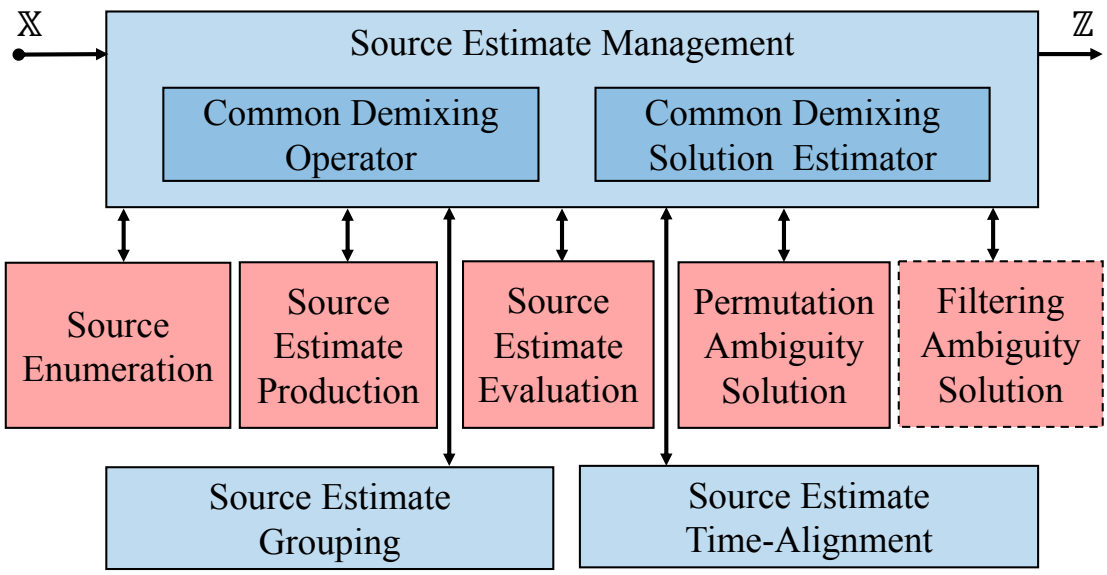

Figure 4. The basic components of a general MASS framework. The items in pink represent a decomposition of the general source separation (SS) problem, and items in blue address general MASS issues. The Source Estimate Management (SEM) system controls data flow and task execution in the MASS framework, where X and Z are general input and output datasets, respectively. The Common Demixing Operator (CDO) and Common Demixing Solution (CDS) Estimator allow SS methods to collaborate. The Source Enumeration (SNUM) plugin estimates the number of sources. The Source Estimate Production (SEP) system is comprised of multiple (SS) SEP plugins running in parallel and provides source estimates. The Source Estimate Evaluation (SEE) plugin evaluates the source estimates to provide a composite source estimate (CSE) set. The Permutation Ambiguity Solution (PAS) plugin assures the CSE for particular source is consistently output on the same channel. The Filtering Ambiguity Solution (FAS) plugin guarantees source fidelity. The Source Estimate Grouping (SEG) and Source Estimate Time-Alignment (SETA) plugins provide general functionality for the SEE plugin.

That said, the remaining blue items in Figure 4, i.e. the Source Estimate Management (SEM) system, the Common Demixing Operator (CDO), and the Common Demixing Solution (CDS) estimator, are critical to our strategy for MASS. The SEM system controls all information flow and task execution within the MASS framework, and although we give an overview of the SEM system in the next section, the SEM system's responsibilities are detailed in Ch. 3. The CDO provides a common language for SS methods with disparate approaches to communicate their demixing information to the SEM as well as other SS methods, as we shall see in Sect. 2.1.2. The CDS estimator

allows the SEM to estimate a particular demixing solution in the CDO format for a particular source estimate, where once again, we will delve into more detail in Sect. 2.1.2.

2.1.1 A General Block-Processing Constraint for MASS

A fundamental assumption that we make in this work is that any individual SS method used in the MASS framework is capable of processing data in a block-sequential manner. We impose this restriction since some SS methods are designed for block-processing, while others that are not, can be modified to do so. For instance, an SS method that is designed to work one *sample* at a time, can be implemented in a SEP plugin to sequentially process each sample in a block of data, essentially becoming a block-processing SS method. Furthermore, our formulation of a MASS framework does not dictate the minimum or maximum block lengths, rather the various plugins inform the SEM system of their range of data block lengths that are acceptable, and the SEM system, typically, chooses the shortest data block length that is acceptable to all. For example, if a set of mixtures is processed by a MASS system in which all plugins only require a minimum block size of one sample, then MASS would work sample-sequentially. For another example, batch methods that work on all samples at once typically have a minimum number of samples in which the batch requirement is met, thus providing a minimum block length. This block-processing constraint is not without consequences, however, and we now briefly describe some of the issues encountered.

The implication of converting non-block-processing SS methods into block-processing SS methods via the MASS SEP plugin is not lost on the author, and in fact, the MASS framework provides a way to study these effects. Even though the point of this work is to study *multiple* algorithms running in parallel, the MASS system can be configured to run a single SS method, and therefore, the effects of the various MASS parameters, including data-block length, on a single algorithm can readily be evaluated.

Although any MASS plugin can specify a minimum data-block length, there are a few factors that determine the actual data-block length that MASS uses. If multiple algorithms are running, MASS uses the maximum block length across all SEP plugins' minimum data block, thereby ensuring the minimum block length is satisfied for all methods, if possible. Furthermore, the user can configure the MASS system to use a data block length that is greater than the length just specified. Assuming a CDO that involves

convolution, then once this preliminary block length, N, has been determined, the data block length, M, is set to be the preliminary block length plus the length of the CDS filter, L, minus one sample, i.e. $M = N + L - 1$. Even though we supply the algorithms with M samples, we use the central part of the convolution to provide N samples at the output.

In a block-processing context, an issue of source estimate continuity arises. That is, the solution used to create a particular CSE may change from block to block and a discontinuity, e.g. a click in audio tracks, can occur at the boundary between two blocks. The FAS plugin is designed to deal with smoothing these boundary issues by determining a global filtering of a particular source's CSE that is constant with respect to time. For example, once a CSE is constructed, we can use the $Im\{\}$ operator to image the CSE onto a particular mixture, so that the CSE represents the approximate image of the source in the mixture. That said, we do not actually deal with this issue, nor have we actually defined or implemented the FAS plugin in this work, and we will only address this issue in future work.

2.1.2 A Fundamental Strategy for MASS: The Common Demixing Operator

In this work, we propose to use multiple, diverse SS algorithms running in parallel to perform the SS task. To perform this MASS task, we need to evaluate the performance of the algorithms' source estimates to create a set of CSEs, as well as provide the demixing solution necessary to produce the CSE set. That said, the diversity of algorithms comes with a diversity of SS approaches, and more importantly, a diversity of demixing operations. This presents the problem of how to describe the demixing operation for an individual CSE, when that CSE is comprised of source estimates from algorithms which have differing demixing operations. Many solutions to this problem exist, e.g. simply convey the CSE demixing operation as a combination of the constituent demixing operations, design a framework that does not need the CSE demixing operation, etc., however we take the approach of requiring that the demixing operation for any source estimate be transformed into a parameterized *common demixing operation* (CDO). That is, for any demixing operation present in the SS algorithm set, we require a method to convert the algorithm-specific demixing operation to a single demixing operation.

This approach provides a general strategy for MASS. Conceptually, we can describe any source estimate or CSE as the application of the CDO to the observation data, so that any source estimate only differs by the parameter values of the CDO. This alleviates the need to specify each individual algorithm's demixing operation and operation-specific parameter set when speaking of the MASS framework in abstraction. Practically, the CDO provides a uniform method to compute source estimates from observation data and gives individual algorithms a way to use SS solution information produced by MASS, as we shall see in the next section. In general, for any source estimate (or CSE), we term the combination of the CDO and the estimate-specific parameter values as a common demixing *solution* (CDS), since in practice, the parameterized CDO applied to the observation data generates the source estimate.

From a general standpoint, the specific CDO definition along with any general assumptions defines the scope of a particular MASS framework. For instance, in this work, we will define the CDO to be the linear demixing operator given in (72) along with the assumptions that all SS algorithms can process data at the block level, and that the estimates produced at each block by any SS algorithm can reasonably be modeled as a time-invariant application of the CDO to the observation data block. Thus, we term the framework presented here as MASS under a block, linear, time-invariant (BLTI) filtering constraint, and we will go into more detail in Sect. 2.2

The general CDO approach to MASS also points out the difference between a Source Estimate Production (SEP) plugin and an SS algorithm in the current MASS framework. An SS algorithm simply estimates sources, whereas a SEP plugin is an SS algorithm in conjunction with a CDO transform, so that the source estimates can be expressed as a CDS applied to the observations. Once again, we cover this issue in more detail in in Sect. 2.2.

2.1.3 Algorithm-Level Processing Modes: Competition and Cooperation

In the MASS framework, the SS algorithms can individually be set to operate in either a *competitive* or *cooperative* mode. In competition mode, an algorithm will simply work independently to produce source estimates, whereas in cooperation mode, the algorithm will use demixing information provided through the CDS set to inform the algorithm's estimation process. That is, a competing method's only interaction with

MASS is in providing source estimates, while a cooperating method will collaborate (indirectly) with all the other SS methods via the CDS set.

The notion of competition is fairly straightforward, but the notion of cooperation is a bit more nuanced. Under the general MASS CDO strategy, cooperation implies that the CDO transform supplied by an SEP plugin be (partially) *invertible*. That is, the SEP plugin needs to convert the demixing information contained in the CSE CDS set into the algorithm-specific demixing solution format of the SEP plugin in order to cooperate. We note that this conversion is the inverse of the SEP plugin's CDO transform, i.e. the transform converts the algorithm-specific demixing solution into a CDS, and we will formalize this invertible transform requirement for cooperation in Sect. 2.2. Furthermore, cooperation entails its own strategies, in general, and we will briefly explore some ideas in Sect. 2.2.2.

Since the competition or cooperation processing decision is made at the algorithm-level, we note that MASS can be run in a mixed-mode state with respect to competition and cooperation. That is, when multiple SS algorithms run in parallel, a subset of the SS methods can compete, while the rest cooperate. Although all methods will provide demixing information that can be incorporated into the CDS, only cooperating methods will incorporate the CDS information into their learning process. Thus, with a finite SEP set, the MASS framework allows multiple configurations just by varying the mixed-mode processing. For instance, when MASS is used as a SS research and development tool, a new algorithm can be run in a competitive mode against established SS methods in order to gain insight into the new method's benefits, drawbacks, and deficiencies. On the other hand, when MASS is used as a SS platform to solve a particular problem, both cooperation and competition can be beneficial.

2.1.4 MASS-Level Processing Modes: Standard and Self-Competition

If MASS allows multiple SS algorithms to compete to produce a CSE set, and MASS can be considered as an SS approach in its own right, then a natural extension of MASS is to allow MASS to compete against itself. The standard MASS configuration simply allows multiple SS algorithms to collaborate, but the MASS framework easily allows exactly this MASS *self-competition*.

To clarify self-competition, consider a set of SS algorithms in MASS that determine a CDS, $\boldsymbol{D}(k)$, given the k^{th} block of data. At the next $(k+1)^{th}$ block of data, the SS algorithms compute their individual sets of source estimates, $\boldsymbol{D}(k)$ is applied to the $(k+1)^{th}$ data block to produce an additional set of source estimates, and *all* of the source estimates are evaluated by the MASS system to produce a CSE set with a corresponding CDS, $\boldsymbol{D}(k+1)$. This procedure is then repeated at each data block, so that the previous CDS always competes with the current solutions produced by the individual SS methods. Inherently, MASS self-competition provides memory to MASS, which has two applications in this work.

Fundamentally, a system that has memory can provide fast solutions to rapidly evolving problems. That is, given the SEP/SEE relationship in MASS, an SEP that "remembers" a past solution that is deemed appropriate by the SEE to the current observations can significantly decrease processing time and increase SS performance, especially in a parallel processing environment. Although we only include one past CDS as the competitor in this version of MASS, we can easily generalize to an entire "brain of memories" for future work. Our main purpose for self-competition is to study time-invariant or slowly time-varying environments, where a past solution could easily outperform the best guess of a learning method at a given data block. However, we also find that MASS self-competition is helpful for implementing and/or developing SS methods that have no inherent memory, e.g. non-adaptive methods, as we shall see in chapters 4 and 5.

2.1.5 Side-Information: Blind, Semi-Blind, and Supervised MASS

As we stated in the introduction, some source separation algorithms can use outside information to produce a solution, and depending on the type of outside information used, SS methods can generally be classified as supervised, semi-blind, or blind. Supervised methods use explicit knowledge of the scene, e.g. source signals, room impulse responses, etc., semi-blind methods use only general knowledge about a scene, e.g. statistical models, and *truly* blind methods have no knowledge of the scene. More generally, we note than *any* MASS plugin implementation's method, e.g. SEP, SEE, PAS, etc., could potentially need side information and could also be classified in terms of the method's "blindness". Not only does a general MASS framework allow for multiple

SS algorithms to run in parallel, the framework necessitates that multiple plugins (SEPs, SEE, PAS, etc.) work in conjunction, thus the terms "supervised", "semi-blind", and "blind" take on new meanings within the context of MASS.

To illustrate this issue, let a "configuration" denote a finite set of plugin implementations employed for a particular usage of MASS. Obviously, if all plugins' underlying methods are supervised or blind, we would designate the entire MASS configuration as supervised or blind, respectively. However, if in the supervised configuration, multiple methods use different sources of side-information, the MASS configuration is still supervised, but we need to recognize that MASS under multiple sources of side-information is different from a single SS method using all of these side-information sources.

That said, MASS can be configured in a number of semi-blind ways, and we now consider a simple MASS configuration that only consists of two SEP plugins (two SS methods) and a SEE plugin. If any one of the plugins is supervised, and any of the other plugins is blind or semi-blind, we could certainly consider the MASS configuration to be semi-blind. However, two blind SEP plugins with a supervised SEE plugin could also be considered semi-blind MASS, as could two supervised SEP plugins and a blind SEE plugin. Although all of these configurations could be put under the umbrella of semi-blind MASS, these configurations potentially provide drastically different results. This configuration-blindness issue points out another area of MASS versatility, and anyone delving into the MASS concept must understand how small choices in configurations can have large impacts on MASS results.

That said, a general framework for MASS should be capable of acquiring side-information for delivery to the constituent methods. In this version of the framework we only provide access to one type of side-information, the underlying source signals, so that we can support supervised methods. We leave the acquisition and dispersal of general side-information for future work.

2.2 Source Estimate Production: Issues and Assumptions

The source estimate production system (SEPs) is composed of some number, M, of source estimate production (SEP) plugins running in parallel, where each SEP plugin contains an SS algorithm that produces source estimates. In this work, the only

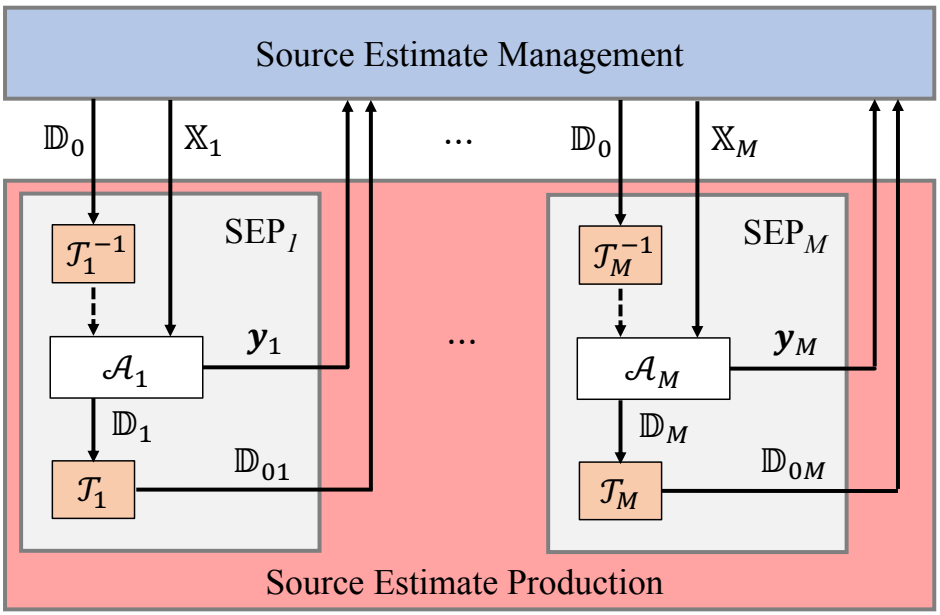

Figure 5. A general model for the Source Estimate Production system. Multiple Source Estimate Production (SEP) methods run in parallel. The m[th] method, SEG_m, takes in the CDS for the current CSE set, $\mathbb{D}_0$, and a set of signal data, $\mathbb{X}_\text{m}$ (observations, side-information, etc.), to produce a set of source estimates, $\mathbf{y}_\text{m}$, and a CDS, $\mathbb{D}_{0\text{m}}$, by transforming its demixing solution, $\mathbb{D}_\text{m}$. If SEG_m wishes to cooperate with MASS, SEG_m must transform $\mathbb{D}_0$ into its algorithm demixing solution format, $\mathbb{D}_\text{m}$, before processing.

restrictions placed on the underlying SS algorithms are that **they can process data *one block at a time*, and that they be able to convey *how they produced their source estimates* to the MASS system**. For the latter constraint, we will introduce the concept of an invertible demixing transform that will allow each algorithm to translate its solution into a common demixing form and vice versa. For the former, a block-processing constraint is required.

As we noted in Sect. 1.3.3, an SS algorithm, $\mathcal{A}$, produces source estimates, $\boldsymbol{y}_\mathcal{A}$, via the algorithm's demixing operation (ADO), $\mathcal{D}_\mathcal{A}$, on the mixtures, $\boldsymbol{x}$, where $\mathcal{D}_\mathcal{A}$ can be parameterized by the set $\mathbb{P}_\mathcal{A}$, i.e. Eq. (67) reproduced here as,

$$\boldsymbol{y}_\mathcal{A}(n) = \mathcal{D}_\mathcal{A}\{\boldsymbol{x}(n); \mathbb{P}_\mathcal{A}(n)\}.$$

The set $\mathbb{D}_\mathcal{A} = \{\mathcal{D}_\mathcal{A}, \mathbb{P}_\mathcal{A}\}$ is termed an *algorithm demixing solution* (ADS), since when a set of mixtures are provided, this set provides the method to produce source estimates for a particular algorithm.

As we mentioned in Sect. 2.1.2, we will require that each SEP plugin be able to transform its underlying ADS into a format that is common across the MASS framework. That is, the MASS framework will define a common demixing operator (CDO), $\mathcal{D}_0$, and the associated parameter set, $\mathbb{P}_0$, so that a common demixing solution (CDS) is given as the set $\mathbb{D}_0 = \{\mathcal{D}_0, \mathbb{P}_0\}$. Thus, in this work, we require the transformation of *any* ADS into this CDS format.

As shown in Figure 5, each SEP plugin has two fundamental parts; an algorithm, $\mathcal{A}$, that produces source estimates, and a transform, $\mathcal{T}$, to translate the algorithm's ADS, $\mathbb{D}_{\mathcal{A}}$, to a CDS format, $\mathbb{D}_0$. Stating this explicitly for each algorithm, $\mathcal{A}_m$, running in the SEP system, $\mathcal{A}_m$ estimates a set of sources as,

$$\boldsymbol{y}_m(n) = \mathcal{D}_m\{\boldsymbol{x}(n); \mathbb{P}_m(n)\}, \qquad m = 1, \dots, M, \tag{81}$$

where $\mathcal{D}_m$ is the demixing operation of the m^{th} algorithm, $\boldsymbol{y}_m(n) = [y_{1m}(n) \quad \cdots \quad y_{Qm}(n)]^T$, $y_{qm}(n)$ is the q^{th} source estimate of the m^{th} algorithm at time-index n, and $\mathbb{P}_m(n)$ is a set of parameter values associated with the m^{th} algorithm's demixing operation. Each algorithm's demixing solution transform, $\mathcal{T}_m$, should satisfy the equality,

$$\mathbb{D}_{0m} = \mathcal{T}_m\{\mathbb{D}_m\}, \qquad m = 1, \dots, M, \tag{82}$$

where $\mathbb{D}_{0m}$ is the m^{th} algorithm's demixing solution translated into a CDS form. Furthermore, if the algorithm wishes to use the information contained in the current MASS CDS, we require that the algorithm's demixing solution transform be invertible so that,

$$\mathbb{D}_m = \mathcal{T}_m^{-1}\{\mathbb{D}_0\}, \qquad m = 1, \dots, M, \tag{83}$$

where $\mathbb{D}_0$ is the CDS used to produce the most current CSE set.

2.2.1 A Block-Level, Linear, Time-Invariant Common Demixing Solution

In this work, we will assume that the source estimates, $\boldsymbol{y}_m(n)$, produced from an algorithm, $\mathcal{A}_m$, over some finite time-interval can reasonably be modeled as a time-invariant filtering on the observation data, $\boldsymbol{x}(n)$, and the CDO is the linear demixing operation given in (72), i.e. $\mathcal{D}_0 = \mathcal{D}_l$. Under these two assumptions, we can use simple linear estimation techniques to provide an exemplary, linear CDS transform.

We make the above two assumptions, because any algorithm which meets these conditions can provide information to the MASS system via the system identification operator defined in Section 1.2.3. That is, we can approximate the demixing operation as a linear demixing matrix, i.e. $\mathcal{D}_m(n) \cong \boldsymbol{D}_m^T \boldsymbol{x}(n)$, where $\boldsymbol{D}_m$ has the same structure as $\boldsymbol{D}_l$. If we let $y_{qm}(n)$ be the q^{th} output of $\mathcal{D}_m(n)$, then we can estimate the q^{th} column of $\boldsymbol{D}_m$ as,

$$\widehat{\boldsymbol{d}}_{qm}(n) = Id\{\boldsymbol{x}(n), y_{qm}(n);\ L_d\}, \qquad q = 1, \dots, Q, \tag{84}$$

where $Id\{\cdot\}$ is the system identification operator defined in (31), and L_d is a filter length. From this we can construct the estimated linear demixing matrix as,

$$\widehat{\boldsymbol{D}}_m(n) = \left[\widehat{\boldsymbol{d}}_{1m}(n), \widehat{\boldsymbol{d}}_{2m}(n), \dots, \widehat{\boldsymbol{d}}_{Qm}(n)\right]. \tag{85}$$

Thus, the system identification operator is a valid transform for any SEP plugin whose outputs satisfy the block-level, linear, time-invariant (BLTI) filtering assumption.

Furthermore, for algorithms whose ADS format is equivalent to or compatible with the linear CDS, the system identification operator constitutes an *invertible* transform that can be used when the SEP plugin runs in cooperative mode. As we pointed out in Sect. 2.1.3, a SEP plugin can operate in a cooperative manner by transforming the past block's CDS into the SEP plugin's ADS and using the solution to inform the SEP plugin's source estimation for the current block of data. Given that the CDS conforms to the BLTI assumption, and the SEP plugin's ADS is equivalent to the CDS, we can simply use the $Id\{\cdot\}$ operator to estimate the CSE set's CDS set.

2.2.2 A Note on Cooperation Strategies

The MASS system works in a block-sequential manner, so that a block of data is input into the system and the various SEP components produce solutions which are then evaluated to produce a CSE set. When the MASS system pulls in the next consecutive block of data, an algorithm that is working in *cooperation* with the MASS system needs to incorporate the MASS CDS into its own methodology. In this section, we will briefly give some simple strategies of how to incorporate the CDS into a SEP component's particular solution. The set of ideas presented here is not exhaustive and is simply meant to bring awareness of the issue, and SEP designers are encouraged to use and develop cooperation methods that are the most suited to their problems.

In the simplest case, a SEP component will simply transform the CDS from the previous block of data as an initialization to its processing of the next data block. This constitutes a memoryless system that assumes that the most recent CDS is the best demixing solution. The uses for this type of system includes SEPs with slow transient phase learning, a MASS self-competitive solution, etc.

The next case of incorporation allows the SEP component to compete for a while before using the MASS CDS solution, and this constitutes an *interval* memoryless solution. In this case, the SEP waits some N number of blocks of data before substituting the MASS CDS for its initialization to process the next block of data. Thus, the SEP component competes for N blocks of data before using the community solution. This approach is generally useful for adaptive systems, so that the method an individual SEP component uses has time to make its best determination before being reeled in by the common consensus.

The SEP component can also choose to use a time-averaging or forgetting factor into the above two methods which would constitute memory (competition) in cooperation. This is a very interesting case, since by averaging (linear or non-linear) past solutions for the current initialization, the SEP component is competing with the MASS system, even while the SEP component is cooperating. To see this, we can simply trace a couple of iterations of a SEP component that acts in this manner. That is, a SEP component produces a demixing solution, the MASS system provides a CDS, the SEP component averages the MASS solution and the SEP component's previous solution, and the initialization to the SEP component at the current data block is neither the SEP component's nor the MASS's CDS, thus the averaged solution is a competitor, even if for only a block or an interval. Thus, this methodology constitutes *interval competitive cooperation*. In the author's studies, multiple adaptive filters, i.e. MSE cost, operating in interval competitive cooperation mode perform better in MSE than purely competitive adaptive filters.

2.3 Source Estimate Evaluation: Issues and Strategies

The main task of the source estimate evaluation (SEE) plugin is to produce a CSE set, and this section provides an overview of the concept, including some of the pitfalls associated with the subject and some general strategies for approaching the problem.

SEE methods can generally be decomposed into two parts, where the first part deals with how source estimates are combined and the second deals with the number of source estimates being evaluated. Both of these topics are complicated, but here we will give them fairly simple treatments in order to clearly convey how they pertain to MASS. Thus, we boil the first issue down to a question of *blending* or *selecting*, and we simplify the second item to a matter of *joint* evaluation of sets of source estimates versus *individually* evaluating each estimate. To delve into the SEE subject, we will begin in the next subsection by delineating three subjects pertinent to adaptive SS and MASS; separation criteria, learning rules, and of course, source estimate evaluation. We will end this section by exploring general approaches to the SEE problem.

2.3.1 Separation Criteria, Optimization, and Source Estimate Evaluation

The SS problem admits many different approaches, e.g. see [92, Ch. 2,4,7,10,12,13], and at a fundamental level, all of these approaches make particular assumptions about what constitutes a source and how that source exists in its environment. The assumptions about how one source (or group of sources) can be distinguished from another source (or group of sources) lead to the formation of separation criteria which, in learning methods, are manifested in a cost (contrast) function for BSS. Learning techniques search for a demixing solution that operates near the desired, or optimal, values of the criteria by minimizing the cost (maximizing the contrast). In this context, *we model an algorithm as a cost function in conjunction with an optimization technique*. In order to determine how well an algorithm performs at some point in time, the criteria need to be evaluated under the current solution, e.g. the cost function is evaluated using the current solution. In this section, we will point out some of the difficulties in blind performance evaluation for source separation.

We begin with a simple example using mutual information (MI) as a cost function. For two random variables, $\boldsymbol{x} = [x_1 \quad x_2]^T$, the amount of information shared between the two is given by (21), and recreated here as,

$$\mathcal{J}_{MI}(\boldsymbol{x}) = \mathcal{H}_1 + \mathcal{H}_2 - \mathcal{H}_{\boldsymbol{x}} \tag{86}$$

where $\mathcal{H}_1$ and $\mathcal{H}_2$ are the marginal entropies of x_1 and x_2, respectively, and $\mathcal{H}_{\boldsymbol{x}}$ is the joint entropy of $\boldsymbol{x}$. When x_1 and x_2 are statistically independent $\mathcal{J}_{MI}(\boldsymbol{x}) = 0$, and when x_1 and x_2 are completely dependent $\mathcal{J}_{MI}(\boldsymbol{x}) = \min\{\mathcal{H}_1, \mathcal{H}_2\}$. Now let $x_1(\alpha) = s_1 +$

αs_2 and $x_2(\beta) = s_2 + \beta s_1$, so that $\boldsymbol{x}(\alpha, \beta) \triangleq [x_1(\alpha) \quad x_2(\beta)]^T$, where s_1 and s_2 are statistically independent RVs, and α and β are scalars. If we let $\alpha = \beta = 0$, then $\mathcal{J}_{MI}(\boldsymbol{x}(0,0)) = 0$, since no information is shared between two statistically independent RVs, and $\mathcal{J}_{MI}(\boldsymbol{x})$ constitutes a proper cost function for source separation; it is nonnegative, and it is only zero when $\hat{\boldsymbol{s}} = \{s_1, s_2\}$ and order is inconsequential. Thus, under the separation criteria that the sources are mutually statistically independent, MI provides a cost function whose *optimization* satisfies the separation criteria. Evaluating MI for source estimate evaluation is not as straightforward, and in fact, evaluating MI for mixtures can be ambiguous and misleading, as we will now show.

First off, we initially observed that if x_1 and x_2 are completely dependent then $\mathcal{J}_{MI}(\boldsymbol{x}) = \min\{\mathcal{H}_1, \mathcal{H}_2\}$, thus the upper bound on the MI measure is dependent on the random variables (RVs) at hand. Although we can provide a general upper bound on MI as the entropy of a Gaussian RV with the same variance as $\boldsymbol{x}$, the upper bound is not fixed, so the range of values of MI can change due to the nature of the problem, e.g. nonstationary sources, adaptive separation, etc.

Next, we look at the two cases of $\hat{\boldsymbol{s}}_1 = \boldsymbol{x}(0,1)$ and $\hat{\boldsymbol{s}}_2 = \boldsymbol{x}(1,0)$. Clearly, $\mathcal{J}_{MI}(\hat{\boldsymbol{s}}_1) > 0$ and $\mathcal{J}_{MI}(\hat{\boldsymbol{s}}_2) > 0$, thus MI is not a good measure of the performance for an *individual* source estimate, since in both estimates a source has been completely isolated. Mutual information is a measure of the *joint* performance of a set of source estimates. However, if we now let $\hat{\boldsymbol{y}}_1 = \boldsymbol{x}(\alpha, \beta)$ and $\hat{\boldsymbol{y}}_2 = \boldsymbol{x}(\beta, \alpha)$ and consider the case where s_1 and s_2 are identically distributed, we see that $\mathcal{J}_{MI}(\hat{\boldsymbol{y}}_1) = \mathcal{J}_{MI}(\hat{\boldsymbol{y}}_2)$ for any choice of α and β, and MI has no "preference" for either source estimate set. Furthermore, $\mathcal{J}_{MI}(\hat{\boldsymbol{s}}_1) \neq \mathcal{J}_{MI}(\hat{\boldsymbol{s}}_2)$, generally, thus MI has a bias when the sources are not identically distributed (effectively the basis for using negentropy). The above examples are provided to highlight the potential difficulties in using a valid BSS cost function to evaluate and compare the performance of a set of source estimates.

A major dilemma in the SEE problem, indeed in the MASS problem, is how to evaluate a set of source estimates when the underlying SS methods producing the estimates differ in their respective SS criteria. For example, in Chapter 4 we will define three BSS methods that use drastically different criteria for SS: mutual statistically independent sources, scene geometry, and source time-structure along with scene geometry, respectively. Although these three methods provide valid criteria for

separating sources, the method for SEE is very unclear in this case. In fact, SEE is an open question in the general MASS problem, and the MASS framework has been designed to study this problem. One potential approach is provided by Cardoso [62] who shows that combinations of valid SS cost functions also produces a valid SS cost function, and one can imagine multiple SEE plugins working in tandem to oversee multiple SS methods with disparate SS criteria. Furthermore, the multiple algorithm/model approaches in the adaptive prediction domain, e.g. see [3], show a hierarchy that could potentially be extended to the MASS problem where multiple MASS systems run in parallel and oversee particular classes of algorithms. For now, we simply note that this is an ongoing, open research topic.

2.3.2 MASS SEE and Extant Work in Other Domains

The main task of the SEE plugin is to construct a set of composite source estimates (CSEs). Conceptually, the SEE plugin uses a cost function, $\mathcal{J}$, to evaluate source estimates, a combination function, $\mathcal{C}$, that creates the CSEs from the underlying source estimates, and creates the common demixing solution (CDS), i.e. the demixing solution which when applied to the mixtures produces the CSEs. In the previous section, we gave an overview of issues involved with $\mathcal{J}$, but here we are only concerned with whether $\mathcal{J}$ evaluates multiple source estimates *jointly*, or if $\mathcal{J}$ evaluates each source estimate *individually*, and we shall see that even this distinction can be blurred. The combination function, $\mathcal{C}$, either involves a *blending* of the source estimates, so that CSEs are a combination of the source estimates, or *selecting* individual source estimates, so that the CSEs are produced by picking from the available source estimates. The approach to create a set of CSEs depends on the combination of joint-individual evaluation method and the blended-selected combination method. The strategies that we explore in the next section are inspired by the multiple algorithm/model approaches in the prediction, e.g. [3]–[9], adaptive filtering, e.g. [10]–[19], adaptive beamforming, e.g. [20]–[22], and sequential compound decision, e.g. [23]–[28], problems. Before we iterate through the various evaluation-combination pairs to give examples of simple SEE approaches, we will now compare the multiple algorithm work in other domains to the MASS SEE problem.

MASS shares the fundamental multiple algorithm approach of the work in other domains, but generally this is the only similarity, where the main differences stem from the fact that SS is inherently a multi-estimate problem, SS is most useful in a *blind* context, and convolutive SS involves a *filtering ambiguity*. The work in multiple algorithm adaptive prediction (AP), adaptive filtering (AF), and sequential compound decisions (SCD) typically involve multiple methods running in parallel to estimate a *single* signal, thus the general SS problem is quite different. That said, if we consider multiple estimates of a single source as our competing estimate set, then we can start to use general strategies from these domains. Adaptive beamforming (AB) certainly allows for multiple-estimate solutions, but generally beamformers require knowledge of sensor array geometry, which leads to the next issue of *blindness*.

Although SS can be run in a supervised mode, we must always account for the *blind* SS (BSS) task in this work. Although the underlying quantity to be estimated in the AP, AF, and AB problems is unknown, these domains require *a priori* knowledge of the problem. AB methods generally have knowledge of the sensor array geometry, and AP and AF methods have explicit knowledge of an underlying signal that drives the estimation process.

Speaking generally, the AP, AF, AB, and SCD problems involve an instantaneous weighting of the quantities being estimated, whereas the convolutive BSS problem involves a filtering ambiguity that complicates blending procedures. That is, two valid estimates of a single source can differ by two drastically different transfer functions applied to the source, thus a simple instantaneous combination of the estimates is generally not applicable. That said, this is not an insurmountable problem, since the blending problem just involves a higher-dimension solution space that can be solved either by brute force or by a transformation of the source estimates to a common, overall filtering, easily achieved via imaging.

Given the issues involved with using multiple algorithm methods' solutions for the MASS SEE problem, we note that the extant work in other domains drives our general strategies for the SEE problem. The general notion of selection is inspired by the switching methods in the literature, e.g. [8], [28], and the blending-based methods are inspired by the work in the AP, AF, and AB domains. Many of the works in the AP, AF, AB, and SCD domains involve the combination of individually evaluated estimates, thus

the notion of individual evaluation is inspired by these problems along with the source extraction problem. Joint estimation is most closely tied to the BSS problem, but our inspiration for joint evaluation comes predominantly from the AF domain, where [11] is an invaluable source on this topic. That said, the joint evaluation topic has different meanings when we are evaluating a group of estimates for one source, as in AF, or when we are evaluating a set of estimates that contain multiple estimates of multiple sources, as in MASS.

2.3.3 Simple Approaches for Composite Demixing Solutions

Having briefly explored some general issues involved with SEE in the previous section, we now outline some general strategies for developing SEE methods. The approaches presented here are not meant to be exhaustive, and we present them simply to broach the approach-specific issues that SEE developers can encounter. For reference, we give an example of a (supervised) individually evaluated selection method in Sect. 4.10 and a (blind) jointly-evaluated selection method in Sect. 4.11.

2.3.3.1 Simple Joint-Selection (Algorithm Selection)

The simplest approach belongs to the jointly evaluated, and selected class of CSE production. Consider M algorithms running in parallel, so that the set of source estimates from the m^{th} algorithm is denoted as $\boldsymbol{y}_m$. We can simply evaluate $\mathcal{J}(\boldsymbol{y}_m)$ for $m = 1, \ldots, M$ and select the outputs that minimize the cost, i.e.

$$\boldsymbol{z} = \arg\min_{\boldsymbol{y}_m} \mathcal{J}(\boldsymbol{y}_m) \tag{87}$$

where $\boldsymbol{z}$ is the CSEs. This particular strategy, which we denote as *algorithm selection*, has a limited scope and is typically used when all the underlying algorithms are all estimating the same number of sources and all the algorithms either belong to the same class (cost function) or are working in cooperation. To see this limitation, we first note that we make no assumption about the number of sources that an individual algorithm estimates, so that the dimension of $\boldsymbol{y}_m$ may be dependent on m. Therefore, if an algorithm which estimates a small number of sources has a slightly better cost than an algorithm that estimates a large number of sources, at best the cost evaluation is ambiguous and at worst the cost evaluation is biased against algorithms that "do more".

For example, consider an algorithm that mildly separates two sources, and an algorithm that perfectly separates four sources but includes a fifth source estimate that is a highly mixed combination of the sources. If the cost of the former algorithm is better than the latter, we have missed an opportunity to separate the sources. One way to alleviate this problem is to correct for this bias, but we will leave that for future work. We should also note that if the cost function explicitly requires a joint (comparison) evaluation, then algorithms which only produce one source estimate will be excluded, and therefore, will never contribute to the CSEs.

We now consider the case where all algorithms are estimating the same number (>1) of sources. Although we have alleviated the bias discussed above, the cost function evaluation might be ambiguous or misleading as we discussed in Sect. 2.3.1. For instance, consider two algorithms, i.e. algorithm 1 and algorithm 2, that are trying to separate two sources, i.e. source 1 and source 2. We can imagine a scenario in which algorithm 1 does an excellent job in separating source 1, but does a poor job in isolating source 2, and conversely, algorithm 2 separates source 2 very well, but performs poorly for source 1. The simple joint-selection method would only allow the SEE to select one algorithm's outputs, thus as above, we have missed an opportunity to create a set of CSEs that outperforms any of the underlying algorithms. In the next section, we will use a simple extension of algorithm selection to alleviate these problems.

2.3.3.2 Joint-Selection

In order to alleviate the problems associated with the algorithm selection approach, we can take a slightly more sophisticated approach to the joint-selection problem. Instead of considering the source estimates from the various algorithms as distinct groups, we will consider them collectively, or more precisely, we will consider multiple joint evaluations of all the combinations of the algorithms' outputs. Let the vector of all source estimates be denoted as as $\boldsymbol{y} = Vec\{\boldsymbol{y}_1, \boldsymbol{y}_2, \dots, \boldsymbol{y}_M\}$, where $\boldsymbol{y}_m$ are the outputs of the m^{th} algorithm and let $\bar{y}_i = [\boldsymbol{y}]_i$ be the i^{th} entry in $\boldsymbol{y}$. If we denote the set of indices in $\boldsymbol{y}$ as $\mathbb{I} = \{1, \dots, N\}$, where N is the number of elements of $\boldsymbol{y}$, then we will denote the set of all k-element combinations in $\mathbb{I}$ as $\mathbb{I}_k = \left\{\mathbb{I}_{kj} : \mathbb{I}_{kj} \subset \mathbb{I}, \left|\mathbb{I}_{kj}\right| = k, j = 1, \dots, N_k\right\}$, where $N_k = \binom{N}{k}$. Letting $\mathbb{Y}_{kj} = \left\{\bar{y}_i : i \in \mathbb{I}_{kj}\right\}$ be the set of source estimates

associated with $\mathbb{I}_{kj}$, so that $\overline{\boldsymbol{y}}_{kj} = Vec\{\mathbb{Y}_{kj}\}$, then we will select the set of source estimates that minimizes the cost, or

$$\boldsymbol{z} = \arg\min_{\overline{\boldsymbol{y}}_{kj}} \mathcal{J}(\overline{\boldsymbol{y}}_{kj}) . \tag{88}$$

At some point, evaluating $\mathcal{J}(\overline{\boldsymbol{y}}_{kj})$ for all values of k and j can become prohibitive. One way to reduce the number of evaluations is to limit the range of values that $k \in \{k_{min}, ..., k_{max}\}$ can take on, i.e. define a range of the number of source estimates produced. At a minimum, we need to estimate $k_{min} = 2$ sources to use a joint method, so this becomes the lower value for k, unless outside knowledge of the problem exists. For an upper value of k, N is the maximum value possible, but if we assume that the mixtures are critically-determined, then we can set the maximum value to $k_{max} = P$, where P is the number of mixtures. Unless *a priori* knowledge exists about the number of sources present in the observations, then at some level, limiting the range of values for k is arbitrary. That said, evaluating the cost over all possible values of k, and looking at the maximum value that produces a CSE set, constitutes a source enumeration method. In fact, as we noted in Sect. 2.1, we provide a source enumeration (SNUM) plugin to explicitly estimate the number of sources. That said, we can still evaluate the cost over the entire range of k, as a double-check of the SNUM plugin's output.

