Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Reimplementation of AMICA in Julia

Alexander Lulkin

Course of Study:	Informatik
Examiner:	Jun.-Prof. Dr. Benedikt Ehinger
Supervisor:	Jun.-Prof. Dr. Benedikt Ehinger
Commenced:	March 20, 2023
Completed:	September 20, 2023

Abstract

Adaptive Mixture Independent Component Analysis (AMICA) is one of the best performing ICA algorithms. But despite being commonly used, the already existing implementations lack the readability and extensibility needed to allow for future development of the algorithm. This work uses a MATLAB implementation of the AMICA algorithm as reference to develop an easier to understand version. This new implementation is written in the scientific programming language Julia. An evaluation of the correctness, the performance and of how well this implementation achieves it’s goals is provided in this thesis.

Contents

List of Figures

Acronyms

AMICA Adaptive Mixture Independent Component Analysis. 7

BSS Blind Source Separation. 13

CLT Central Limit Theorem. 14

EEG Electroencephalography. 11

EM Expectation-maximization. 19

GGD Generalized Gaussian Distribution. 18

GGMM Generalized Gaussian Mixture Model. 17

ICA Independent Component Analysis. 13

PDF Probability Density Function. 14

1 Introduction

Electroencephalography (EEG) is a method to record electric signals in the brain. This is achieved by placing electrodes along the scalp. The non-invasiveness of this method is it's biggest advantage, but it also comes with some drawbacks. The signals might mix and interfere with each other, making it harder to identify different sources in the brain. Noise also poses an issue. For example, discharges from neuro-muscular activity might get recorded alongside the brain activity and need to be filtered out with post-processing techniques. This problem is generally known as the Blind Source Separation Problem [CL96]. One technique, that can be used to solve the BSS problem, is Independent Component Analysis (ICA) [Com94]. ICA algorithms model the data as a linear combination of the source signals. By using the fact that mixed signals tend to be more dependent on each other than unmixed signals, ICA can separate the data by increasing the independence between the components. Different variations of ICA exist, one of which is Adaptive Mixture Independent Component Analysis (AMICA) [HPP+18]. AMICA is a multi-layered mixing network which combines multiple ICA mixture models with mixtures of generalized Gaussian distributions. The method has been successfully implemented and used in different programming languages, such as MATLAB or Fortran. These implementations however come with their own disadvantages. For example, the Fortran implementation is hard to maintain, understand and further develop. The MATLAB implementation on the other hand is easier to develop, but suffers from worse performance. This thesis presents an implementation which is easier to understand and to extend than both the MATLAB and the Fortran version. The new implementation is called *Amica.jl* and is written in the scientific programming language Julia [BEKS17]. An evaluation of the implementation with regard to the correctness, the performance, the readability and extensibility is provided. The main goal of this work is to lay a solid foundation for the future, with the hope that at one point Amica.jl becomes a well rounded, easy to use and well performing AMICA implementation.

2 Independent Component Analysis

2.1 Blind Source Separation

In a Blind Source Separation (BSS) problem, multiple receivers receive different mixtures of signals from multiple sources [CL96]. These mixtures typically have the form of linear combinations. The goal is to identify the original sources from the mixtures alone, without any information about the structure of those mixtures. A well-known example of this problem is the so called *Cocktail Party Effect* [Aro92]. Imagine two people talking simultaneously (Fig. 2.1). Two microphones are placed in the room and are recording the conversation. As both microphones will be able to record both people, the recorded sound will have the form of linear mixtures. Those mixtures however will have a key difference: As the microphones are not in the same place, they will record both voices at different intensities. This difference can be used to extract the original sources by applying a method called Independent Component Analysis (ICA) [LCG10].