Another way to reduce the number of cost evaluations performed for the joint-selection approach, utilizes sorting. If we sort all of the algorithms' outputs into groups, so that each group only contains estimates of a particular source, then the number of combinations is reduced along two dimensions of the problem; the number of sources and the pool of allowable combinations. First off, the number of groups is an estimate of the number of active sources, therefore the value of k is fixed. Second, instead of the need to evaluate over the N_k index combinations, we only need to evaluate the combinations in which the k indices come from distinct groups. Clearly, the reduction in the search space comes with the computational overhead of the sorting task, but as mentioned before (Sect. 2.1), the sorting task is provided in the source estimate grouping (SEG) plugin, so that sorting becomes independent of the evaluation method.

2.3.3.3 Pairwise Joint-Selection

Although the general joint-selection approach above outlines a reasonable solution to the problem, we will now briefly give an alternative joint-selection approach that approximates each joint cost function evaluation as an average of the joint evaluations of each pair of source estimates in a k-element set. Let us first denote the pairwise cost function evaluation as $\mathcal{J}^{(2)}$, so that the evaluation of an N-element vector of source estimates, $\boldsymbol{y}$, produces an NxN matrix, $\boldsymbol{J} = \mathcal{J}^{(2)}(\boldsymbol{y})$, whose $(i,j)^{th}$ element is given by $[\boldsymbol{J}]_{ij} = \mathcal{J}([\bar{y}_i\ \bar{y}_j])$. If we now average the pairwise cost evaluations in the $\mathbb{Y}_{kj}$ set then we will approximate the joint cost evaluation of the entire $\mathbb{Y}_{kj}$ set as the average pairwise cost. The benefit of this approach is the potential for computational savings; each pairwise (2x2) cost evaluation is performed once for all $\sim N^2$ combinations, whereas the joint evaluation (kxk) must be computed for all N_k combinations. If we assume the cost function is asymmetric, i.e. $[\boldsymbol{J}]_{ij} \neq [\boldsymbol{J}]_{ji}$, then the number of pairwise cost function evaluations is $N^{(2)} = N^2 - N$, since we are unconcerned with the diagonal elements of $\boldsymbol{J}$. If the cost function is symmetric, i.e. $[\boldsymbol{J}]_{ij} = [\boldsymbol{J}]_{ji}$, then the number of pairwise cost function evaluations is $N^{(2)} = \frac{N^2-N}{2}$. Thus, when $N^{(2)} < N_k$, the pairwise approach will reduce the computation outright, even if we do not consider the potential added computation involved with a (2x2) versus a (kxk) cost evaluation. Furthermore, the pairwise method can be improved by using a sorting method, just like the joint-selection method.

2.3.3.4 Joint-Blending

The joint-blending approach can take on many forms, but here, we will briefly give an example that is the simplest to describe, but potentially the most difficult to evaluate, generally; an over-determined BSS algorithm. That is, consider a BSS algorithm, $\mathcal{A}_B$, that has the following qualities; $\mathcal{A}_B$ can handle over-determined data, $\mathcal{A}_B$ can work in batch mode, and $\mathcal{A}_B$ has an invertible CDO transform. Thus, if we specify the number of sources to estimate,

$$\boldsymbol{z} = \mathcal{A}_B(\boldsymbol{y}). \tag{89}$$

If $\boldsymbol{y}$ are the estimates from all algorithms during a block, we necessitate that $\mathcal{A}_B$ handle over-determined data, because, presumably, multiple algorithms will estimate the same

source, and we necessitate that $\mathcal{A}_B$ work in batch mode, since we only present $\mathcal{A}_B$ with solutions that are valid for the current data block. The reason for inversion of $\mathcal{A}_B$'s solution is exactly the same as any SEP; conversion to a CDS. All that said, this method relies upon a BSS method to take as input, not the observation mixtures, but source estimates to come up with a combination that improves upon the estimates in the current set. Generally speaking, BSS methods that are capable of self-organizing, and allow for quick adaptation are ideal candidates for this method.

2.3.3.5 Individual-Selection

We now begin to look at cost functions that evaluate individual source estimates. For the *selection* problem under individual evaluation, blind source extraction (BSE) cost functions with a sorting preprocessor are sufficient. In fact, sorting is a necessary component for individual source estimate evaluation, generally. To see this, we consider the negentropy cost function, $\mathcal{J}_{NE}$, described in Sect. 1.2.2. Although $\mathcal{J}_{NE}$ is a proper BSE cost function, it is not a proper BSS cost function without some comparison with other source estimates, e.g. coherence. That is, BSE cost functions are only concerned with optimizing characteristics of a single source, independently of any other source. Thus, if we wish to use a BSE cost function from a BSS evaluation perspective, we need to isolate our individual performance measure from other measures within the BSS task. As pointed out in Sect. 1.3.5, one strategy in sequential BSE is to remove a source estimate from the observations and move on to the next source estimate. For performance evaluation of multiple source estimates (potentially multiple estimates of the same source), this strategy is overly complicated, and produces a combinatorial problem in the paths of deflation.

Once again, a simple solution is to sort the source estimates, so that we can compare the individual estimates within each group estimating one source. If we let $\mathbb{Y}_q = \{\bar{y}_i : i \in \mathbb{I}, \bar{y}_i \approx s_q\}$ be the set of source estimates that are approximating the q^{th} source for $q = 1, \dots, \hat{Q}$, where $\hat{Q}$ is an estimate of the number of sources, then we can select a best estimate for the q^{th} source as,

$$z_q = \underset{y \in \mathbb{Y}_q}{\arg\min}\, \mathcal{J}(y), \qquad q = 1, \dots, \hat{Q}, \tag{90}$$

so that the CSE is constructed as,

$$\boldsymbol{z} = Vec\{z_1, z_2, \dots, z_{\hat{Q}}\}. \tag{91}$$

2.3.3.6 Individual-Blending

Blending under the individual cost assessment is, perhaps, the most complicated problem for the SEE system, even when sorting is employed. In fact, the problem without sorting is highly overdetermined to the point that a satisfactory solution does not exist at present, and we leave this endeavor for future work. That said, the individual-blending method has the most potential for rewarding research, since it aims to use grouped estimates of a single source to improve the estimate of that source; a benefit that joint-blending, joint-selection and individual-selection methods cannot claim.

The individual-blending evaluation method removes most BSE cost functions from contention since a BSE cost function optimization will always try to isolate the *single* source with the optimal characteristics. That is, if we consider that an algorithm is a cost function in conjunction with an optimization technique, and blending is a method to combine source estimates to reduce the cost, blending for individual evaluation of a BSE cost function will result in estimation of a *single* (or small subset of) source(s), regardless of the pool (sorted groups) used to optimize the estimate. Negentropy is a good example of this concept, since as long as every source estimate has some miniscule contamination, i.e. including the least/most negentropic source, blending will result in isolating the extreme negentropic source. Of course, we can always constrain the blending problem in some way, but in general, constrained solutions incur slow progress or suboptimal results, and given that the problem we deal with here is, generally, overdetermined (more than one estimate of source q) the problem is intensified.

CHAPTER 3: A SOFTWARE FRAMEWORK FOR MASS

Software frameworks are devoted to a domain-level task, and as such, well designed frameworks involve collaboration between experts in the domain and experts in software engineering. The last chapter covered the issues associated with the MASS domain, and in this chapter, we will use this domain-level knowledge to construct a MASS framework in software. As we shall see, the MASS problem when decomposed into its elementary tasks, as defined in Sect. 2.1, presents an ideal candidate for implementation as a software framework. Thus, this chapter will fully define a software framework for MASS, beginning with an overview of software frameworks, generally.

3.1 An Overview of Software Framework Architecture

From a fundamental standpoint, a software framework is one of a number of approaches to solve the *reusability* problem in the software engineering field. The reusability problem is essentially a problem of software development efficiency, where the goal in developing a new application is to reuse as much relevant past work as possible. That is, for any new application that uses a "wheel", the software developer should not need to create, or reinvent, the wheel for each application. Although the definition of a framework varies in the software engineering literature, the definitions are generally not exclusive of one another, and the varied definitions simply point out the complexity in approaching the reusability subject [90, p. 3]. This work combines a few framework definitions from the literature so as to cover as many framework topics as possible.

We define a framework as *a reusability strategy that deconstructs a domain-specific system into a number of concrete and abstract functionality classes plus a definition of the way instances of these classes interrelate*. Let's quickly inspect this definition. First, a framework is a code *reuse* strategy, and once the framework has been sufficiently developed, developers can use the framework efficiently to build applications. Second, a framework is focused on solving a problem or set of problems within a *specific domain*, and typical framework development involves software engineers working with domain experts to inform the development. Third, a framework defines a *decomposition* of a domain-level system into a set of sub-systems that can

operate independently to perform the overall domain-level task. Fourth, the decomposition of the system defines a set of *abstract*, user-extensible, functionalities and a set of immutable, non-user-extensible, functionalities, both of which are realized as *concrete* implementations of their respective functionalities. Finally, a framework defines how all the pieces of the system *interrelate* by defining the immutable workflow and interfaces that users must comply with in order to utilize the framework. [90, pp. 21–22], [91, p. 403], [93]

Software frameworks occupy the middle ground between two well-known reusability concepts: *components* and *design patterns*. A component is an implemented solution to a particular low-level problem, and an application developer does not need to know how a particular component performs its task and can simply connect various components together to create an application. A design pattern is similar to a component in that it focuses on solving a particular problem, however a design pattern provides a *strategy* to solve the problem, and the solution is abstract. Applications can be built in abstraction by connecting various design patterns, but to implement the application, a developer needs to fully understand each design pattern to create the specific functionality. Thus, both components and design patterns provide a level of modularity, however components act as "black boxes" where a developer needs to know nothing of the implementation of a particular component, and most design patterns act as "white boxes" where a developer needs to know everything about the underlying strategy to create a concrete application. [91, pp. 197-204,413], [93]

Beyond some unique characteristics of frameworks that we will discuss shortly, a framework can be seen as a combination of the component and design pattern concepts. On one side, a *white-box* framework defines each part of the user-extensible functionality in terms of an interface, an abstract definition of the functionality, and concrete examples of the functionality as *reference components*. In a *black-box* framework, a component library is supplied, and an application developer can simply plug in various components to the framework to create her specific application. Given that a framework is focused on solving a particular (domain-level) problem, the white-box framework generally resembles a pattern design, and the black-box framework resembles a component from a high-level perspective. For example, consider a framework for sandwich-making. A white-box framework provides abstract plugin definitions for *how to describe* the bread,

the condiments, and the filling, while a black-box framework would come with plugins that *describe* various breads, condiments, and fillings. Thus, the white-box framework provides the essence of the framework that a developer can use to create customized plugins at the cost of needing to fully understand the abstract plugin definitions, and the black-box framework provides precompiled plugins ready to use but is limited by the versatility of the supplied component library. [90, p. 17] [93]

In practice, frameworks are typically developed in an iterative manner, where at each iteration the framework is factored into the parts that did not change and the parts that did. The parts that do not change are classified as "frozen spots" in the framework, since they represent sections that are not user-extensible, e.g. tools, utility functions, program execution, etc. The parts that do change are denoted as "hot spots", since the functionality of these parts is either folded into existing abstract plugin definitions or is used to create new abstract plugin definitions that users can extend. Thus, the abstract definitions contained in the white-box framework fully define all of the hot spots in the framework, and a framework is "well-designed if it provides adequate hot spots for adaptations." [90, p. 379]

There are several benefits to using frameworks, and here we will confine our attention to four such benefits: reusability, extensibility, modularity, and inversion of control. As we have already stated, frameworks are a reusability strategy, and clearly both white-box and black-box frameworks exhibit this property. For instance, consider the black-box sandwich-making framework that contains plugins defining the condiments and fillings for a turkey sandwich as well as the plugins to define white and wheat bread. The application developer can use the framework to create a turkey sandwich on either white or wheat bread simply by plugging in the appropriate bread plugin without developing new code.

That said, extensibility provides the framework the power of versatility. In the white-box framework, the developer can extend any of the plugins to provide custom implemented functionality. That is, if the developer would like to create a turkey sandwich on pumpernickel, then she would need to extend the abstract definition of the bread plugin to define pumpernickel, and then plug pumpernickel into the framework. We should recognize this as a "grey-box" framework, since we are using the abstract bread plugin definition from the white-box framework to create a new bread component,

while re-using the condiment and filling plugin components for a turkey sandwich from the black-box framework.

We should also note that this independent plugin extension is made possible because of the modularity of the plugins. From a low-level perspective, we can see that the plugin definitions in the white-box framework act as design patterns and the plugins in the black-box framework act as components. Well-designed design patterns and components exhibit modularity, such that a particular design pattern's or a component class's functionality does not overlap with functionality of other methods, a concept known as separation of concerns (SoC). Not only does SoC reduce redundancy in implementing the same functionality, SoC allows a particular plugin to be developed independently without affecting the behavior of other plugins.

One of the main distinguishing characteristics of a framework from other application development concepts is the notion of inversion of control (IoC) [90, p. 5]. To create an application without a framework, a developer typically calls the specific functionality they need from a library, and the developer dictates the flow of execution of the program. Frameworks, on the other hand, control the flow of execution in the application, and the developer only specifies the particular plugins used in the application. Continuing with our sandwich-making framework, a developer simply specifies the plugins for a particular bread, particular condiments, and particular fillings, and the framework is responsible for actually creating the sandwich. IoC allows the developer to focus on the particular needs of the application without worrying about how to assemble the final program. We note that IoC is made possible by the well-defined structure and interaction of the frozen and hot spots in the framework.

Frameworks have three major drawbacks: design and development, documentation, and first-time usage. For the first drawback, framework design is a difficult task, given that a framework should handle any problem within the framework's problem domain. For a framework to be successful, the designers must foresee every problem in the domain that can be addressed by the framework and design accordingly. Even if the designers are experts in the problem domain, designing a system that fully encompasses and properly modularizes the problem from the very beginning of development is a daunting, if not impossible, objective. Thus, like any software system, a framework evolves over time, but the initial development strives to be comprehensive

enough to accommodate all foreseeable scenarios in which the framework can be used, as well as providing flexibility to accommodate unforeseen scenarios, either through hot spot or framework development. These requirements make the initial framework design and development a time and cost intensive task. Thus, reasonable strategies for framework development typically involve development of the white-box and black-box versions simultaneously while evaluating a few disparate examples that can be encountered in the framework's domain. [90, pp. 21–22], [94]

The second drawback comes in describing, or documenting, the framework, and frameworks are notoriously difficult to describe. Generally speaking, frameworks are difficult to describe because they not only define all of the components that can be used, but frameworks further demand a specified behavior of the components within the framework's workflow while also needing to simultaneously explain the framework's workflow. Thus, a circular definition problem arises; components need to be defined within a workflow context, and a workflow is defined by usage of the components. There are two general philosophies in *how* to describe a framework: detailed documentation, and example-based "cookbooks". For the former, a thorough, detail-laden description of the framework is supplied, and the user is tasked with extracting any necessary development information related to her application. For the second treatment, simple initial examples are provided, and then incrementally more advanced examples of framework usage are presented, successively building upon previous examples, thus creating a set of "recipes" that formulate a framework "cookbook". In this work, we lean toward the detailed documentation approach, but we also provide a substantial cookbook. [90, pp. 498–501]

The third drawback comes with first-time usage, since there is a steep learning curve associated with frameworks. Given the complexity in developing and describing a framework, this last drawback should not be surprising. Thus, the cookbook approach typically provides the tutorials necessary for a user to become acquainted with the framework, and the documentation approach allows the user to explore the nuances of the framework once they have a grasp of the fundamental capabilities of the framework. That said, the complexity of learning depends on whether the user is studying the white-box version for future framework development or if the user is learning the roles and capabilities of library components in the black-box framework. There are many

strategies to learn a framework, but just as in development, a mix of white-box and black-box framework studies provides the fastest depth and breadth of framework knowledge. [90, pp. 505–520]

This chapter will detail all of the functionality of the MASS white-box framework except the reference components, while the next chapter will focus on a component library. The component library in the next chapter can be construed as either a completion of the white-box framework, or as a black-box framework in its own right, and this depends on how the software package is delivered. The first-time reader is encouraged to skim this chapter and the next, reading only topic descriptions, and then reread for the details. In this manner, a reader will gain a top-level perspective of how the MASS framework components interact, before diving into the details of a particular concept or system.

3.2 An Overview of the MASS Framework

The MASS problem readily lends itself to the software framework paradigm, and in this chapter, we will build upon the general framework given section 2.1 to provide a complete white-box framework for MASS. The white-box framework we present here consists of a set of utility classes, a set of plugin classes, a configuration class, and a centralized control mechanism in the form of the source estimate management system (SEM) class. The utility classes, the configuration class, and the SEM are all immutable, thus they define the frozen spots in the framework, while all plugins are meant to be developed by end-users, thus the plugins are the hot spots of the framework. The utility classes provide data transport mechanisms and various signal processing functions that are useful in plugin development. We will detail the utility classes in section 3.4, but in this section we will mainly focus on the plugins, the configuration, and the SEM.

The MASS framework is controlled by the SEM, and any user of the MASS framework controls the SEM via a *Configuration* object. A *Configuration* object represents a MASS scenario and contains information such as signal processing parameters, plugins to be used, plugin parameters, data sources, etc. Thus, a user constructs a *Configuration* object that specifies the data sources, plugins, and signal processing parameters needed to accomplish her task, and instantiates the SEM with this object. The SEM is then responsible for validating and executing the scenario defined in

a *Configuration*. Before we look at the SEM's workflow, we will overview the MASS framework's plugin architecture.

The MASS framework has several plugin types as shown in Figure 6, and these plugins can be broadly classified into three groups: configuration, input/output (I/O), and source separation. The configuration group contains one plugin, the *Config* plugin, whose sole responsibility is to create a valid *Configuration* object and pass it to the SEM.

The I/O plugins are tasked with acquiring data from and disseminating data to external sources via the data acquisition (DA) plugin and the data output (DO) plugin, respectively. Examples of acquired data include observation signals, source signals, and side-information, while examples of output data include CSEs and analysis results. The procedure to access a data source can vary dramatically, e.g. data stream, a file, a memory location, etc., and the I/O plugins are included for generality and completeness. The source separation plugins can be further subdivided into two groups: primary and secondary. The primary group provides functionality that is essential to performing MASS, or even SS, and is comprised of a source estimate production (SEP) system comprised of multiple SEP plugins, a source estimate evaluation (SEE) plugin, and a permutation ambiguity solution (PAS) plugin. The SEP plugin is a wrapper around a SS method, thus the purpose of an SEP plugin is to provide source estimates. The SEE plugin is tasked with evaluating the source estimates produced by the SEPs and create a CSE set. The PAS plugin is responsible for ensuring that the individual CSEs are consistently presented at the same outputs.

The secondary source separation plugins support the primary plugins. The source enumeration (SNUM) plugin provides an estimate of the number of active sources present in an observation set. The source estimate grouping (SEG) plugin organizes the outputs of the SEPs into groups, where each group contains estimates of a particular source. The source estimate time-alignment (SETA) plugin provides signal time-alignment functionality in the framework. The source estimate analysis (SEA) plugin provides analyses of the various functionality of the framework. As we stated earlier, the

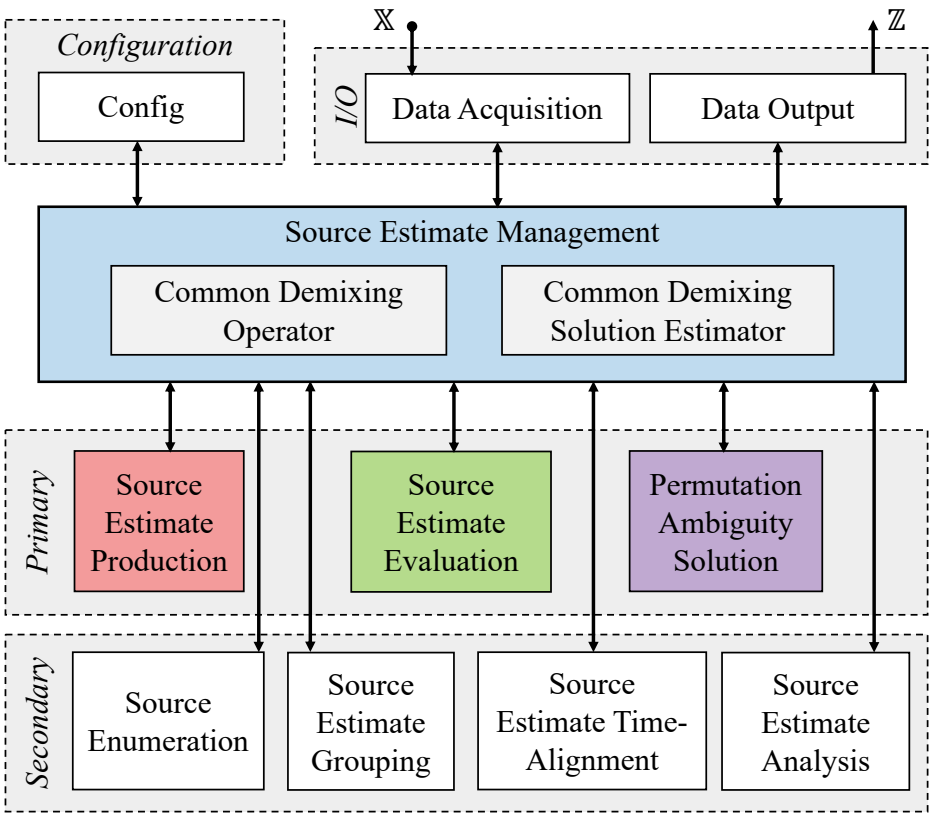

Figure 6. MASS Framework Plugin Architecture. MASS plugins are shown in grey, dash-lined boxes. The MASS framework has four fundamental processing layers/functionality: Configuration, Input/Output (I/O), Primary, and Secondary. The I/O layer deals with acquiring data (X) from and disseminating data (Z) to external sources, respectively. The Core layer is comprised of the non-extensible Source Estimate Management (SEM) system and is responsible for coordinating all tasks of the MASS framework. The SEM contains functionality that may be exposed as plugins in future versions of the framework, i.e. the Common Demixing Operator (CDO) and the Common Demixing Solution (CDS) estimator. The Primary layer contains the functionality crucial to performing BSS in the MASS framework. The Secondary layer includes functionality that either accommodates the necessities of other plugins or is useful for independently evaluating the performance of the MASS system.

SEM is a centralized control mechanism in the MASS framework, and Figure 6 shows that all plugins communicate only with the SEM.

Figure 6 also shows two pieces of functionality that are exposed to the user in the MASS framework: the common demixing operator (CDO) and the common demixing solution estimator (CDSE). The CDO allows a user to apply a CDS to a set of

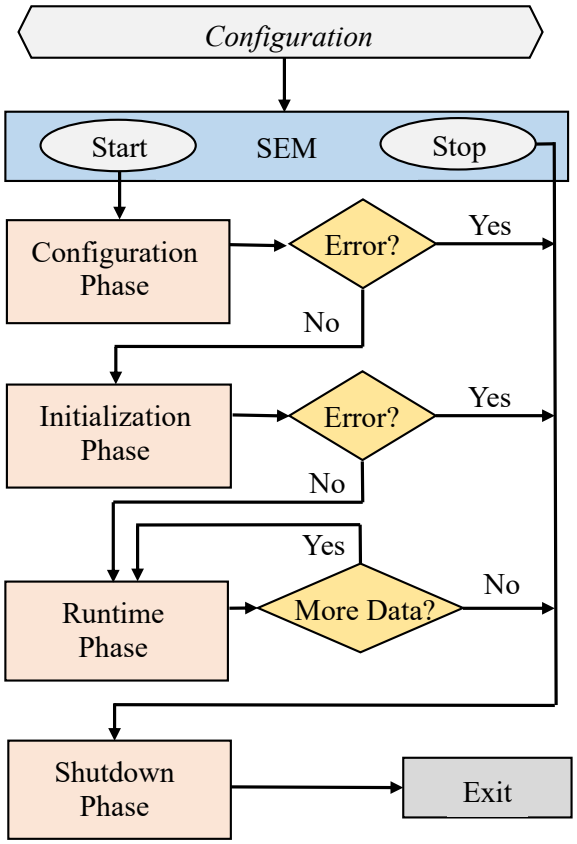

Figure 7. The source estimate management (SEM) workflow overview. The SEM workflow consists of four phases. The SEM validates *Configuration* information in the configuration phase, initializes the plugins in the initialization phase, performs MASS in the runtime phase, and performs clean up tasks prior to exiting in the shutdown phase.

observation data, while the CDSE allows a user to estimate a CDS given a set of source estimates. Although these items of functionality are exposed through one of the utility classes in this version of the framework, future framework versions may implement the CDO and the CDSE as plugins, thus they are included in Figure 6.

As we have stated previously, the SEM controls and manages the workflow of the framework, and as shown in Figure 7, the workflow is decomposed into four phases: configuration, initialization, runtime, and shutdown. Once the SEM object has been instantiated with a *Configuration*, the user begins the SEM's workflow by calling the SEM's Start() function. The configuration phase determines if the provided *Configuration* object is valid, as well as instantiating each plugin specified in the

Configuration and determining various signal processing parameters. For example, if a *Configuration* contains a SEE plugin that requires grouped source estimates, then a valid *Configuration* must also contain a SEG plugin. Thus, the SEM must ensure that all information contained in the *Configuration* is compatible with the MASS framework's capabilities.

In the initialization phase, the SEM allows each of the plugins to perform any necessary tasks prior to performing the plugin's MASS task. For instance, the DA plugin will need to open connections to data sources, and an SEP plugin may need to revise its parameters based upon the SEM's signal processing parameters. The initialization simply allows plugins to prepare for the MASS task.

In the runtime phase, the SEM coordinates the plugins efforts in performing MASS. That is, while the SEM is able to acquire data from the DA plugin, the SEM calls on the SEPs to update their source estimates and passes these estimates to the various plugins used to create a CSE set. The plugins that are called depend upon the particular *Configuration*, but the SEE and PAS plugins are always called, and a valid configuration must always contain at least one SEP plugin, as well as a SEE and PAS plugin.

If an error occurs at any point, or if the SEM is unable to acquire more input data, the shutdown phase begins. In the shutdown phase, the SEM allows each of the plugins to perform any necessary clean up tasks before the application exits. For instance, the DO and DA plugins might need to terminate their connections to external data sources. The shutdown phase is included to ensure that a MASS application will exit cleanly. The SEM workflow will be detailed in section 3.7.

3.3 Notation and MATLAB Object-Oriented Programming Overview

The initial MASS framework is implemented using MATLAB's object-oriented programming (OOP) and function-based language. A comprehensive overview of OOP concepts and design is well beyond the scope of this work, but in this section, we will define a simple set of concepts and terminology within MATLAB's OOP language that will be used to define the components of the MASS framework in following sections.

Table 3. Keywords used in the MATLAB object-oriented description of the MASS framework.

Keyword	Abbreviation	Description
Abstract	none	This declaration keyword denotes that the class only contains the structural definition. This type of class cannot be instantiated.
Sealed	none	This declaration keyword denotes that the class cannot be extended.
public	pub	This keyword designates that any class can access the field.
protected	prot	This keyword designates that only the defining class or an extension of the class can access the field.
private	priv	This keyword designates that only the defining class can access the field.
SetAccess	Set	This keyword followed by *pub*|*prot*|*priv* denotes which classes can modify the property.
GetAccess	Get	This keyword followed by *pub*|*prot*|*priv* denotes which classes can view the property value or execute the function.

3.3.1 General MATLAB Notation and Keywords

In this work we will use keywords to easily define a variable's attributes, most of these being generally accepted concepts in all OOP languages, and here we confine our attention to MATLAB's notion of the attributes. The first two keywords, *Abstract* and

Sealed, are class-level declarations, the next three keywords, *public*, *protected*, and *private*, describe the accessibility of a class field, and the last two keywords, *SetAccess* and *GetAccess,* are accessibility sub-categorizations used to conveniently qualify the previous three accessibility keywords. These terms are summarized in Table 3, and we will now quickly describe these keyword sets.

In this work we will use two special types of classes given by the class declaration keywords *Abstract* and *Sealed*. An *Abstract* class simply defines the *structure* of the class without necessarily defining any specific functionality or variable values. An *Abstract* class cannot be instantiated, and therefore, can only be extended. A *Sealed* class, on the other hand, fully defines and populates its fields and can be instantiated, but it cannot be

extended. A *Sealed* class can extend other classes, including *Abstract* classes, but once a class is declared as *Sealed*, the class is fixed and cannot be extended by other classes. Examples of *Abstract* and *Sealed* classes are given in Table 4 and Table 5, respectively.

OOP languages, generally, have a notion of scoping that determines whether and how a field can be accessed, and we will use three basic accessibility types: *public*, *protected*, and *private*. If a field of a class is declared as *public*, then the field can be fully accessed by any class, i.e. a *public* variable can be read or modified by any class, and a *public* function can be run by any class. A *protected* field can only be accessed by the class defining the field or any class that extends the defining class. A *private* field can only be accessed by the defining class. In MATLAB, a set of properties' or methods' accessibility is controlled by the keyword "*Access*", so that the declaration *Access* = [*public* | *protected* | *private*] gives a group of variables or functions a particular accessibility.

Variables in MATLAB have the additional accessibility keywords of *SetAccess* and *GetAccess*. The *SetAccess* keyword determines the accessibility for *modifying* a variable value, and *GetAccess* determines the accessibility for *reading* a variable value. In many languages, set() and get() functions are written for a particular variable to control the setting or getting, respectively, of a variable's value. The *SetAccess* and *GetAccess* are convenience keywords that effectively determine whether and how a variable can be accessed just with the variable name. For example, if *Class1* declares the variable *var1* as (*GetAccess = public*, *SetAccess = protected*), then *Class1.var1* can be read by any class but can only be modified by *Class1* or extensions of *Class1*.

Although MATLAB is not a strongly-typed language, we will specify the data type of a field within a class definition using the set of general (primitive) data types given in Table 6. We will also use square brackets, [], following a data type to indicate that the variable is an array of the specified data type. For instance, if we denote a variable as *Float []*, then the variable is designated as array of floating point numbers.

Table 4. An example Abstract class in MATLAB.

```
classdef (Abstract) ExampleAbstractClass
    % An example Abstract class

    %-----------------------------------------------------------------
---
    properties (Access = protected)
        % Define Protected variables here.
        % Protected Variables can only be accessed by class
extensions
        protectedVar1;      % Variable definition, without a default
value
        protectedVar2 = 0;  % Variable definition, with a default
value
    end

    %-----------------------------------------------------------------
---
    properties (GetAccess = public, SetAccess = protected)
        % Define Mixed Access variables here.
        % GetAccess = public: Variables defined here are read-
accessible
        %   by all classes.
        % SetAccess = protected: Variables defined here can only be
        %   modified by class extensions.
        var3;       % Variable definition, without a default value
        var4 = 0;   % Variable definition, with a default value
    end

    %-----------------------------------------------------------------
---
    methods (Access = public)
        % Define Public functions here.
        % Public functions can be run by any class
        obj = function1(obj,input1,input2); % Function signature
only.

        % NOTE: "obj" is MATLAB's version of "self" or "this" in
other
        %       OOP languages.  "obj" is a reference to this
instance
        %       of ExampleAbstractClass.
    end

    %-----------------------------------------------------------------
---
    methods (Access = protected)
        % Define Protected functions here
        % Protected Functions can only be accessed by class
extensions
        [output1, output2] = function3(obj,inputA); % Function
signature only.
    end
end
```

Table 5. An example Sealed class in MATLAB.

```
classdef (Sealed) ExampleSealedClass < ExampleAbstractClass
    % An example Sealed class extending an Abstract class.
    % The Sealed class cannot be extended.

    %-------------------------------------------------------------
---
    properties (Access = protected)
        % Define additional Protected variables here.
        % Protected Variables can only be accessed by class
extensions.
        % Since ExampleSealedClass is non-extensible, Protected
Access is
        %   equivalent to Private Access, here.
        varA = [];
    end

    %-------------------------------------------------------------
---
    methods (Access = public)
        % Define Public functions here.
        % Public functions can be run by any class.

        % Example Constructor:
        function obj = ExampleSealedClass(Var1,Var2,VarA)
            % Populate variables defined in ExampleAbstractClass:
            obj.protectedVar1 = Var1;
            obj.protectedVar2 = Var2;
            % Populate the variable defined in ExampleSealedClass:
            obj.varA = VarA;
        end

        % Fully define the Abstract function1 operation:
        function obj = function1(obj,input1,inputA)
            % function1 method goes here, for example:
            obj.protectedVar1 = input1;
            obj.varA = inputA;
        end

        % New function for ExampleSealedClass:
         function [output2] = function2(obj,inputA)
            obj.varA = inputA;
            output2 = obj.protectedVar2;
        end
    end

    %NOTE: ExampleSealedClass does not define or use
    %      function3 from ExampleAbstractClass
end
```

Table 6. Primitive data types used in the object-oriented description of the MASS framework.

Data Type	Abbreviation	Description
String	*Str*	A string (array) of characters, where a character is usually a Unicode encoding.
Integer	*Int*	A general integer. Here, we assume a signed integer, but we will leave the bit depth (value range) unspecified.
Float	none	A general floating point number. In MATLAB the default is double-precision.
Boolean	*Bool*	A binary variable, flagging "True" or "False" conditions.
Void	none	Nothing. Typically used to describe functions that have no output.

3.3.2 General Class Definition Format

In the rest of this chapter, we will define numerous classes used in the MASS framework. Instead of subjecting the reader to viewing MATLAB code for each class definition, we will now define a general format for defining classes, using the following format to define a class: give a short description of the class, list the class constructor(s) (if applicable), display a table containing all of the fields of the class along with data types and accessibility, and then give a list detailing the functions in the class. For example, we now define the *ExampleSealedClass* in this format.

(Sealed) ***ExampleSealedClass*** extends *ExampleAbstractClass*

Description: The *ExampleSealedClass* class provides an example of a *Sealed* class that extends the *Abstract* class *ExampleAbstractClass*.

Constructor: *ExampleSealedClass(Var1,Var2,VarA)*

Fields Summary:

Field Name	Type	Set	Get	Description
varA	*Float*	prot	prot	Example protected variable
function1()	*Void*	NA	pub	Example overridden public function
function2()	*Function*	NA	pub	Example public function

Inherited Fields: *protectedVar1, protectedVar2, var3, var4, function1(), function3()*

Property Details:

varA	*Float, SetAccess=Protected, GetAccess=Protected*

A detailed description of *VarA* goes here. Generally, properties can be succinctly described in the Fields Summary table, but if needed, a list of detailed variable descriptions appears after the Fields Summary.

Function Details:

(Constructor)	*ExampleSealedClass(Var1,Var2,VarA)*

The description of this constructor goes here. If multiple constructors exist, a new section is created for each constructor.

Var1 – A description of *Var1* goes here.

Var2 – A description of *Var2* goes here.

VarA – A description of *VarA* goes here.

function1	*function1 (input1, inputA)*

The description of *function1* goes here. Typically, simple functions that have no arguments and have no outputs can be succinctly described in the Fields Summary table. Functions with a single output are labeled in the Fields Summary table with the data type of the output variable, functions with no outputs are labeled as data type *Void*, and functions with multiple outputs are labeled as data type *Function*. If a *GetAccess* scoping property is not given for the function, then the function is assumed to be publicly accessible.

input1, Float – A description of each input or output goes on a new line. The description of *input1* goes here. Note that the data type of the variable follows the variable name.

inputA, Integer [] – A description of *inputA* goes here. We denote an array of a certain data type with square brackets, i.e. [], following the data type.

Function2	*function1 (inputA)*

The description of *function2* goes here.

inputA, Integer [] – A description of *inputA* goes here.

3.4 The Immutable Utility Classes

The MASS framework utility classes provide various tools to interact with a MASS system, from efficient transfer of data to a set of queries. We now define the MASS framework utility classes.

3.4.1 The SourceEstimate Class

(Sealed) ***SourceEstimate***

Description: The *SourceEstimate* class is a Sealed class with two public fields: a float array (*Float []*) of source estimates' samples and a float array containing the corresponding CDS used to produce the source estimates. Various systems produce source estimates, and the *SourceEstimate* class is a container for the signals and the CDS used to create the estimates. This class is provided for generality and may be developed in the future to contain more information about the process leading to the source estimates.

Constructor: *SourceEstimate()*

Fields Summary:

Field Name	Type	Set	Get	Description
sig	*Float []*	pub	pub	An array of source estimate samples
cds	*Float []*	pub	pub	A CDS that produces *sig* when applied to the observations via the CDO.

3.4.2 The ExternalSignal Class

(Sealed) ***ExternalSignal***

Description: The Sealed *ExternalSignal* class defines data sources or destinations used to acquire input data from or output MASS system data to locations external to the MASS system, respectively. In this version of MASS, the Data Acquisition plugin uses *ExternalSignal*s to acquire observations and side-information, and the Data

Output plugin uses an *ExternalSignal* to define a destination for MASS system outputs.

Constructor: *ExternalSignal()*

Fields Summary:

Field Name	Type	Set	Get	Description
name	*String*	pub	pub	A descriptive name for identifying the *ExternalSignal*
groupId	*Integer*	pub	pub	An identifier used to group a set of *ExternalSignal*s
location	*String*	pub	pub	A data location, e.g. a file location, an input device, a memory location, etc.
varName	*String*	pub	pub	A variable name at *location*, e.g. a variable in a .mat file
channel	*Integer*	pub	pub	A channel number for the data source.
sampRate	*Float*	pub	pub	The sampling rate of the signal.
type	*String*	pub	pub	A data type descriptor for the signal source, e.g. “file”, “channel”, “memloc”, etc. used by the *DAPlugin* and *DOPlugin* to determine how to acquire the signal.

3.4.3 The *MASSInfo* Class

(Sealed) ***MASSInfo***

Description: The *MASSInfo* class provides access to various pieces of information about the MASS system. Each plugin in the MASS framework is provided with a *MASSInfo* object, so that the plugin can query the MASS system. The *MASSInfo* object is initialized and populated during the configuration phase, and the plugins have access to all of the *MASSInfo*’s capabilities during the initialization, runtime, and shutdown phases.

Constructor: *MASSInfo()*

Fields Summary:

Field Name	Type	Access	Description
getBlkLen()	*Integer*	pub	Returns the data block length.
getBlkStep()	*Integer*	pub	Returns the data block step.
getSampRate()	*Integer*	pub	Returns the sampling rate set in configuration.
getNumSrc()	*Float []*	pub	Returns the number of sources.
getNumObs()	*Integer*	pub	Returns the number of observation signals.
getCDS()	*Function*	pub	Returns the current CDS.
getCDSPrev()	*Function*	pub	Returns the CDS from the previous block.
getCDSFiltLen()	*Integer*	pub	Returns the filter length used in the CDS.
getCDSFiltOffset()	*Integer*	pub	Returns the delay applied in the CDS.
getConfig()	*Function*	pub	Returns the configuration used
CDO()	*multiple outputs*	pub	The common demixing operation.
SysIdOp()	*multiple outputs*	pub	The system identification operation.
ImgOp()	*multiple outputs*	pub	The imaging operation
CDSEstimator()	*multiple outputs*	pub	An estimator for a CDS.

Function Details:

---

getNumSrc *[numSrc] = getNumSrc()*

This function returns an array containing the (possibly time-varying) number of sources that are active in each mixture in a data block.

numSrc, Float [] – An *[NxP]* array whose $(n,p)^{th}$ element is an estimate of the number of sources at the n^{th} sample of the p^{th} observation. If an SNUM plugin is defined in the MASS configuration, then the SNUM plugin's output populates the *numSrc* output of *getNumSrc* (see Sect. 0).

CDO *[out, img] = CDO(Sig, Krnl, Cnv)*

The CDO function provides the Common Demixing Operation used in the MASS framework, and in this version, it is the BLTI filtering operation given in Sect. 2.2.1.

Sig, Float [] – An [*N*x*P*] array of signal samples to be filtered and summed to produce the *Q* output signals.

Krnl, Float [] – An [*L*x*P*x*Q*] sized array of filter coefficients used to filter and sum the *P* inputs to produce the *Q* outputs, where *L* is the filter length.

Cnv, String – A string to denote which part of the convolution to use, and can be any of the strings below, and the default is 'last'.

'full' – Produces the full convolution which has a length of $M=N+L-1$, where *N* is the length of the input signal and *L* is the length of the filter.

'same', 'first', 'last' – These produce an output whose length is equal to the length of the input signal, e.g. $M=N$, where 'same' uses the central part of the full convolution, and 'first' and 'last' produce the first and last parts of the full convolution, respectively.

'valid' – Produces the part of the convolution that did not use zero-padding. For a signal of length *N* and a filter of length *L*, the result will be of length $M=N-L+1$ if $M>0$, or 0, otherwise.

out, *Float []* – An [*M*x*Q*] array of output signal samples, where *Q* is the output dimension of the *Krnl* variable above, and *M* is the signal length determined by the convolution method used in the *Cnv* variable above.

img, Float [] – An [*M*x*P*x*Q*] array of images used to produce the *out* variable above, where *M* is the length determined by the convolution method used in the *Cnv* variable above, and *P* and *Q* are dimensions of the *Krnl* variable above. If we consider that the q^{th} output is the sum of the filtered input signals, then the $(m,p,q)^{th}$ element of *img* is the m^{th} sample of the filtered p^{th} input used to produce the q^{th} output.