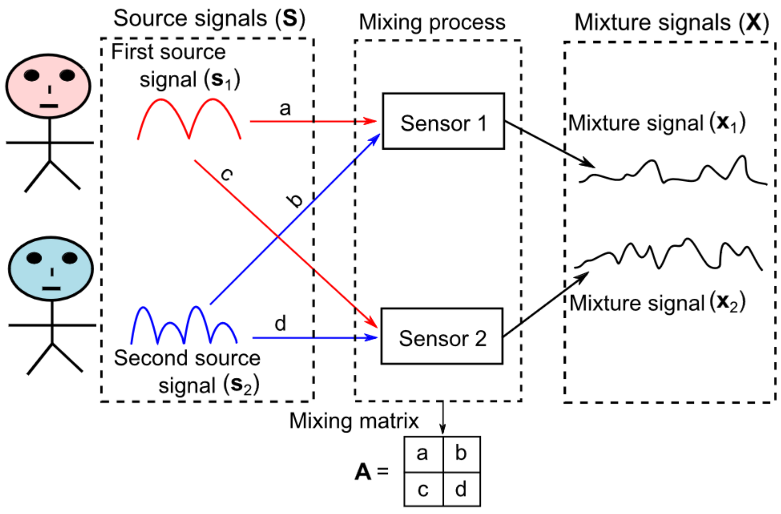

Source signals (**S**)
First source signal ($\mathbf{s}_1$)
Second source signal ($\mathbf{s}_2$)
a
b
c
d
Mixing process
Sensor 1
Sensor 2
Mixture signals (**X**)
Mixture signal ($\mathbf{x}_1$)
Mixture signal ($\mathbf{x}_2$)
Mixing matrix
A =
a
b
c
d

Figure 2.1: Two different mixtures are recorded from two source signals (Image by Alaa Tharwat, 2018) [Tha21].

2.2 ICA

By modeling data as a linear combination of independent source signals, ICA is able to decompose the data into different components which are maximally independent from each other [LCG10]. Mathematically this can be expressed as

$$x = A \cdot s \tag{2.1}$$

x being the n-channeled input data with N samples, A being the $(n \times n)$ mixing matrix and s the original source signals $(n \times N)$. Recovering the sources equals to computing the *unmixing* matrix $W = A^{-1}$ so that

$$s = W \cdot x \tag{2.2}$$

2.2.1 Conditions

Not all signal mixtures are actually separable. In order for ICA to be able to recover the unmixing matrix, certain conditions must be met. These are the following [CL96; LCG10]:

1. The source signals must be statistically independent from each other. Two random variables x_1 and x_2 with Probability Density Function (PDF) $p_i(x_i)$ are considered temporarily mutually independent, if $p(x_1, x_2) = p_1(x_1) \cdot p_2(x_2)$.
2. Mixtures must be linearly independent from each other, as a linearly dependent mixture does not provide additional information.
3. The source signals must be noise free, as noise can be interpreted as an additional independent component.
4. The Data must be centered (this is not always a requirement).
5. Only a single source signal is allowed to have a gaussian PDF.

2.2.2 How does ICA work

Many popular ICA algorithms, like InfoMax or FastICA, work by maximizing the non-Gaussianity of the estimated source signals [LCG10]. This is possible by making use of condition 5 and the Central Limit Theorem (CLT), which states that the average of multiple random variables will have a distribution that tends towards the normal distribution, even if the random variables themselves are not normally distributed. Since the source signals are expected to not be normally distributed, but their mixture is, recovering them should be possible by rotating the signals until they are as non-Gaussian as possible. However, there is a little caveat. The data is not 100% recoverable, the estimated source signals can differ from the original sources by permutation and scaling [NK11].

Permutation: It is not possible for ICA to recover the original order of the source signals. This is because both the mixing matrix and the source signals can be permuted by a permutation matrix P and it's inverse P^{-1}, so that the resulting data x will still be the same:

$$\begin{aligned} x &= AP^{-1}Ps \qquad (2.3)\\ &= A's' \qquad (2.4) \end{aligned}$$

Scaling: Because only the mixed data is known, it is impossible to determine whether the source signals have been scaled or not. This is made clear by the following equation:

$$x = A \cdot s \tag{2.5}$$

$$= \sum_{j=1}^{N} a_j s_j \tag{2.6}$$

$$= \sum_{j=1}^{N} (a_j / \alpha_j)(\alpha_j s_j) \tag{2.7}$$

Choosing different values for the scaling factor α will still produce the same data x, so it will not be possible to recover α during ICA.