SysIdOp *[krnl, offset] = SysIdOp (SigArr, Sig, FiltLen, Offset)*

The *SysIdOp* implements the system identification operator, *Id{}*, defined in Sect. 1.2.3.

SigArr, *Float []* – An [*N*x*K*] column-oriented array of signal samples to filtered and summed to estimate the signal in *Sig*, where *N* is the length of each signal, and *K* are the number of signals.

Sig, Float [] – An [*N*x*1*] column-oriented vector of signals samples to be estimated via the filtered sum of the samples in *SigArr*.

FiltLen, *Integer* – The filter length(s) used to estimate *Sig* from *SigArr.*

Offset, Integer – An integer offset applied to *Sig* before estimation. A positive value is an advance, and a negative value is a delay.

krnl, Float [] – An [*L*x*K*x*1*] array of filter coefficients such that *CDO(SigArr,krnl)* is an estimate of *Sig*, where *L=FiltLen* and *K* is the number of signals in *SigArr*.

offset, Integer – Before estimation, *Sig* is time-shifted so that the maximum cross-correlation between *Sig* and the signal in *SigArr* with maximum advance relative to *Sig* is zero, and the *Offset* is applied. *offset* is the total offset applied to *Sig.*

ImgOp *[img, filt, offset] = ImgOp (Sig1, Sig2, FiltLen, Offset)*

The *ImgOp* implements the imaging operation, *Im{},* defined in Sect. 1.2.3. The image estimation is a special case of *SysIdOp* above when the first argument is a single signal. Once the *krnl* is acquired from *SysIdOp*, the signal is filtered via *CDO* to produce the image.

Sig1, Float [] – An [*N*x*1*] column-oriented vector of signal samples, which when filtered, provides an estimate of *Sig2.*

Sig2, Float [] – An [*N*x*1*] column-oriented vector of signal samples to be estimated via filtered vesion of *Sig1.*

FiltLen, Integer – The filter length used to estimate *Sig2* from *Sig1.*

Offset, Integer – An integer offset applied to *Sig2* before estimation. A positive value is an advance, and a negative value is a delay.

img, Float []– The filtered version of *Sig1* estimating *Sig2*.

filt, Float [] – The filter used to estimate *Sig2* from *Sig1*. *filt* is the *krnl* variable obtained from *SysIdOp(Sig1, Sig2, FiltLen, Offset)*. See *SysIdOp* above. The *img* signal is produced by either output of *CDO(Sig1, filt)*. See *CDO* above.

offset, Integer – The total offset applied to *Sig2* before estimation. See "*offset"* in *SysIdOp* above.

CDSEstimator *[krnl, offset] = CDSEstimator (Sig)*

The *CDSEstimator* function estimates the CDS for a source estimate. In this work, the CDS is a linear filter array, and *CDSEstimator* is equivalent to *SysIdOp* when the *SigArr* variable is the observation data, *FiltLen = MASSInfo.getCDSFiltLen()*, and *Offset = MASSInfo.getCDSFiltOffset()*.

Sig, Float [] – An [*N*x*1*] column-oriented vector of signal samples for which a CDS is to be estimated from the [*N*x*P*] array of observation samples.

krnl, Float [] – An [*L*x*P*x*1*] sized array of filter coefficients used to filter and sum the *P* observation signals to produce an estimate of *Sig*, where *L* is the filter length given in *getCDSFiltLen()*.

offset, Integer – An offset applied to *Sig* prior to CDS estimation. See "*offset"* in *SysIdOp* above.

3.4.4 The *PluginInst* Class

(Sealed) ***PluginInst***

Description: The Sealed *PluginInst* class provides plugin instantiation information in the MASS framework. The SEM only recognizes one particular constructor for any plugin, i.e. *PluginClass*(*PluginParameterSetId*), where *PluginClass* is a plugin class, and *PluginParameterSetId* is an integer identifying a particular parameter set that *PluginClass* will use. The *PluginInst* class defines the plugin class to be instantiated, as well as the parameter set identification argument used when the SEM instantiates a plugin.

Constructor: *PluginInst* ()

Fields Summary:

Field Name	Type	Set	Get	Description
pluginClass	*String*	*pub*	*pub*	A name of a plugin class.
pluginParamSetId	*Integer*	*pub*	*pub*	A plugin class parameter set identification number.

3.5 The Abstract Configuration Class (*Configuration*)

(Abstract) ***Configuration***

Description: *The* Configuration *class is responsible for defining all the information necessary for the SEM to initialize itself and all of the user-extended MASS plugins. The* method *of acquiring configuration information can be modified by the end-user via the extensible* ConfigPlugin *class (see Sect. 3.6.2) whose only purpose is to produce a valid* Configuration *object.*

The *Configuration* class uses the *ExternalSignal* class to define two external data signal sets; an observation/mixture signal set (*semObsSig)* and an underlying source signal set (*semSrcSig*). The observation signals are required by source separation methods, while a set of underlying source signals allow some plugins to operate in a *supervised* manner, e.g. the *SEAPlugin* may implement an SIR metric. Although the *semSrcSig* is only required to be defined when a particular plugin needs this information, the *semObsSig* is required for any configuration.

The *Configuration* class also uses the *ExternalSignal* class to define a destination for the data output. In this version of MASS, we require that a data output destination be defined for a configuration to be valid.

The *Configuration* class also uses the *PluginInst* class to define instantiations of the various plugins by the SEM. The *PluginInst* object simply contains the name of a plugin class, *PluginClass,* and a parameter set identification number, *PluginParamSetId*. For every *PluginInst* in the *Configuration*, the SEM will instantiate a *PluginClass* object via the constructor *PluginClass(PluginParamSetId)* for the appropriate plugin, or plugin set.

Fields Summary:

Field Name	Type	Set	Get	Description
semSelfComp	*Boolean*	prot	pub	Determines if MASS work in self-competition mode
semBlkLen	*Integer*	prot	pub	The data block length used in MASS processing
semBlkStep	*Integer*	prot	pub	The data block step used in MASS processing
semCDSFiltLen	*Integer*	prot	pub	The CDS filter length (BLTI)
semCDSOffset	*Integer*	prot	pub	A global offset (delay) in the CDS
semPluginError	*Integer*	prot	pub	Flag to determine operation of SEM after a plugin error occurs. 0=Stop,1=Remove plugin and continue
semPluginUndef	*Integer*	prot	pub	Flag to determine operation of SEM if a plugin is undefined in the configuration. 0=Stop,1=Use appropriate reference component
semVerbose	*Integer*	prot	pub	Flag to determine if the SEM should display information during processing. 0=No,1=Display summary info
daObsExtSig	*ExternalData[]*	prot	pub	An array of *ExternalData* objects defining the observation signals.
daSrcExtSig	*ExternalData[]*	prot	pub	An array of *ExternalData* objects defining the underlying source signals.
doDestExtSig	*ExternalData*	prot	pub	An *ExternalData* object defining the destination of MASS data output
SEP	*PluginInst []*	prot	pub	An array of SEP plugin construction definitions
SEE	*PluginInst*	prot	pub	SEE plugin construction definition
PAS	*PluginInst*	prot	pub	PAS plugin construction definition
SNUM	*PluginInst*	prot	pub	SNUM plugin construction definition
SEG	*PluginInst*	prot	pub	SEG plugin construction definition
SEA	*PluginInst*	prot	pub	SEA plugin construction definition
SETA	*PluginInst*	prot	pub	SETA plugin construction definition
DA	*PluginInst*	prot	pub	DA plugin construction definition
DO	*PluginInst*	prot	pub	DO plugin construction definition

3.6 The Abstract Plugin Classes

In this section we will completely define the various plugins used by the MASS framework. The plugin class definitions provided here are all abstract, thus they cannot be instantiated, but any implementation of a MASS plugin must extend and adhere to these definitions.

3.6.1 General Plugin Class Definition Overview

Although all of the MASS sub-systems have focused and non-overlapping tasks, all MASS framework plugins implementing these sub-systems share traits, and in an effort to reduce redundancy in describing the various plugins in the MASS framework, we will describe three different sets of fields that are categorized as *common* fields, *plugin requirement* fields, and *SEM data* fields. The common fields contain plugin identification information, the requirement fields allow a plugin to communicate its data needs to the MASS system, and the SEM data fields are used by the SEM to supply plugins with their required data before the plugin performs its task.

All plugins in the MASS framework are required to have the common fields, which are listed in Table 7. The identifying information of a plugin, i.e. the *pluginName*, *pluginVersion*, *pluginAbbr*, and *pluginDescr* fields, should be populated at construction. In this version of MASS, only the *pluginName* field is required to be populated, but we strongly encourage all MASS plugin developers to utilize these various identifiers, as they can be very useful in facilitating MASS development and usage, e.g. in using databases to manage MASS configurations, in developing GUIs for users to efficiently use the MASS system, etc. All reference components provided in this work supply this basic set of identifying information.

The *massInfo* field contain a *MASSInfo* object that allows a plugin to query the MASS system. This object is instantiated and initialized during the configuration phase, so that a plugin can use its *massInfo* field in the initialization and runtime phases, when the SEM calls the plugin's *Init()* and *Update()* functions, respectively. See Sect. 3.4.3 for detailed functionality and information provided by the *MASSInfo* class.

Table 7. MASS Framework Common plugin fields. All plugins in the MASS framework are required to have the common fields. These fields provide plugin identifying information, a MASSInfo object used to query the MASS system, as well as functions that are called by the SEM during the initialization and runtime phases.

Field Name	Type	Set	Get	Description
pluginName	*String*	prot	pub	A string with a full name of the plugin.
pluginVersion	*String*	prot	pub	A string with the plugin's development version.
pluginAbbr	*String*	prot	pub	A string with an abbreviation of *pluginName*, typically used in user displays with strict character limits.
pluginDescr	*String*	prot	pub	A string containing a description of the plugin's implemented functionality.
pluginStatus	*Integer*	prot	pub	A flag denoting the state of the plugin.
paramSetId	*Integer*	prot	pub	A plugin-specific parameter set identification number
massInfo	*MASSInfo*	pub	prot	Allows the plugin to query the MASS system for particular parameters and functionality using the *MASSInfo* object.
Init()	*Void*	NA	pub	Plugin-specific initialization procedure. Called by SEM during initialization phase.
Update()	*Void*	NA	pub	Plugin-specific task procedure. Called by SEM during runtime after a new data-block has been acquired.
Shutdown()	*Void*	NA	pub	Plugin-specific shutdown procedure. Called by SEM during shutdown phase.

The *Init()* function is executed by the SEM in the initialization phase. Initialization procedures are specific to both a plugin type and the actual implementation of a plugin, thus we cannot specify any generalized procedure. That said, the plugin's *massInfo* object is usable at the time that *Init()* is called, thus the plugin can query the MASS system for information to modify plugin parameters, for instance.

The *Update()* function is executed by the SEM in the runtime phase. After acquiring a new block of data, the SEM will call the plugins' *Update()* functions in the order specified in Table 11. When a plugin's *Update()* function is called, the plugin should perform its designated task using the current block of data in order to populate the plugin's *update fields*. Update fields are used to supply the SEM with the plugin's outputs, thus after the SEM calls the *Update()* function and the function completes its

task, the SEM then collects the results from the Update fields. The update fields can only be populated by the plugin, but can be read by the SEM. Except for one plugin class, all plugins have update fields that are specific to their plugin class. For example, the SEP plugin class (see Sect. 3.6.5) produces source estimates in the call to the *Update()* function, and the SEP plugin class has an *Update Field* named "*se*" of type *SourceEstimate*. At the end of the *Update()* function, the plugin will populate the *se* variable with the plugin's current source estimate and CDS, and the SEM will then access and collect the information contained in the plugin's *se* update field.

In Table 8 we provide a complete set of the *requirement* fields used by various plugins. The requirement fields should be set during the construction of a plugin, so that the SEM can determine the plugin's data requirements during the configuration phase, and therefore ensure that the plugin's necessary data needs are met during the initialization and runtime phases. We note that usage of the fields is plugin class specific, where some plugins use all of the requirement fields, some use none of the requirement fields, and still more use various subsets of the requirement fields. We will see these usages when we define the various plugins in Sects. 3.6.2 - 3.6.11.

The last general set of plugin fields are the *SEM Data* fields, and these are summarized in Table 9. All plugins except one, have at least one of these fields in their class definitions. Depending on a particular plugin's data requirements, the SEM will update the plugin's SEM data fields either before initialization or during runtime, depending on the type of data being supplied. For instance, the Data Acquisition plugin is supplied with a data block length and a block step, before the plugin's *Init()* function is called, while the SEM will provide an SEP plugin with a new block of observation data during runtime before the SEP plugin's *Update()* function is called. Once again, the SEM data fields that are included for a plugin are highly dependent on the plugin type, as we shall see when we define the various plugins in section 3.6.

Beginning in the next sub-section, we provide the various plugin class definitions. These definitions will have a similar appearance to other classes we have already defined with the notable addition of the new fields we have defined in this section. In order to reduce redundancy, we will simply list the various fields defined here, and the reader can use this section as a reference to understand the various common, plugin requirement, and SEM data fields. We begin with the simplest plugin definition with regards to the

fields used, the configuration plugin, but we shall soon see the complicated field structures that are possible when we continue the plugin definitions in the following sub-sections.

Table 8. MASS Framework Requirement plugin fields. The requirement fields are used by the plugins to communicate data needs to the SEM, and based upon these requirements, the SEM ensures that the MASS system is properly configured. This is a complete set of requirement fields, but the subset of fields used are dictated by a particular plugin class.

Field Name	Type	Set	Get	Description
reqBlkLenMin	*Float*	prot	pub	The minimum amount of data (the data block length) required by the plugin. Positive values will indicate the nearest integer value of samples, while a negative value will indicate the number of seconds.
reqBlkLenMax	*Float*	prot	pub	The maximum number of data samples (the data block length) a plugin can handle. Positive values will indicate the nearest integer value of samples, while a negative value will indicate the number of seconds.
reqSrcSig	*Bool*	prot	pub	Denotes whether or not the plugin requires underlying source signals. Supervised methods set this variable to "true" or 1.
reqSrcNum	*Bool*	prot	pub	Denotes whether or not the plugin requires an estimate of the source number. Setting this to "true" ensures that a source enumeration plugin is available.
reqSrcEstCDS	*Bool*	prot	pub	Denotes whether or not the plugin requires the CDS for every source estimate. Setting this to "true" ensures that all source estimates' CDSs are available.
reqSrcEstTA	*Bool*	prot	pub	Denotes whether or not the plugin requires source estimates to be time-aligned. Setting this to "true" ensures that a SETA plugin is available.
reqSrcEstGroup	*Bool*	prot	pub	Denotes whether or not the plugin requires source estimates to be grouped. Setting this to "true" ensures that a SEG plugin is available.
reqSrcEstGroupTA	*Bool*	prot	pub	Denotes whether or not the plugin requires source estimates to be grouped and then time-aligned. Setting this to "true" ensures that a SEG plugin and a SETA plugin are both available.

Table 9. SEM data fields. SEM data fields are populated at each data block before a plugin's Update function is called during the runtime phase. This table lists a complete set of SEM data fields. The subset of fields used by a particular plugin is dependent on the plugin's class.

Field Name	Type	Set	Get	Description
semObsSig	*Float []*	pub	pub	The current block of observation signals' samples
semSrcSig	*Float []*	pub	pub	The current block of source signals' samples
semCSE	*SourceEstimate*	pub	pub	The current CSE
semSrcEstArr	*SourceEstimate*	pub	pub	An array of source estimates produced from SEPs
semSrcEstAlg	*SourceEstimate []*	pub	pub	Source estimates grouped by SEP
semSrcEstGroup	*SourceEstimate []*	pub	pub	Source estimates grouped by estimated source
semSrcEstArrTA	*SourceEstimate*	pub	pub	An array of time-aligned source estimates
semSrcEstGroupTA	*SourceEstimate []*	pub	pub	Time-aligned source estimates grouped by estimated source
semCDSPrev	*Float []*	pub	pub	CDS from previous block
semSEA	*Float []*	pub	pub	An array of CSE analysis values
semObsExtSig	*ExternalSignal*	pub	pub	Data used to acquire observation signals
semSrcExtSig	*ExternalSignal*	pub	pub	Data used to acquire source signals
semDataDestExtSig	*ExternalSignal*	pub	pub	Data destination for MASS output
semBlkLen	*Integer*	pub	pub	Data block length
semBlkStep	*Integer*	pub	pub	Data block step
semSampRate	*Float*	pub	pub	Data sampling rate

3.6.2 The Configuration Plugin (*ConfigPlugin*)

(Abstract) ***ConfigPlugin***

Description: The *ConfigPlugin*'s only task is to provide the SEM with a *Configuration* class-extended object, so that the SEM can configure the MASS system. The MASS framework requires a valid configuration to operate. In this version of the framework, the SEM has three constructors: one which takes a *Configuration*-extended object

directly as an argument, and the other two either use a default *ConfigPlugin* extension when no argument is supplied or take a *ConfigPlugin* extension object as an argument. Extensions of the *ConfigPlugin* class could potentially present a GUI, so that a user could create and save particular configurations at will. In Appendix B, we provide a simple, default *ConfigPlugin* extension that allows a user to choose from pre-defined configurations from the command line in MATLAB.

Update Fields Summary:

Field Name	Type	Set	Get	Description
config	*Configuration* class extension	prot	pub	An extension of the MASS framework *Configuration* class.

Common Fields: *pluginName, pluginVersion, pluginAbbr, pluginDescr, pluginStatus, paramSetId, massInfo, Init(), Update()*, *Shutdown()*

Requirement Fields: None

SEM Data Fields*:* None

3.6.3 The Data Acquisition Plugin (*DAPlugin*)

(Abstract) ***DAPlugin***

Description: The Data Acquisition plugin is responsible for acquiring external signals such as observations or side-information used by the various MASS plugins. The data acquisition process is potentially a very complicated task, and here we try to generalize that task to accommodate processes we cannot imagine. The *DAPlugin* is not only responsible for acquiring the external signals, but it is also tasked with providing the SEM with new blocks of data each time the SEM calls the plugin's *Update()* function, thus the *DAPlugin* extensions are responsible for resolving any buffering and memory issues necessary to provide data to the MASS system. We provide a simple implementation of this plugin class in 4.1, that simply reads in audio files, and block-sequentially steps through the data at each *Update()* execution. The *DAPlugin* is also responsible for ensuring that all signals are sampled at *semSampRate* set in configuration.

Update Fields Summary:

Field Name	Type	Set	Get	Description
obsSig	*Float []*	prot	pub	An array of observation signal samples at the current data block
srcSig	*Float []*	prot	pub	An array of source signal samples at the current data block
numObsSig	*Integer*	prot	pub	The number of observation signals
numSrcSig	*Integer*	prot	pub	The number of source signals

Common Fields: *pluginName, pluginVersion, pluginAbbr, pluginDescr, pluginStatus, paramSetId, massInfo, Init(), Update()*, *Shutdown()*

Requirement Fields: None

SEM Data Fields: *semObsExtSig, semSrcExtSig, semBlkLen, semBlkStep, semSampRate*

Specialized Functions: *Start(), Stop()*

Function Details:

Init *Init ()*

The *DAPlugin* class is special, in that the SEM data fields are populated in the configuration phase *before* the *Init()* function is called. Thus, the *DAPlugin* class has access to the information contained in *semObsExtSig, semSrcExtSig, semBlkLen, semBlkStep,* and *semSampRate* when the *Init()* function is called. The *DAPlugin* is responsible for populating the *numObsSig* variable with the number of observation signals specified in the *semObsExtSig* object, and when a *semSrcExtSig* object is supplied, the *DAPlugin* is responsible for populating the *numSrcSig* variable with the number of source signals.

Start *Start ()*

The *Start* function is called after initialization and before the runtime phase begins so that the *DAPlugin* can perform any operations necessary to start acquiring data. We include this functionality for generality to accommodate more complicated data acquisition scenarios, such as capturing real-time data streams.

Update	*Update ()*

Each time the *Update()* function is called, the *obsSig* variable is updated with a current block of data drawn from the *semObsExtSig*. The dimension of the block of data is *obsSig* [*N* x *semBlkLen*] where *N* is the number of signals defined in *semObsExtSig*. Each subsequent call of the *Update()* function increments the block by *semBlkStep* samples (or the equivalent number of samples if *semBlkStep* is given in seconds). If *semSrcExtSig* is supplied, the same update routine is conducted to populate the *srcSig* variable at each block.

Stop	*Stop ()*

The *Stop* function is typically called at the end of the runtime phase before shutdown but can be called at any time during the runtime phase. This function allows the *DAPlugin* to perform any necessary operations to cease acquiring data. We include this functionality for generality to accommodate more complicated data acquisition scenarios, such as capturing real-time data streams.

3.6.4 The Source Enumeration Plugin (*SNUMPlugin*)

(Abstract) ***SNUMPlugin***

Description: The *SNUMPlugin* is responsible for determining the number of active sources at each sample in each observation signal. Although we do not require the *SNUMPlugin* to be as precise as the initial description implies, we do require that any *SNUMPlugin* implementation follow the format of reporting a source number estimate at each sample of each observation signal. Therefore, the data that the *SNUMPlugin* reports is an [*N*x*P*] array of source number estimate values, where *N* is the length of the current data block and *P* is the number of observation signals, and the $(n,p)^{th}$ value is an estimate of the number of sources active at the n^{th} sample in the p^{th} observation. Although the number of sources active in a mixture is a positive integer (or infinity!), we let the data type of these source number estimates be a *Float*, since source enumeration is an estimation process, and real number values are

informative, e.g. 3.9 sources in a statistical sense helps inform the decision "probably 4 sources".

Update Fields Summary:

Field Name	Type	Set	Get	Description
srcNum	*Float []*	prot	pub	An estimate of the number of sources at each sample for each observation.

Common Fields: *pluginName, pluginVersion, pluginAbbr, pluginDescr, pluginStatus, paramSetId, massInfo, Init(), Update()*, *Shutdown()*

Requirement Fields: *reqSrcSig, reqBlkLenMin, reqBlkLenMax*

SEM Data Fields: *semObsSig, semSrcSig*

3.6.5 The Source Estimate Production Plugin (*SEPPlugin*)

(Abstract**)** ***SEPPlugin***

Description: The *SEPPlugin* is responsible for producing source estimates in the MASS framework, and methods for SS are implemented in this class of plugin. The *SEPPlugin* class has one update field of type *SourceEstimate*, where the *SEPPlugin* places its current source estimates for a block in the *sig* field and optionally the corresponding CDS into the *cds* field. If a CDS is not supplied, the MASS system assumes that the source estimates are BLTI, and a CDS will be generated using the *MASSInfo.CDSEstimator()* function. When multiple *SEPPlugin* implementations run simultaneously in MASS, the SEM collects all source estimates and populates two fields. The first field is a single *SourceEstimate* object, and all source estimates and CDS are placed in the *sig* and *cds* field, respectively. The second field is an array of *SourceEstimate* objects, where each object corresponds to the output of one *SEPPlugin*, i.e. the estimates are grouped according to the algorithms. MASS plugins requiring source estimates are provided with both fields, the first being supplied in the plugin's *semSrcEstArr* SEM data field, and the second is supplied via the *semSrcEstAlg* SEM data field.

Update Fields Summary:

Field Name	Type	Set	Get	Description
se	*SourceEstimate*	prot	pub	The current source estimates, and optional CDS.

Common Fields: *pluginName, pluginVersion, pluginAbbr, pluginDescr, pluginStatus, paramSetId, massInfo, Init(), Update()*, *Shutdown()*

Requirement Fields: *reqSrcNum, reqSrcSig, reqBlkLenMin, reqBlkLenMax*

SEM Data Fields: *semCDSFiltLen, semCDSFiltOffset, semObsSig, semSrcSig, semCDS, semSrcNum*

3.6.6 The Source Estimate Grouping Plugin (*SEGPlugin*)

(Abstract**)** ***SEGPlugin***

Description: The *SEGPlugin* is responsible for sorting an array of source estimates into groups where each group consists of the source estimates estimating one source. Each group of source estimates is placed in a *SourceEstimate* object, and the array of all groups' *SourceEstimate* objects constitutes the plugin's update field. The SEM will use the output of the *SEGPlugin* to populate the *semSrcEstGroup* SEM data field of any plugin requiring this data.

Update Fields Summary:

Field Name	Type	Set	Get	Description
seg	*SourceEstimate []*	prot	pub	An array of source estimates. Each *SourceEstimate* object contains a group of similar source estimates

Common Fields: *pluginName, pluginVersion, pluginAbbr, pluginDescr, pluginStatus, paramSetId, massInfo, Init(), Update()*, *Shutdown()*

Requirement Fields: *reqSrcNum, reqSrcSig, reqBlkLenMin, reqBlkLenMax*

SEM Data Fields: *semObsSig, semSrcSig, semSrcNum, semSrcEstArr, semSrcEstAlg*

3.6.7 The Source Estimate Time-Alignment Plugin (*SETAPlugin*)

(Abstract**)** ***SETAPlugin***

Description: The *SETAPlugin* class is tasked with time-aligning source estimates. Given that source estimates can be supplied as either a single signal array, i.e. *semSrctEstArr*, or as multiple grouped signal arrays, i.e. *semSrcEstGroup*, the *SETAPlugin* is required to time-align signals in both the *semSrctEstArr* and *semSrcEstGroup* source estimate sets. Signals will always be provided in *semSrctEstArr* field, but the *semSrcEstGroup* SEM data field will only be populated for configurations in which a *SEGPlugin* is employed, and the *SETAPlugin* is responsible for checking this field for data. The SEM uses the outputs of a *SETAPlugin* to populate the *semSrcEstArrTA* and *semSrcEstGrouptTA* SEM data fields. The *SETAPlugin* can also generate the corresponding CDSs for the newly time-aligned signal sets and populate the appropriate *cds* fields, but this is not required. We note that there is a third group of source estimates provided in the *semSrcEstAlg* variable, but currently, time-alignment for this group is unsupported.

Update Fields Summary:

Field Name	Type	Set	Get	Description
setaArr	*SourceEstimate*	prot	pub	A *SourceEstimate* object in which the source estimates from the *semSrctEstArr* SEM data field have been time-aligned.
setaGroup	*SourceEstimate []*	prot	pub	An array of *SourceEstimate* objects in which the source estimates from the *semSrcEstGroup* SEM data field have been time-aligned.

Common Fields: *pluginName, pluginVersion, pluginAbbr, pluginDescr, pluginStatus, paramSetId, massInfo, Init(), Update()*, *Shutdown()*

Requirement Fields: *reqSrcSig, reqBlkLenMin, reqBlkLenMax, reqSrcEstGroup*

SEM Data Fields: *semObsSig, semSrcSig, semSrcNum, semSrcEstArr, semSrcEstAlg, semSrcEstGroup*

3.6.8 The Source Estimate Evaluation Plugin (*SEEPlugin*)

(Abstract**)** ***SEEPlugin***

Description: The *SEEPlugin* is charged with evaluating the performance of a set of source estimates and producing a CSE set. An overview of the evaluation topic for MASS was given in Sect. 2.3, and any SEE method should be implemented in an *SEEPlugin* extension class.

Update Fields Summary:

Field Name	Type	Set	Get	Description
cse	*SourceEstimate*	prot	pub	A composite source estimate set.

Common Fields: *pluginName, pluginVersion, pluginAbbr, pluginDescr, pluginStatus, paramSetId, massInfo, Init(), Update()*, *Shutdown()*

Requirement Fields: *reqSrcNum, reqSrcSig, reqBlkLenMin, reqBlkLenMax, reqSrcEstGroup, reqSrcEstTA, reqSrcEstCDS*

SEM Data Fields: *semObsSig, semSrcSig, semSrcNum, semSrcEstArr, semSrcEstAlg, semSrcEstGroup, semSrcEstTA, semSrcEstGroupTA*

3.6.9 The Permutation Ambiguity Solution Plugin (*PASPlugin*)

(Abstract**)** ***PASPlugin***

Description: The *PASPlugin* resolves the permutation ambiguity in the source separation. Although it has access to all SEM source estimate data, its main task is to ensure that the order of source estimates from the *SEEPlugin*, i.e. the *semCSE* SEM data field, stays the same from block to block.

Update Fields Summary:

Field Name	Type	Set	Get	Description
cse	*SourceEstimate []*	prot	pub	A composite source estimate set, in which the permutation ambiguity has been resolved.

Common Fields: *pluginName, pluginVersion, pluginAbbr, pluginDescr, pluginStatus, paramSetId, massInfo, Init(), Update()*, *Shutdown()*

Requirement Fields: *reqSrcNum, reqSrcSig, reqBlkLenMin, reqBlkLenMax, reqSrcEstGroup, reqSrcEstTA, reqSrcEstCDS*

SEM Data Fields*:* *semObsSig, semSrcSig, semSrcNum, semSrcEstArr, semSrcEstAlg, semSrcEstGroup, semSrcEstTA, semSrcEstGroupTA, semCSE, semCDSPrev*

3.6.10 The Source Estimate Analysis Plugin (*SEAPlugin*)

(Abstract**)** ***SEAPlugin***

Description: In this version of the MASS framework, the *SEAPlugin* analyzes the current CSE set and returns a *Float []* array of values whose dimensions equal the dimensions of the current CSE. In future versions of the framework, we may expand the framework and define different types of analysis plugins, e.g. observation analysis, source analysis, source estimate evaluation analysis, general CDS analysis, etc., but for simplicity this version concentrates on analysis of the MASS composite source estimates output.

Update Fields Summary:

Field Name	Type	Set	Get	Description
sea	*Float []*	prot	pub	An array of analysis values with the same size as a block of the current CSE data

Common Fields: *pluginName, pluginVersion, pluginAbbr, pluginDescr, pluginStatus, paramSetId, massInfo, Init(), Update()*, *Shutdown()*

Requirement Fields: *reqSrcNum, reqSrcSig, reqBlkLenMin, reqBlkLenMax, reqSrcEstGroup, reqSrcEstTA, reqSrcEstCDS*

SEM Data Fields: *semObsSig, semSrcSig, semSrcNum, semSrcEstArr, semSrcEstAlg, semSrcEstGroup, semSrcEstTA, semSrcEstGroupTA, semCSE, semCDSPrev*

3.6.11 The Data Output Plugin (*DOPlugin*)

(Abstract**)** ***DOPlugin***

Description: The *DOPlugin* is tasked with outputting the various data produced by the MASS framework. As shown below, the *DOPlugin* has access to all SEM Data Fields, but in this version of the MASS framework, only the *semCSE, semCDS*, and *semSEA* fields are updated prior to calling the *DOPlugin*'s *Update()* method at each

block. Thus, the *DOPlugin* is responsible for delivering the CSE set, the associated CDS set, and any data provided by a *SEAPlugin* to the destination provided by the *doDestExtSig* object given in the configuration. The *DOPlugin* does not have any update fields, since it updates an external data destination at each call of the *DOPlugin*'s *Update()* method.

Common Fields: *pluginName, pluginVersion, pluginAbbr, pluginDescr, pluginStatus, paramSetId, massInfo, Init(), Update()*, *Shutdown()*

Requirement Fields: None

SEM Data Fields: *semSrcNum, semSrcEstAlg, semSrcEstGroup, semSrcEstTA, semSrcEstGroupTA, semCSE, semCDS, semSEA, semBlkLen, semBlkStep, semSampRate, semDataDestExtSig*

Specialized Functions: *Start(), Stop()*

Function Details:

Start	*Start ()*

The *Start* function is called after initialization and before the runtime phase begins so that the *DOPlugin* can perform any operations necessary to start outputting data. We include this functionality for generality to accommodate more complicated data output scenarios, such as real-time data streams.

Stop	*Stop ()*

The *Stop* function is typically called at the end of the runtime phase before shutdown but can be called at any time during the runtime phase. This function allows the *DOPlugin* to perform any necessary operations to cease outputting data. We include this functionality for generality to accommodate more complicated data output scenarios, such as real-time data streams.

3.7 Using the MASS Framework: The Immutable Source Estimate Management Class (*MASS_SEM*)

By far, the source estimate management class (*MASS_SEM)* represents the most complicated, fully implemented system in the MASS framework. However, since the *MASS_SEM* class cannot be modified by users and the *MASS_SEM* class only exposes a couple of functions, the description of the *MASS_SEM* class is very simple. The *MASS_SEM* class is the user's access to starting and stopping the MASS framework's signal processing workflow, thus an understanding of the *MASS_SEM* class is equivalent to understanding how to run the MASS framework for an application.

3.7.1 Overview of the *MASS_SEM* Workflow

In order to use the MASS system, the user must create an instance of the SEM, and SEM instantiation requires several important pieces of information: SEM signal processing parameters, a set of plugins, a data source for observations (mixtures), a data source for side-information, and a data destination for MASS outputs. In the configuration phase, the SEM assimilates and validates all of these configuring data to initialize the MASS system, and an overview of the configuration procedure is provided in Table 10.

The initialization phase allows the plugins to incorporate any information gleaned from the configuration phase into their respective processing tasks and methods before the runtime phase begins. Every plugin in the MASS system is required to have an initialization routine that

Table 10. Overview of the MASS configuration procedure. Once a SEM system has been instantiated, the SEM follows these steps to configure the MASS system.

Step	SEM Task Description
1	Check to see if SEP, SEE, PAS, DA, DO plugins are specified. *The MASS framework requires this minimum set of plugins.*
2	Gather external data source information
3	Gather MASS output destination information
4	Gather general signal processing parameters, e.g. data block length, sampling rate, etc.
5	Instantiate the SEPs, SEE, and PAS plugins, and gather the plugins' requirements, e.g. external data, data generated by other MASS plugins, minimum/maximum data block lengths, etc.
6	If supplied, instantiate the SNUM plugin and gather the plugin's requirements.
7	If supplied, instantiate the SEG plugin and gather the plugin's requirements.
8	If supplied, instantiate the SETA plugin and gather the plugin's requirements.
9	If supplied, instantiate the SEA plugin and gather the plugin's requirements.
10	Validate and adjust signal processing parameters, e.g. block length, CDO filter length, etc., based on all plugins' requirements.
11	Instantiate the DA plugin, supply the DA plugin with external data source information and relevant processing parameters, e.g. sampling rate, block length, etc.
12	Instantiate the DO plugin, supply the DO plugin with data destination information and relevant processing parameters, e.g. sampling rate, block length, etc.
13	Check that all required data sources and plugins have been supplied to meet all plugins' requirements.
13 a	If all needs have not been met, warn the user, and halt
13 b	Otherwise, update all plugins with a function that allows them to access all public MASS functionality and parameters, e.g. the CDO, block length, etc., and proceed to Initialization phase.

is accessible from the SEM. What a particular plugin does when that routine is called is wholly up to the plugin, and the point here is that the SEM *will* call a configuration-specified plugin's initialization function. In fact, the MASS initialization phase is entirely devoted to the SEM calling all specified plugins' initialization functions, thus making initialization the simplest phase, from the SEM's point of view. From a plugin's point of view, the initialization phase allows the plugin to modify any plugin-specific functionality or parameters using MASS system information specified in the

configuration phase, e.g. variable memory allocation, sampling rate dependent parameters, CDS specific parameters, etc., before the MASS runtime phase begins.

Once the MASS system has been configured and all relevant plugins have been initialized, the runtime phase begins, and the task of MASS is underway. At a basic level, the runtime phase consists of the SEM sequentially feeding blocks of data to the relevant sub-systems and acquiring their outputs. Similar to the SEM-accessible initialization function, all plugins in the MASS framework are required to have an "update" function accessible to the SEM. In the runtime phase, the SEM calls each relevant plugin's update function, the plugin performs its task, the plugin updates its outputs, and the SEM collects the plugin's product. As summarized in Table 11, the SEM calls each plugin in a very specific order at each data block, and this defines the runtime phase. Additionally, we use the phrase "*SEM Data*" to denote block- and plugin-specific data supplied to a particular plugin from the SEM at a particular data block. For example, the SEE plugin requires source estimates produced by the SEPs at each block, thus the SEM will supply said source estimates as *SEM Data* at each block. Furthermore, the DA and DO plugins have special "*Start"* and "*Stop*" functions, and these are built in for dealing with real-time DA and DO processes. All of these plugin-specific data requirements and specialized functions will be detailed later in this chapter.

The final phase of the MASS workflow is the shutdown phase. Once the runtime phase completes, the SEM simply iterates through the plugins and calls their respective *Shutdown()* functions. We provide this function for generality, so that the MASS system can exit cleanly by allowing each of the plugins to perform any cleanup tasks before the exit.

Table 11. Overview of the MASS runtime procedure. After configuration and initialization phases are complete, the SEM follows these steps to perform MASS.

Step	SEM Task Description
1	**Execute the DA and DO** *Start* functions
2	**While the DA plugin is acquiring data**, do steps **3-14** below:
3	**Call the DA plugin** *Update* function, and gather a new block of data from the DA plugin
4	If specified, **evaluate the SNUM plugin**: Pass *SEM Data* to the SNUM plugin. Call the SNUM plugin *Update* function. Collect the Source Enumeration estimate
5	For each SEP plugin, **evaluate each SEP plugin**: Pass *SEM Data* to the SEP plugin. Call the SEP plugin *Update* function. Collect the Source Estimates
6	**Estimate CDS** for any SEP plugins that provide Source Estimates without a CDS.
7	If MASS is configured to work in **Self-Competition mode**: Apply the last block's CDS to the current Observation data. Collect the Source Estimates.
8	If specified, **evaluate the SEG Plugin**: Pass *SEM Data* to the SEG plugin. Call the SEG plugin *Update* function. Collect the Grouped Source Estimates
9	If specified, **evaluate the SETA Plugin**: Pass *SEM Data* to the SETA plugin. Call the SETA plugin *Update* function. Collect the Grouped, Time-Aligned Source Estimates.
10	**Evaluate the SEE Plugin**: Pass *SEM Data* to the SEE plugin. Call the SEE plugin *Update* function. Collect the CSEs and CDS.
11	**Evaluate the PAS Plugin**: Pass *SEM Data* to the PAS plugin. Call the PAS plugin *Update* function. Collect the CSEs and CDS.
12	**Evaluate the SEA Plugin**: Pass *SEM Data* to the SEA plugin. Call the SEA plugin *Update* function. Collect Source Estimate Analysis data.
13	**Evaluate the DO Plugin**: Pass *SEM Data* to the DO plugin. Call the DO plugin *Update* function.
14	**Repeat** process starting at **2**
15	**Execute the DA and DO** *Stop* functions
16	**Exit**

3.7.2 *MASS_SEM* Class Definition

(Sealed) ***MASS_SEM***

Description: The *MASS_SEM* class coordinates all of the efforts of the various plugins in the MASS framework. In order to use the MASS framework, a user must create and configure an instance of the *MASS_SEM* class, and then call the *MASS_SEM's Start()* function.

Specialized Functions: *Start(), Stop()*

Constructors: *SEM*(*PluginInst*), *SEM*(*Configuration*)

Function Details:

MASS_SEM	*MASS_SEM* (*Configuration*)

This constructor takes a *Configuration* object as an input. The *Configuration* object must contain a valid set of configuring information for the MASS framework to operate. Once the configuring information is acquired, the procedure in Table 10 is executed to validate and instantiate the configuration information.

MASS_SEM	*MASS_SEM* (*PluginInst*)

This constructor takes a *PluginInst* object that contains the name and parameter set id for a valid *ConfigPlugin* object. As soon as the *MASS_SEM* object is instantiated via this constructor, the designated *ConfigPlugin* is called to obtain a valid *Configuration* object. The *Configuration* object must contain a valid set of configuring information for the MASS framework to operate. Once the configuring information is acquired, the procedure in Table 10 is executed to validate and instantiate the configuration information.

Start	*Start ()*

Once the *MASS_SEM* object is instantiated, the user needs to call the *Start()* function in order to begin the MASS signal processing. Once the *Start()* function is called, the initialization procedure is run (see Sect. 3.2), and then the runtime procedure

in Table 11 is executed. The runtime phase continues until a shutdown signal is received, e.g. when the end of a finite dataset is reached, an error occurs, the *MASS_SEM's Stop()* function is called, etc. Once the shutdown signal is received, the *MASS_SEM* runs the shutdown phase procedure (see Sect. 3.2) before exiting.

Stop	*Stop ()*

The *Stop()* function can be called after the *Start()* function has been called to initiate the shutdown phase (see Sect. 3.2) of the processing. After the shutdown phase is complete, *MASS_SEM* will exit. The *Stop()* function has no affect if the *Start()* function has not been called.

CHAPTER 4: A PLUGIN COMPONENT LIBRARY FOR MASS

Chapter 3 specified a white-box framework for MASS without reference components, and this chapter defines a set of plugin components that can either be construed as the completion of a basic white-box framework, or as a basic plugin component library necessary for a rudimentary black-box framework. In this chapter, the semantics of the white-/black- box distinction are somewhat irrelevant, since we simply provide concrete examples of how to create useful plugin components for the MASS task. However, we will treat the components in this chapter as if they are plugin components in a black-box framework, thus a developer or user only needs understand the underlying conceptual methods employed in the components. We will study the *Configuration* class in detail in Chapter 5, and here we will confine our attention to concrete examples of the plugin classes that developers can utilize "out of the box" to create their applications.