2.2.3 Preprocessing

Before applying ICA to a given data set, it is advised to apply some preprocessing steps. The most notable ones are *centering* and *sphering/whitening* [NK11].

Centering is the practice to subtract the mean from the input data. This allows to assume a zero mean during the calculation, which simplifies the ICA algorithm. This does not affect the estimation of the mixing matrix, because the mean can always be added back to estimate the original source signals [NK11]:

$$x_c = x - E(x) \tag{2.8}$$

$$s = A^{-1}(x_c + E(x)) \tag{2.9}$$

Whitening (also known as **sphering**) is a method which transforms the input data in such a way, so that the components have unit variance and satisfy the following equation:

$$E(x_w \cdot x_w^T) = I \tag{2.10}$$

x_w being the whitened input data and $E(x_w \cdot x_w^T)$ being the covariance matrix of x_w. Whitening the data causes the mixing matrix to be orthogonal, which reduces the amount of elements to be estimated from n^2 to $n(n-1)/2$, which significantly reduces the complexity of the calculation [NK11].

2.2.4 Use Cases

While ICA has a lot of uses in all of signal processing, it is especially useful in EEG [SLB05]. In EEG, electric brain signals are recorded with the help of electrodes which are placed on the scalp. While this method has the advantage of being non-invasive, it does also have some drawbacks. Because the electrodes are not placed directly on the source of the signals, they can only record mixtures of those signals. Another issue is noise. For example discharges from neuromuscular activity can get recorded alongside the brain signals. ICA is well suited to identify the underlying components in the recordings and to filter out the noise.

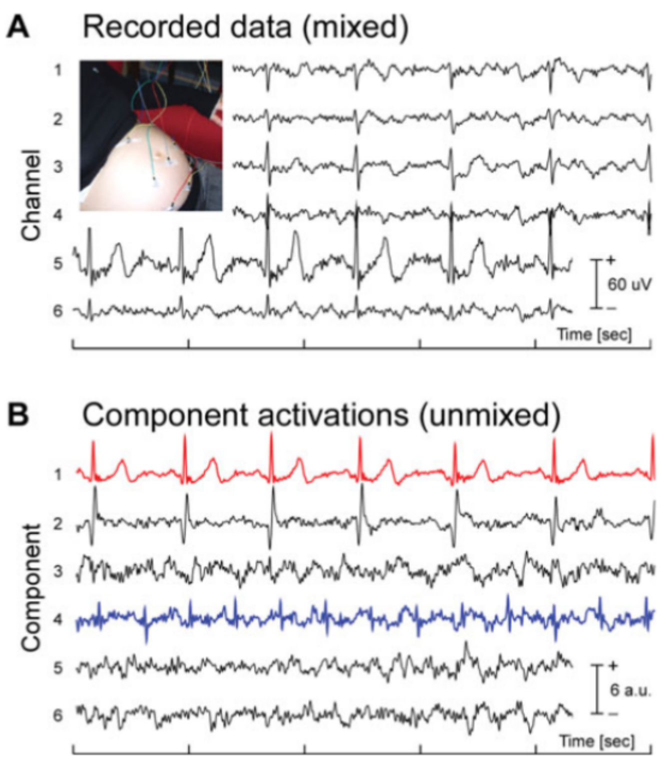


Figure 2.2: Decomposition of electric bio-potential recordings via electrodes on the womb of a pregnant woman (Image by Filipa Campos Viola et al., 2010) [VDTS10].

Figure 2.2 shows recordings of signals on a pregnant woman's womb. After applying ICA, multiple components were identified from the mixed signals. For example the red line shows the mothers heartbeat, while the blue line shows the unborn baby's heartbeat, which was previously not visible [VDTS10].

3 Adaptive Mixture Independent Component Analysis

A widely used ICA algorithm is AMICA [HPP+18]. AMICA has two unique core features, which make it stand out from other ICA algorithms:

1. It allows to model the given data with multiple ICA mixture models at the same time, with different models applying to different samples. This can be advantageous in EEG, as there is evidence that brain activities can vary depending on the brain state, requiring different models for each state [HJ17; HPP+18].
2. The PDFs of the estimated source signals are approximated by an adaptive Generalized Gaussian Mixture Model (GGMM), which is better suited for modeling non-Gaussian source signals than a pre-defined PDF [PKM06].

Combined with the regular ICA assumption of the data **x** being a mixture of independent components, it becomes clear why AMICA truly deserves to have mixture in it's name. The algorithm ends up having three separate layers of mixtures: A mixture of ICA models, a mixture of independent components and a mixture of generalized Gaussians. This is illustrated in figure 3.1.

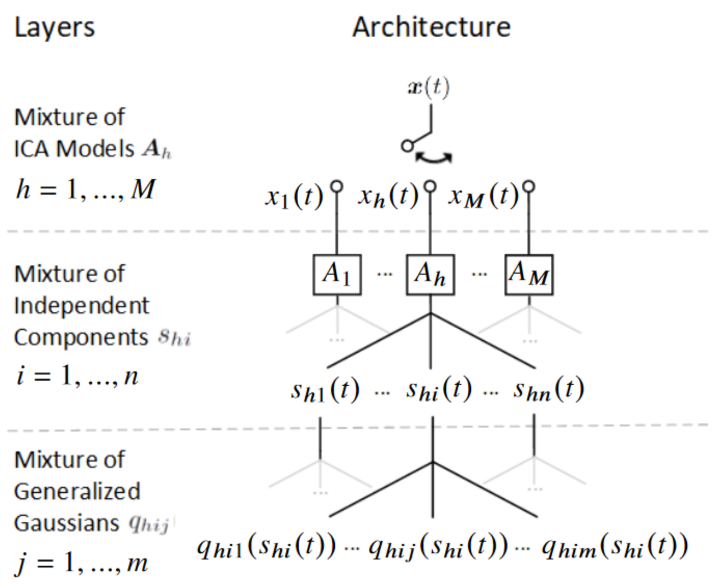

Layers
Architecture
$\boldsymbol{x}(t)$
Mixture of ICA Models $\boldsymbol{A}_h$
$h = 1, \ldots, M$
$x_1(t)$ $x_h(t)$ $x_M(t)$
A_1 ··· A_h ··· A_M
Mixture of Independent Components s_{hi}
$i = 1, \ldots, n$
$s_{h1}(t)$ ··· $s_{hi}(t)$ ··· $s_{hn}(t)$
Mixture of Generalized Gaussians q_{hij}
$j = 1, \ldots, m$
$q_{hi1}(s_{hi}(t))$ ··· $q_{hij}(s_{hi}(t))$ ··· $q_{him}(s_{hi}(t))$

Figure 3.1: The architecture of AMICA (Image adapted from Sheng-Hsiou Hsu et al., 2018) [HPP+18].

3.1 Mixture of ICA models

Just like in regular ICA, the data **x** is given as an $n \times N$ matrix (n channels, N time samples). In AMICA however, a different model might apply to a different time sample, so **x** is modeled by a mixture of M models with model index $h = 1, ..., M$ and time $t = 1, ..., N$:

$$\mathbf{x}(t) = \mathbf{x}_h(t) = \mathbf{A}_h \mathbf{s}_h(t) + \mathbf{c}_h \tag{3.1}$$