To expedite usage of this document, we further confine the scope of this chapter to plugin components that are utilized in the Chapter 5 cookbook, thus not all abstract plugin classes have components in the forthcoming library presented in this chapter. We note that the limited component library presented in this chapter covers all of the *required* plugins as given in Chapter 3, i.e. the DA, DO, SEP, SEE, and PAS plugins. Although we do not provide SEG and SETA plugin components in this work, the plugin components that we do provide in this chapter and the configurations given in Chapter 5 do not utilize SEG or SETA plugins. Thus, the immutable classes, the plugin component library presented in this chapter and the configuration component library given in Chapter 5 define a barebones, but usable, black-box framework for MASS.

The plugin components given in this chapter are black-boxes themselves, and as such, the user of this document only needs to understand the fundamental method being implemented in a given component along with any constraints of applicability and any parameters used to implement that method. Thus, each section will contain a "Description" section that gives a summary of the method being implemented by a component followed by the class constructor and a table of common plugin fields' values used to programmatically describe a plugin, i.e. *pluginName*, *pluginVersion, pluginAbbr*, and *pluginDescr*. After that, each section will include a table of the *requirement* fields' values of the particular component where appropriate (not all plugins have requirement

fields) so the user can determine applicability. Each plugin component section will explain all parameters used by the component and then provide a table that summarizes the parameters, followed by a description of the parameter sets produced by the constructor argument, *ParamSetId*. When a plugin has no parameters, any integer will suffice for the *ParamSetId* constructor variable, and we leave this structure in place for possible future development.

This document is both a dissertation and a user manual for the MASS framework. The section titles in this chapter have the format " ([*extended class*]) [*component name*] [: *descriptor*]", where [*component name*] is the name of the plugin component to be described, [*extended class*] is the name of the abstract plugin class to which the component belongs, and [: *descriptor*] indicates whether the method is blind, semi-blind, supervised, or not-applicable (empty). This notation allows for easy reference in the table of contents for framework users.

4.1 (*DAPlugin*) DA_FileRead

DA_FileRead Extends *DAPlugin*

Description:

The data acquisition plugin (*DAPlugin*) *DA_FileRead* reads data from files specified in the *semObsExtSig* and *semSrcExtSig ExternalSignal* objects contained in the configuration. The *DA_FileRead* component ensures that signals contained in the *semObsExtSig* and *semSrcExtSig* files are sampled at *semSampRate*, also contained in configuration. The *DA_FileRead* component reads into memory all data contained in the files specified by *semObsExtSig* and *semSrcExtSig*, but only delivers time-incremented blocks of data to the SEM with each *Update()* function call according to the *DAPlugin* specification.

Constructor: *DA_FileRead (ParamSetId)*

Parameters: *None*

Parameter Sets: *None*

Common Fields:

Field Name	Value
pluginName	Data Acquisiton from Files
pluginVersion	1.0.1
pluginAbbr	DA_FileRead
pluginDescr	Acquires data from audio or mat files

Requirement Fields: *None*

4.2 (*DOPlugin*) DO_FileWrite

DO_FileWrite Extends *DOPlugin*

Description:

The data output plugin (*DOPlugin*) *DO_FileWrite* saves into a .mat file the *semCDS*, *semCSE*, and *semSEA* data at each block, as well as the configuration information used to produce these data. The .mat file location and name is defined in the *semDestExtSig* configuration variable. This component is very simple, in that it concatenates successive blocks of *semCSE* values into a single array that is saved out in the *CSE* variable in the .mat file. This component also concatenates the *semSEA* data into a single *SEA* array in the .mat file. Each CDS at every block is stored in a structure array *CDS* where *CDS(m).values* contains the CDS of the m^{th} block. The configuration is saved out in a structure array *CFG* variable whose structure is the same as the *Configuration* object used to produce the data.

In future versions of this component, the output .mat file will be modified to contain at least these three variables: *CSE*, *Block*, and *CFG*. The *CSE* and *CFG* variables will be the same as described above. The *Block* variable will be a structure array with at least four fields: *CSE, CDS, OBS,* and *SEA*. The m^{th} entry in *Block*, i.e. *Block(m)*, will contain the m^{th} data block's CSE, CDS, observation data (*OBS*), and *SEAPlugin* results (*SEA*) as *Block(m).CSE, Block(m).CDS, Block(m).OBS,* and *Block(m).SEA*, respectively. This will allow the *SEAPlugin* to output more general (dimensioned) data than currently given in the *SEAPlugin*'s definition and allow precise post-processing analysis by providing the exact samples of data needed to precisely recreate the MASS outputs at each block. The *Block* variable also allows for other types of block data unforeseen at

this moment to be added in the future, and we note that this document provides a "beta" overview of the MASS framework. We will discuss even more opportunities for MASS framework improvement in Chapter 6.

Constructor: *DO_FileWrite (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	Data Output to a MAT File
pluginVersion	1.0.1
pluginAbbr	DA_FileWrite
pluginDescr	Writes MASS configuration, CDS, CSE, and SEA data to a .mat file

Requirement Fields: *None*

Parameters: *None*

Parameter Sets: *None*

4.3 (*SNUMPlugin*) SNUM_NumObs : Blind

SNUM_NumObs Extends *SNUMPlugin*

Description:

The source enumeration plugin (*SNUMPlugin*) *SNUM_NumObs* assumes a critically-determined source number, *Q*, to sensor number, *P*, scenario and simply reports the number of sources as the number of observation signals, i.e. *Q=P*. The *srcNum* field is an array the size of *semObsSig* populated with the sole value *P* which is the second dimension of the *semObsSig* variable. Thus, this naïve method is blind and assumes all sources are active in every channel of the observations at all times.

Constructor: *SNUM_NumObs (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	SNUMPlugin that assumes critically-determined mixtures
pluginVersion	1.0.1
pluginAbbr	SNUM_NumObs
pluginDescr	Estimates the number of sources as the number of observations (blind, critically-determined)

Requirement Fields:

Field Name	Value	Comment
reqSrcSig	0	Blind method, does NOT require source signals
reqBlkLenMin	[]	Minimum data-block length is unspecified
reqBlkLenMax	[]	Maximum data-block length is unspecified

Parameters: None

Parameter Sets: None

4.4 (*SEPPlugin*) SEP_Static

SEP_Static Extends *SEPPlugin*

Description:

The source estimate production plugin (*SEPPlugin*) *SEP_Static* applies an identity CDS to the observation data, thus passing through the observation data as a CSE set for further processing in the MASS framework. Although conceptually simple, the *SEP_Static* plugin does take the (computational) time to apply the identity CDS contained in *se.cds* to the observation data via the plugin's *massInfo.CDO* operation to produce the *se.cse* values. Although this method could have been named "*SEP_Identity*", we note that this particular plugin component is extraordinary in the MASS framework for several reasons: *SEP_Static* gives the MASS framework a simple, top-level debugging mechanism, *SEP_Static* allows the MASS framework to be used solely for signal analysis, *SEP_Static* is the only *component* that is meant to be extended by users, and the *SEP_Static* component concisely defines MASS block-processing concepts for use in future development of the MASS framework. For the first reason, we simply note that a user can designate any signal set as the observation data and then design a configuration to study a particular set of plugins' behaviors, and for the second reason, we devote Sect. 5.3 to exploring a couple of signal analysis applications.

Thirdly, the *SEP_Static* plugin is abnormal in the component library, since we envision users extending the *SEP_Static* plugin component and modifying the *se.cds* variable with a particular CDS that will be applied at every block to the observation data. In this way, a user can apply a pre-defined CDS to the observations and allow MASS to

determine if this CDS is ever "the best" in terms of a particular configuration. In this manner, we give the user a method to apply static solutions to the data and study the MASS solution.

Furthermore, the *SEP_Static* method can describe the MASS framework's application of a CDS to the observation data via a CDO concisely. That is, any method that uses this operation, e.g. a SEP component, the MASS CSE, etc., can be described as a function that determines a CDS followed by a *SEP_Static* method that applies the CDS to a block of data. The *SEP_Static* method, applied at the block-level, encapsulates the block-processing concept of MASS, and future development of the MASS framework will use *SEP_Static* as a building block.

Constructor: *SEP_Static (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	SEP Static Filter
pluginVersion	1.0.1
pluginAbbr	*SEP_Static*
pluginDescr	Static filtering of observation data

Requirement Fields:

Field Name	Value	Comment
reqSrcNum	0	Method does NOT require an *SNUMPlugin*
reqSrcSig	0	Blind method, does NOT require source signals
reqBlkLenMin	-1	Minimum data-block length is 1s
reqBlkLenMax	inf	Maximum data-block length is unspecified

Parameters: None

Parameters *Sets:* None

4.5 (*SEPPlugin*) SEP_SupSysId : Supervised

SEP_SupSysId Extends *SEPPlugin*

Description:

The source estimate production plugin (*SEPPlugin*) *SEP_SupSysId* is a supervised method that estimates each of the source signals as a linear combination of the observation signals via the *massInfo.SysIdOp* (see Sects. 1.2.3 and 3.4.3) function. This

estimation of the source signals from the observation signals uses all CDO properties, e.g. filter length, filter offset, etc., defined for the CDO in the configuration. The CDS produced by *massInfo.SysIdOp* populates the *se.cds* variable, and the CSE produced by the application of the *massInfo.CDO* function to the observation data with *se.cds* as input populates the *se.cse* variable, at each block.

Constructor: *SEP_SupSysId(ParamSetId)*

Common Fields:

Field Name	Value
pluginName	Supervised FIR Deconvolution
pluginVersion	1.0.1
pluginAbbr	*SEP_SupSysId*
pluginDescr	Uses System Identification Operator with known sources to produce source estimates.

Requirement Fields:

Field Name	Value	Comment
reqSrcNum	0	Method does NOT require an *SNUMPlugin*
reqSrcSig	1	Supervised method, requires source signals
reqBlkLenMin	-2	Minimum data-block length is 2s
reqBlkLenMax	inf	Maximum data-block length is unspecified

Parameters: None

Parameter Sets: None

4.6 (*SEPPlugin*) SEP_ABYK : Blind

SEP_ABYK Extends *SEPPlugin*

Description:

The source estimate production plugin (*SEPPlugin*) *SEP_ABYK* implements the BSS method given by Aichner et al. in [95], with a slight modification. We use the shorthand notation of "ABYK" when referring to the algorithm (each letter is the first letter of the authors' last names in [95]). Given our historical perspective of BSS in Sect. 1.3.1, we classify this BSS algorithm as a convolutive, natural gradient descent for output mutual information minimization under a Gaussian source assumption with a non-

holonomic constraint. Any user of this work should refer to [95] for a more detailed explanation of ABYK.

Constructor: *SEP_ABYK (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	SEP via the ABYK algorithm
pluginVersion	1.0.1
pluginAbbr	*SEP_ABYK*
pluginDescr	Convolutive BSS via Natural Gradient of SOS

Requirement Fields:

Field Name	Value	Comment
reqSrcNum	0	Method does NOT require an *SNUMPlugin*
reqSrcSig	0	Blind method, does NOT require source signals
reqBlkLenMin	-2	Minimum data-block length is 2s
reqBlkLenMax	inf	Maximum data-block length is unspecified

Parameters:

The ABYK algorithm is a block-processing method, and for a given "composite" block of data (not to be confused with the MASS data block), the ABYK method further subdivides the composite block into K sub-blocks of N samples each. The ABYK algorithm tries to minimize the cross-correlation function values (using N samples) between each pair of output signals over L time-lags. The method calculates a (natural) gradient descent update of the demixing solution at each sub-block and averages the updates over the K sub-blocks. The composite block demixing solution is updated as in

Parameter	Description	Range	Default
L	The filter length in samples of the demixing solution and the number of lags over which to de-correlate outputs.	$L \geq 1$	1024
FLTDLY	The delay applied in the initial (identity) demixing solution	$FLTDLY \in [0, L-1]$	512
N	The number of samples used in the output cross-correlation calculation and the number of samples in a sub-block (potentially overlapping with other sub-blocks).	$N \geq L$	$2L$
K	The number of sub-blocks.	$K \geq 1$	4
T	The number of samples to step between successive composite blocks of data	$T > 0$	KL
MU	The composite block demixing solution natural gradient update stepsize.	$MU \in [0,1]$	0.04
JMAX	The number of iterations to perform composite block gradient updates.	$JMAX \geq 1$	7
LAMBDA	Scalar convex weighting of current demixing solution with past block's demixing solution.	$LAMBDA \in [0,1]$	0.6
SIGMA0	Stability factor / noise floor	$SIGMA0 \geq 0$	0.01
DMAX	Stability factor	$DMAX \geq 0$	1

gradient descent by adding the past (iteration) demixing solution with the averaged sub-block update weighted by a stepsize *MU*. The updated demixing solution is applied to the observation data to produce outputs, and these outputs are input into the sub-block level natural gradient descent and composite demixing solution update, as just described above, for *JMAX* iterations. At this point, the algorithm updates the global demixing solution as the weighted sum of the last block's composite demixing solution and the current block's composite demixing solution, i.e.,

$$\boldsymbol{W}(m) = \lambda \boldsymbol{W}(m-1) + (1-\lambda)\widetilde{\boldsymbol{W}}(m),$$

where $\boldsymbol{W}(m)$ is the demixing solution at block m, λ (parameter *LAMBDA*) is a scalar forgetting factor, and $\widetilde{\boldsymbol{W}}(m)$ is the final demixing solution produced after iterating over the sub-blocks *JMAX* times. The algorithm then steps through the data by *T* samples and proceeds to the next, $(m+1)^{th}$, block of data.

During the update calculation of the sub-block demixing solution, the cross-correlation values used to update estimates for the q^{th} output are normalized by $\sigma_q^2 + \delta_q$, where σ_q^2 is the power of the q^{th} output and δ_q is a stability term given as,

Parameter Set Id	Parameter Values	*SEP Mode*
0	Given by *SEP_ABYK_params_default()* function	Competitive
1	Given by *SEP_ABYK_params()* function	Competitive
2	Given by *SEP_ABYK_params()* function, except for the following sample-valued parameters derived from the *Configuration*: *L = semCDSFiltLen* *N = 2 semCDSFiltLen* *FLTDLY = semCDSFiltOffset* *T = K semCDSFiltLen / 2*	Competitive
10	Same as Parameter Set 0	Cooperative
11	Same as Parameter Set 1	Cooperative
12	Same as Parameter Set 2	Cooperative

$$\delta_q = \delta_{max} e^{-\sigma_q^2/\sigma_0^2}.$$

When the power of the output is low and approaches or is less than σ_0^2 (parameter *SIGMA0*), the δ_q approaches or equals the value in δ_{max}. This mechanism relieves the normalization from a divide-by-zero instability problem for low power sections of the signal, but also has the function of declaring a noise floor level, so that updates produced by signals with power that are approximatley σ_0^2 will have little influence on the overall demixing update.

Parameter Sets

The *SEP_ABYK* plugin has three fundamental parameter sets whose parameter values are either fully or partially defined by the external functions *SEP_ABYK_params_default()* and *SEP_ABYK_params()*. The *SEP_ABYK_params_default()* is a non-modifiable function that returns a structure array with fields names given by the parameter names in the table above whose corresponding values are the default values given in the table above. The *SEP_ABYK_params()* function also returns a structure array of parameter values, but the user can modify this function to provide their own specific parameter values. Without modification, *SEP_ABYK_params()* function returns the values in *SEP_ABYK_params_default()*. A third parameter set uses configuration information to populate all of the sample-valued parameters and put the *SEP_ABYK* solution in the CDS format. All other parameters are populated from the *SEP_ABYK_params()* function.

Using the three general parameters sets above, the *SEP_ABYK* method can work either in a competitive or cooperative manner. Thus, the *SEP_ABYK*, has six parameter sets that are summarized in the table below.

4.7 (*SEPPlugin*) SEP_MCLP : Blind

SEP_MCLP Extends *SEPPlugin*

Description:

The source estimate production plugin (*SEPPlugin*) *SEP_MCLP* implements the convolutive blind source extraction (BSE) method given by Delcroix et al. in [45] and [46]. This underlying method is a *multi-channel* version of the linear prediction problem (see Sect. 1.2.3) and was originally intended as a blind *dereverberation* method for an acoustic source in the presence of a compact noise source. However, the method is actually a BSE method under certain source and sensor location geometries. In particular, the *SEP_MCLP* can extract a source of interest (SOI) if the SOI is closer to a particular sensor than any other source, and all the other sources are closer than the SOI to all the other sensors.

Constructor: *SEP_MCLP (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	SEP via Multi-Channel Linear Prediction
pluginVersion	1.0.1
pluginAbbr	*SEP_MCLP*
pluginDescr	Potential Source Extraction via Multi-Channel Linear Prediction

Requirement Fields:

Field Name	Value	Comment
reqSrcNum	0	Method does NOT require a *SNUMPlugin*
reqSrcSig	0	Blind method, does NOT require source signals
reqBlkLenMin	-2	Minimum data-block length is 2s
reqBlkLenMax	inf	Maximum data-block length is unspecified

Parameters

If we let $\boldsymbol{x_p}(n-1) = [x_p(n-1), \ldots x_p(n-L)]^T$ be a vector of the past L samples of the p^{th} observation signal, $x_p(n)$, then we can construct a vector consisting of the past L samples of all P observation signals as $\boldsymbol{x}(n-1) = Vec\{\boldsymbol{x}_1(n-1),\ldots, \boldsymbol{x}_P(n-1)\}$. For the p^{th} observation signal, multi-channel linear prediction (MCLP) then aims to minimize the prediction error power, where the prediction error is given by,

$$e_p(n) = x_p(n) - \boldsymbol{w}_p^T \boldsymbol{x}(n-1), \quad p = 1, \ldots, P,$$

where $\boldsymbol{w}_p = Vec\{\boldsymbol{w}_{1p}, \ldots, \boldsymbol{w}_{Pp}\}$ is a vector of prediction filter coefficients, $\boldsymbol{w}_{ip}$ is the L-length filter applied to $\boldsymbol{x_i}(n-1)$ for $i = 1, \ldots P$. We denote the filter length, L, as the parameter *filtLen* in the *SEP_MCLP* plugin. The error signal, $e_p(n)$, is a whitened version of the p^{th} estimate produced by the *SEP_MCLP* plugin.

The precise de-whitening method given in [45] is computationally expensive and impractical for complicated multi-source scenarios, thus we have included a simplified de-whitening method. To begin we image the p^{th} error signal onto the p^{th} observation via the *massInfo* imaging operation, $\tilde{e}_p(n) = massInfo.ImgOp\big(e_p(n), x_p(n), siFiltLen, siFiltLag\big)$ where *siFiltLen* is the filter length of the imaging filter, and *siFiltLag* is an offset (delay) applied to $e_p(n)$ during the estimation process to ensure a causal solution. We then estimate an *arOrder*-length prediction filter, $a_p(\kappa)$, for $\tilde{e}_p(n)$ to produce the poles of an autoregressive spectrum, i.e. $a_p(\kappa) \overset{z}{\Leftrightarrow} A_p(z)$, where $A_p(z)$ is the z-transform of the prediction filter. We then apply the magnitude of the autoregressive spectrum to the whitened error signal to produce the p^{th} source estimate, i.e., $y_p(n) \overset{z}{\Leftrightarrow} E_p(z)/\big|A_p(z)\big|$, where $y_p(n)$ is the p^{th} source estimate and $E_p(z)$ is the z-transform of $e_p(n)$.

Parameter	Description	Range	Default
filtLen	Prediction filter length in samples	*filtLen* > *0*	400
arOrder	Number of poles in estimated AR spectrum for de-whitening	*arOrder* > 0	20
siFiltLen	Imaging filter length in samples in de-whitening process	*siFiltLen* > 0	200
siFiltLag	Number of samples to delay error signal in imaging for de-whitening process	0 ≤ *siFiltLag* < *siFiltLen*	20

Parameter Sets

The non-modifiable *SEP_MCLP_params_default()* function returns a structure array whose fields correspond to the parameter names given above and whose corresponding values are the default values given in the table above. The user-modifiable *SEP_MCLP_params()* function returns a structure array of parameters whose values are populated by a user. Without modification, the *SEP_MCLP_params()* function returns the values from *SEP_MCLP_params_default()*.

Parameter Set Id	Parameter Values	SEP Mode
0	Given by *SEP_MCLP_params_default()* function	Competitive
1	Given by *SEP_MCLP_params()* function	Competitive

4.8 (*SEPPlugin*) SEP_TTSE : Blind

SEP_TTSE Extends *SEPPlugin*

Description:

The source estimate production plugin (*SEPPlugin*) *SEP_TTSE* implements a convolutive blind source extraction (BSE) method for sources that intermittently become silent. The method is dubbed the turn-taking source extraction (TTSE) method for the way intermittent sources tend to take turns, e.g. conversational speech. The method makes many assumptions, but crucially, when the mixing system is critically- or over-determined with an intermittent source, a source extraction solution for the intermittent source is available via a system identification problem and a full discussion of the method is given in Appendix A.

Constructor: *SEP_TTSE (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	Source Extraction for Intermittent Sources
pluginVersion	0.0.1
pluginAbbr	*SEP_TTSE*
pluginDescr	Source Extraction when some sources are intermittently silent

Requirement Fields:

Field Name	Value	Comment
reqSrcNum	0	Method does NOT require a *SNUMPlugin*
reqSrcSig	0	Blind method, does NOT require source signals
reqBlkLenMin	-2	Minimum data-block length is 2s
reqBlkLenMax	inf	Maximum data-block length is unspecified

Parameters:

The TTSE method (see Appendix A) divides the data block into sub-blocks, and produces multiple candidate source extraction solutions at each sub-block. The size of a sub-block, in seconds, is contained in the parameter *reqWinLen*, and *reqWinStep* denotes the interval, in seconds, at which successive sub-blocks are evaluated, i.e. a rectangular window of size *reqWinLen* stepping through the data at *reqWinStep* intervals.

The TTSE method also contains two other significant parameters that are currently exposed through the *Configuration*, but are not directly available for *SEP_TTSE*-specific modification: the extraction solution filter length (see Appendix A) and the delay introduced for causal extraction solutions (see Sect. A.3). The filter length is given by the *semCDSFiltLen* variable and the delay term is given by the *semCDSFiltOffset* variable, where *semCDSFiltLen* and *semCDSFiltOffset* are specified in a valid *Configuration* object used to configure the MASS framework usage.

Parameter	Description	Range	Default
reqWinLen	Sub-block length, in seconds	$reqWinLen > 1$	1
reqWinStep	A delay to ensure causal solutions, in seconds	$reqWinStep \geq 0$	0.25

Parameter Sets

Similar to the *SEP_ABYK* and *SEP_MCLP* plugins, the *SEP_TTSE* plugin has two utility functions that return a structure array with field names given by the parameter names in the above table. *SEP_TTSE_params_default()* is an unmodifiable function that returns the parameter structure array with the default values given in the table above. *SEP_TTSE_params()* is a user-modifiable function that returns a parameter structure array with user-specified parameter values, and without modification, *SEP_TTSE_params()* returns the default parameter values.

Parameter Set Id	Parameter Values	SEP Mode
0	Given by *SEP_TTSE_params_default()* function	Competitive
1	Given by *SEP_TTSE_params()* function	Competitive

4.9 (*SEEPlugin*) SEE_Identity

SEE_Identity Extends *SEEPlugin*

Description:

The source estimate production plugin (*SEEPlugin*) *SEE_Identity* takes a set of source estimates as input and passes those source estimates to the output. The *SEE_Identity* plugin is useful for evaluating the performance of an individual *SEPPlugin*, *SEGPlugin*, or *SETAPlugin*, or some combination of those plugins, by removing the *SEEPlugin* from the analysis.

Constructor: *SEE_Identity(ParamSetId)*

Common Fields:

Field Name	Value
pluginName	Identity (Pass Through) SEE Plugin
pluginVersion	1.0.1
pluginAbbr	SEE_Identity
pluginDescr	Passes source estimates from input to output

Requirement Fields:

Field Name	Value	Comment
reqSrcEstGroup	0	Method does not require a *SEGPlugin*
reqSrcEstTA	0	Method does not require a *SETAPlugin*
reqSrcEstCDS	0	Method does not require that the CDS for each source estimate be supplied
reqSrcNum	0	Method does NOT require an *SNUMPlugin*
reqSrcSig	0	Blind method, does NOT require source signals
reqBlkLenMin	[]	Minimum data-block length is unspecified
reqBlkLenMax	[]	Maximum data-block length is unspecified

Parameters: None

Parameter Sets: None

4.10 (*SEEPlugin*) SEE_SupSIRSelect : Supervised

SEE_SupSIRSelect Extends *SEEPlugin*

Description:

The source estimate evaluation plugin (*SEEPlugin*) *SEE_SupSIRSelect* is a supervised method that selects the source estimates based upon SIR (see Sect. 1.3.6). That is, for a set of Q known sources and a set of source estimates (SEs), the *SEE_SupSIRSelect* plugin evaluates the SIR for each known source in each SE and then passes through the Q SEs that provide the maximum SIR for each of the known sources. The method assumes that (only) Q sources are present in any CSE set, thus it always produces a CSE set containing Q source estimates.

Constructor: *SEE_SupSIRSelect (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	Selection based SEE via SIR Evaluation
pluginVersion	1.0.1
pluginAbbr	SEE_SupSIRSelect
pluginDescr	Selection based SEE via SIR Evaluation

Requirement Fields:

Field Name	Value	Comment
reqSrcEstGroup	0	Method does not require a *SEGPlugin*
reqSrcEstTA	0	Method does not require a *SETAPlugin*
reqSrcEstCDS	0	Method does not require that the CDS for each source estimate be supplied
reqSrcNum	0	Method does NOT require an *SNUMPlugin*
reqSrcSig	1	Supervised method, requires source signals
reqBlkLenMin	-1	Minimum data-block length is 1s
reqBlkLenMax	inf	Maximum data-block length is unspecified

Parameters:

The *SEE_SupSIRSelect* plugin estimates SIR, e.g. (79), using source images estimated from the imaging operator, e.g. (80). The imaging operator is implemented via the *massInfo* object's *ImgOp()*, thus the estimated image of the q^{th} source in the i^{th} source estimate is $\tilde{\boldsymbol{s}}_{qi}$=*massInfo.ImgOp (*$\boldsymbol{s}_q$*,* $\boldsymbol{y}_i$*, filtLen, filtOffset)* where $\boldsymbol{s}_q$ are the

samples of the q^{th} source, $\boldsymbol{y}_i$ are the samples of the i^{th} source estimate, *filtLen* is the imaging filter length, and *filtOffset* is a delay applied $\boldsymbol{s}_q$ to ensure a causal solution. The source samples are provided in *semSrcSig* and the source estimates are provided in *semSrcEstArr*.

Parameter	Description	Range	Default
filtLen	Imaging filter length, in samples	*filtLen* > 1	*semCDSFiltLen*
filtOffset	A delay to ensure causal solutions, in samples	*filtOffset* ≥ 0	50

Parameter Sets:

The *SEE_SupSIRSelect_params_default()* function provides the default value for *filtOffset*, and the *SEE_SupSIRSelect_params()* function allows users to modify the *filtOffset* value. Although the *SEE_SupSIRSelect_params_default()* and the *SEE_SupSIRSelect_params()* contain a field for *filtLen*, the current version uses a parameter set dependent multiple of *semCDSFiltLen* as the *filtLen* value.

Parameter Set Id	Parameter Values	SEE Mode
0	*filtLen* = *semCDSFiltLen* *filtOffset* is given by *SEE_SupSIRSelect_params()*	Block
1	*filtLen* = 0.5 x *semCDSFiltLen* *filtOffset* is given by *SEE_SupSIRSelect_params()*	Block
2	*filtLen* = 2 x *semCDSFiltLen* *filtOffset* is given by *SEE_SupSIRSelect_params()*	Block
3	*filtLen* = 3 x *semCDSFiltLen* *filtOffset* is given by *SEE_SupSIRSelect_params()*	Block
4	*filtLen* = 4 x *semCDSFiltLen* *filtOffset* is given by *SEE_SupSIRSelect_params()*	Block
10	*filtLen* = *semCDSFiltLen* *filtOffset* is given by *SEE_SupSIRSelect_params()*	Batch
11	*filtLen* = 0.5 x *semCDSFiltLen* *filtOffset* is given by *SEE_SupSIRSelect_params()*	Batch
12	*filtLen* = 2 x *semCDSFiltLen* *filtOffset* is given by *SEE_SupSIRSelect_params()*	Batch
13	*filtLen* = 3 x *semCDSFiltLen* *filtOffset* is given by *SEE_SupSIRSelect_params()*	Batch
14	*filtLen* = 4 x *semCDSFiltLen* *filtOffset* is given by *SEE_SupSIRSelect_params()*	Batch

Furthermore, the *SupSIRSelect* plugin operates in two modes, block and batch. In the block mode, the SIR is calculated using the data-block's source and source estimate

samples. For finite length signals, e.g. audio files, the SIR can be calculated in batch mode by using *all* samples of the source signal as well as using the source estimates produced by applying the CDSs to *all* samples of the observations. The batch mode is applicable when the mixing system is considered time-invariant. The batch mode is supplied to produce more accurate estimates.

4.11 (*SEEPlugin*) SEE_MinXCorrSelect : Semi-Blind

SEE_MinXCorrSelect Extends *SEEPlugin*

Description:

The source estimate evaluation plugin (*SEEPlugin*) *SEE_MinXCorrSelect* is a blind method that selects a set of sources such that the maximum cross-correlation (over a range of lags) between each pair of source estimates is minimized. The method is semi-blind, because it requires that the number of sources be known or estimated, thus any valid *Configuration* using the *SEE_MinXCorrSelect* plugin must also include a source enumeration plugin (*SEEPlugin*).

Constructor: *SEE_MinXCorrSelect (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	Selection based SEE via Minimum Source Estimate Cross-correlation
pluginVersion	1.0.1
pluginAbbr	*SEE_MinXCorrSelect*
pluginDescr	Selection based SEE via Minimum Source Estimate Cross-correlation

Requirement Fields:

Field Name	Value	Comment
reqSrcEstGroup	0	Method does not require a *SEGPlugin*
reqSrcEstTA	0	Method does not require a *SETAPlugin*
reqSrcEstCDS	0	Method does not require that the CDS for each source estimate be supplied
reqSrcNum	1	Semi-Blind method, requires an *SNUMPlugin*
reqSrcSig	0	Source signals are NOT source signals
reqBlkLenMin	-2	Minimum data-block length is 2s
reqBlkLenMax	-60	Maximum data-block length is 60s

Parameters: None

Parameter Sets: None

4.12 (*PASPlugin*) PAS_Identity

PAS_Identity Extends *PASPlugin*

Description:

The permutation ambiguity solution plugin (*PASPlugin*) *PAS_Identity* simply passes the CSEs given as input directly to the output. The *PAS_Identity* plugin is useful for evaluating the performance of an individual *SEPPlugin*, *SEGPlugin*, *SETAPlugin*, *SEEPlugin* or some combination of those plugins, by removing the *PASPlugin* from the signal processing and analysis.

Constructor: *PAS_Identity (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	Identity (Pass Through) PAS Plugin
pluginVersion	1.0.1
pluginAbbr	PAS_Identity
pluginDescr	Passes source estimates from input to output

Requirement Fields:

Field Name	Value	Comment
reqSrcEstGroup	0	Method does not require a *SEGPlugin*
reqSrcEstTA	0	Method does not require a *SETAPlugin*
reqSrcEstCDS	0	Method does not require that the CDS for each source estimate be supplied
reqSrcNum	0	Method does not require a *SNUMPlugin*
reqSrcSig	0	Blind method, source signals are NOT required
reqBlkLenMin	[]	Minimum data-block length is unspecified
reqBlkLenMax	[]	Maximum data-block length is unspecified

Parameters: None

Parameter Sets: None

4.13 (*PASPlugin*) PAS_CSEPrevXcorr : Blind

PAS_CSEPrevXcorr Extends *PASPlugin*

Description:

The permutation ambiguity solution plugin (*PASPlugin*) *PAS_CSEPrevXcorr* is a blind method that maximizes the cross-correlation between current and the last block's CDSs to resolve the permutation ambiguity. The method is primitive in that, if the last block's CDS estimates Q sources, then *PAS_CSEPrevXcorr* will output Q CSEs. Thus, any error in estimating the number of sources will be propagated by *PAS_CSEPrevXcorr*. To understand the implications of this deficiency, we now give a quick technical description of the method.

Let $\boldsymbol{D}(m)$ represent the CDS produced by a *SEEPlugin* at the m^{th} block of data, and let $\boldsymbol{y}(n,m) = [y_1(n,m), \dots, y_B(n,m)]$ and $\boldsymbol{z}(n,m) = \left[z_1(n,m), \dots, z_Q(n,m)\right]$ be the source estimates produced by applying $\boldsymbol{D}(m)$ and $\boldsymbol{D}(m-1)$, respectively, to the m^{th} block of data, where $y_j(n,m)$ is the j^{th} source estimate in $\boldsymbol{y}(n,m)$ for $j = 1, \dots, B$, $\boldsymbol{D}(m)$ estimates B number of sources, $z_q(n,m)$ is the k^{th} source estimate in $\boldsymbol{z}(n,m)$ for $k = 1, \dots, Q$, and $\boldsymbol{D}(m-1)$ estimates Q number of sources. Furthermore, let $r_{jk}(\tau) = \mathcal{E}\{y_j(n,m)\, z_k^*(n+\tau,m)\}$ be the cross-correlation function between $y_j(n,m)$ and $z_k(n,m)$, where τ is a lag-index, and for clarity we drop the time-dependence on $r_{jk}(\tau)$, so the assumption is that $y_j(n,m)$ and $z_k(n,m)$ are wide-sense stationary, or $r_{jk}(\tau)$ is the sample cross-correlation function for the data-block.

The *PAS_CSEPrevXcorr* plugin selects the current CSEs based upon the maximum cross-correlation of the source estimates produced by the past CDS and the current CDS. That is, if we let $r'_{jk} = \max_{\tau}\{r_{jk}(\tau)\}$, then the k^{th} CSE output by the *PAS_CSEPrevXcorr* plugin, $x_k(n,m)$, is given by,

$$x_k(n,m) = y_{q(k)}(n,m), \qquad k = 1, \dots, Q, \tag{92}$$

where

$$q(k) = \arg\max_j r'_{jk}, \qquad k = 1, \dots, Q. \tag{93}$$

Thus, *PAS_CSEPrevXcorr* will always output Q CSEs, which causes redundant current CSEs to be output when $Q > B$, and causes $B - Q$ current CSEs to be omitted when $Q < B$. Future versions of this component will remedy these issues.

Constructor: *PAS_CSEPrevXcorr (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	PAS via past and current CDS Cross-Correlation
pluginVersion	1.0.1
pluginAbbr	PAS_CSEPrevXcorr
pluginDescr	PAS using cross-corrleation of CSEs produced from current and past CDSs applied to current observations

Requirement Fields:

Field Name	Value	Comment
reqSrcEstGroup	0	Method does not require a *SEGPlugin*
reqSrcEstTA	0	Method does not require a *SETAPlugin*
reqSrcEstCDS	0	Method does not require that the CDS for each source estimate be supplied
reqSrcNum	0	Method does not require a *SNUMPlugin*
reqSrcSig	0	Blind method, source signals are NOT required
reqBlkLenMin	-1	Minimum data-block length is 1s
reqBlkLenMax	-60	Maximum data-block length is 60s

Parameters: None , Parameter Sets: None

4.14 (*SEAPlugin*) *SEA_SIR* : Supervised

SEA_SIR Extends *SEAPlugin*

Description:

The source estimate evaluation plugin (*SEAPlugin*) *SEA_SIR* is a supervised method that calculates SIR (see Sect. 1.3.6) for each source and each CSE per data block.

Constructor: *SEA_SIR (ParamSetId)*

Common Fields:

Field Name	Value
pluginName	SIR Analysis
pluginVersion	1.0.1
pluginAbbr	SEA_SIR
pluginDescr	SIR Analysis

Requirement Fields:

Field Name	Value	Comment
reqSrcEstGroup	0	Method does not require a *SEGPlugin*
reqSrcEstTA	0	Method does not require a *SETAPlugin*
reqSrcEstCDS	0	Method does not require that the CDS for each source estimate be supplied
reqSrcNum	0	Method does not require a *SNUMPlugin*
reqSrcSig	1	Supervised method, source signals are required
reqBlkLenMin	-1	Minimum data-block length is 1s
reqBlkLenMax	inf	Maximum data-block length is unspecified

Parameters:

The *SEA_SIR* plugin estimates SIR, e.g. (79), using source images estimated from the imaging operator, e.g. (80). The imaging operator is implemented via the *massInfo* object's *ImgOp()*, thus the estimated image of the q^{th} source in the i^{th} source estimate is $\tilde{\boldsymbol{s}}_{qi}$=*massInfo.ImgOp (*$\boldsymbol{s}_q$*,* $\boldsymbol{y}_i$*, filtLen, filtOffset)* where $\boldsymbol{s}_q$ are the samples of the q^{th} source, $\boldsymbol{y}_i$ are the samples of the i^{th} source estimate, *filtLen* is the imaging filter length, and *filtOffset* is a delay applied $\boldsymbol{s}_q$ to ensure a causal solution. The source samples are provided in *semSrcSig* and the source estimates are provided in *semSrcEstArr*.

Parameter	Description	Range	Default
filtLen	Imaging filter length, in samples	*filtLen* > 1	*semCDSFiltLen*
filtOffset	A delay to ensure causal solutions, in samples	*filtOffset* ≥ 0	50

Parameter Sets:

The *SEA_SIR_params_default()* function provides the default value for *filtOffset*, and the *SEA_SIR_params()* function allows users to modify the *filtOffset* value. Although the *SEA_SIR_params_default()* and the *SEA_SIR_params()* contain a field for *filtLen*, the current version uses a parameter set dependent multiple of *semCDSFiltLen* as the *filtLen* value.

Furthermore, the *SEA_SIR* plugin operates in two modes, block and batch. In the block mode, the SIR is calculated using the data-block's source and source estimate samples. For finite length signals, e.g. audio files, the SIR can be calculated in batch mode by using *all* samples of the source signal as well as using the source estimates produced by applying the CDSs to *all* samples of the observations. The batch mode is applicable when the mixing system is considered time-invariant. The batch mode is supplied to produce more accurate estimates.

Parameter Set Id	*Parameter Values*	*SEA Mode*
0	*filtLen* = *semCDSFiltLen* *filtOffset* is given by *SEA_SIR_params()*	Block
1	*filtLen* = 0.5 x *semCDSFiltLen* *filtOffset* is given by *SEA_SIR_params()*	Block
2	*filtLen* = 2 x *semCDSFiltLen* *filtOffset* is given by *SEA_SIR_params()*	Block
3	*filtLen* = 3 x *semCDSFiltLen* *filtOffset* is given by *SEA_SIR_params()*	Block
4	*filtLen* = 4 x *semCDSFiltLen* *filtOffset* is given by *SEA_SIR_params()*	Block
10	*filtLen* = *semCDSFiltLen* *filtOffset* is given by *SEA_SIR_params()*	Batch
11	*filtLen* = 0.5 x *semCDSFiltLen* *filtOffset* is given by *SEA_SIR_params()*	Batch
12	*filtLen* = 2 x *semCDSFiltLen* *filtOffset* is given by *SEA_SIR_params()*	Batch
13	*filtLen* = 3 x *semCDSFiltLen* *filtOffset* is given by *SEA_SIR_params()*	Batch
14	*filtLen* = 4 x *semCDSFiltLen* *filtOffset* is given by *SEA_SIR_params()*	Batch

CHAPTER 5: USING THE MASS FRAMEWORK: CONFIGURATIONS

Although the purpose of this work is multiple algorithm source separation (MASS), in this chapter we will find that the MASS framework also admits applications for single algorithm source separation (SASS) and signal analysis. In this chapter we will generate many instantiable extensions of the *Configuration* class that fully inform the MASS framework how to operate, and we denote each of these extensions as a "Configuration". If a user wishes to use the MASS framework in her application and randomly picks a particular Configuration from this chapter out of a hat, for example a configuration named "*Config_out_of_a_hat*", these are the three lines of code she would need to implement the MASS framework in her MATLAB application:

```
config = Config_out_of_a_hat();
mass = MASS_SEM( config );
mass.Start();
```

The entirety of this chapter is devoted to the inner workings of *Config_out_of_a_hat*, and we will show how Configurations direct MASS applications.

Even with the small library of plugin components given in Chapter 4 and the limited number of configuration options given by the abstract *Configuration* class in Sect. 3.5, the combinations of these two sets for a Configuration are large. Configurations represent a realization, or application, of the MASS framework, and the number of possible Configurations within the scope of this limited document is immense. In a real sense, the limits of a Configuration define the scope of study for the MASS topic. Thus, the more diverse a black-box MASS framework becomes, we can either gain depth or breadth of knowledge/probing depending on the direction the diversity of methods delves into or expands the field, respectively. Although all parts of the MASS framework are important, Configurations are the key to performing MASS.

In this version of the MASS framework, Configurations are objects with a moderately complicated structure. In this chapter we begin by inspecting the structure of Configurations and provide a tabulated summary notation to succinctly define the various Configurations used in this chapter. An important part of all source separation problems is the observations and the conditions of the observations; thus, we will also define the various datasets that will be used throughout this chapter. Beyond that, we will define

the Configurations that allow us to explore three general applications of the MASS framework: signal analysis, single algorithm source separation (SASS), and multiple algorithm source separation (MASS).