A_h being a mixing matrix, $\mathbf{s}_h(t)$ the original source signals and $\mathbf{c}_h$ the bias. Together with $h = h(t)$ they make up the dominant model at time t. The *likelihood function* of the data being generated by the ICA model mixture can then be written as:

$$p(\mathbf{X}|\Theta) = \prod_{t=1}^{N} \sum_{h=1}^{M} p(\mathbf{x}(t)|C_h, \theta_h) \cdot p(C_h) \tag{3.2}$$

$\Theta = \{\theta_1, ..., \theta_M\}$ are the model parameters and $p(C_h)$ the probability of model h being active at time t. The prior probabilities of a certain model being active satisfy $\sum_{h=1}^{M} p(C_h) = 1$. The probability of the data being generated by a specific ICA model can be calculated with the *likelihood function*:

$$p(\mathbf{x}(t)|C_h, \theta_h) = \left|det\mathbf{A}_h^{-1}\right| \cdot \prod_{i=1}^{n} p(s_{hi}(t)) \tag{3.3}$$

This makes use of the independence assumption for the original source signals, which states that $p(x_1, x_2) = p(x_1) \cdot p(x_2)$. The determinant of the unmixing matrix A^{-1} is supposed to account for possible scaling of the data by the mixing process.

3.2 Mixture of Generalized Gaussians

While other ICA algorithms might assume a fixed PDF for the source signals, AMICA instead models them as a mixture of generalized Gaussian distributions [HPP+18]. A parameterized Generalized Gaussian Distribution (GGD) is defined by:

$$q(s; \rho, \beta, \mu) = \frac{\rho}{2\beta \cdot \Gamma(1/\rho)} e^{\left(-\left|\frac{s-\mu}{\beta}\right|\right)^{\rho}} \tag{3.4}$$

The shape of the GGD is set by parameter ρ, the scale by β and the location by μ. The mixture of multiple GGDs which approximates the PDF of the source signals is therefore defined by:

$$p(s_{hi}(t)) = \sum_{j=1}^{m} \alpha_{hij} \cdot q(s_{hi}(t); \rho_{hij}, \mu_{hij}, \beta_{hij}) \tag{3.5}$$

with each GGD being weighted differently by the weight parameter α.

In summary, the likelihood of the data being generated by a given ICA model can be expressed as:

$$L_{h(t)} = p(C_h) \cdot \left|det\mathbf{A}_h^{-1}\right| \cdot \prod_{i=1}^{n} \sum_{j=1}^{m} \alpha_{hij} \cdot q(s_{hi}(t); \rho_{hij}, \mu_{hij}, \beta_{hij}) \tag{3.6}$$

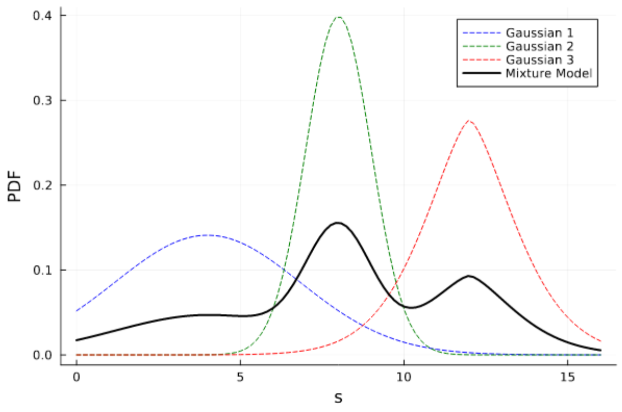


Figure 3.2: Gaussian mixture model consisting of three equally weighted parameterized Gaussians (Created with distributions.jl [BPA+19]).

And finally, the probability of a model with index h being active at time t can be calculated by normalizing $L_{h(t)}$:

$$p(h(t)) = L_{h(t)} / \sum_{h=1}^{M} L_{h(t)} \tag{3.7}$$

3.3 Structure

In order to model the data, AMICA needs to estimate the model parameters Θ with

$$\Theta = \{\mathbf{A_h^{-1}}, \mathbf{c}_h, \gamma_h, \alpha_{hij}, \beta_{hij}, \rho_{hij}, \mu_{hij}\} \tag{3.8}$$

for ICA models $h = 1, ..., M$, channels $i = 1, ..., n$ and Gaussian mixture models $j = 1, ..., m$. γ_h is the probability of model h being active with $\gamma_h = p(C_h)$. AMICA works as an Expectation-maximization (EM) algorithm in order to iteratively maximize the model parameters.