5.1 Configuration Summary Notation

As we pointed out in Sect. 3.7 and in the introduction to this chapter, the SEM requires a valid *Configuration* object to initialize and run the MASS framework, and here we denote a general child of the abstract *Configuration* class as a "Configuration". Thus, to use the MASS framework, a user must create a valid Configuration object, and here, we will give many such Configuration implementations.

A Configuration contains four general pieces of information: a description of the configuration, processing parameters, a set of plugins, and a set of data input and output location information. To create an application that involves the MASS framework, one needs to understand the capabilities and parameters of the various plugin components, e.g. Chapter 4, as well as how to construct a valid Configuration. Although a major future development opportunity for the MASS framework involves creating a GUI-based Configuration builder, the current version of MASS simply defines configuration information as an extension of the base *Configuration* class. To give the user a feel for how to construct a valid Configuration and ultimately use the MASS framework, we now look at a couple of Configurations in MATLAB code for elucidation.

For example, the MATLAB code given in Table 12 is a Configuration used in Sect. 5.5.2 to study a semi-blind MASS scenario. In line 1, the Configuration class *Config_MASS_Ex2_GS1_3x3* is declared to be an extension of the parent class *Configuration* that was specified in Sect. 3.5. Line 3 gives the constructor for the *Config_MASS_Ex2_GS1_3x3* class. Lines 5-7 give descriptive information about the Configuration in the *configName*, *configDescr*, and *configAbbr* fields. Lines 9-16 set the

Table 12. MATLAB source code for an example Configuration, i.e. an instantiable child class of the *Configuration* class. A Configuration contains four general pieces of information: a description of the configuration, processing parameters, a set of plugins, and a set of data input and output information.

```
1     classdef Config_MASS_Ex2_GS1_3x3 < Configuration
2         methods
3             function obj = Config_MASS_Ex2_GS1_3x3()
4
5   Description
6                 obj.configName  = 'MASS Example 2';
7                 obj.configDescr = 'MASS Example 2: Semi-Blind MASS';
8                 obj.configAbbr  = class(obj);
9
10  Processing Parameters
11                obj.semSampRate = 16000;
12                obj.semSelfComp = 0;
13                obj.semBlkLen = -5;
14                obj.semBlkStep = -5;
15                obj.semCDSFiltLen = -1/16;
16                obj.semCDSFiltOffset = obj.semCDSFiltLen/2;
17                obj.semPluginError = 0;
18                obj.semVerbose = 0;
19
20  Plugins
21                obj.SEP  = [PluginInst('SEP_MCLP',0);
22                            PluginInst('SEP_TTSE',0);
23                            PluginInst('SEP_ABYK',2);];
24                obj.SEE  = PluginInst('SEE_SupSIRSelect',10);
25                obj.PAS  = PluginInst('PAS_CSEPrevXCorr',0);
26                obj.SEG  = [];
27                obj.SNUM = [];
28                obj.SETA = [];
29                obj.SEA  = PluginInst('SEA_SIR', 10);
30                obj.DA   = PluginInst('DA_FileRead',0);
31                obj.DO   = PluginInst('DO_FileWrite',0);
32
33  Inputs
34  Dataset
35                dataSet = 'DS_SS1-3_GS1_3x3_RT60-0.1s.mat';
36                obsFQN = fullfile('./Datasets',dataSet);
37                load(dataSetFQN, 'SrcSet');
38                srcFQN = fullfile('./Datasets','SourceSets',SrcSet);
39
40  Observations
41                obj.daObsExtSig = ExternalSignal();
42                obj.daObsExtSig.location = obsFQN;
43                obj.daObsExtSig.varName = 'OBS';
44                obj.daObsExtSig.sampRate = obj.semSampRate;
45                obj.daObsExtSig.name = [dataSet '_' obj.daObsExtSig.varName];
46                obj.daObsExtSig.groupId = [];
47                obj.daObsExtSig.channel = [];
48                obj.daObsExtSig.type = 'file';
49
50  Sources
51                obj.daSrcExtSig = ExternalSignal();
52                obj.daSrcExtSig.location = srcFQN;
53                obj.daSrcExtSig.varName = 'SRC';
54                obj.daSrcExtSig.sampRate = obj.semSampRate;
55                obj.daSrcExtSig.name = [dataSet '_' obj.daSrcExtSig.varName];
56                obj.daSrcExtSig.groupId = [];
57                obj.daSrcExtSig.channel = [];
58                obj.daSrcExtSig.type = 'file';
59
60  Output
61                outputFQN = fullfile('./Outputs',[class(obj) '-OUT.mat']);
62                obj.doDestExtSig = ExternalSignal();
63                obj.doDestExtSig.location = outputFQN;
64                obj.doDestExtSig.sampRate = obj.semSampRate;
65                obj.doDestExtSig.varName = [];
66                obj.doDestExtSig.name = [];
67                obj.doDestExtSig.groupId = [];
68                obj.doDestExtSig.channel = [];
69                obj.doDestExtSig.type = 'file';
70            end
71        end, end
```

Table 13. MATLAB source code for a Configuration component that is an extension of another Configuration component. Object-oriented inheritance allows a user to easily create configuration components that differ slightly from already created components.

```
1       classdef Config_MASS_Ex2_GS1_3x3_Alg1Analysis < Config_MASS_Ex2_GS1_3x3
2           methods
3               function obj = Config_MASS_Ex1_GS1_3x3_Alg1Analysis()
4                   obj.configName = 'MASS Example 2: MCLP Analysis';
5                   obj.configDescr = 'MASS Example 2: MCLP Analysis';
6                   obj.configAbbr = class(obj);
7
8                   obj.SEP = [PluginInst('SEP_MCLP',0)];
9                   obj.SEE = [PluginInst('SEE_Identity',0)];
10              end
11          end
12      end
```

processing parameters used in the Configuration, and for example, the sampling rate is set to 16kHz, self-competition is turned off, and the data block length is set to 5s. The *pluginError* field is unused as of this writing, but we include it for future compatibility.

Lines 18-28 define the plugins used in the configuration, and although all of the SEP plugins are blind methods, i.e. *SEP_ABYK*, *SEP_MCLP*, and *SEP_TTSE*, the Configuration performs semi-blind MASS since the SEE plugin, *SEE_SupSIRSelect*, is a supervised method. We note that all of the plugins are defined using the *PluginInst* utility class, thus each of the
plugins are defined by a plugin name and a parameter set identification number.

Lines 30-51 define the *daObsExtSig* and *daSrcExtSig ExternalSignal* objects that contain the information to acquire the observation and source signals, respectively, and lines 53-61 specify the *doDestExtSig ExternalSignal* object which the data output plugin, *DO_FileWrite*, uses to store processed data. Lines 30-33 define the location of the input dataset files, and we will give more information about these in the next section. Noting that we are using the *DA_FileRead* data acquisition plugin, line 36 defines the location of a .mat file containing the observation signals, and line 37 indicates that the observation signals are contained in the variable "OBS" within the .mat file. Lines 45 and 46 contain similar location and variable name information, respectively, for the source signals.

One benefit of objectifying the Configurations is that we can create a slightly modified version of a Configuration by extension. For instance, if we want to evaluate the *SEP_MCLP* individually but under the same processing conditions and for the same data that were used in *Config_MASS_Ex2_3x3*, we can simply extend the Configuration as given in Table 13. In line 1 of Table 13, we create a new Configuration class,

Table 14. Tabulated shorthand notation for specifying the fields of a Configuration. Descriptive information is omitted. In this work, all Configurations use the DA_FileRead and DO_FileWrite data acquisition and output plugins, respectively, thus the DO and DA plugin fields are also omitted. Extensions to the component are listed along with the fields/values that are modified, and otherwise the extensions use all of the inherited field values from the parent class.

Config_MASS_Ex2_3x3			
DATASET: DS_3x3_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGINST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_MCLP*, 0
semSelfComp	0		*SEP_TTSE*, 0
semCDSFiltLen	-1/16		*SEP_ABYK*, 2
semCDSFiltOffset	*semCDSFiltLen/2*	***SEE***	*SEE_SupSIRSelect*, 10
semBlkLen	-5	***PAS***	*PAS_CSEPrevXCorr*, 10
semBlkStep	-5	***SEA***	*SEA_SIR*, 10

EXTENSIONS			
Config_MASS_Ex2_3x3_Alg1Analysis			
SEP	*SEP_MCLP*, 0	***SEE***	*SEE_Identity*, 0

Config_MASS_Ex2_3x3_Alg1Analysis, that extends *Config_MASS_Ex2_3x3*, thus all fields and values contained in *Config_MASS_Ex2_3x3*, are inherited by the new *Config_MASS_Ex2_3x3_Alg1Analysis* class. After modifying the descriptive information in lines 4-6, we define a sole source estimate production plugin, *SEP_MCLP*, in line 8 and a new source estimate evaluation plugin, *SEE_Identity*, in line 9. Thus, we can now run the individual algorithm under the same conditions as the multiple SEP plugin Configuration.

Now that we have a feel for the usage and underlying code for using Configuration objects, we will use a shorthand to describe a Configuration and its various extensions, and an example is given in Table 14. The first piece of information given (besides the configuration class) is the dataset, and as we shall see in the next section, we define datasets in this work in a way that all input and output *ExternalSignals* for the DA and DO plugins, respectively, can be created, thus we simply note a dataset file to cover these properties. Our shorthand notation provides the processing parameters for the Configuration, and we exclude the *semVerbose* and *semPluginError* flags, since they are inconsequential to the MASS performance.

The tabulated shorthand notation in Table 14 lists all of the plugins used in the Configuration by the name of the *PluginInst* field name, followed by the plugin component name and the comma separated *ParameterSetId* used to instantiate the plugin, with some notable exceptions. All Configurations used in this work utilize the *DA_FileRead* and *DO_FileWrite* data acquisition and output plugins, respectively, thus we omit them from our shorthand notation. Besides the DO and DA plugins, any *PluginInst* field that is not listed is not used, e.g., the Configuration in Table 14 does not use a SNUM, SEG, or SETA plugin.

The extensions of the configuration are listed below the parameters (dataset, processing parameters, and plugins) of the Configuration parent class. For example in Table 14, the *Config_MASS_Ex2_3x3_Alg1Analysis* class inherits all of the field values provided by the *Config_MASS_Ex2_3x3* configuration, but overrides the *SEP* and *SEE* plugin instance definitions by replacing, respectively, a multi-SEP scenario with the single SEP plugin, *SEP_MCLP*, and using the *SEE_Identity* plugin instead of the *SEE_SupSIRSelect* plugin component given in the original definition of *Config_MASS_Ex2_3x3*. We will use this format to concisely define various Configurations in Sects. 5.3-5.5, and we now turn our attention to the datasets used in the Configurations to exemplify various MASS framework capabilities.

5.2 Datasets

All Configurations require a set of inputs, and for the example Configurations we give in this chapter, we will use various datasets to evaluate the MASS framework. Along with other information to be described later, each dataset used here is comprised of a set of source signals, a set of observations, and a set of room impulse responses (RIRs), where each observation is created by filtering the sources with a subset of the RIRs and summed. Furthermore, we use *synthetic* RIRs produced from a method in the literature, and this allows us to precisely specify and control the source locations, the sensor locations, the dimensions of a room, and the reverberation characteristics of a room, while also producing reasonable approximations of scenarios encountered in the physical world. That is, the problematic parts of the source separation problem, e.g. determinedness, long reverberation times, etc., can be realistically and easily modeled via

synthetic RIRs. In this section we will detail the source signals, RIRs, and datasets that will be employed throughout this chapter.

5.2.1 Source Signal Sets

The total set of source signals that we utilize in this chapter is comprised of two female talkers, a male talker, and a white Gaussian noise source. All of the signals are sampled at 16kHz, have a duration of 30s, are zero mean, and have unit variance. The first source signal is running speech from a female talker, and we will refer to this signal as "Source 1" or "Female Talker #1" to distinguish it from the other female talker's signal. The second source signal, "Source 2", is a synthetically generated white Gaussian noise (WGN) sequence. The time-domain signals for Source 1 and Source 2 are shown in Figure 8.

"Source 3" and "Source 4" are running speech of a female talker ("Female Talker #2") and a male talker, respectively, that have been randomly windowed to simulate conversational speech between the two talkers. Figure 9 a) shows Source 3 and the windowing function used to produce the signal from running speech of Female Talker #2. The fundamental windowing function was constructed by creating successive cycles of active and silent regions within the signal, and the duration in seconds of each active and silent region was chosen, independently for each, from a uniform random variable on the range $[0.5s\ 2.5s]$. To avoid audible discontinuities due to abruptly turning the signal on and off, we applied a raised cosine function, or Tukey window, to the leading and trailing 25ms of the active region.

To define active and inactive regions, we let $D_a(m) \sim U(8000,40000)$ and $D_i(m) \sim U(8000,40000)$, for $m = 1,2,3, \ldots$, be sequences of independent uniform random variables on the range 8000 to 40000, i.e. the sample range for $0.5s$ to $2.5s$ at 16kHz sampling rate. Furthermore, we let $N_a(m)$ and $N_i(m)$ be realizations of $D_a(m)$ and $D_i(m)$, respectively, where $N_a(m)$ and $N_i(m)$ are the duration in samples of the m^{th} active and inactive regions, respectively. If we define the raised cosine functions,

$$w_{in}(\kappa) = \begin{cases} \frac{1}{2} - \frac{1}{2}\cos\frac{\kappa\pi}{399}, & \kappa = 0,1, \ldots, 399 \\ 0, & \text{otherwise} \end{cases}, \tag{94}$$

and

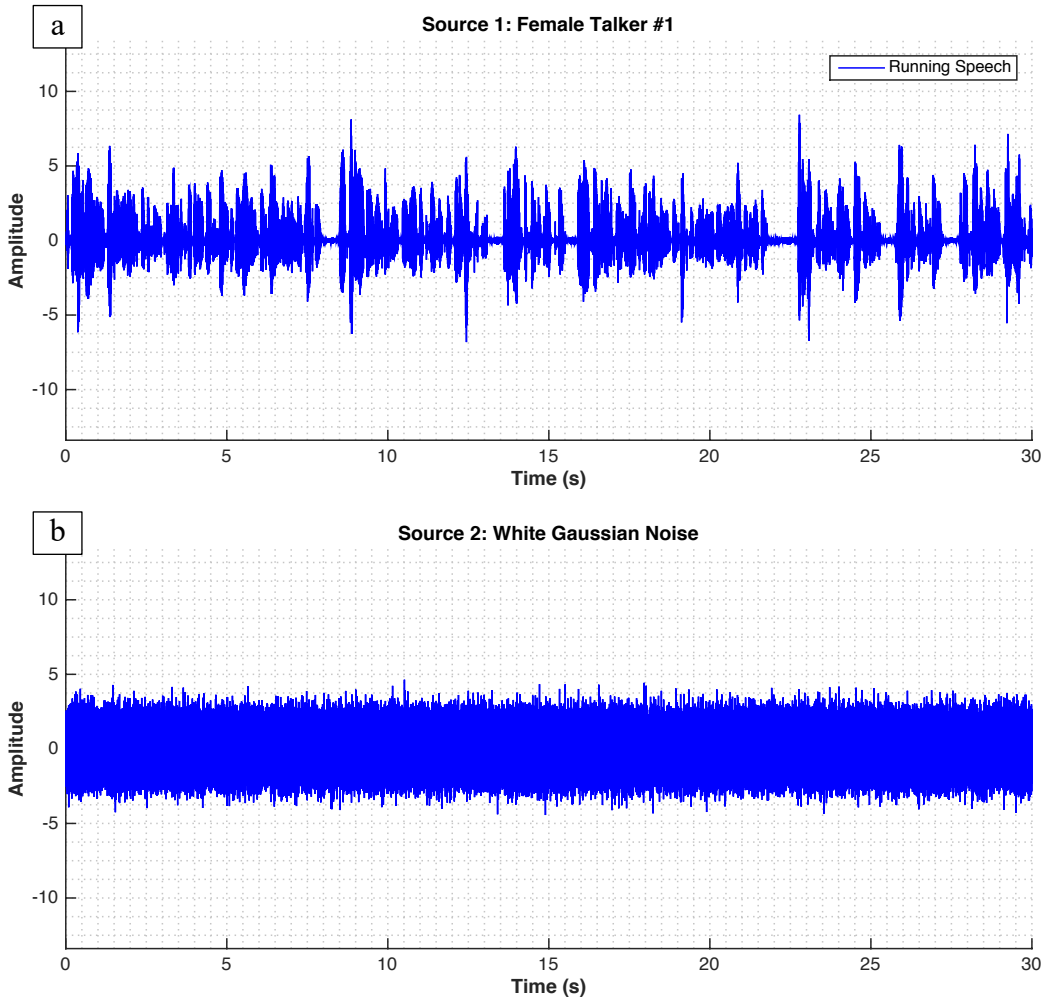

Figure 8. Time-domain plots of Sources 1 and 2: Source 1 is running speech from Female Talker #1, and Source 2 is stationary, white Gaussian noise. The sources are sampled at 16kHz and are 30s in duration.

$$w_{out}(\kappa) = \begin{cases} \frac{1}{2} + \frac{1}{2}\cos\frac{\kappa\pi}{399}, & \kappa = 0,1,\dots,399 \\ 0, & \text{otherwise} \end{cases}, \tag{95}$$

where κ is a lag-index, then we can define the k^{th} sample of the $N(m)$-length window for Female Talker #2 during the m^{th} windowing cycle as,

$$w_F(m,k) = \begin{cases} w_{in}(k), & k = 0,\dots,399 \\ 1 & k = 400,\dots,N_a(m) - 401 \\ w_{out}(k - N_a(m) + 400),, & k = N_a(m) - 400,\dots,N_a(m) - 1 \\ 0, & k = N_a(m),\dots,N(m) - 1 \end{cases}, \tag{96}$$

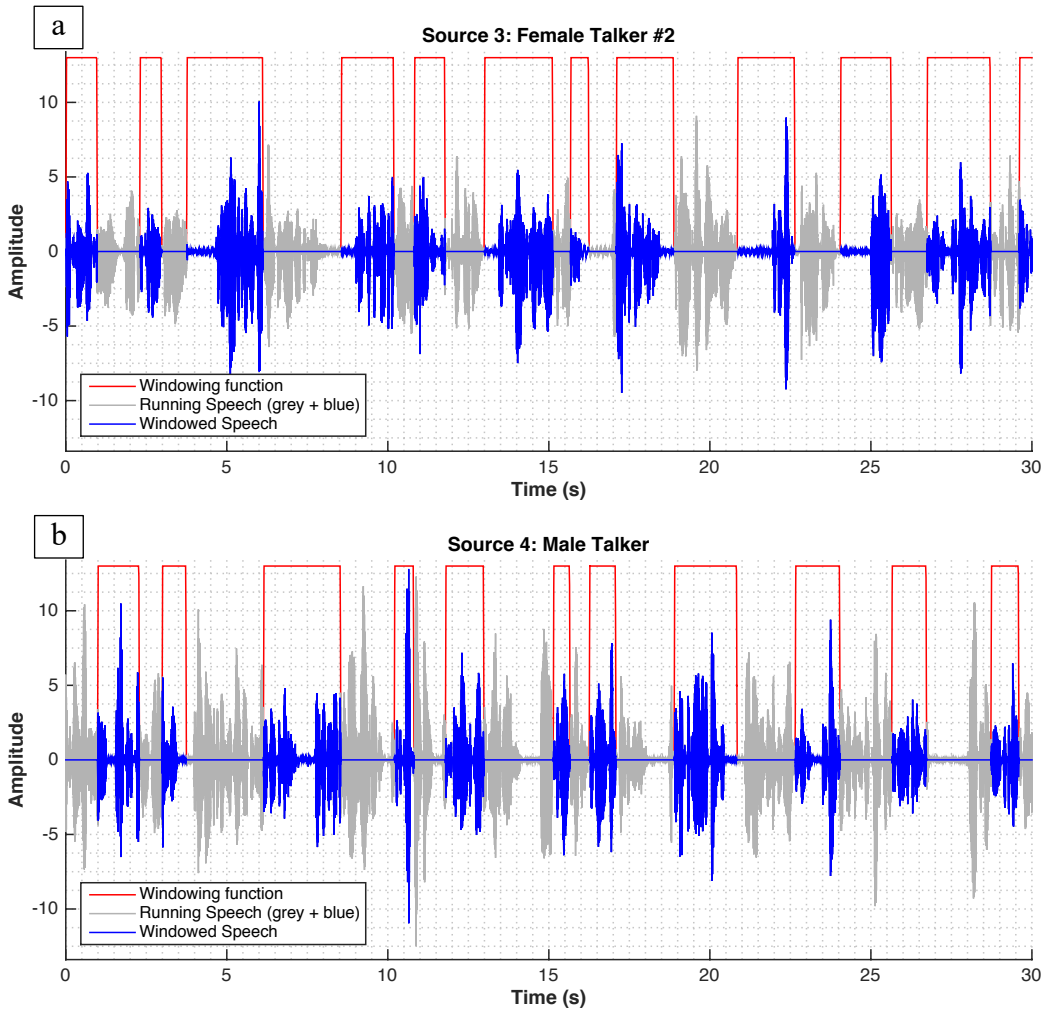

Figure 9. Time-domain plots of Sources 3 and 4. Source 3 is running speech from Female Talker #2 that has been windowed to produce random intervals of silence. Source 4 is running speech from a Male Talker that has been windowed to produce silent intervals during Source 3's active intervals and active intervals during Source 3's silent intervals. The sources are sampled at 16kHz and are 30s in duration.

for $m = 1,2,3, \ldots$, where $N(m) = N_a(m) + N_i(m)$. Letting $N_{tot}(m) = \sum_{l=0}^{m-1} N_a(l) + N_i(l)$, we create the windowed signal by applying $w_F(m,k)$ to the samples of the Female Talker #1's running speech signal at time-indices $n = N_{tot}(m) + 1, \ldots, N_{tot}(m) + N(m)$, for $m = 1,2,3, \ldots$, where $N_a(0) = N_i(0) = 0$.

We can construct a windowing function for the Male talker (Source 4) similarly, except that we make the Male talker active during Female Talker #2's silent regions and

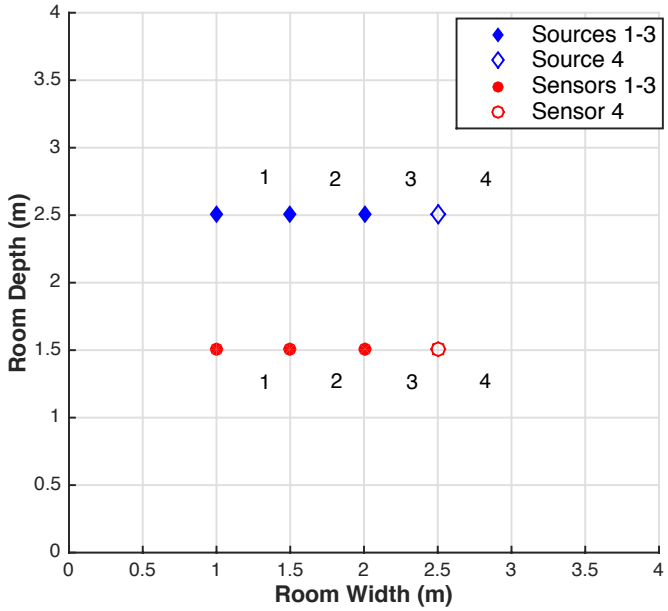

Figure 10. Horizontal geometry of source and sensor locations within a room for synthetically generated room impulse response sets used in this work. The room height is 3m and the speed of sound is set at 343 m/s.

make the Male talker silent during the female talker's active regions. The windowing function for the male talker during the m^{th} cycle is therefore given as,

$$w_M(m,k) = \begin{cases} 0, & k = 0, \dots, N_a(m) - 1 \\ w_{in}(k), & k = N_a(m), \dots, N_a(m) + 399 \\ 1, & k = N_a(m) + 400, \dots, N(m) - 401 \\ w_{out}(k - N(m) + 400), & k = N(m) - 400, \dots, N(m) - 1 \end{cases}, \quad (97)$$

and Source 4 is shown in Figure 9 b) along with the windowing function.

For convenience later on we store the source signals in two .mat files. The first .mat file is named "SS1-3.mat", and it contains a [480000x3] matrix named "*SRC*" where the samples of Source q are contained in the q^{th} column of *SRC*, for $q = 1,2,3$. The second file, named "SS1-4.mat", contains all 4 sources's samples similarly in the [480000x4] *SRC* array with Source 4 in the fourth column of *SRC*.

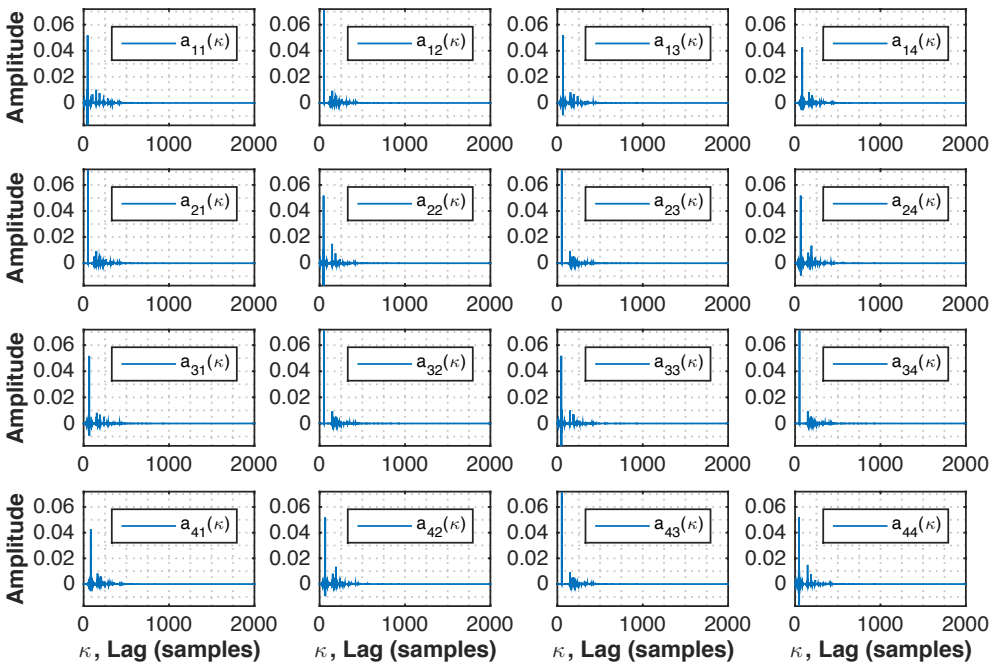

Figure 11. RT_{60} = 100ms room impulse response (RIR) set for 4 sources and 4 mixtures. Each RIR is sampled at 16kHz. Each RIR is 3200 samples long, but only 2000 samples are shown.

5.2.2 Room Impulse Response Sets

In this work, we will synthetically generate room impulse responses using the "imaging" method given by Allen and Berkley [96] implemented in Habets [97]. To construct a room impulse response set we need the room dimensions, a set of source locations within the room, a set of sensor locations within the room, the reverberation characteristics of the room and its surfaces, and the speed of sound. For all RIR sets defined here, the room dimensions will be 4m wide x 4m deep x 3m high, and we use a constant 343m/s for the speed of sound. From there, we can create RIR sets that are only dependent upon the source and sensor locations, as well as the reverberation characteristics of the room.

For any specified room reverberation characterization, the RIR sets used in this work are subsets of, or are equal to, the total set of RIRs generated by using all of the source and sensor locations given in Figure 10. That is, if we let $a_{qp}(\kappa)$ be the filter

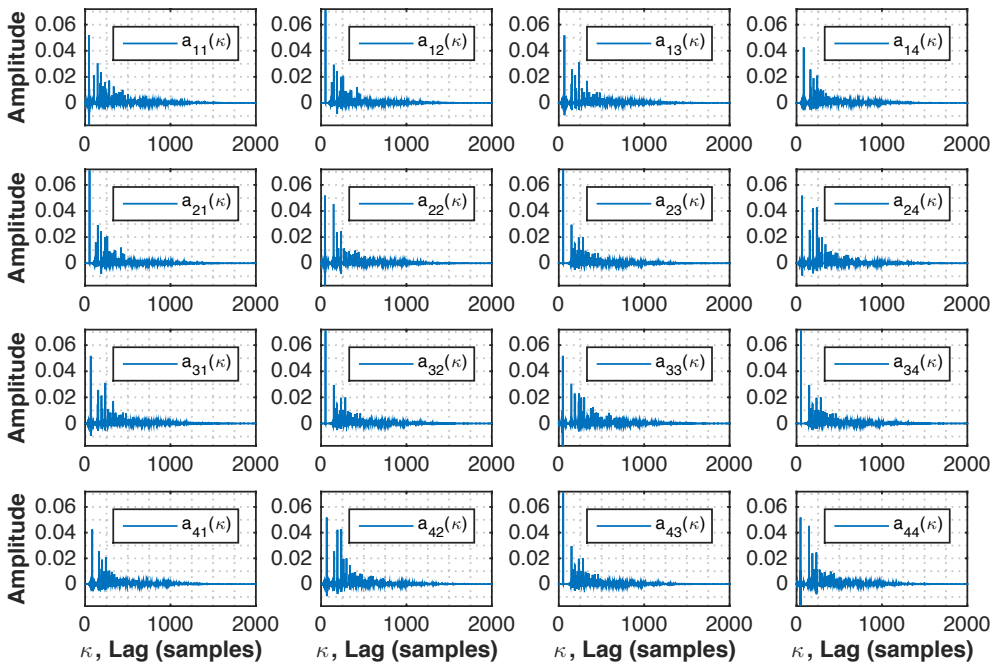

Figure 12. RT_{60} = 150ms room impulse response (RIR) set for 4 sources and 4 mixtures. Each RIR is sampled at 16kHz. Each RIR is 3200 samples long, but only 2000 samples are shown.

representing the channel between the q^{th} source and the p^{th} sensor, then the geometry in Figure 10 implies that there are 16 such filters in the 4 source and 4 sensor case. Denoting this 4 source and 4 sensor scenario as the "4x4" case, we can use subsets of the 4x4 RIR set to construct other scenarios, as we will see in the next section.

In this work we generate three 4x4 RIR sets based upon three -60 dB reverberation time, or RT_{60}, values. The RT_{60} is the time for the power of all cumulative echoes resulting from the room surface reflections of an emanating source to equal -60 dB relative to the power of the originating source, and the RT_{60} is mainly determined by the speed of sound, the volume of the enclosure, and the absorption/reflection characteristics of each surface within the enclosure. As mentioned above, we use a constant speed of sound, and a constant room volume here, and we now specify the RT_{60} values.

In this work we will use three values for the RT_{60} time to produce three sets of 4x4 RIRs. Although the method in [97] allows us to specify the absorption coefficient for each surface in the room, we use the alternative usage of [97] in which we can specify the RT_{60} time directly so that all surfaces have equal absorption coefficients, i.e. all

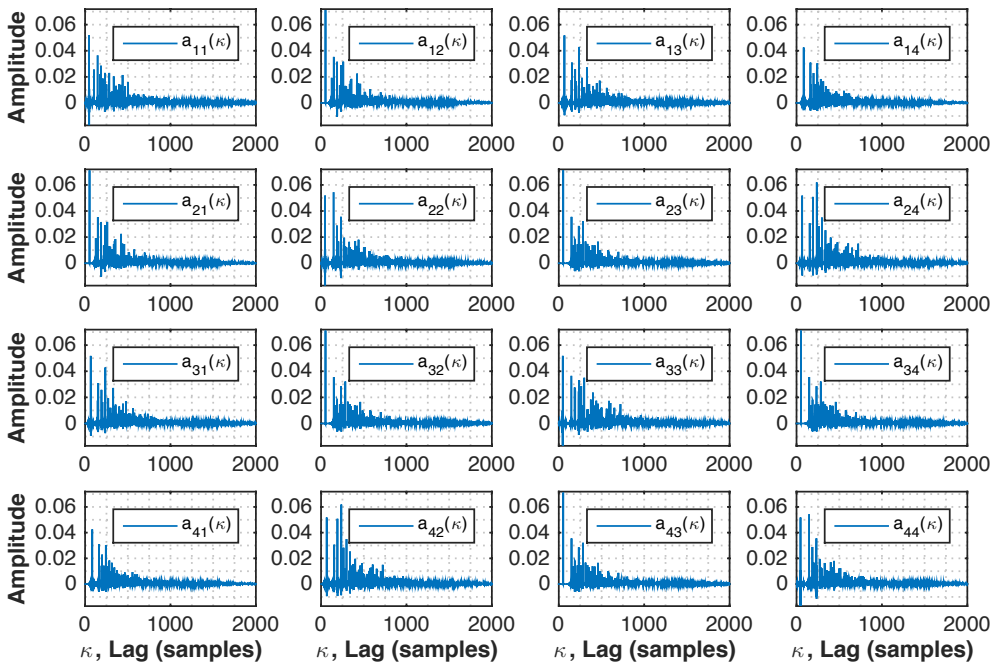

Figure 13. RT_{60} = 200ms room impulse response (RIR) set for 4 sources and 4 mixtures. Each RIR is sampled at 16kHz. Each RIR is 3200 samples long, but only 2000 samples are shown.

surfaces (including floor and ceiling) are equally reflective. The RT_{60} values we use are 100ms, 150ms, and 200ms and Figures 11-13 show the 4x4 RIR sets generated with these values. We display these RIR sets graphically to
give the reader a feel for how smaller RT_{60} values compare with larger RT_{60} values, e.g. the 100ms RT_{60} RIRs have most non-trivial structure at low-valued lags, while 200ms RT_{60} RIRs have significant structure at all lags. All of the RIRs in Figures 10-12 have a length of 3200 samples, but we only show the first 2000 lags for presentation and ease of comparison.

Using the three 4x4 RIR sets just defined, we can create several RIR subsets that will be used in the next section to create the datasets used to evaluate the MASS framework. For a given RT_{60} value, Figure 14 shows the specific RIR subsets that we will use to create our datasets, where the 4x4 set is outlined in black, the 3x4 subset is outlined in red, the 3x3 subset is outlined in blue, and the 3x1 subset is outlined in green. For convenience of framework use, all RIR sets are contained in separate .mat files in a [QxPx3200] matrix called "RIR", along with all of the information used to create the set, e.g. speed of sound, sampling rate, room dimensions, source locations, reverberation

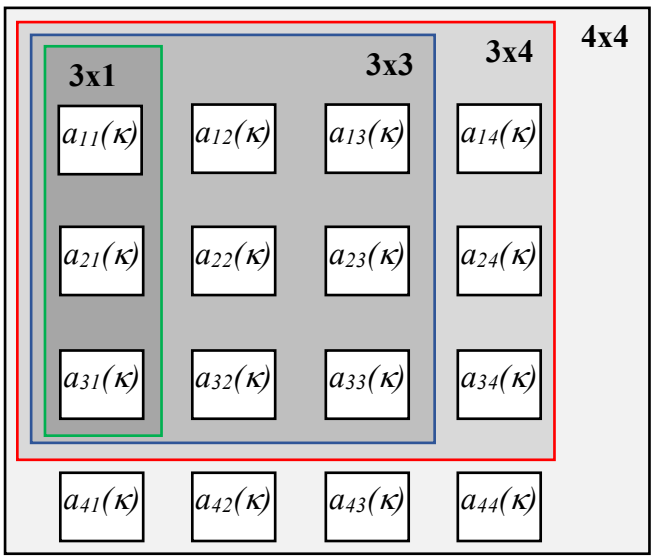

Figure 14. Room impulse response (RIR) subsets. For each previously defined 4x4 RIR set, we define the particular 3x1, 3x3, and 3x4 RIR subsets above.

time, etc., where Q is the number of sources and P is the number of sensors. The naming convention for the RIR set .mat file names is "RIR_[QxP]_RT60-[RT_{60} value in seconds]s.mat" and the "RIR Set" column of Table 15 lists all RIR sets used in this work.

5.2.3 Dataset Definitions

We now combine the source sets defined in Sect. 5.2.1 and the room impulse response (RIR) sets given in Sect. 5.2.2 to create the datasets used in subsequent sections of this paper to evaluate the MASS framework. All three-source scenarios, e.g. 3x1, 3x3, and 3x4, use Sources 1-3 provided in SS1-3.mat, and all four-source scenarios use Sources 1-4 provided in SS1-4.mat, as described in Sect. 5.2.1

Although the 3x1 case has a special usage, all datasets are provided in a .mat file for convenience with the naming convention "DS_[QxP]_RT60-[RT_{60} value in seconds]s.mat", where Q is the number of sources, P is the number of observation signals, and RT_{60} is the RIR set reverberation. Each dataset .mat file contains four variables, *Fs*, *SRCset*, *RIRset*, and *OBS* where *Fs* is the sampling rate, *SRCset* is the name of the source set .mat file, *RIRset* is the name of the RIR set .mat file, and *OBS* is a [480000xP] array of observation samples produced from filtering and summing the sources using the filters in the RIR set. For example, DS_3x4_RT60-0.2.mat has *Fs* =

Table 15. Summary of datasets supplied with the MASS framework. A dataset is comprised of the sampling rate, a set of source signals, a set of room impulse repsonses (RIRs), and a set of observations that are the result of filtering and combining the sources through the RIRs. All signals used in this work are sampled at 16kHz.

Dataset Name (.mat)	RIR Set (.mat)	Source Set (.mat)	RT_{60}	Number of Mixtures
DS_3x1_RT60-0.1s_SIRStudy	RIR_3x1_RT60-0.1s	SS1-3	100ms	4*
DS_3x3_RT60-0.1s	RIR_3x3_RT60-0.1s	SS1-3	100ms	3
DS_3x3_RT60-0.15s	RIR_3x3_RT60-0.15s	SS1-3	150ms	3
DS_3x3_RT60-0.2s	RIR_3x3_RT60-0.2s	SS1-3	200ms	3
DS_3x4_RT60-0.1s	RIR_3x4_RT60-0.1s	SS1-3	100ms	4
DS_3x4_RT60-0.15s	RIR_3x4_RT60-0.15s	SS1-3	150ms	4
DS_3x4_RT60-0.2s	RIR_3x4_RT60-0.2s	SS1-3	200ms	4
DS_4x4_RT60-0.1s	RIR_4x4_RT60-0.1s	SS1-4	100ms	4
DS_4x4_RT60-0.15s	RIR_4x4_RT60-0.15s	SS1-4	150ms	4
DS_4x4_RT60-0.2s	RIR_4x4_RT60-0.2s	SS1-4	200ms	4

16000, *RIRset* = "RIR_3x4-0.1s.mat", *SRCset* = "SS1-3.mat", and the p^{th} column of the *OBS* variable are the first 480000 samples of $\sum_{q=1}^{3} a_{qp}(\kappa) * s_q(n)$, for $p = 1,..,4$, where $a_{qp}(\kappa)$ is the channel filter between the q^{th} source and the p^{th} sensor for the RT_{60} = 100ms case contained in RIR_3x4-0.1s.mat, $s_q(n)$ is Source q from SS1-3.mat, and $*$ denotes convolution. Table 15 summarizes the datasets supplied with the MASS framework, and we will now further explain the 3x1 dataset "DS_3x1_RT60-0.s_SIRStudy".

The 3x1 dataset was designed specifically for the Configurations given in Sect. 5.3, and although it contains all of the information that other datasets have, e.g. a source set, a RIR set, and a set of observations, we generate the observation signals a bit differently. The p^{th} observation, $x_p(n)$ is of the form,

$$x_p(n) = \tilde{s}_1(n) + \gamma\big(SIR_p\big)[\tilde{s}_2(n) + \tilde{s}_3(n)], \qquad p = 1,..,4, \tag{98}$$

where,

$$\tilde{s}_q(n) = \Big(a_{q1}(n) * s_q(n)\Big)(n), \qquad q = 1,\dots,3, \tag{99}$$

is the image of the q^{th} source in the 1^{st} sensor, $a_{q1}(n)$ is the channel filter between the q^{th} source in the 1^{st} sensor as given in RIR_3x1_RT60-0.1s.mat, and $\gamma(SIR_p)$ is a scalar used to produce mixtures with various SIR values for Source 1. Specifically, if we wish to produce an SIR of SIR_p dB for Source 1, e.g.,

$$SIR_p = 10\log_{10}\left[\frac{\sigma_1^2}{\sigma_{23}^2\gamma^2(SIR_p)}\right], \tag{100}$$

where the all signals are zero-mean, $\sigma_1^2 = \mathcal{E}\left\{\left(\tilde{s}_1(n)\right)^2\right\}$, and $\sigma_{23}^2 = \mathcal{E}\left\{\left(\tilde{s}_2(n) + \tilde{s}_3(n)\right)^2\right\}$, then

$$\gamma(SIR_p) = 10^{\frac{-SIR_p}{20}}\left(\frac{\sigma_1^2}{\sigma_{23}^2}\right)^{0.5}. \tag{101}$$

In this work the first observation has an SIR for Source 1 of $SIR_1 = 5\text{dB}$, for mixture 2 $SIR_2 = 10\text{dB}$, and $SIR_3 = 15\text{dB}$ and $SIR_4 = 20\text{dB}$ for observations three and four, respectively.

5.3 Signal Analysis

In this section, we will give a set of Configurations that display how to use the MASS framework to simply analyze a set of input signals using our SEA plugin. Although the signal analysis capability is useful for providing measures of source separation performance, we can also use the signal analysis in developing various plugin components in the MASS framework. Beyond looking at the signal analysis Configurations, we note that this section plays an important role in analyzing the MASS framework, generally, since the examples that we give here characterize the performance of our *SEA_SIR* plugin which is used throughout the rest of this chapter to analyze source separation performance.

5.3.1 Signal Analysis Example 1: SIR Imaging Filter Length

In this section we will analyze the SIR for an input signal set using the *SEA_SIR* plugin, i.e. the SEA plugin that reports SIR information, as well as study the effect that filter length has in the SIR estimation process. All of the Configurations in this section use the DS_3x1_RT60-0.1s_SIRStudy dataset, where each of the four observation signals

Table 16. Configurations used in Signal Analysis Example 1. The *SEA_SIR* plugin performance is studied over a range of imaging filter lengths.

Config_SigAnalysis_Ex1_500			
DATASET: DS_3x1_RT60-0.1s_SIRStudy.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_Static*, 0
semSelfComp	0	***SEE***	*SEE_Identity*, 0
semCDSFiltLen	-1/16	***PAS***	*PAS_Identity*, 0
semCDSFiltOffset	*semCDSFiltLen/2*	***SEA***	*SEA_SIR*, 11
semBlkLen	-30		
semBlkStep	-30		

EXTENSIONS		
Config_SigAnalysis_Ex1_1000	***SEA***	*SEA_SIR*, 10
Config_SigAnalysis_Ex1_2000	***SEA***	*SEA_SIR*, 12
Config_SigAnalysis_Ex1_3000	***SEA***	*SEA_SIR*, 13
Config_SigAnalysis_Ex1_4000	***SEA***	*SEA_SIR*, 14

were generated to contain a mixture in which Source 1 (Female Talker #1 running speech) has a prescribed SIR value relative to the other sources, as explained in Sect. 5.2.3. The SIR values for Source 1 are 5dB in observation 1, 10dB in observation 2, 15dB in observation 3, and 20dB in observation 4. In this section, we will simply analyze the input signals' Source 1 SIRs, and thus do not perform any source estimation.

The Configurations used in this section are given in Table 16, and the notable common plugins of all the Configurations are the *SEP_Static*, *SEE_Identity*, *PAS_Identity*, and *SEA_SIR* plugins. The *SEP_Static*, *SEE_Identity*, and *PAS_Identity* plugins are used with respective parameter sets that, for each plugin, passes the input of the plugin directly to the plugin output. The parameter set for *SEA_SIR* is varied over the Configurations so that a Configuration has the naming convention "*Config_SigAnalysis_Ex1_X*" where *X* denotes the number of coefficients used in the SIR imaging filter, i.e. *X=500, 1000, 2000, 3000,* and *4000* for the Configurations used here.

Another notable common feature of all Configurations used in this section is that the *semBlkLen* and the *semBlkStep* processing parameters are both set to 30s. Given that the all of the signals contained in the dataset are of length 30s, the Configurations are

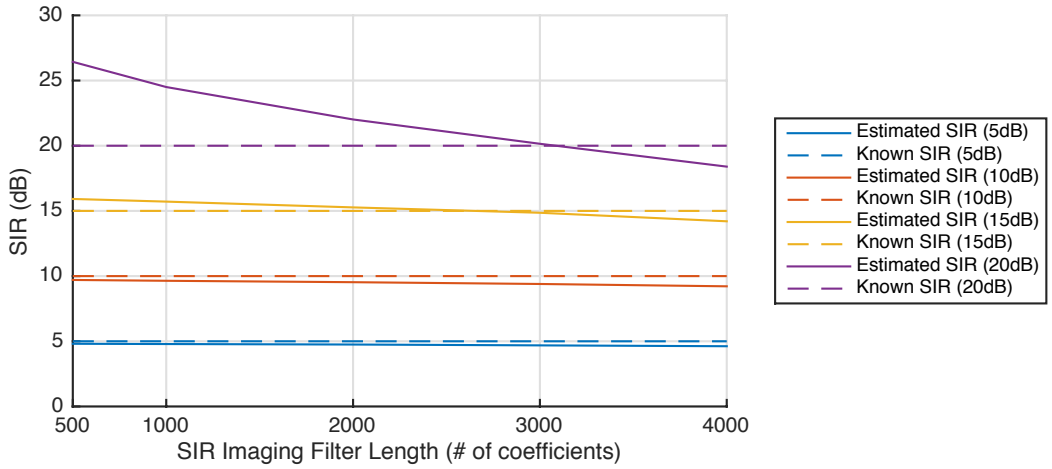

Figure 15. Signal Analysis via the MASS Framework, Example 1: SIR Imaging filter length study. Three sources were mixed using the 3x1 RT_{60}=100ms RIRs to produce four signals with SIRs for Source 1 of 5, 10, 15, and 20 dB, respectively (dashed lines). For each mixture, the SIR was estimated via the SEA_SIR plugin using different imaging filter lengths (solid lines). This figure shows the SIR estimation error as a function of filter length where the optimal filter length lies in the 500-1000 tap range.

executed in batch mode. That is, the first, and only, data-block contains all samples of the observation data, and the SIR analysis is performed using all observation samples.

Figure 15 plots the known SIR (dashed lines) and the estimated SIR (solid lines) of Source 1 in the four signals as a function of the imaging filter length used. The major trend that we observe across all estimated SIR values is a downward bias in SIR as we use progressively longer imaging filter lengths, and this bias is due to the error incurred in using excessively long filters which is further compounded by the SIR measure. To see this, we first note that all the significant RIR filter coefficients for the RT_{60}=100ms dataset are contained in the first ~500-1000 taps (see Figure 11), thus the optimal imaging filter length will lie somewhere in this 500-1000 tap range, e.g. see [39], [40]. Since the RIR coefficients are zero beyond the optimal length, then using an imaging filter length greater than the optimal length will incur excess error for each tap that is non-zero beyond the optimal length, i.e. longer filters will incur more excess error than shorter filters.

Next, we note that in the estimation of the SIR in our 3x3 case, we need to estimate the three images of Sources 1-3, respectively, and then form the SIR as the ratio

of the Source 1 image power to the sum of the image powers of Sources 2 and 3. If we assume that the excess imaging filter estimation error is approximately the same for any source in a particular mixture, then this excess error adds the same amount of power to all imaged sources. In the SIR, the excess error contribution is twice as much in the denominator as the numerator, thus biasing the SIR value downward by an amount that is proportional to the imaging filter length.

The next significant feature in Figure 15 is the overestimation of SIR for the 20dB and 15dB cases within the optimal imaging filter-length range, e.g. 500 and 1000 taps, where the 15dB case is a mild overestimation, and the 20dB case is a large overestimation. The SIR image filter estimation is a standard signal estimation in the presence of noise problem, and high SIR scenarios present two extremes of the filter estimation problem. On one hand, estimating the image filter for Source 1 at high Source 1 SIR begins to approach image estimation without noise, while estimating the image filters of Sources 2 or 3 in a high Source 1 SIR scenario is signal estimation in the presence of overwhelming noise. The *SEA_SIR* uses the *sample* cross-correlation function in image filter estimation, and in the high Source 1 SIR cases given in this section the Source 2 and 3 image power estimates are biased downward, thereby biasing the SIR estimate for Source 1 upwards. Although there are other complicating factors such as the non-stationarity of Sources 1 and 3 and filter length mismatch, sample correlation is the culprit in over- and under-estimating SIR in high and low SIR signals, respectively.

5.3.2 Signal Analysis Example 2: SIR Analysis and Data Block Length

In this section we will analyze the SIR for an input signal set using the *SEA_SIR* plugin, i.e. the SEA plugin that reports SIR information, as well as study the effect that data-block length has in the SIR estimation process. All of the Configurations in this section use the DS_3x1_RT60-0.1s_SIRStudy dataset, where each of the four observation signals were generated to contain a mixture in which Source 1 (Female Talker #1 running speech) has a prescribed SIR value relative to the other sources, as explained in Sect. 5.2.3. The SIR values for Source 1 are 5dB in observation 1, 10dB in observation 2, 15dB in observation 3, and 20dB in observation 4.

Table 17. Configurations used in Signal Analysis Example 2. The *SEA_SIR* plugin performance is studied over a range of data-block lengths.

Config_SigAnalysis_Ex2_1000_2			
DATASET: DS_3x1_RT60-0.1s_SIRStudy.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_Static*, 0
semSelfComp	0	***SEE***	*SEE_Identity*, 0
semCDSFiltLen	-1/16	***PAS***	*PAS_Identity*, 0
semCDSFiltOffset	*semCDSFiltLen/2*	***SEA***	*SEA_SIR*, 0
semBlkLen	-2		
semBlkStep	-2		

EXTENSIONS				
Config_SigAnalysis_Ex2_1000_5	***semBlkLen***	-5	***semBlkStep***	-5
Config_SigAnalysis_Ex2_1000_10	***semBlkLen***	-10	***semBlkStep***	-10
Config_SigAnalysis_Ex2_1000_30	***semBlkLen***	-30	***semBlkStep***	-30

The Configurations used in this section are given in Table 17 and the notable common plugins of all the Configurations are the *SEP_Static*, *SEE_Identity*, *PAS_Identity*, and *SEA_SIR* plugins. The *SEP_Static*, *SEE_Identity*, and *PAS_Identity* plugins are used with respective parameter sets that, for each plugin, passes the input of the plugin directly to the plugin output. The parameter set for *SEA_SIR* is set to use a 1000 tap imaging filter for all Configurations. In this set of Configurations, we vary the data-block length, and the naming convention for the Configurations is "*Config_SigAnalysis_Ex1_1000_X*" where *X* denotes the data-block length in seconds. Here we will use data-block lengths of 2s, 5s, 10s, and 30s, and the Configurations are given in Table 17.

Figure 16 displays the known SIR (dashed lines) and the average estimated SIR (solid lines) of Source 1 in the four observation signals as a function of the data block length for a 1000 tap imaging filter. For a given block-length, the SIR shown in Figure 16 is the average over all blocks, and we note that the 30s data block-length (with a 1000 tap imaging filter) is the same as the 1000 tap imaging filter SIR values given in Figure 15. Figure 16 shows that there is a general downward bias in the SIR estimates for data-block lengths of 10s or less. As we noted in the previous section, the image filter estimation in the *SEA_SIR* plugin uses the sample cross-correlation method, and the sample cross-correlation's precision is fundamentally tied to the number of signal

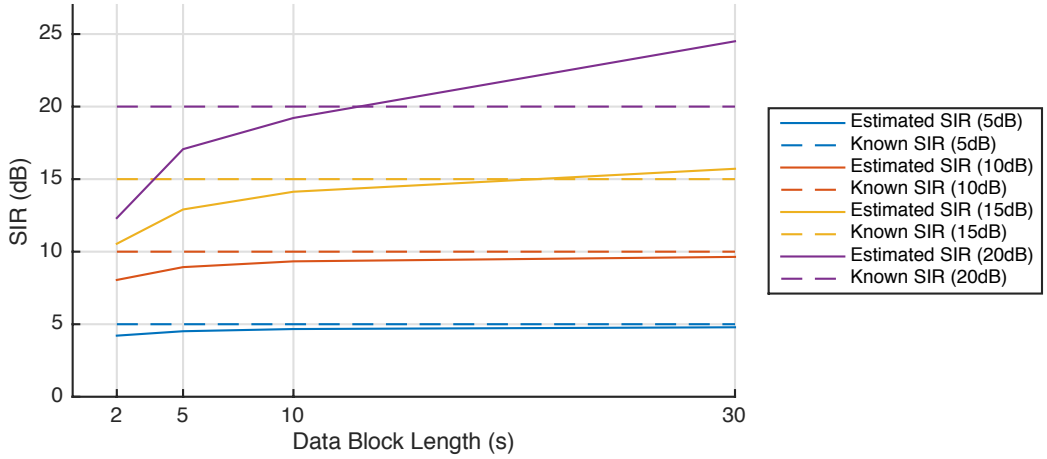

Figure 16. Signal Analysis via the MASS Framework, Example 2: SIR data block length study. Three sources were mixed using the 3x1 RT_{60}=100ms RIRs to produce four signals with SIRs for Source 1 of 5, 10, 15, and 20 dB, respectively (dashed lines). For each mixture, the SIR was estimated via the SEA_SIR plugin using different data block lengths (solid lines), and this figure shows the SIR estimation error as function of data block length.

samples used in the estimate. Thus, low-sample cross-correlation estimates incur a higher error, and in essence, the low data block-lengths in Figure 16 manifest the cross-correlation error in the SIR estimate, e.g. see [98].

To see this, we can model the image estimate for any source as the true source image plus an error term. Furthermore, we simply assume that the sample-size (data block-length) dependent cross-correlation error power is directly related to the image estimation error power and is independent of the source being imaged. For a given data block-length, the SIR will be biased downward in signals that involve more than two sources, simply because the error power in the denominator is (Q-1) times larger than the error power in the numerator, where Q is the number of sources with Q>2. Although this is a simplified analysis of the results, when we couple this with the analysis in the previous section we can see that the ratio of an estimated source's image power to the estimation process's error power becomes significant in SIR estimation.

Figure 16 also points out a general asymptotic quality of SIR estimation with respect to data block-length, where our best estimates of SIR for RT_{60}=100ms data with the *SEA_SIR* plugin come with moderate Source 1 SIR situations, e.g. 5dB and 10dB, at

Table 18. Configurations used in SASS Example 1. The supervised deconvolution SEP plugin, *SEP_SupSysId*, is evaluated via the block and batch analysis modes of the *SEA_SIR* plugin.

Config_SASSAnalysis_Ex1_2			
DATASET: DS_3x3_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_SupSysId*, 0
semSelfComp	0	***SEE***	*SEE_Identity*, 0
semCDSFiltLen	-1/16	***PAS***	*PAS_Identity*, 0
semCDSFiltOffset	*semCDSFiltLen/2*	***SEA***	*SEA_SIR*, 0
semBlkLen	-2		
semBlkStep	-2		

EXTENSIONS				
Config_SASSAnalysis_Ex1_5	***semBlkLen***	-5	***semBlkStep***	-5
Config_SASSAnalysis_Ex1_10	***semBlkLen***	-10	***semBlkStep***	-10
Config_SASSAnalysis_Ex1_30	***semBlkLen***	-30	***semBlkStep***	-30
Config_SASSAnalysis_Ex1_2_30	***SEA***	*SEA_SIR*, 10		
Config_SASSAnalysis_Ex1_5_30	***SEA***	*SEA_SIR*, 10		
	semBlkLen	-5	***semBlkStep***	-5
Config_SASSAnalysis_Ex1_10_30	***SEA***	*SEA_SIR*, 10		
	semBlkLen	-10	***semBlkStep***	-10
Config_SASSAnalysis_Ex1_30_30	***SEA***	*SEA_SIR*, 10		
	semBlkLen	-30	***semBlkStep***	-30

the largest data block-lengths, e.g. 10s and 30s. This further bolsters the simple model of sample-size dependent cross-correlation estimation and SIR-dependent SIR estimation bias from the previous section. Thus, for a 1000 tap imaging filter with RT_{60}=100ms data we can reasonably trust that the values produced by the *SEA_SIR* plugin when we use 30s or more of data, and when the SIR value is in the range ~ [-15dB 15dB]. Outside of the range we assume the magnitude of SIR is overestimated. That said, for source separation problem with air acoustic sources, 15dB SIR represents a significant, audible isolation of a source. Although we will probe the SIR analysis plugin's performance a bit further in the next section, we will use the 1000 tap imaging filter in batch mode (30s) extensively throughout the rest of this chapter.

5.4 Single Algorithm Source Separation (SASS) with Performance Analysis

As we have noted previously, the MASS framework can be used to perform standard single algorithm source separation (SASS), and here we note that the MASS framework also provides a mechanism to analyze SASS performance via the SEA plugin. In this section we will give multiple Configurations that allow the MASS framework to perform SASS, and we will study how the SEA plugin is affected by Configuration-level and SEA plugin-level processing characteristics.

5.4.1 SASS Example 1: Block vs. Batch SIR Analysis

Our first SASS example uses the supervised SEP plugin, *SEP_SupSysId*, along with the supervised SEA plugin, *SEA_SIR*, and we will look at how top-level configuration processing parameters and *SEA_SIR* component-specific parameters affect the performance analysis of a single source separation algorithm. We choose the *SEP_SupSysId* plugin for this example because it is supervised, and although some of the plugin's and Configurations' parameters chosen for this experiment may not be ideal, we can rely upon this method to perform optimally (in a FIR MSE sense) for these sub-optimal conditions. Given this qualitative summary of *SEP_SupSysId*'s performance characteristics, we will continue to study the performance characteristics of the supplied SEA plugin, *SEA_SIR* (see Sect. 5.3), but in the less controlled source *estimation* context.

The various Configurations that we use in this section are given in Table 18. As we have already stated, we will be using the *SEP_SupSysId* and *SEA_SIR* plugins, and we also note that we use the pass-through versions of the SEE and PAS plugins, *SEE_Identity* and *PAS_Identity*, respectively, for all experiments performed in this section. The *SEP_SupSysId* parameter set identification number 0 says that we use a *semCDSFilten*-length filter with a system delay of *semCDSFiltOffset* to estimate the sources, and for all Configurations in this section *semCDSFilten*=1000, and *semCDSFiltOffset*=500.

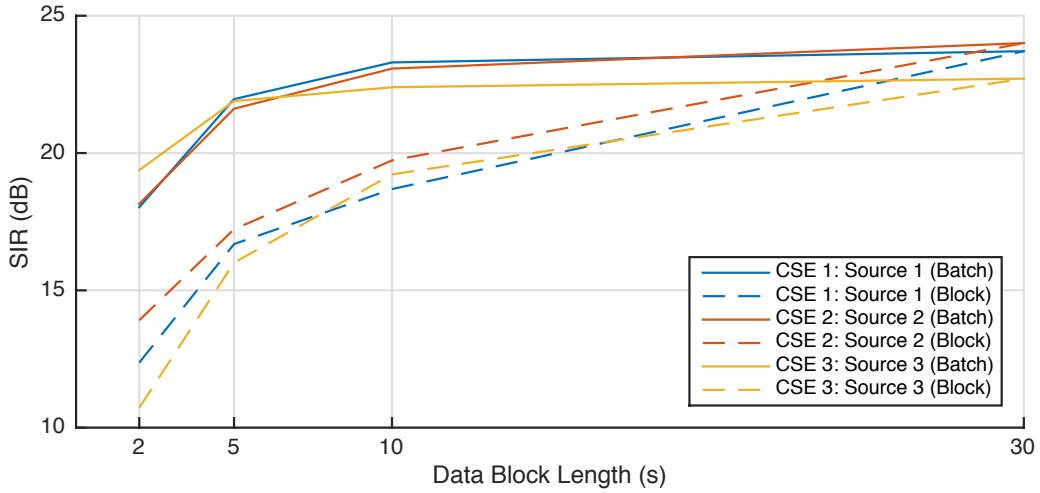

Figure 17. Single Algorithm Source Separation Performance Analysis via the MASS framework, Example 1: Block versus Batch SIR Analysis. Using a set of three mixtures consisting of three sources in a mildly reverberant environment (RT_{60} = 100ms), the SEP_SupSysId plugin was used to deconvolve the sources using a 1000 sample CDS filter length with data block lengths given on the horizontal axis. The solid lines indicate average estimated SIR value when the current data block's CDS was applied to all mixture set samples (batch), and the dashed line shows the average estimated SIR value when the current data block's CDS was applied to the current data block's samples, only (block).

For this set of examples/Configurations, we will vary the Configuration-level data-block-length and the *SEA_SIR* plugin-level analysis mode. The data-block-length specifies the amount of data that the *SEP_SupSysId* uses to provide a CDS, and in this section, we let the *SEP_SupSysId* plugin operate on the source/observation data at data block-lengths of 2s, 5s, 10s, and 30s.

The *SEA_SIR* modes determine whether the SIR of the sources is calculated using a block of data (block mode) or the entire signal (batch mode). In block mode, the *SEA_SIR* calculates the SIR of each source, at each block, using only the source/observation samples present in a block. In batch mode, *SEA_SIR* calculates the SIR of each source from the signals produced by applying the current data-block's CDS to all samples of the observations. Noting that all signals used in this work are 30s long, the 30s data-block length case produces equivalent SIR results whether the *SEA_SIR* plugin is in block or batch mode.

For each MASS CSE, Figure 17 shows the average SIR of the dominant source in the CSE as a function of block-length for the block SEP processing using batch (dashed) and block (solid) SIR analysis Configurations given in Table 18. The *SEP_SupSysId* plugin is a supervised SEP method, thus the dominant source in CSE 1 is Source 1, the dominant source in CSE 2 is Source 2, and the dominant source in CSE 3 is Source 3, and in a general blind SEP or MASS scenario we hold no assumption of a particular source being estimated at a particular CSE/MASS output. For a given block-length, the SIR in Figure 17 is the average SIR over all blocks.

In Section 5.3 we gave a cursory analysis of the *SEA_SIR* plugin for known SIR situations, and here we will apply those findings to interpret the unknown SIR situations given in Figure 17. For instance, all of the block-analysis (dashed lines) source SIR estimates in Figure 17 generally follow the 20dB SIR data block-length specific line in Figure 16 (solid purple). Given our simple analysis of the *SEA_SIR* plugin in Section 5.3, we can reasonably assume that all of the source estimates from the *SEP_SupSysId* plugin provide a true SIR of approximately 20dB.

Although a more in-depth analysis of the SIR method, *SEA_SIR*, is warranted, we can see in Figure 17 that block SIR analysis under-estimates the SIR for block lengths less than 10s. Although block-processing and block-analysis are very relevant in time-varying scenes, all of the scenes (datasets) studied in this work are time-invariant. We have used the studies in this and the previous section to inform our MASS studies beginning in Sect. 5.5, where we will use a 1000 tap imaging filter for the *SEA_SIR* plugin operating in batch (30s) mode and a SEP data block-length of 5s for the RT_{60}=100ms datasets. We recognize that the *SEA_SIR* plugin needs improvement, but we are confident that the analysis provided by *SEA_SIR* with the above parameters can be usefully interpreted.

5.4.2 SASS Example 2: CDS Filter Length Mismatch

In our second supervised SASS example using the MASS framework, we will only concern ourselves with supervised batch SEP processing and supervised batch SEA performance, and we will continue our probing of SIR estimates by looking at the effect of under-specified filter-lengths in the *SEA_SIR* plugin. Although we could choose a

Table 19. Configurations used in SASS Analysis Example 2. The *SEP_SupSysId* SEP plugin is evaluated by a batch mode *SEA_SIR* plugin over varied RT_{60} datasets.

Config_SASSAnalysis_Ex2_100			
DATASET: DS_3x3_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_SupSysId*, 0
semSelfComp	0	***SEE***	*SEE_Identity*, 0
semCDSFiltLen	-1/16	***PAS***	*PAS_Identity*, 0
semCDSFiltOffset	*semCDSFiltLen/2*	***SEA***	*SEA_SIR*, 10
semBlkLen	-30		
semBlkStep	-30		

EXTENSIONS	
Config_SASSAnalysis_Ex2_150	**DATASET:** DS_3x3_RT60-0.15s.mat
Config_SASSAnalysis_Ex2_200	**DATASET:** DS_3x3_RT60-0.2s.mat

particular dataset and vary the filter length used by *SEA_SIR*, here we choose to keep the filter length static and use our various RT_{60} datasets to display a more general problem. That is, if we set our data-block length to 30s, as in Table 19, then a *SEP_SupSysId* plugin will produce a single CDS using the entire 30s of observation and source data, and a *SEA_SIR* plugin in batch mode will estimate the SIR of each source using the entire 30s of CSE and source data using a parameter set dependent imaging filter-length. Table 19 indicates that the Configurations in this section keep the imaging filter-length of *SEA_SIR* fixed at *semCDSFiltLen*=1000 taps and use three datasets with RT_{60} values of 100ms, 150ms, and 200ms, respectively.

The results of the Configurations used in this section are given in Figure 18. Figure 18 gives the estimated SIR of the dominant source in each CSE for each RT_{60} dataset. For the batch processing and batch analysis used here, the general trend given in Figure 18 is a progressively lower estimated SIR value for progressively higher RT_{60} values, respectively. Figure 18 is fairly uninformative, because even with our past studies on the *SEA_SIR* plugin, we cannot discern whether the *SEP_SupSysId* plugin with a fixed CDS length or the *SEA_SIR* plugin with a fixed imaging filter length (or both) produces poor estimates at higher RT_{60} values. We provide this example simply to inform and warn the user of what is possible using the MASS framework. That said, we can simply extend the various MASS signal analysis Configurations to study the SIR

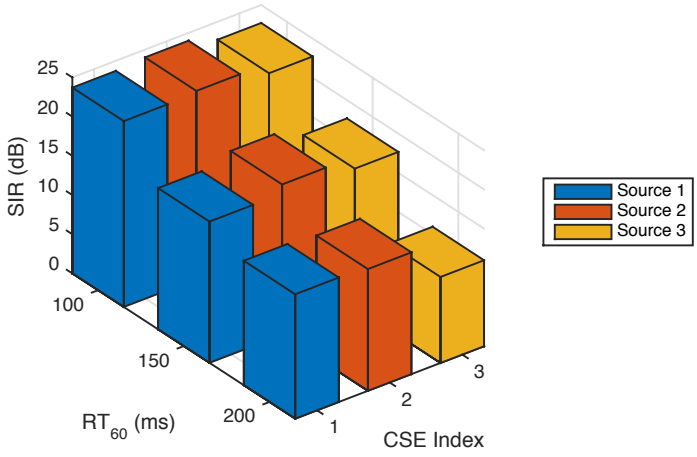

Figure 18. Single Algorithm Source Separation Performance Analysis via the MASS framework, Example 2: CDS Filter Length Mismatch. The SEP_SupSysId plugin was used to deconvolve 3 sources from multiple 3 mixture sets using a 1000 sample CDS filter length, where the RIRs used to create the mixtures varied for RT_{60}=100ms, 150ms, and 200ms. The results in the figure are inconclusive, since the degradation in SIR for higher RT_{60} values could be due to insufficient CDS filter length in the deconvolution or insufficient filter length in the SIR analysis.

imaging filter length with respect to RT_{60} to clarify our results here, but for now we leave this as future work, since the rest of the Configurations in this chapter use the RT_{60}=100ms data.

5.5 Multiple Algorithm Source Separation

In this section we will give Configurations that display how to use the MASS framework to perform multiple algorithm source separation, i.e. MASS. As we will see in the next section, even individual source separation algorithms can benefit from the MASS framework by using the self-competition capability

5.5.1 MASS Example 1: SASS Enhancement via MASS Self-Competition

In this section we will display the self-competition capability of the MASS framework by enhancing SASS Configurations. That is, we will give several Configurations that only contain one SEP plugin, but we will run the MASS framework

Table 20. Configurations used in MASS Example 1a. The *SEP_SupSysId* plugin is evaluated for SASS as a lone SEP plugin and using blind and supervised SEE plugins with MASS self-competiton.

Config_MASS_Ex1a_SysId			
DATASET: DS_3x3_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_SupSysId*, 0
semSelfComp	0	***SEE***	*SEE_Identity*, 0
semCDSFiltLen	-1/16	***PAS***	*PAS_CSEPrevXCorr*, 1
semCDSFiltOffset	*semCDSFiltLen/2*	***SEA***	*SEA_SIR*, 10
semBlkLen	-5		
semBlkStep	-5		

EXTENSIONS					
Config_MASS_Ex1a_SysId_BSC					
semSelfComp	1	***SEE***	*SEE_MinXCorrSelect*, 0	***SNUM***	*SNUM_NumObs*, 0
Config_MASS_Ex1a_SysId_SSC					
semSelfComp	1	***SEE***	*SEE_SupSIRSelect*, 0		

in self-competition mode so that the single SEP plugin's current demixing solution will compete with the SEP plugin's past demixing solutions.

All of the Configurations presented in this section will use several identical Configuration property values. Each Configuration uses the 3x3 100ms RT_{60} dataset, the sampling rate is set to 16kHz, the CDS filter length is 1000 taps, the CDS filter delay is set to a 500 sample lag, and the blind *PAS_CSEPrevXcorr* is used as the PAS plugin. The data block-length and block-step are each set to 5s, thus the SEP plugins work on 5s of data at a time. The *SEA_SIR* plugin is used with parameter set id 10, thus the SIR imaging filter length is equal to the CDS filter length (1000 taps) and the SIR analysis works in batch mode, i.e. over the entire 30s of signal data.

We will look at four SEP plugins in this section, and for each SEP plugin we will create three Configurations which differ by the combination of self-competition mode and the SEE plugin employed. For example, Table 20 gives the Configuration for studying self-competition with the *SEP_SupSysId* SEP method. The base Configuration, *Config_MASS_Ex1a_SysId*, uses the *SEE_Identity* SEE plugin but does not use self-competition, thus this configuration is essentially the Configuration for *SEP_SupSysId*

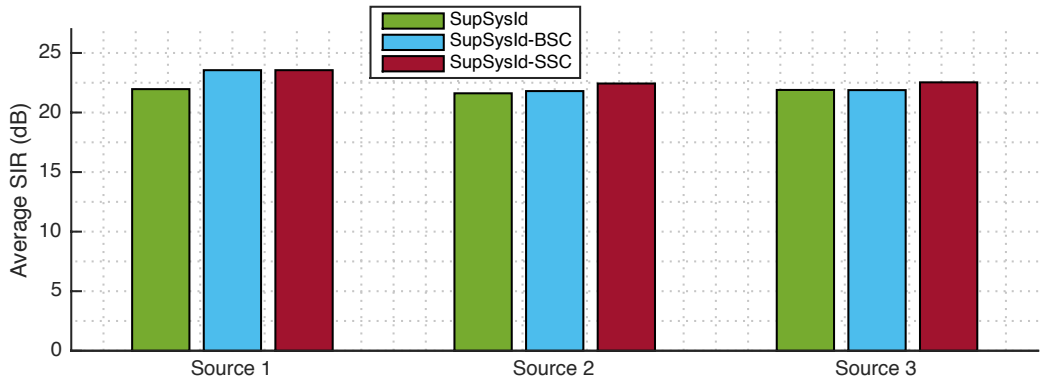

Figure 19. Single Algorithm Source Separation Enhancement via MASS Self-Competition using the SEP_SupSysID plugin. For a three source, three mixture set, the SEP_SupSysId supervised source deconvolution plugin was run by itself (green), using self-competition with the blind SEE_MinXCorrSelect evaluation plugin (blue), and using self-competition with the supervised SEE_SupSIRSelect evaluation plugin (red).

SASS. The next configuration, *Config_MASS_Ex1a_SysId_BSC*, uses a blind SEE plugin, *SEE_MinXCorrSelect*, and works
with MASS self-competition turned on, and we term this a blind self-competition (BSC) Configuration. The last Configuration, *Config_MASS_Ex1a_SysId_SSC*, is termed supervised self-competition (SSC), since the SEE method, *SEE_SupSIRSelect*, is supervised, and the Configuration works with MASS self-competition turned on.

If we note that *SEP_SupSysId* is a supervised SEP method, then the combination of *SEP_SupSysId* with either *SEE_Identity* or *SEE_SupSIRSelect* results in a supervised MASS application, whereas the combination of *SEP_SupSysId* with the blind *SEE_MinXCorrSelect* plugin results in a semi-blind MASS application. Tables 21-24 give the example Configurations when we use the blind *SEP_MCLP*, *SEP_TTSE*, and *SEP_ABYK* plugins, respectively, and we can make analogous statements about MASS application blindness. That is, using a blind SEP plugin with a pass-through SEE plugin, e.g. *SEE_Identity*, or a blind SEE plugin, e.g. *SEE_MinXCorrSelect*, results in a blind MASS application, whereas using a blind SEP plugin with a supervised SEE plugin, e.g. *SEE_SupSIRSelect*, results in a semi-blind MASS application.

Figures 18-21 show the average SIR over the data blocks for the maximum SIR CSE of each source for the *SEP_SupSysId*, *SEP_MCLP*, *SEP_TTSE*, and *SEP_ABYK*

Table 21. Configurations used in MASS Example 1b. The *SEP_MCLP* plugin is evaluated for SASS as a lone SEP plugin and using blind and supervised SEE plugins with MASS self-competiton.

Config_MASS_Ex1b_MCLP			
DATASET: DS_3x3_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_MCLP*, 0
semSelfComp	0	***SEE***	*SEE_Identity*, 0
semCDSFiltLen	-1/16	***PAS***	*PAS_CSEPrevXCorr*, 1
semCDSFiltOffset	*semCDSFiltLen/2*	***SEA***	*SEA_SIR*, 10
semBlkLen	-5		
semBlkStep	-5		

EXTENSIONS					
Config_MASS_Ex1b_MCLP_BSC					
semSelfComp	1	***SEE***	*SEE_MinXCorrSelect*, 0	***SNUM***	*SNUM_NumObs*, 0
Config_MASS_Ex1b_MCLP_SSC					
semSelfComp	1	***SEE***	*SEE_SupSIRSelect*, 0		

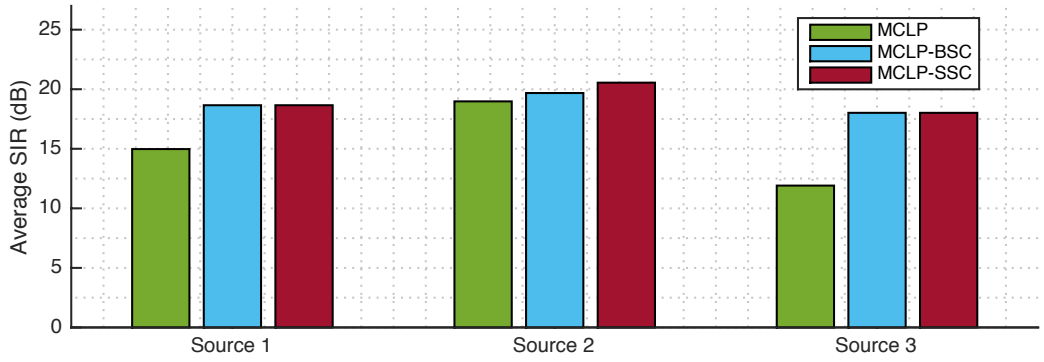

Figure 20. Single Algorithm Source Separation Enhancement via MASS Self-Competition using the SEP_MCLP plugin. For a three source, three mixture set, the SEP_MCLP blind source extraction plugin was run by itself (green), using self-competition with the blind SEE_MinXCorrSelect evaluation plugin (blue), and using self-competition with the supervised SEE_SupSIRSelect evaluation plugin (red).

plugins, respectively, when the SEP plugin is run by itself, with blind self-competition (BSC), and with supervised self-competition. To begin, we note that all sources in all mixtures of the DS_3x3_RT60-0.1s dataset have SIRs on the range -7dB to -1dB. Figure

19 shows that the *SEP_SupSysId* plugin provides an average SIR of greater than 20dB for all sources and for all cases, which is excellent separation for convolutive mixtures. The SSC case provides a marginal 1-2dB gain over the lone plugin case, while the BSC case provides a marginal SIR gain for Source 1 only, with Sources 2 and 3 performing almost identically to the lone plugin case.

In the *SEP_MCLP* cases given in Figure 20, the lone plugin separates Sources 1 and 2 fairly well by providing an average SIR of ~15dB and ~18dB respectively, while the lone *SEP_MCLP* only provides a 12dB SIR for Source 3. Both of the BSC and SSC cases provide a 3-5dB SIR gain over the lone plugin case for Sources 1 and 3 and provides a marginal gain for Source 2.

Figure 21 gives the results for the *SEP_TTSE* cases. The lone plugin provides an average 10dB SIR for Source 1, 16dB SIR for Source 3, and -20dB SIR for Source 2. Although the BSC and SSC have comparable performance they only provide a marginal 1-3dB gain over the lone plugin's Source 1 and 3 estimates. The SSC provides a massive 18dB gain for Source2 while the BSC provides a -3dB loss for Source 2, but with that said, the SSC has an average of -2dB SIR for Source 2.

The *SEP_TTSE* plugin is designed to separate out intermittent sources, and Source 3 was designed to be an intermittent source, thus we would expect that the *SEP_TTSE* method would separate out Source 3 well, and the results in Figure 21 support this. Source 2 is a stationary noise signal, thus the *SEP_TTSE* plugin is incapable of separating out this source. Source 1 is a continuous speech, and in theory the *SEP_TTSE* plugin will not be able to isolate this source. However, about 25s into the Source 1 signal, the talker pauses briefly to take a breath, and the *SEP_TTSE* method is able to use this momentary silence to provide a demixing solution. In fact, as we will see in the next section, the *SEP_TTSE* is actually estimating Source 3 at all outputs except at the 25s data-block, where *SEP_TTSE* is able to isolate Source 1. The *SEP_TTSE* provides a 20dB SIR for Source 1 at the 25s data-block, but when averaged with the poor performance in previous data-blocks, the *SEP_TTSE* method provides an average 12dB SIR for Source 1.

Table 22. Configurations used in MASS Example 1c. The *SEP_TTSE* plugin is evaluated for SASS as a lone SEP plugin and using blind and supervised SEE plugins with MASS self-competition.

Config_MASS_Ex1c_TTSE			
DATASET: DS_3x3_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_TTSE*, 0
semSelfComp	0	***SEE***	*SEE_Identity*, 0
semCDSFiltLen	-1/16	***PAS***	*PAS_CSEPrevXCorr*, 1
semCDSFiltOffset	*semCDSFiltLen/2*	***SEA***	*SEA_SIR*, 10
semBlkLen	-5		
semBlkStep	-5		

EXTENSIONS					
Config_MASS_Ex1c_TTSE_BSC					
semSelfComp	1	***SEE***	*SEE_MinXCorrSelect*, 0	***SNUM***	*SNUM_NumObs*, 0
Config_MASS_Ex1c_TTSE_SSC					
semSelfComp	1	***SEE***	*SEE_SupSIRSelect*, 0		

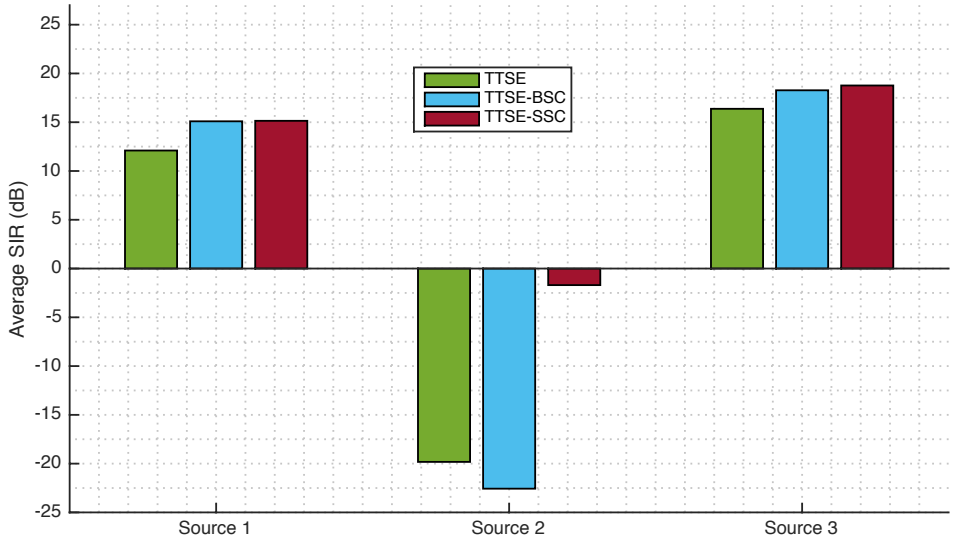

Figure 21. Single Algorithm Source Separation Enhancement via MASS Self-Competition using the SEP_TTSE plugin. For a three source, three mixture set, the SEP_TTSE blind source extraction plugin was run by itself (green), using self-competition with the blind SEE_MinXCorrSelect evaluation plugin (blue), and using self-competition with the supervised SEE_SupSIRSelect evaluation plugin (red).

Table 23. Configurations used in MASS Example 1d. The *SEP_ABYK* plugin is evaluated for SASS as a lone SEP plugin and using blind and supervised SEE plugins with MASS self-competiton.

Config_MASS_Ex1d_ABYK			
DATASET: DS_3x3_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_ABYK*, 2
semSelfComp	0	***SEE***	*SEE_Identity*, 0
semCDSFiltLen	-1/16	***PAS***	*PAS_CSEPrevXCorr*, 1
semCDSFiltOffset	*semCDSFiltLen/2*	***SEA***	*SEA_SIR*, 10
semBlkLen	-5		
semBlkStep	-5		

EXTENSIONS					
Config_MASS_Ex1d_ABYK_BSC					
semSelfComp	1	***SEE***	*SEE_MinXCorrSelect*, 0	***SNUM***	*SNUM_NumObs*, 0
Config_MASS_Ex1d_ABYK_SSC					
semSelfComp	1	***SEE***	*SEE_SupSIRSelect*, 0		

Figure 22 gives the results for the *SEP_ABYK* Configurations, where the lone plugin, the BSC, and the SSC cases all provide comparable results. The *SEP_ABYK* method provides an average 17dB SIR for Source 3, while only providing 7dB SIR for Source 1 and 3dB SIR for Source 2. The *SEP_ABYK* is a learning method, so the average SIR is biased against this method. Figure 23 displays the time-varying SIR for the *SEP_ABYK* method, and we can see that even though SEP_ABYK provides 10dB, 6dB, and 19dB SIR for Source 1,2, and 3, respectively, at t=30s, the average SIR presented in Figure 22 biases the SIR down due to the learning phase.

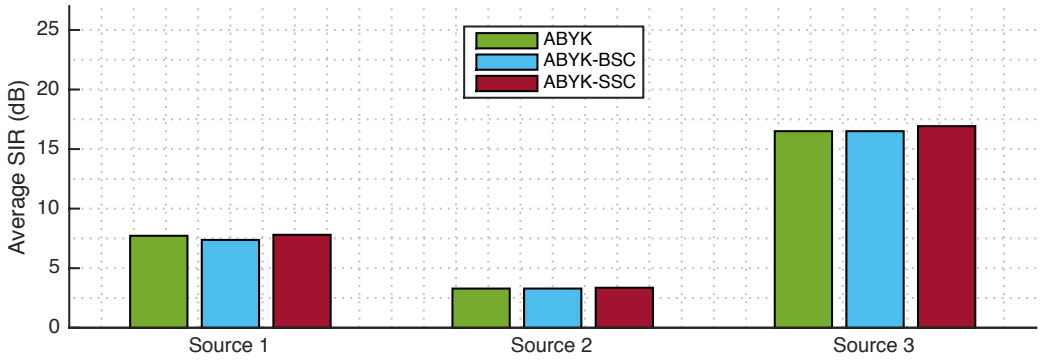

Figure 22. Single Algorithm Source Separation Enhancement via MASS Self-Competition using the SEP_ABYK plugin. For a three source, three mixture set, the SEP_ABYK blind source separation plugin was run by itself (green), using self-competition with the blind SEE_MinXCorrSelect evaluation plugin (blue), and using self-competition with the supervised SEE_SupSIRSelect evaluation plugin (red).

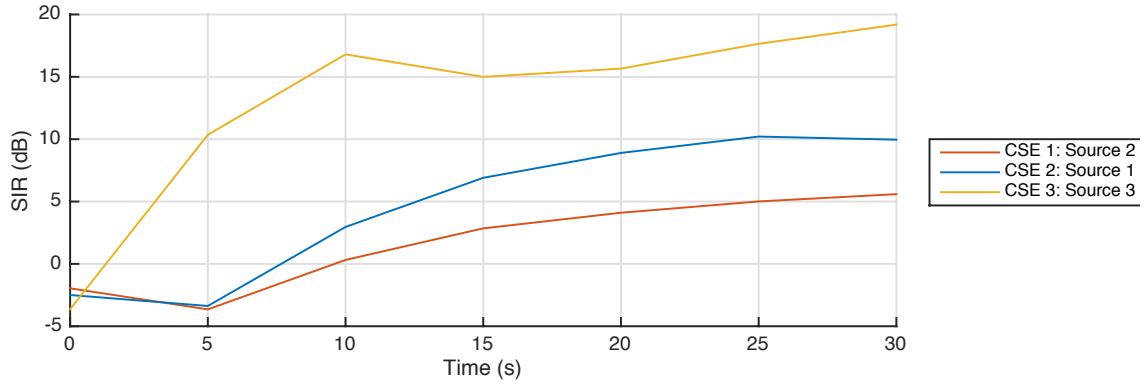

Figure 23. Time-varying SIR of SEP_ABYK. The dominant source SIR in each CSE is displayed, where CSE 1 estimates Source 2, CSE 2 estimates Source 1, and CSE 3 estimates Source 3. SEP_ABYK is an adaptive method, thus the sources' demixing solutions are learned over time.

5.5.2 MASS Example 2: Semi-Blind Competitive MASS

In this example, we will allow multiple blind SEP algorithms to compete under a supervised, selective SEE method to produce a CSE set. We will study this MASS behavior for the three RT_{60}=100ms datasets, DS_3x3_RT60-0.1s, DS_3x4_RT60-0.1s,

Table 24. Configurations used in MASS Example 2a. Semi-blind multiple algorithm source separation using the blind *SEP_MCLP*, *SEP_TTSE*, and *SEP_ABYK* SEP plugins and the supervised *SEE_SupSIRSelect* SEE plugin with individual analysis of the constituent SEP plugins for the DS_3x3_RT60-0.1s dataset.

Config_MASS_Ex2a_3x3			
DATASET: DS_3x3_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_MCLP*, 0
semSelfComp	0		*SEP_TTSE*, 0
semCDSFiltLen	-1/16		*SEP_ABYK*, 2
semCDSFiltOffset	*semCDSFiltLen/2*	***SEE***	*SEE_SupSIRSelect*, 0
semBlkLen	-5	***PAS***	*PAS_CSEPrevXcorr*, 1
semBlkStep	-5	***SEA***	*SEA_SIR*, 10

EXTENSIONS				
Config_MASS_Ex2a_3x3_MCL	***SEP***	*SEP_MCLP, 0*	***SEE***	*SEE_Identity*, 0
Config_MASS_Ex2a_3x3_TTSE	***SEP***	*SEP_TTSE, 0*	***SEE***	*SEE_Identity*, 0
Config_MASS_Ex2a_3x3_ABYK	***SEP***	*SEP_ABYK, 2*	***SEE***	*SEE_Identity*, 0

and DS_4x4_RT60-0.1s, where the datasets differ in the number of sources and number of sensors. For each of these datasets, we will create four Configurations, where the first Configuration allows the blind *SEP_MCLP, SEP_TTSE*, and *SEP_ABYK* SEP plugin components to compete under the supervised *SEE_SupSIRSelect* SEE method, and the other three Configurations give the behavior of the individual SEP algorithms under the *SEE_Identity* SEE method. All Configurations in this section use the same processing parameters, the same *PAS_CSEPrevXcorr* PAS plugin, and the same batch mode *SEA_SIR*, and the Configurations for the three datasets are given in Tables 24-26.

For each set of Configurations, we will produce a cluster of four bar graphs, where each bar graph summarizes the performance of a particular Configuration. The bars in the graphs display are ordered by the Configuration's CSE index, the color coding indicates the particular source being estimated, and the height of the bar indicates the SIR level. For each cluster, subplot a) is the MASS Configuration results, and b)-d) are the results of the individual constituent SEP plugins used in the MASS Configuration. For example, Figure 24 a) shows the results of the MASS performance when the *SEP_MCLP*, *SEP_TTSE*, and *SEP_ABYK* compete and process the DS_3x3_RT60-0.1s dataset. For the MASS results, Source 1 has the maximum average SIR in CSE 1, Source 2 has the

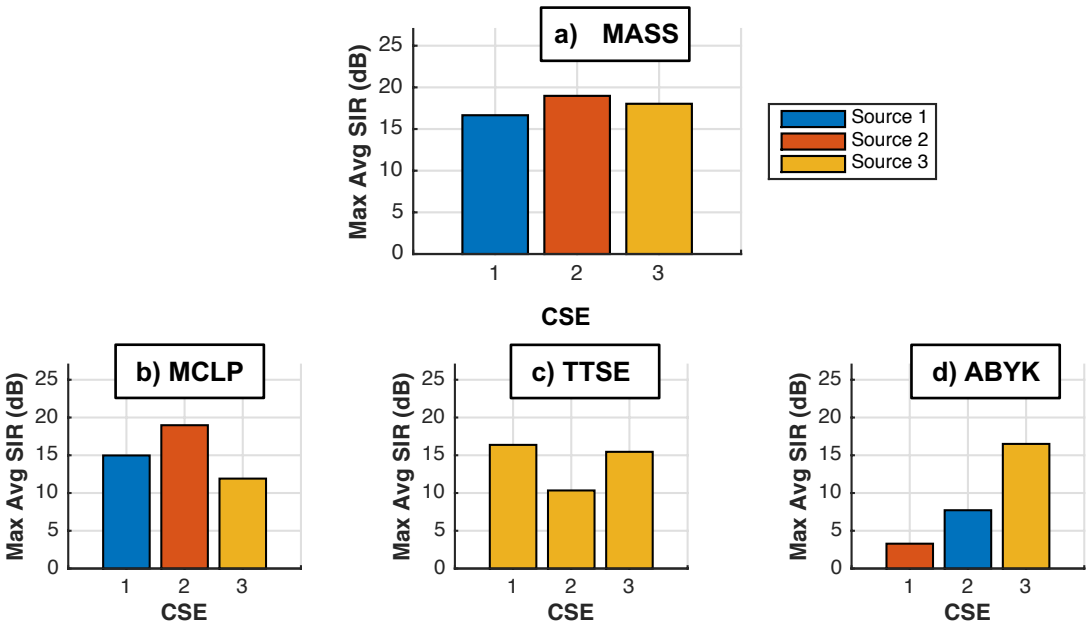

Figure 24. Semi-Blind Multiple Algorithm Source Separation via the MASS Framework, Example 2a: three source, three mixture set. Three blind SEP plugins, SEP_MCLP, SEP_TTSE, and SEP_ABYK, compete under the supervised SEE_SupSIRSelect evaluation plugin. Figure a) gives the maximum average SIR for each of the CSEs, where the maximum is taken over the source set, i.e. the maximum average SIR over all sources for CSE 1 corresponds to Source 1. Figures b), c), and d) give the maximum average SIR results for respective constituent algorithms, SEP_MCLP, SEP_TTSE, and SEP_ABYK.

maximum average SIR in CSE 2, and Source 3 has the maximum average SIR in CSE 3. Figure 24 b)-d) show the results for the individual plugin Configurations, and the color of the bar shows which source has the maximum average SIR in a particular source estimate.

Figure 24 points out several interesting characteristics of our Configurations. First, the results for the individual *SEP_TTSE* Configuration given in Figure 24 c) show that the *SEP_TTSE* method outputs three estimates of Source 3. As we mentioned earlier, the *SEP_TTSE* method is designed to estimate intermittent sources, and Source 3 was designed to be an intermittent source. The *SEP_TTSE* plugin outputs *three* source estimates because it is a blind method that simply assumes that the observations are critically-determined, and therefore, it produces the same number of source estimates as the number of observations.

As shown in Figure 24 d), the *SEP_ABYK* plugin's CSE 1 is an estimate of Source 2, CSE 2 is an estimate of Source 1, and CSE 3 is an estimate of Source 3. The individual *SEP_ABYK* uses a blind SEE method, *SEE_Identity*, thus a permutation of the sources is perfectly valid. We could make a similar statement about the *SEP_TTSE*'s performance in Figure 24 c) but to an extreme, since estimating a single source at all outputs is also perfectly valid. We use a supervised SEA plugin, *SEE_SupSIRSelect*, for all the Configurations in this sections, so the SEE plugin's outputs will necessarily be ordered by the source signals in the datasets. Although we could have used the simpler *PAS_Identity* plugin to solve the permutation problem and achieved the same MASS results in Figure 24 a), we simply note that we use the blind *PAS_CSEPrevXcorr* PAS plugin in all Configurations in this section, and this method does not fail. The previous statement is a long way from providing strong confidence in our blind PAS plugin, and we are simply stating that it must handle this simple case to even be considered as a valid PAS method.

The most striking feature shown in Figure 24, is that the MASS estimates of Sources 1 and 3 achieve a higher average SIR than any of the individual constituent SEP plugins employed. That is, Figure 24 a) shows that the MASS CSE 1 produces a 16dB SIR estimate of Source 1 and a 18dB SIR estimate of Source 3, when the maximum SIR across all constituent SEP methods for Source 1 is 15dB from *SEP_MCLP* (Figure 24 b)), and the maximum SIR estimate for Source 3 is essentially a tie between *SEP_TTSE* and *SEP_ABYK* at 16dB. Although these are marginal SIR gains, they are gains, and we are only dealing with an SEE plugin that uses selection.

To better understand how a simple selection process can improve the quality of our source estimates, we will now look at the composition of the MASS CSE 1 and CSE 3 in Figures 23 and 24, respectively. Figure 25 displays the (batch) estimated SIR for CSE 1 of our MASS output, as well as CSE 1 from *SEP_MCLP* and CSE 3 from *SEP_TTSE*, at each 5s data-block interval. Clearly for the first 20s, the MASS CSE 1 simply selects the output of *SEP_MCLP*, but at 25s the *SEP_TTSE* plugin provides the best estimate of the source that had been estimated in CSE 1, i.e. Source 1. In the last section, we mentioned that Source 1 is continuous speech, but during the 25s data-block, the talker took a brief pause to breathe, and the *SEP_TTSE* was able to produce a demixing solution for Source 1. In the MASS setting, we are able to take advantage of

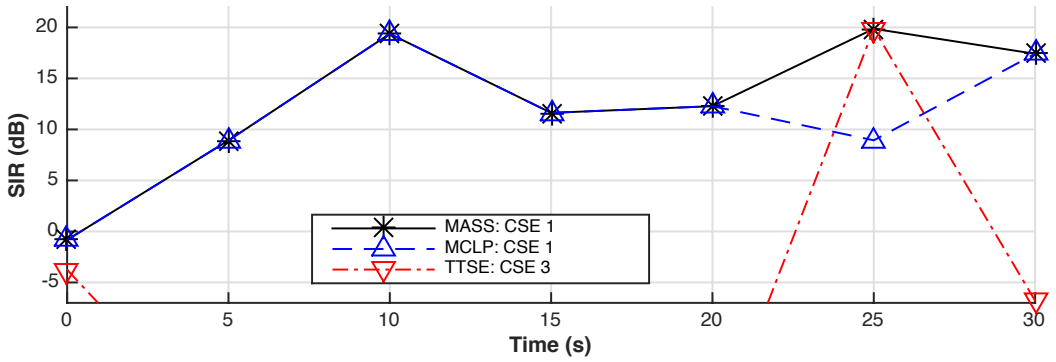

Figure 25. The components of MASS CSE 1 in MASS Example 2a. The time-varying SIR for MASS CSE 1 from MASS Example 2a is given by the solid black line, marked with an asterisk. The MASS CSE 1 is comprised of the source estimates produced by the SEP_MCLP plugin's CSE 1 (dashed blue line, marked by upward-pointed triangle), and the SEP_TTSE plugin's CSE 2 (dash-dot red line, marked by downward-pointed triangle).

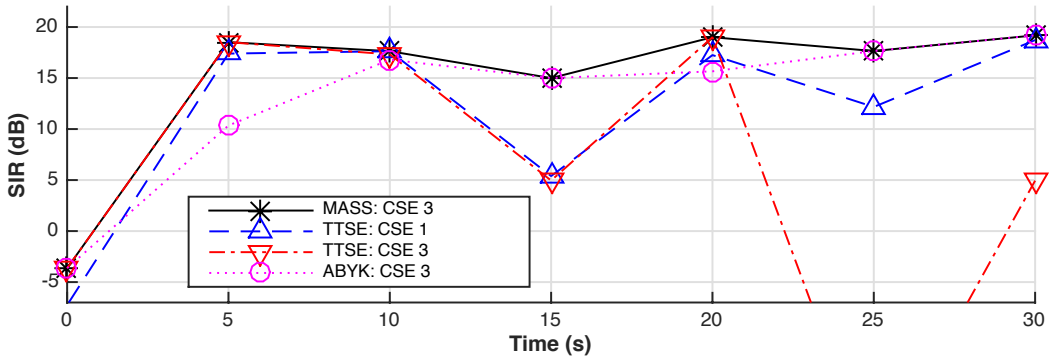

Figure 26. The components of MASS CSE 3 in MASS Example 2a. The time-varying SIR for MASS CSE 3 from MASS Example 2a is given by the solid black line, marked with an asterisk. The MASS CSE 3 is comprised of the source estimates produced by the SEP_TTSE plugin's CSE 1 (dashed blue line, marked by upward-pointed triangle) and CSE 3 (dash-dot red line, marked by downward-pointed triangle), as well as the source estimate produced from the SEP_ABYK plugin's CSE 3 (dotted purple line, marked by circle).

these localized source estimate enhancements, but since we were not using self-competition in this Configuration, the *SEP_MCLP* estimate in the last data-block is chosen for MASS CSE 1 even though the *SEP_TTSE* estimate at 25s was a better estimate. Figure 26 shows that MASS CSE 3 can similarly be decomposed into the contributions from the various SEP plugins' source estimates.

Table 25. Configurations used in MASS Example 2b. Semi-blind multiple algorithm source separation using the blind *SEP_MCLP*, *SEP_TTSE*, and *SEP_ABYK* SEP plugins and the supervised *SEE_SupSIRSelect* SEE plugin with individual analysis of the constituent SEP plugins for the DS_3x4_RT60-0.1s dataset.

Config_MASS_Ex2b_3x4			
DATASET: DS_3x4_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_MCLP*, 0
semSelfComp	0		*SEP_TTSE*, 0
semCDSFiltLen	-1/16		*SEP_ABYK*, 2
semCDSFiltOffset	*semCDSFiltLen/2*	***SEE***	*SEE_SupSIRSelect*, 0
semBlkLen	-5	***PAS***	*PAS_CSEPrevXcorr*, 1
semBlkStep	-5	***SEA***	*SEA_SIR*, 10

EXTENSIONS				
Config_MASS_Ex2b_3x4_MCLP	***SEP***	*SEP_MCLP, 0*	***SEE***	*SEE_Identity*, 0
Config_MASS_Ex2b_3x4_TTSE	***SEP***	*SEP_TTSE, 0*	***SEE***	*SEE_Identity*, 0
Config_MASS_Ex2b_3x4_ABYK	***SEP***	*SEP_ABYK, 2*	***SEE***	*SEE_Identity*, 0

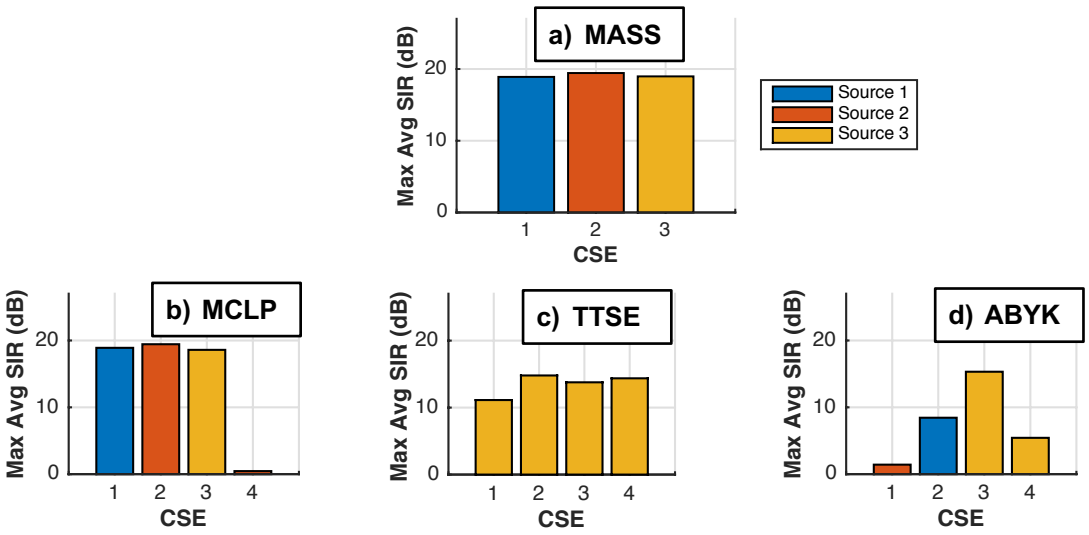

Figure 27. Semi-Blind Multiple Algorithm Source Separation via the MASS Framework, Example 2b: three source, four mixture set. Three blind SEP plugins, SEP_MCLP, SEP_TTSE, and SEP_ABYK, compete under the supervised SEE_SupSIRSelect evaluation plugin. Figure a) gives the maximum average SIR for each of the CSEs, where the maximum is taken over the source set, i.e. the maximum average SIR over all sources for CSE 1 corresponds to Source 1. Figures b), c), and d) give the maximum average SIR results for respective constituent algorithms, SEP_MCLP, SEP_TTSE, and SEP_ABYK.

Table 26. Configurations used in MASS Example 2c. Semi-blind multiple algorithm source separation using the blind *SEP_MCLP*, *SEP_TTSE*, and *SEP_ABYK* SEP plugins and the supervised *SEE_SupSIRSelect* SEE plugin with individual analysis of the constituent SEP plugins for the DS_4x4_RT60-0.1s dataset.

Config_MASS_Ex2c_4x4			
DATASET: DS_4x4_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_MCLP*, 0
semSelfComp	0		*SEP_TTSE*, 0
semCDSFiltLen	-1/16		*SEP_ABYK*, 2
semCDSFiltOffset	*semCDSFiltLen/2*	***SEE***	*SEE_SupSIRSelect*, 0
semBlkLen	-5	***PAS***	*PAS_CSEPrevXcorr*, 1
semBlkStep	-5	***SEA***	*SEA_SIR*, 10

EXTENSIONS				
Config_MASS_Ex2c_4x4_MCLP	***SEP***	*SEP_MCLP, 0*	***SEE***	*SEE_Identity*, 0
Config_MASS_Ex2c_4x4_TTSE	***SEP***	*SEP_TTSE, 0*	***SEE***	*SEE_Identity*, 0
Config_MASS_Ex2c_4x4_ABYK	***SEP***	*SEP_ABYK, 2*	***SEE***	*SEE_Identity*, 0

Our next Configuration set is given in Table 25, and it only differs from the previous example by using a 3x4 dataset, and the results of these Configurations are given in Figure 27. One of the main distinctions between this Configuration set and the 3x3 Configuration set is that all of the individual SEP plugins output four source estimates, because all of the SEP plugins blindly assume critically-determined observations. However, the SEE plugin, *SEE_SupSIRSelect*, is supervised, thus it knows the number of sources and only selects three estimates for the MASS CSE set output.

The second distinction from the 3x3 dataset is that *SEP_MCLP* consistently provides the best estimates for all sources, thus the supervised SEE plugin simply selects, at each data-block, the *SEP_MCLP* source estimates that provide maximum SIR for all sources. The underlying source extraction technique used in *SEP_MCLP* is capable of extracting sources in a critically-determined scenario, but the method excels in over-determined environments such as the 3x4 dataset.

The last set of Configurations and the corresponding results use the DS_4x4_RT60-0.1s dataset and are given in Table 26 and Figure 28, respectively. The MASS CSE set is simply comprised of the estimates for Sources 1, 2, and 4 from the *SEP_MCLP* plugin, and the estimate of Source 3 from the *SEP_ABYK* plugin. The

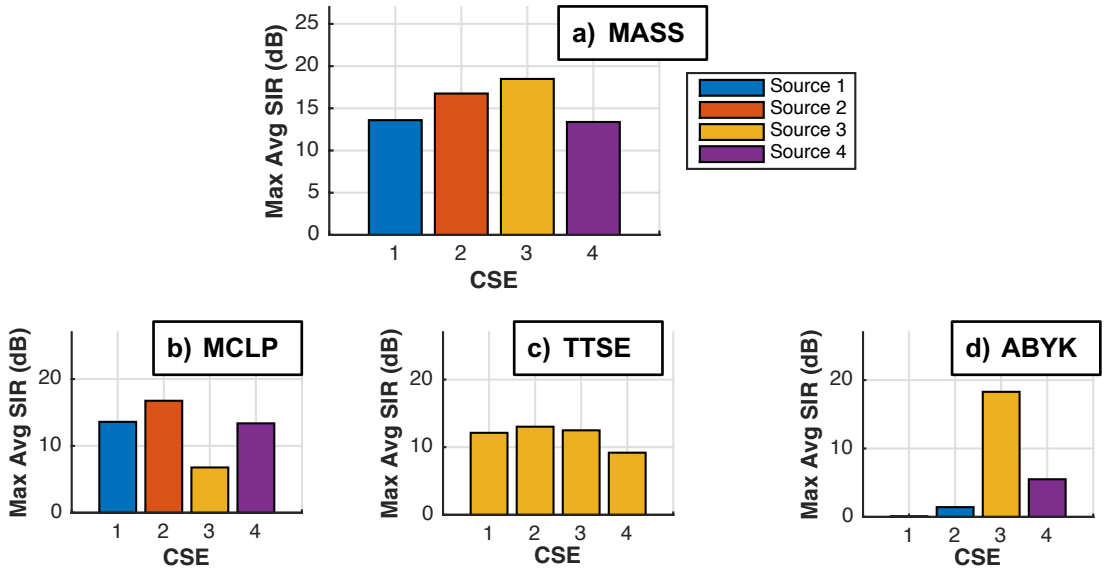

Figure 28. Semi-Blind Multiple Algorithm Source Separation via the MASS Framework, Example 2c: four source, four mixture set. Three blind SEP plugins, SEP_MCLP, SEP_TTSE, and SEP_ABYK, compete under the supervised SEE_SupSIRSelect evaluation plugin. Figure a) gives the maximum average SIR for each of the CSEs, where the maximum is taken over the source set, i.e. the maximum average SIR over all sources for CSE 1 corresponds to Source 1. Figures b), c), and d) give the maximum average SIR results for respective constituent algorithms, SEP_MCLP, SEP_TTSE, and SEP_ABYK.

overall separation performance of the MASS Configuration is better than any of the constituent BSE/BSS plugin methods.

5.5.3 MASS Example 3: Blind MASS and Blind Self-Competiton

In this section we will study multiple blind SEP algorithms competing under a blind SEE method, and we will look at the MASS behavior with and without self-competition. We will study this MASS behavior for the three RT_{60}=100ms datasets, i.e. DS_3x3_RT60-0.1s, DS_3x4_RT60-0.1s, and DS_4x4_RT60-0.1s, where the datasets differ in the number of sources and number of sensors. For each of these datasets, we will create two Configurations, where the first Configuration allows the blind *SEP_MCLP, SEP_TTSE*, and *SEP_ABYK* SEP plugin components to compete under the blind *SEE_MinXCorrSelect* SEE method without MASS self-competition, and the other Configuration is identical except that MASS self-competition is employed. All

Table 27. Configurations used in MASS Examples 3a)-c). Blind multiple algorithm source separation using the blind *SEP_MCLP*, *SEP_TTSE*, and *SEP_ABYK* SEP plugins with and without MASS self-competition using the blind *SEE_MinXCorrSelect* SEE plugin over the DS_3x3_RT60-0.1s, DS_3x4_RT60-0.1s , and DS_4x4_RT60-0.1s datasets.

Config_MASS_Ex3a_3x3			
DATASET: DS_3x3_RT60-0.1s.mat			
PROCESSING PARAMETERS		***PLUGININST* FIELDS**	
semSampRate	16000	***SEP***	*SEP_MCLP*, 0
semSelfComp	0		*SEP_TTSE*, 0
semCDSFiltLen	-1/16		*SEP_ABYK*, 2
semCDSFiltOffset	*semCDSFiltLen/2*	***SNUM***	*SNUM_NumObs*, 0
semBlkLen	-5	***SEE***	*SEE_MinXCorrSelect*, 10
semBlkStep	-5	***PAS***	*PAS_CSEPrevXcorr*, 1
		SEA	*SEA_SIR*, 10

EXTENSIONS			
Config_MASS_Ex3a_3x3_BSC	***semSelfComp***	1	
Config_MASS_Ex3b_3x4	***semSelfComp***	0	**DATASET:** DS_3x4_RT60-0.1s.mat
Config_MASS_Ex3b_3x4_BSC	***semSelfComp***	1	**DATASET:** DS_3x4_RT60-0.1s.mat
Config_MASS_Ex3c_4x4	***semSelfComp***	0	**DATASET:** DS_4x4_RT60-0.1s.mat
Config_MASS_Ex3c_4x4_BSC	***semSelfComp***	1	**DATASET:** DS_4x4_RT60-0.1s.mat

Configurations in this section use the same processing parameters, the same *PAS_CSEPrevXcorr* blind PAS plugin, and the same batch mode *SEA_SIR*, and the Configurations for the three datasets are given in Table 27. Thus, the results in this section represent a fully blind MASS framework application.

For each of the datasets used here, we will compare the blind MASS results, the blind MASS using self-competition results, and the semi-blind SEE MASS results from the previous section. For example, Figure 29 shows the average SIR for each MASS CSE and compares the performances of the blind SEE plugin, *SEE_MinXCorrSelect*, using and not using MASS self-competition with the supervised SEE Plugin, *SEE_SupSIRSelect*, not using MASS self- competition given in Figure 24 a) and re-displayed in Figure 29. Figures 28 and 29 display the results corresponding to the 3x4 and 4x4 cases, respectively, where the semi-blind SEE MASS results are drawn from Figures 23 a) and 24 a), respectively.

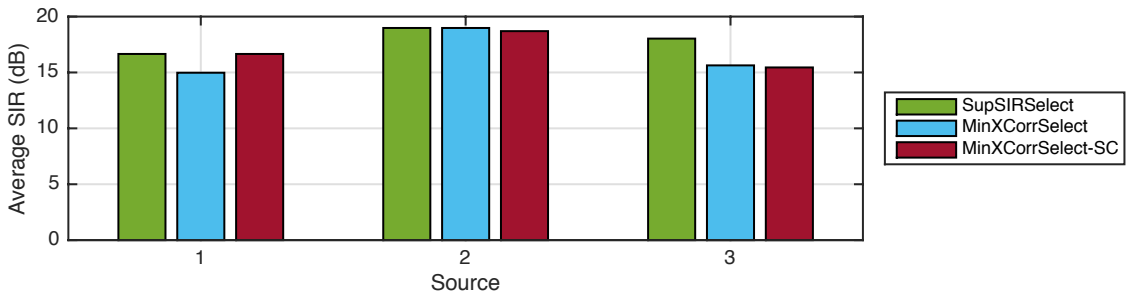

Figure 29. Blind Multiple Algorithm Source Separation via the MASS Framework, Example 3a: three source, three mixture set. Three blind SEP plugins, SEP_MCLP, SEP_TTSE, and SEP_ABYK, compete under the blind SEE_MinXCorrSelect evaluation plugin when the MASS system uses self-competition (red) and when the algorithms just compete (blue). The MASS results from the supervised evaluation experiment in Example 2a) are presented here (green) for comparison.

If we take in the comparisons provided by Figures 27-29 as a whole, the general conclusion that we draw is that semi-blind MASS performs marginally better than blind MASS, whether blind MASS uses self-competition or not. The maximum SIR advantage that semi-blind MASS holds over the blind MASS cases is given as a 4dB SIR gain for Source 3 in the 3x4 case shown in Figure 30, and otherwise blind MASS with and without self-competition is either comparable or exceeds the SIR of the semi-blind MASS case. From the previous section, we saw that the semi-blind MASS cases outperformed each of the respective individual blind SASS cases, and here, we see that the blind MASS cases perform comparably to the semi-blind MASS cases. We recognize that the limited datasets presented here allow for a limited probing of MASS potential, but the fact that blind MASS Configurations using diverse BSS and BSE methods can produce comparable results to semi-blind MASS Configurations using the same SEP diversity is very encouraging.

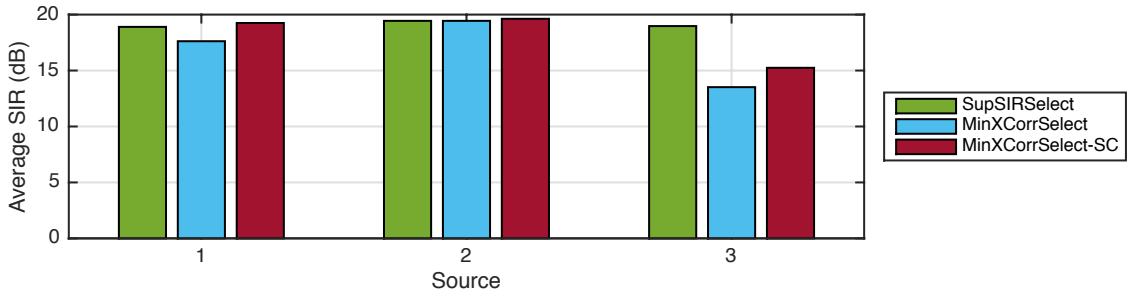

Figure 30. Blind Multiple Algorithm Source Separation via the MASS Framework, Example 3b: three source, four mixture set. Three blind SEP plugins, SEP_MCLP, SEP_TTSE, and SEP_ABYK, compete under the blind SEE_MinXCorrSelect evaluation plugin when the MASS system uses self-competition (red) and when the algorithms just compete (blue). The MASS results from the supervised evaluation experiment in Example 2b) are presented here (green) for comparison.

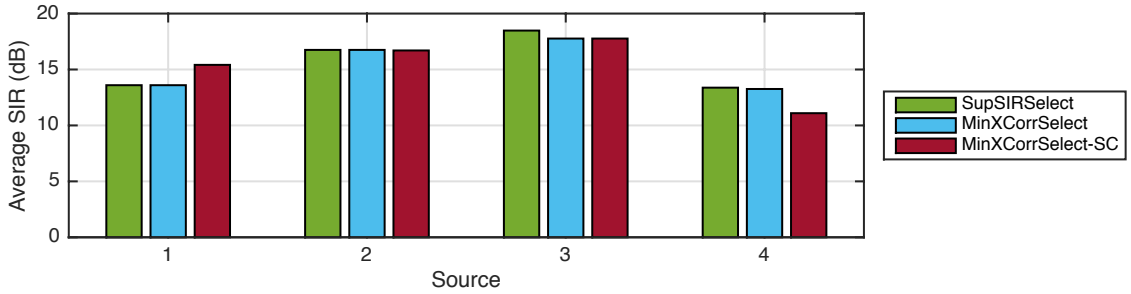

Figure 31. Blind Multiple Algorithm Source Separation via the MASS Framework, Example 3c: four source, four mixture set. Three blind SEP plugins, SEP_MCLP, SEP_TTSE, and SEP_ABYK, compete under the blind SEE_MinXCorrSelect evaluation plugin when the MASS system uses self-competition (red) and when the algorithms just compete (blue). The MASS results from the supervised evaluation experiment in Example 2c) are presented here (green) for comparison.

CHAPTER 6: CONCLUSION

In this work, we have discussed the inherent problems in the general source separation (SS) problem, introduced the subject of multiple algorithm source separation (MASS) as a decomposition of the general SS problem, defined a MATLAB-based software framework to perform MASS, given examples of how to use the MASS framework, and provided results that display the usefulness of MASS. In this chapter, we will briefly discuss future MASS research opportunities.

6.1 Future MASS Domain Research

Although the main research aim of this work is to simply consider the problem of multiple, disparate, convolutive source separation and/or extraction methods running in parallel to produce a viable set of source estimates, a loftier goal of *universal* source separation is now driving this work. Although we have presented evidence that MASS can produce source estimates that, as a set, perform as well or better than any source estimate set produced by a constituent SS method, we make no guarantees here that this result will hold for all cases, i.e. over all combinations of SS methods, over all mixing conditions, over all source signals, etc. Furthermore, given this source estimate set mentality, we certainly cannot guarantee each individual source estimate from MASS is as good as or better than the best estimate of that source over all the SS methods employed. This result is due to the fact that we have only used *joint* evaluation methods in the SEEs presented here. If we wish to ensure that all source estimates produced by MASS are as good as the best estimates from the SS methods, we need to develop *individual* SEE methods, whether they be selected or blended.

Although the universal convolutive source separation problem has multiple dimensions to consider, the definition of a source almost entirely defines a universe and drives individual SEE methods within that universe. Sources can be defined as being mutually, statistically independent or uncorrelated as in ICA, but they can also be defined as being spatially distinct, as in beamforming. For example, in a room containing a talker and an air conditioning unit, the sources are both statistically uncorrelated and spatially distinct, but two loudspeakers driven by the same signal in a room are spatially distinct

but highly correlated. Thus, SEE development will necessarily be geared towards specific universes.

Furthermore, universal methods typically attain the performance of some functional that is best suited to the particular observations. In some sense, we have begun looking at this issue by employing disparate SEP methods, e.g. ABYK, TTSE, and MCLP, however we note that the inclusion of this widely varying set of SEPs defines a very large universe. Indeed, future work should narrow the universe to problems where an individual method provides the best parameterized solutions to all problems in the universe. For example, we could study scenes in which ABYK provides valid solutions that are dependent upon the filter length employed and develop a universal method with ABYK as the underlying method. This narrowed focus could provide insight into issues that will be encountered in a larger universe, where we can combine the smaller universal solutions in MASS.

With this in mind, another area of research moving forward is *blended* SEE methods, both individual and joint. Both joint and individual blending are complex topics, and once again are highly dependent upon the definition of a source. These topics are complicated in the convolutive BSS problem by the fact that an infinite number of valid solutions exist due to the filtering ambiguity. However, blended solutions provide the potential of improving source estimates, and will be a focus of future research.

6.2 Future MASS Framework Development

The work presented here defines an elementary MASS framework, and although the current framework is usable, we have identified several development items to enhance the framework. These items range from making the framework easier to use, to making the framework more robust, and to making the framework accessible and promoting its use.

In Chapter 5 we saw how to use the MASS framework by manipulating Configurations. Although Configurations fully define how the MASS framework is used and are a necessary part of using the framework, the Configurations themselves can be quite cumbersome. To make the framework easier to use, we will create a *ConfigPlugin* component that will guide the user via a graphical user interface (GUI) to create Configurations. Although the output of a *ConfigPlugin* is a valid Configuration, we will

unburden the user from needing to manually generate a Configuration. In fact, we envision two different GUI-based *ConfigPlugins*: a Q&A style sequence of displays that queries the user for their specific needs and an "advanced" display that simply allows the user to fill in or select the various components and parameters they wish to use. That said, users will still be able to create Configurations manually, if they choose, and use the framework as described in this work.

Once we have an easy method for creating Configurations, we will need to develop a database so that Configurations can be saved for later use. For instance, we created 50 Configurations to produce the results presented in Chapter 5 and although the examples give a comprehensive look at the MASS framework's fundamental capabilities, the cases studied in Chapter 5 are not exhaustive. Thus, a database is a compact method for storing and retrieving configuration data for a large number of Configurations. Coupling the database and the GUI-based *ConfigPlugin* will allow the end-user to efficiently create and retrieve Configurations.

At present, the MASS framework only allows one SEA component to be active in a Configuration, and we will extend the SEA functionality to allow multiple SEA components to run per Configuration. In this work, we gave only one example of a plugin component, *SEA_SIR*, which measures the SIR of each source in each CSE. That said, we can also create a SEA component that measures the mutual information (MI) of the CSE set, as well as various other source separation measures. However, we may want to know *both* the SIR and the MI of the CSE set, and this SEA expansion would allow us to write one component for SIR and one component for MI, and run them both in one Configuration, just like we can run multiple SEP components in one Configuration.

In general, we would like to expand the plugin component library for every class of plugin, but we note a special expansion related to the SEA plugin. If we note that the SEA plugin is named the "source estimate analysis" plugin and not the "source separation analysis" plugin, we see that we can implement various measures that have nothing to do with source separation. For instance, if we are performing source separation on a mixture of speech signals, we might want to do speech-related analyses, and we can implement those analyses in an SEA component. Thus, future versions of the SEA component library will include various components unrelated to source separation.

At this point we should note that the definition of the SEA plugin allows the SEA plugin access to the observations and the individual SEP plugins' source estimates, and future versions of MASS will include examples that analyze these signals. Using multiple SEA plugins (the SEA expansion above), the analysis of the observations in conjunction with the source separation task and its relevant analyses, removes the need for running MASS in a separate *signal analysis* Configuration to analyze the observations. Similarly, when MASS is configured for all SEP plugins to compete, the analysis of the SEPs' source estimates during the MASS task, removes the need to run a separate signal analysis Configuration for each SEP method. When MASS is configured with cooperating SEPs, the SEPs' source estimate analysis during the MASS Configuration will generally differ from the individual SEP signal analysis Configuration for each of the cooperating SEPs.

A more robust MASS framework will include at least four new plugins: the filtering ambiguity solution plugin, a general pre-processing plugin, a general post-processing plugin, and a data visualization management plugin. As we discussed in Chapter 2, the filtering ambiguity solution (FAS) plugin is responsible for ensuring that individual source estimates are consistent from block to block. The FAS plugin is necessary when processing audio data, so that the CSE audio signals do not contain discontinuities at data block edges or have different overall spectral shaping from block to block.

The pre- and post-processing plugins are provided as a convenience, so that end-users can decouple any general processing tasks from the rest of the system. That said, we envision the output of the pre-processor to feed into the MASS framework for use with the various other plugins, while the output of the post-processor can either become the CSE set (along with the associated CDS set), or the output can become a separate output of MASS. For instance, the post-processor might be a dereverberation method for which the output coupled with the source separation of MASS becomes a deconvolution estimate, and this may or may not be appropriate for informing some SEP methods. Thus, this dereverberation output can simply be a secondary output of MASS, or it can become the CSE and CDS that is used for cooperating SEP plugins where appropriate

MASS provides many signals to analyze and multiple SEA components can provide multiple figures per SEA component to provide visualization of the component's

results. Thus, we can conceive of a data visualization management plugin to help coordinate the various graphics that can be generated in the framework. That is, the data visualization management plugin ensures that the results of one SEA component plotted in a particular set of figures are not overwritten by another SEA component. Although this may seem like a minor task, the data visualization management's role becomes more apparent when the number of active SEA components increases.

The last item on our list of MASS framework development tasks is not a software development task, but a community development task. Just as the signal processing power of the MASS framework comes from a multitude of diverse source separation methods, another power of the MASS framework comes from a community of diverse developers and researchers using and developing the framework. We see the MASS framework as a potentially indispensable tool in a research community aimed towards advancing the source separation topic. In Chapter 5, we saw that the MASS framework provides three general capabilities, i.e. signal analysis, SASS with performance analysis, and MASS. Thus, a researcher who is unconcerned with the MASS topic can, reproducibly, examine their underlying signals and perform standard SASS. Although this may seem trivial, we argue that the reproduction of results is at the heart of scientific research, and when the MASS framework is accessible to *all* researchers, the MASS framework provides both reproducibility of results and a common basis for which all results can be examined. That said, the MASS framework also provides the capability of performing multiple algorithm source separation, for anyone who is interested.

Furthermore, when the MASS framework becomes a shared development process, every developer and researcher benefits from every developer's and researcher's contributions. Although we have expended much time and energy in developing the MASS framework, the author is only one person with one imagination, thus the author clearly concedes that he has not thought of every possibility for a set of signals, much less approaches to source separation, and even less, approaches to MASS. Thus, a diverse community involved with the MASS framework can only provide greater insight to all of the various topics involved with SASS and MASS. This work, along with the MATLAB-based framework code and data, will be publicly accessible, and the author plans to make a concerted effort to make researchers aware of the tool presented here via various methods including, but not limited to, submissions to conferences and journals,

contacting the chairs of source separation competitions, and informing relevant corporations interested in public advancement of source separation.

6.3 A Multi-Algorithm Signal Processing Framework

Even though the focus of this work has been on source separation using multiple algorithms, we can see that the current software framework for MASS could easily be extended to study multiple algorithm signal processing (MASP) techniques, generally. In fact, we can easily implement various multiple algorithm methods from the literature, e.g. [7], [11], [20], in the current MASS framework, simply by creating the appropriate SEP and SEE plugin components. For instance, if we create N SEP plugin components, where the n^{th} plugin is a n^{th}-order linear predictor for $n = 1, \ldots, N$, and we also create a SEE plugin that implements the blending method in [7], then we can recreate the universal linear prediction algorithm given in [7]. In fact, with no modification to the MASS framework, we can implement all of the competitive adaptive filtering methods given in [14].

That said, to create a general and robust MASP framework, we will need to make some modifications to the MASS framework. At the simplest level, we need to change the nomenclature that is specific to source separation. For instance, we will remove the word "source" or "demixing" from all plugin names and functions, thus the SEP plugin becomes the EP plugin, the SEE plugin becomes the EE plugin, the CSE becomes the CE, and the CDS estimator becomes the CS estimator, etc.

Next, we need to generalize the common solution (CS) estimator, i.e. the CDS in the MASS framework., and we start by exposing the CS as a plugin. Exposing the CDS in the MASS framework will allow end-users to define the common demixing operation (CDO), thus CS exposure in the MASP framework will allow end-users to define the common estimation operation (CEO). That said, not all multiple algorithm methods will employ a CEO, since the very notion of a CEO runs counter some multiple algorithm methods. For example consider two least mean square (LMS) filters running in two EP plugins. If both the LMS filters are of the same length, L, then a natural CS format is an L-length FIR filter. If the LMS filters differ in filter length, then it is up to the framework user to determine whether an N-length FIR CS format is appropriate.

The next task in creating a MASP framework involves determining what to do with application specific plugins. For instance, the PAS and FAS plugins were created as a decomposition of the BSS problem, and they will have little or no use in an adaptive filtering application. That said, the PAS and FAS could be used in a general, blind, multi-channel signal processing application. One approach to the problem is to have a base, or parent, MASP framework that contains all functionality that is common to every multiple signal processing task imaginable. From there, class-specific child frameworks can be created by adding plugins that are specific to a particular class of multiple algorithm methods. Alternatively, the MASP framework can simply contain all plugins necessary for all classes of multiple algorithm methods, and a user simply utilizes the plugins necessary for her class-specific application. We will leave this development choice for future work.

Another foreseeable task in developing a MASP framework involves creating *spawn* and *prune* functionality for EP plugins. The spawn functionality would allow the MASP framework to add one or many new EP components at any data-block, and this functionality is exemplified in the work by Moon and Weissman, e.g [18], where a new EP component would be added at every data-block step. With the introduction of a spawn method, we need to introduce a prune method for EP components so that we can remove EP components as they become insignificant contributors, or if the array of EP components becomes too large, as is the case in most tree-based methods, e.g. [99]–[101].

This generalization of the MASS framework to a MASP framework points to a further generalized signal processing (GSP) framework, given that the MASS framework provides capabilities beyond performing multiple algorithm source separation. As we displayed in Chapter 5, the MASS framework provides three general capabilities: signal analysis, SASS, and MASS. The MASP framework would generalize and include MASS framework functionality, thus the SASS functionality would translate to a SASP functionality, which is just a general signal processing capability. Thus, a MASP framework would simply be a child of a larger parent GSP framework, all of which is built from the simple idea of having multiple source separation algorithms running in parallel toward a common solution. The GSP framework will be immensely useful, but immensely difficult to develop, and for now, we will simply enjoy the MASS framework.

APPENDIX A: THE TURN-TAKING SOURCE EXTRACTION METHOD

The turn-taking source extraction (TTSE) method is a convolutive blind source extraction (BSE) method for sources that intermittently become silent. The method is dubbed the turn-taking source extraction method for the way some intermittent sources tend to take turns, e.g. conversational speech. The method makes many assumptions, but crucially, when the mixing system is critically- or over-determined with an intermittent source, a source extraction solution for the intermittent source is available via a system identification problem. Before we detail the method, we now give the multiple assumptions that must hold for this method to extract an intermittent source.

Assumption 1 (A1): Any source of interest (SOI) that is to be extracted is intermittent, so that in some (frequent) time blocks, the SOI falls silent and adds no appreciable power to the observed mixtures.

Assumption 2 (A2): The mixing system is either critically- or over-determined when the SOI is active, and that the mixing system is over-determined when the SOI is inactive.

Assumption 3 (A3): No new sources become active *while* and *after* the SOI is inactive.

Assumption 4 (A4): All of the sources complying with *A1-A3* do not move *while* the SOI is silent.

Assumption 5 (A5): All of the sources complying with *A1-A4*, except the SOI, do not move *after* the SOI is silent.

Assumption 6 (A6): *After* the SOI is silent, the SOI does not move into a position near any of the other sources.

Although these assumptions may seem prohibitive, many real-world situations fall under exactly these criteria, e.g. conversational speech, a telephone ringing in an office setting, etc.

A.1 Notation

We now give some general notation used in our discussion of the TTSE method. We assume an over- or critically-determined, static scenario, such that Q sources impinge upon a set of P sensors where $Q \leq P$, and all sources and sensors are spatially stationary.

We denote $s_q(n)$ as the q^{th} source signal, $x_p(n)$ as the p^{th} observation signal, and $a_{qp}(\kappa)$ as the impulse response of the channel between the q^{th} source and the p^{th} observation, for $q = 1, \dots, Q$, and $p = 1, \dots, P$, where n is a time-index, and κ is a lag-index indicating that the channel filter is time-invariant. We model the p^{th} observation as the sum of filtered sources in the frequency domain as,

$$X_p(f) = \sum_{q=1}^{Q} A_{qp}(f) S_q(f), \qquad p = 1, \dots, P, \tag{102}$$

where $X_p(f)$, $S_q(f)$, and $A_{qp}(f)$ are the Fourier transforms of $x_p(n)$, $s_q(n)$, and $a_{qp}(\kappa)$, respectively. We define a frequency-domain vector of mixtures as,

$$\boldsymbol{x}(f) = \boldsymbol{A}^T(f)\boldsymbol{s}(f) = [X_1(f), \dots, X_P(f)]^T, \tag{103}$$

where

$$\boldsymbol{s}(f) = \left[S_1(f), \dots, S_Q(f)\right]^T, \tag{104}$$

is a frequency-domain source signal vector and

$$\boldsymbol{A}(f) = [\boldsymbol{a}_1(f), \dots, \boldsymbol{a}_P(f)] \tag{105}$$

is a matrix of channel frequency responses with

$$\boldsymbol{a}_p(f) = \left[A_{1p}(f), \dots, A_{Qp}(f)\right]^T, \qquad p = 1, \dots, P. \tag{106}$$

Letting $\mathcal{Q} = \{1, \dots, Q\}$ be the set of source indexes, we will assume that all sources are zero-mean and mutually uncorrelated, i.e.,

$$\mathcal{E}\{S_i(f)S_j^*(f)\} = 0, \quad i \neq j, i \in \mathcal{Q}, j \in \mathcal{Q}, \tag{107}$$

where * denotes complex conjugate.

In this work, we will need to partition various signal sets. To this end, we will use the identity matrix decomposition given by,

$$\boldsymbol{I}_{|\mathcal{K}'|} = \boldsymbol{H}_{\mathcal{K}} + \bar{\boldsymbol{H}}_{\mathcal{K}}, \tag{108}$$

where $\mathcal{K}' = \{1, \dots, K\}$ is an ordered set of indices with $K = |\mathcal{K}'|$ elements, $\mathcal{K} \subseteq \mathcal{K}'$, $\boldsymbol{I}_{|\mathcal{K}'|}$ is a $[K\text{x}K]$ identity matrix, the $(i,j)^{th}$ element of $\bar{\boldsymbol{H}}_{\mathcal{K}}$ is

$$[\bar{\boldsymbol{H}}_{\mathcal{K}}]_{\boldsymbol{ij}} = \begin{cases} 1, & i = j \notin \mathcal{K} \\ 0, & \text{otherwise} \end{cases}, \tag{109}$$

and $\boldsymbol{H}_{\mathcal{K}} = \boldsymbol{I}_{|\mathcal{K}'|} - \bar{\boldsymbol{H}}_{\mathcal{K}}$. For example, if we let $\mathcal{Q} = \{1, \dots, Q\}$ be the set of source indices, then we can decompose the source signal vector as $\boldsymbol{s}(f) = \boldsymbol{H}_q\boldsymbol{s}(f) + \bar{\boldsymbol{H}}_q\boldsymbol{s}(f)$,

where $\{q\} \subset \mathcal{Q}$ is a one element subset of $\mathcal{Q}$, so that $\boldsymbol{H}_q$ and $\bar{\boldsymbol{H}}_q$ are $[Q\text{x}Q]$ matrices, $\boldsymbol{H}_q\boldsymbol{s}(f)$ is a $[Q\text{x}1]$ vector of all zeros except for the q^{th} element, $s_q(f)$, and $\bar{\boldsymbol{H}}_q\boldsymbol{s}(f)$ is a $[Q\text{x}1]$ vector containing all the source signals except for a zero in place of $s_q(f)$ at the q^{th} element. We note that we are using the shorthand notation $\boldsymbol{H}_q = \boldsymbol{H}_{\{q\}}$ and $\bar{\boldsymbol{H}}_q = \bar{\boldsymbol{H}}_{\{q\}}$ when $\{q\} \subset \mathcal{Q}$ is a single element subset.

A.2 The TTSE Principle

The fundamental idea behind TTSE assumes that a source falls silent intermittently, and we can derive an extraction solution for that source during its silence. Taking the intermittent silence assumption to an extreme for now, let $s_q(n) = 0, \forall n$, such that,

$$\bar{\boldsymbol{s}}_q(f) = \bar{\boldsymbol{H}}_q\boldsymbol{s}(f), \tag{110}$$

and

$$\bar{\boldsymbol{x}}_q(f) = \boldsymbol{A}^T(f)\bar{\boldsymbol{s}}_q(f) = \left[\bar{X}_{q1}(f), \dots, \bar{X}_{qP}(f)\right]^T \tag{111}$$

where $\bar{X}_{qp}(f) = \boldsymbol{a}_{qp}^T(f)\bar{\boldsymbol{s}}_q(f)$. Noting that the mixtures are over-determined in this case, consider a MISO system when $\bar{X}_{qp}(f)$ is the observed output and $\bar{X}_{qk}(f)$ are the known inputs where $\{k: k \in \mathcal{P}, k \neq p\}$ and $\mathcal{P} = \{1, \dots, P\}$. To solve for the system's frequency responses, we construct the error spectrum as,

$$E_{pq}(f) = \boldsymbol{d}_{pq}^T(f)\bar{\boldsymbol{x}}_q(f) \tag{112}$$

where

$$\left[\boldsymbol{d}_{pq}(f)\right]_i = \begin{Bmatrix} 1, & i = p \\ -d_{pi}(f), & i \neq p \end{Bmatrix} \tag{113}$$

and $d_{pi}(f)$ is an estimate of the system frequency response applied to the i^{th} input. The optimal solution for minimizing the error power spectrum, i.e., $\boldsymbol{d}_{pq}(f) = \arg\min_{\boldsymbol{d}} \mathcal{E}\left\{\left|E_{pq}(f)\right|^2\right\}$, is given as,

$$\boldsymbol{d}_{pq}(f) = \boldsymbol{1}_P^T\boldsymbol{H}_p + \bar{\boldsymbol{R}}_{pq}^{+}(f)\bar{\boldsymbol{r}}_{pq}(f) \tag{114}$$

where $\boldsymbol{1}_P$ is a $[P\text{x}1]$ vector of ones, $^{+}$ denotes pseudoinverse, $\bar{\boldsymbol{R}}_{pq} = \bar{\boldsymbol{H}}_p\mathcal{E}\{\bar{\boldsymbol{x}}_q(f)\bar{\boldsymbol{x}}_q^H(f)\}\bar{\boldsymbol{H}}_p$, and $\bar{\boldsymbol{r}}_{pq}(f) = \bar{\boldsymbol{H}}_p\mathcal{E}\{\bar{\boldsymbol{x}}_q(f)X_{qp}^*(f)\}$, and ^{H} denotes complex conjugate transpose.

To see how $\boldsymbol{d}_{pq}(f)$ is a source extraction solution for $s_q(n)$, we now apply $\boldsymbol{d}_{pq}(f)$ to the mixtures $\boldsymbol{x}(f)$, i.e. when $s_q(n)$ is present, as,

$$\begin{aligned}
\boldsymbol{d}_{pq}^T(f)\,\boldsymbol{x}(f) &= \boldsymbol{d}_{pq}^T(f)\boldsymbol{A}^T(f)\boldsymbol{s}(f) \\
&= \boldsymbol{d}_{pq}^T(f)\boldsymbol{A}^T(f)\left[\boldsymbol{H}_q\boldsymbol{s}(f) + \bar{\boldsymbol{H}}_q\boldsymbol{s}(f)\right] \\
&= \left[\boldsymbol{d}_{pq}^T(f)\boldsymbol{A}^T(f)\boldsymbol{H}_q\boldsymbol{s}(f)\right] + \left[\boldsymbol{d}_{pq}^T(f)\boldsymbol{A}^T(f)\bar{\boldsymbol{s}}(f)\right] \\
&= \left[A_{qp}(f)S_q(f) - \sum_{i=1,i\neq p}^{P} d_{pi}(f)A_{qi}(f)S_q(f)\right] + \left[\boldsymbol{d}_{pq}^T(f)\bar{\boldsymbol{x}}(f)\right] \\
&= S_q(f)\left[A_{qp}(f) - \sum_{i=1,i\neq p}^{P} d_{pi}(f)A_{qi}(f)\right] + E_{pq}(f) \\
&= S_q(f)G_{pq}(f) + E_{pq}(f)
\end{aligned}$$

We can see that the demixing solution derived when a source is absent can be applied to the observations when the source is present to isolate the source, up to a filtering, plus an error term.

If we note that the error term only involves the other, interfering sources, the result above allows us to express the maximum achievable signal-to-interference (SIR) ratio of this method for a particular scenario, in the time-domain, as,

$$SIR_{pq} = \frac{\mathcal{E}\left\{\left(s_q(n) * g_{pq}(n)\right)^2\right\}}{\mathcal{E}\left\{\left(e_{pq}(n)\right)^2\right\}} \tag{115}$$

where $g_{pq}(n)$ and $e_{pq}(n)$ are the inverse Fourier transforms of $G_{pq}(f)$ and $E_{pq}(f)$, respectively.

A.3 A Practical TTSE Method

Although the above example gives the foundation for TTSE, this example is ideal in many respects. We know when the *single* intermittent source "turns off" and "turns on". The power spectra needed for the extraction solution are invariant with respect to time. None of the interfering sources move during the extraction solution. These topics define our assumptions on the problem, i.e. *A1-A6*, and to begin covering these topics and our assumptions, we now turn our attention to the practical matters of implementing a TTSE method.

Let $s_q(n)$ be an intermittent source, such that for a region of some number of contiguous samples, $\mathcal{R}_q$, the signal falls silent, i.e. $s_q\left(n \in \mathcal{R}_q\right) = 0$. Our description of TTSE thus far assumes that we estimate the extraction solution for $s_q(n)$ during $\mathcal{R}_q$, which further requires that we estimate $\mathcal{R}_q$. Determining this region in a blind fashion within the multi-source convolutive environment is a daunting, but not insurmountable, task. Here, however, we take a different approach that is simple if not brutish, and as we shall see, is augmented by the MASS framework.

Working in a block-processing context, so that we only receive a contiguous N-sample region of the observation data, e.g. $\boldsymbol{x}(n \in \mathcal{R}) = [x_1(n \in \mathcal{R}), \dots, x_P(n \in \mathcal{R})]^T$, we assume that any silent region of $s_q(n)$ is smaller than the block of data, i.e. $|\mathcal{R}| = N > L = \left|\mathcal{R}_q\right|$. With $\mathcal{R}_q$ unknown, we will step through the observation data and use-sub-blocks of data to calculate $\boldsymbol{d}(f)$ at each step. Let $\mathcal{L} = \{1,2, \dots, L\}$ be an ordered set of time-indices and let $\mathcal{R}_m$ be a contiguous region of L sample time-indices indices at the m^{th} sub-block, i.e. $\mathcal{R}_m = mK + \mathcal{L}$, where K is a sub-block shift in samples. Noting that our extraction solution is performed relative to a particular observation, we now define a set of finite-length, windowed observation data as,

$$\left[\boldsymbol{x}_p(n,m)\right]_i = \begin{cases} x_p(n) & i = p, n \in \mathcal{R}_m \\ 0 & n \notin \mathcal{R}_m, n \in \mathcal{R}' \end{cases}, \; i = 1, \dots, P\,. \tag{116}$$

For each block of data, and for each mixture we will estimate an extraction solution of the form,

$$\left[\boldsymbol{d}_p(f,m)\right]_i = \begin{Bmatrix} e^{-j2\pi f\kappa}, & i = p \\ -d_{pi}(f,m) & i \neq p \end{Bmatrix}, \; i = 1, \dots, P, \; p = 1, \dots, P, \; m = 1, \dots, M \tag{117}$$

where κ is a lag to ensure a causal solution, $d_{pi}(f,m)$ is an estimate of the system frequency response applied to the i^{th} input when $x_p(n \in \mathcal{R}_m)$ is the observed output, and M is the number of sub-blocks. Given the blind and convolutive nature of the problem, we lag the p^{th} observation to ensure that all of the interfering sources are "seen" in the other mixtures before p^{th} mixture, so that the estimation of $\boldsymbol{d}(f)$ can properly remove them. Without the lag, we run the risk of estimating prediction coefficients for the active sources. The solution aims to minimize the error power spectrum at a sub-block, where the error spectrum is given by,

$$E_p(f,m) = \boldsymbol{d}_p^T(f,m)\boldsymbol{x}_p(f,m), \tag{118}$$

where $\boldsymbol{x}_p(f,m)$ is the Fourier transform of $\boldsymbol{x}_p(n,m)$.

Assuming that the sources are spatially stationary during the block $\mathcal{R}$, we apply each of the sub-block extraction solutions to the entire set of data and evaluate the power of the outputs. That is, for each sub-block and each observation, we produce candidate outputs as,

$$Y_p(f,m) = \boldsymbol{d}_p^T(f,m)\boldsymbol{x}(f), \qquad p = 1,\dots,P,\ m = 1,\dots,M, \tag{119}$$

where $\boldsymbol{x}(f)$ is the Fourier transform of $\boldsymbol{x}(n)$. Given the assumption that the observations are critically-determined and further assuming that the largest power outputs produce isolated sources, we choose the P signals from $Y_p(f,m)$ that produce the largest power as the outputs of the TTSE method. Although this selection process is naïve, when the TTSE method is used in a MASS setting, the source estimate evaluation (SEE) system determines whether or not the individual signals produced by the TTSE method are good estimates of an individual source.

BIBLIOGRAPHY

[1] K. Matsuoka and S. Nakashima, "Minimal distortion principle for blind source separation," presented at the Int. Conf. on Independent Component Analysis and Blind Signal Separation, Rome, Italy, 2001, pp. 722–727.

[2] E. Vincent, R. Gribonval, and M. Plumbley, "Oracle estimators for the benchmarking of source separation algorithms," *Signal Process.*, vol. 87, no. 8, pp. 1933–1950, 2007.

[3] N. Merhav and M. Feder, "Universal prediction," *IEEE Trans. Inf. Theory*, vol. 44, pp. 2124–2147, 1998.

[4] V. Vovk, "Aggregating strategies," in *Proc. Third Annual Workshop Computational Learning Theory*, San Mateo, CA, 1990, pp. 371–386.

[5] N. Littlestone and M. K. Warmuth, "The weighted majority algorithm," *Inf. Comput.*, vol. 108, pp. 212–261, 1994.

[6] A. P. Dawid, "Prequential Data Analysis," *Inst. Math. Stat. Lect. Notes - Monogr. Ser.*, vol. 17, pp. 113–126, 1992.

[7] A. C. Singer and M. Feder, "Universal linear prediction by model order weighting," *IEEE Trans. Signal Process.*, vol. 47, pp. 2685–2699, 1999.

[8] S. S. Kozat and A. C. Singer, "Universal switching linear least squares prediction," *IEEE Trans. Signal Process.*, vol. 56, no. 1, pp. 189–204, 2008.

[9] J. Rissanen, "Universal coding, information, prediction, and estimation," *IEEE Trans. Inf. Theory*, vol. IT-30, pp. 629–636, 1984.

[10] S. S. Kozat and A. C. Singer, "Multi-stage adaptive signal processing algorithms," in *Proc. 2000 IEEE Sensor Array and Multichannel Signal Process. Workshop*, Cambridge, MA, 2000, pp. 380–384.

[11] S. S. Kozat, A. T. Erdogan, A. C. Singer, and A. H. Sayed, "Steady-state MSE performance analysis of mixture approaches to adaptive filtering," *IEEE Trans. Signal Process.*, vol. 58, pp. 4050–4063, 2010.

[12] A. T. Erdogan, S. S. Kozat, and A. C. Singer, "Comparison of convex combination and affine combination of adaptive filters," in *Proc. 2009 Int. Conf. on Acoust., Speech and Signal Process.*, Taipei, Taiwan, 2009, pp. 3089–3092.

[13] S. S. Kozat, A. T. Erdogan, S. A.C, and A. H. Sayed, "Transient analysis of adaptive affine combinations," *IEEE Trans. Signal Process.*, vol. 59, pp. 6227–6232, 2011.

[14] A. C. Singer and S. S. Kozat, “A competitive algorithm approach to adaptive filtering,” in *7th Int. Symp. on Wireless Comm. Sys.*, York, United Kingdom, 2010, pp. 350–354.

[15] R. Candido, M. T. M. Silva, and V. H. Nascimento, “Transient and steady-state analysis of the affine combination of two adaptive filters,” *IEEE Trans. Signal Process.*, vol. 58, pp. 4064–4078, 2010.

[16] R. Candido, M. T. M. Silva, and V. H. Nascimento, “Affine combinations of adaptive filters,” in *Proc. 2008 42nd Asilomar Conf. on Signals, Sys. and Computers*, Pacific Grove, CA, 2008, pp. 236–240.

[17] M. T. M. Silva and V. H. Nascimento, “Improving the tracking capability of adaptive filters via convex combination,” *IEEE Trans. Signal Process.*, vol. 56, pp. 3137–3149, 2008.

[18] T. Moon and T. Weissman, “Universal FIR MMSE filtering,” *IEEE Trans. Signal Process.*, vol. 57, no. 3, pp. 1068–1083, 2009.

[19] T. Moon, “Universal switching FIR filtering,” *IEEE Trans. Signal Process.*, vol. 60, no. 3, pp. 1460–1464, 2012.

[20] J. R. Buck and A. C. Singer, “A performance-weighted blended dominant mode rejection beamformer,” in *Proc. 10th IEEE Sensor Array and Multichannel Signal Process. Workshop*, Sheffield, United Kingdom, 2018, pp. 124–128.

[21] M. A. Dmour and M. Davies, “A new framework for underdetermined speech extraction using mixture of beamformers,” *IEEE Trans. Audio Speech Lang. Process.*, vol. 19, pp. 445–457, 2011.

[22] T. Melia and S. Rickard, “A general framework for extending classic array processing techniques to the underdetermined blind source separation problem,” in *Proc. 9th Int. Symp. on Signal Process. and Its Appl.*, Sharjah, United Arab Emirates, 2007, pp. 1–4.

[23] H. Robbins, “Asymptotically subminimax solutions of compound statistical decision problems,” in *Proc. Second Berkeley Symp. on Math. Stat. and Probability*, Berkeley, CA, 1951, pp. 131–149.

[24] J. van Ryzin, “The sequential compound decision problem with m x n finite loss matrix,” *Ann. Math. Stat.*, vol. 37, pp. 954–975, 1966.

[25] E. Samuel, “Sequential compound rules for the finite decision problem,” *J. R. Stat. Soc. Ser. B Methodol.*, vol. 28, pp. 63–72, 1966.

[26] E. Samuel, “The compound statistical decision problem,” *Sankhyā Indian J. Stat. Ser. A*, vol. 29, pp. 123–140, 1967.

[27] N. Merhav and M. Feder, “Universal schemes for sequential decision from individual data,” *IEEE Trans. Inf. Theory*, vol. 39, pp. 1280–1292, 1993.

[28] S. S. Kozat and A. C. Singer, “Universal randomized switching,” *IEEE Trans. Signal Process.*, vol. 58, pp. 1922–1927, 2010.

[29] D.-T. Pham and P. Garat, “Blind separation of mixture of independent sources through a quasi-maximum likelihood approach,” *IEEE Trans. Signal Process.*, vol. 45, no. 7, pp. 1712–1725, 1997.

[30] S. Cruces, L. Castedo, and A. Cichocki, “Asymptotically equivariant blind source separation using cumulants,” *Neurocomputing*, vol. 49, pp. 87–118, 2002.

[31] S. Amari, “Natural gradient works efficiently in learning,” *Neural Comput.*, vol. 10, no. 2, pp. 251–276, 1998.

[32] P. Comon, “Contrasts, independent component analysis, and blind deconvolution,” *Int. J. Adapt. Control Signal Process.*, vol. 18, no. 3, pp. 225–243, 2004.

[33] K. Gilbert and K. Payton, “Competitive algorithm blending for enhanced source separation,” in *Proc. 2014 48th Asilomar Conf. on Signals, Sys. and Computers*, Pacific Grove, CA, 2014, pp. 450–454.

[34] K. Gilbert, K. Payton, R. Goldhor, and J. MacAuslan, “Competitive algorithm blending for enhanced source separation of convolutive speech mixtures,” *J. Acoust. Soc. Am.*, vol. 136, no. 4, pp. 2265–2266, 2014.

[35] A. Papoulis and S. U. Pillai, *Probability, Random Variables, and Stochastic Processes*, 4th ed. New York, NY: McGraw-Hill, 2002.

[36] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. Hoboken, NJ: John Wiley & Sons, Inc., 2006.

[37] A. Hyvärinen, J. Karhunen, and E. Oja, *Independent Component Analysis*. New York, NY: John Wiley & Sons, Inc., 2001.

[38] A. Hyvärinen, “New approximations of differential entropy for independent component analysis and projection pursuit,” *Adv. Neural Inf. Process. Syst.*, vol. 10, pp. 273–279, 1998.

[39] S. Haykin, *Adaptive Filter Theory*, 4th ed. Upper Saddle River, NJ: Prentice Hall, Inc., 2002.

[40] A. H. Sayed, *Adaptive Filters*. Hoboken, NJ: John Wiley & Sons, Inc., 2008.

[41] D. Erdogmus and J. C. Principe, “An error-entropy minimization algorithm for supervised training of nonlinear adaptive systems,” *IEEE Trans. Signal Process.*, vol. 50, no. 7, pp. 1780–1786, 2002.

[42] D. Erdogmus and J. C. Principe, “Comparison of entropy and mean square error criteria in adaptive system training using higher order statistics,” in *Proc. Second*

Int. Workshop on Independent Component Analysis and Blind Signal Separation, Helsinki, Finland, 2000, pp. 75–80.

[43] B. Chen, J. Hu, H. Li, and Z. Sun, "Adaptive filtering under maximum mutual information criterion," *Neurocomputing*, vol. 71, pp. 3680–3684, 2008.

[44] B. Chen, Y. Zhu, J. Hu, and J. C. Príncipe, "Stochastic gradient identification of Wiener system with maximum mutual information criterion," *IET Signal Process*, vol. 5, no. 6, pp. 589–597, 2011.

[45] M. Delcroix, M. Miyoshi, and T. Hikichi, "Precise dereverberation using multichannel linear prediction," *IEEE Trans. Audio Speech Lang. Process.*, vol. 15, no. 2, pp. 430–440, 2007.

[46] M. Delcroix, T. Hikichi, and M. Miyoshi, "Dereverberation and denoising using multichannel linear prediction," *IEEE Trans. Audio Speech Lang. Process.*, vol. 15, no. 6, pp. 1791–1801, 2007.

[47] N. Wiener, *Extrapolation, Interpolation, and Smoothing of Stationary Time-Series with engineering applications*. Cambridge, MA: The MIT Press, 1949.

[48] Y. Bar-Ness, J. Carlin, and M. Steinberger, "Bootstrapping adaptive interference cancellers: some practical limitations," presented at the Globecom '82 - Global Telecomm. Conf., Miami, FL, 1982, pp. 1251–1255.

[49] P. Comon and C. Jutten, Eds., *Handbook of Blind Source Separation: Independent Component Analysis and Applications*. Oxford, United Kingdom: Academic Press, 2010.

[50] J. Herault and B. Ans, "Circuits neuronaux a synapses modifiables: decodage de messages composites par apprentissage non supervise," *C-R Acadamie Sci.*, vol. 299, no. III–13, pp. 525–528, 1984.

[51] B. Ans, J. Herault, and C. Jutten, "Adaptive neural architectures: detection of primitives," in *Proc. COGNITIVA'85*, Paris, France, 1985, pp. 593–597.

[52] J. Herault, C. Jutten, and B. Ans, "Detection de grandeurs primitives dans un message composite par une architecture de calcul neuromimetique en apprentissage non supervise," in *Actes du Xeme colloque GRETSI*, Nice, France, 1985, pp. 1017–1022.

[53] C. Jutten and J. Herault, "Blind separation of sources, part i: an adaptive algorithm based on a neuromimetic architecture," *Signal Process.*, vol. 24, pp. 1–10, 1991.

[54] P. Comon, "Independent component analysis, a new concept?," *Signal Process.*, vol. 36, pp. 287–314, 1994.

[55] A. J. Bell and T. J. Sejnowski, “An information-maximization approach to blind separation and blind deconvolution,” *Neural Comput.*, vol. 7, pp. 1129–1159, 1995.

[56] S. Haykin, Ed., *Unsupervised Adaptive Filtering: Volume 1: Blind Source Separation*. New York, NY: John Wiley & Sons, Inc., 2000.

[57] Cichocki A. and S. Amari, *Adaptive Blind Signal and Image Processing: Learning Algorithms and Applications*. West Sussex, United Kingdom: John Wiley & Sons, Ltd, 2002.

[58] S. Makino, T.-W. Lee, and H. Sawada, Eds., *Blind Speech Separation*. Dordrecht, The Netherlands: Springer, 2007.

[59] S. Makeig, A. J. Bell, T.-P. Jung, and T. J. Sejnowski, “Independent component analysis of electroencephalographical data,” *Adv. Neural Inf. Process. Syst.*, vol. 8, pp. 145–151, 1996.

[60] S. Moussaoui *et al.*, “On the decomposition of Mars hyperspectral data by ICA and Bayesian positive source separation,” *Neurocomputing*, pp. 2194–2208, 2008.

[61] D. Obradovic and G. Deco, “Unsupervised learning for blind source separation: an information-theoretic approach,” in *Proc. 1997 IEEE Int. Conf. on Acoust.., Speech, and Signal Process.*, Munich, Germany, 1997, vol. 1, pp. 127–130.

[62] J.-F. Cardoso, “High-order contrasts for independent component analysis,” *Neural Comput.*, vol. 11, pp. 157–192, 1999.

[63] S.-I. Amari, Cichocki A., and H. H. Yang, “A new learning algorithm for blind signal separation,” in *Advances in neural information processing*, Cambridge, MA: MIT Press, 1996, pp. 757–763.

[64] J.-F. Cardoso and B. Laheld, “Equivariant adaptive source separation,” *IEEE Trans. Signal Process.*, vol. 44, pp. 3017–3030, 1996.

[65] M. Wax and T. Kailath, “Detection of signals by information theoretic criteria,” *IEEE Trans. Acoust. Speech Signal Process.*, vol. ASSP-33, no. 2, pp. 387–392, 1985.

[66] D. T. Pham and J. F. Cardoso, “Blind separation of instantaneous mixtures of nonstationary sources,” *IEEE Trans. Signal Process.*, vol. 49, pp. 1837–1848, 2001.

[67] S.-I. Amari, T.-P. Chen, and A. Cichocki, “Nonholonomic orthogonal learning algorithms for blind source separation,” *Neural Comput.*, vol. 12, pp. 1463–1484, 2000.

[68] A. Cichocki and L. Moszczynski, “A new learning algorithm for blind separation of sources,” *Electron. Lett.*, vol. 28, no. 21, pp. 1986–1987, 1992.

[69] A. Cichocki, R. Unbehauen, and E. Rummert, “Robust learning algorithms for blind separations of signals,” *Electron. Lett.*, vol. 30, no. 17, pp. 1386–1387, 1994.

[70] A. Cichocki and R. Unbehauen, “Robust neural networks with on-line learning for blind identification and blind separation of sources,” *IEEE Trans. Circuits Syst.*, vol. 43, no. 11, pp. 894–906, 1996.

[71] P. Comon, “Analyse en composantes independantes et indentification aveugle,” *Trait. Signal*, vol. 7, pp. 435–450, 1990.

[72] E. Weinstein, M. Feder, and A. V. Oppenheim, “Multi-channel signal separation by decorrelation,” *IEEE Trans. Speech Audio Process.*, vol. 1, pp. 405–413, 1993.

[73] D. Yellin and E. Weinstein, “Criteria for multichannel signal separation,” *IEEE Trans. Signal Process.*, vol. 42, pp. 2158–2168, 1994.

[74] D. Yellin and E. Weinstein, “Multichannel signal separation: methods and analysis,” *IEEE Trans. Signal Process.*, vol. 44, pp. 106–118, 1996.

[75] H. Buchner, R. Aichner, and W. Kellermann, “Blind source separation for convolutive mixtures exploiting nongaussianity, nonwhiteness, and nonstationarity,” in *Proc. Int. Workshop on Acoust. Echo and Noise Control*, Kyoto, Japan, 2003, pp. 275–278.

[76] H. Buchner, R. Aichner, and W. Kellermann, “TRINICON: a versatile framework for multichannel blind signal processing,” in *Proc. 2004 IEEE Int. Conf. on Acoust., Speech, and Signal Process.*, Montreal, Quebec, Canada, 2004, vol. 3, pp. 889–892.

[77] H. Buchner, R. Aichner, and W. Kellermann, “A generalization of blind source separation algorithms for convolutive mixtures based on second-order statistics,” *IEEE Trans. Speech Audio Process.*, vol. 13, pp. 120–134, 2005.

[78] K. Gilbert and K. Payton, “Source enumeration of speech mixtures using pitch harmonics,” in *Proc. 2009 IEEE Workshop on Appl. of Signal Process. to Audio and Acoust.*, New Paltz, NY, 2009, pp. 89–92.

[79] L. Huang, X. Li, and S. Wu, “Reduced-rank MDL method for source enumeration in high-resolution array processing,” *IEEE Trans. Signal Process.*, vol. 55, pp. 5658–5667, 2007.

[80] S. Valaee and P. Kabal, “An information theoretic approach to source enumeration in array signal processing,” *IEEE Trans. Signal Process.*, vol. 52, pp. 1171–1178, 2004.

[81] S. Araki, H. Sawada, R. Mukai, and S. Makino, “Underdetermined blind sparse source separation for arbitrarily arranged multiple sensors,” *Signal Process.*, vol. 87, pp. 1833–1847, 2007.

[82] P. Bofill and M. Zibulevsky, “Underdetermined blind source separation using sparse representations,” *Signal Process.*, vol. 81, pp. 2353–2362, 2001.

[83] P. Georgiev, A. Cichocki, and F. Theis, “Sparse component analysis and blind source separation of underdetermined mixtures,” *IEEE Trans. Neural Netw.*, vol. 16, pp. 992–996, 2005.

[84] O. Yilmaz and S. Rickard, “Blind separation of speech mixtures via time-frequency masking,” *IEEE Trans. Signal Process.*, vol. 52, pp. 1830–1847, 2004.

[85] M. H. Radfar and R. M. Dansereau, “Single-channel speech separation using soft mask filtering,” *IEEE Trans. Audio Speech Lang. Process.*, vol. 15, pp. 2299–2310, 2007.

[86] H. L. Van Trees, *Optimum Array Processing. Part IV of Detection, Estimation, and Modulation Theory*. New York, NY: John Wiley & Sons Inc., 2002.

[87] E. Vincent, R. Gribonval, and C. Fevotte, “Performance measurement in blind audio source separation,” *IEEE Trans. Audio Speech Lang. Process.*, vol. 14, no. 4, pp. 1462–1469, 2006.

[88] E. Vincent, H. Sawada, P. Bofill, S. Makino, and J. P. Rosca, “First stereo audio source separation evaluation campaign: data, algorithms and results,” in *Independent Component Analysis and Signal Separation*, vol. 4666, M. E. Davies, C. J. James, S. A. Abdallah, and M. D. Plumbley, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2007, pp. 552–559.

[89] M. I. Mandel, S. Bressler, B. Shinn-Cunningham, and D. P. W. Ellis, “Evaluating source separation algorithms with reverberant speech,” *IEEE Trans. Audio Speech Lang. Process.*, vol. 18, no. 7, pp. 1872–1883, 2010.

[90] M. E. Fayad, D. C. Schmidt, and R. E. Johnson, Eds., *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. New York, NY: John Wiley & Sons, Inc., 1999.

[91] J. Greenfield and K. Short, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Indianapolis, IN: Wiley Publishing, Inc., 2004.

[92] P. Comon and C. Jutten, Eds., *Handbook of Blind Source Separation: Independent Component Analysis and Applications*. Oxford, United Kingdom: Academic Press, 2010.

[93] R. E. Johnson, “Frameworks = (components + patterns),” *Commun. ACM*, vol. 40, no. 10, pp. 39–42, 1997.

[94] D. Roberts and R. Johnson, “Evolving frameworks: a pattern language for developing object-oriented frameworks,” in *Proc. Third Conf. Pattern Languages and Programming*, Allerton Park, IL, 1996.

[95] R. Aichner, H. Buchner, F. Yan, and W. Kellermann, “A real-time blind source separation scheme and its application to reverberant and noisy acoustic environments,” *Signal Process.*, vol. 86, pp. 1260–1277, 2006.

[96] J. B. Allen and D. A. Berkley, “Image method for efficiently simulating small-room acoustics,” *J. Acoust. Soc. Am.*, vol. 65, no. 4, p. 943, Apr. 1979.

[97] E. A. P. Habets, “Room impulse response generator", *International Audio Laboratories Erlangen*, 2010. [Online]. Available: https://github.com/ehabets/RIR-Generator/blob/master/rir_generator.pdf. [Accessed: Jan. 24, 2019].

[98] P. J. Brockwell and R. A. Davis, *Introduction to Time Series and Forecasting*, 2nd ed. New York, NY: Springer Science + Business Media, LLC, 1996.

[99] F. M. J. Willems, T. J. Tjalkens, and Y. M. Shtarkov, “The context-tree weighting method: basic properties,” *IEEE Trans. Inf. Theory*, vol. 41, pp. 653–664, 1995.

[100] S. S. Kozat, G. C. Zeitler, and A. C. Singer, “Universal piecewise linear prediction via context trees,” *IEEE Trans. Signal Process.*, vol. 55, pp. 3730–3745, 2007.

[101] S. S. Kozat, A. J. Bean, and A. C. Singer, “Universal portfolios via context trees,” in *Proc. 2008 IEEE Int. Conf. on Acoust., Speech and Signal Process.*, Las Vegas, NV, 2008, pp. 2093–2096.