3.3.1 Expectation-Maximization Principle

Maximum likelihood estimation usually requires a complete data set, but an EM algorithm is able to perform the estimation even when there are latent variables present [DLR77]. This is achieved by iteratively optimizing a model, while alternating between expectation and maximization steps. During the E-step, the algorithm calculates a likelihood estimation based on given model parameters. These parameters are then replaced by better fitting ones during the M-step. This is made possible by using optimization methods, e.g. gradient descent, on the previously calculated likelihood function. Since both the model parameters and the actual likelihood function are unknown, the algorithm needs to be initialized using estimates.

AMICA uses this principle to optimize the model parameters (3.8). During the expectation step, the source signals s are estimated using the unmixing matrix A^{-1}. The probability density function of the source signals is estimated by using a mixture of generalized Gaussian distributions. Both of those are then used to calculate an estimation of the likelihood function. During the M-step, AMICA uses the Newton method to maximize the likelihood function and thus generating better model parameters [PMKR08].

3.3.2 Algorithm

Algorithm 3.1 shows a simplified version of the AMICA algorithm. The parameters for the generalized Gaussian distribution mixtures are summarized as $\Omega = \{\alpha, \beta, \rho, \mu\}$. In the reference implementation in MATLAB, some parameters start off as fixed values instead of being initialized randomly. It is also worth mentioning that AMICA possesses an adaptive learning rate, which can decrease if the the likelihood gets worse between iterations.

Algorithm 3.1 AMICA algorithm

```
procedure AMICA($\mathbf{x}$,$M$,$m$,$maxiter$)
    $\mathbf{x} \leftarrow$ RemoveMean($\mathbf{x}$)
    $\mathbf{x} \leftarrow$ Whitening($\mathbf{x}$)
    $\mathbf{W} \leftarrow$ InitializeRandomly                    // Unmixing Matrix $\mathbf{W} = \mathbf{A}^{-1}$
    $\mathbf{\Omega} \leftarrow$ InitializeRandomly                // Gaussian Mixture Model Parameters
    for $iter = 1$ to $maxiter$ do
        $\mathbf{s} \leftarrow W \cdot \mathbf{x}$                                  // Estimate sources
        $\mathbf{Lt} \leftarrow$ CalculateLikelihood($\mathbf{s}$,$\mathbf{\Omega}$)   // Of time points for each model ($M \times N$)
        $\mathbf{LL} \leftarrow$ CalculateMixtureLL($\mathbf{Lt}$)     // Likelihood for whole ICA model mixture
        $\mathbf{dLL}[iter] \leftarrow \mathbf{LL}[iter] - \mathbf{LL}[iter-1]$    // Change in LL since last iter
        if $\mathbf{dLL}[iter] < mindLL$ then
            Terminate                                  // Terminates if LL increase to small
        end if
        $\Omega \leftarrow$ UpdateParameters($\mathbf{\Omega}$)          // Improves Gaussian parameters
        $\mathbf{W} \leftarrow$ NewtonMethod($\mathbf{W}$,$\mathbf{s}$)                     // Improves $\mathbf{W}$
        Reparameterize($\mathbf{W}$,$\mathbf{\Omega}$)                       // Normalizes parameters
    end for
end procedure
```

4 Results

4.1 Implementation

AMICA implementations already exist in Fortran and MATLAB. Both have their own advantages and disadvantages. While the Fortan implementation performs better, future development might be hindered because the code is hard to read. The MATLAB implementation on the other hand is significantly more readable, but suffers from worse performance and also lacks some key features. The MATLAB code, provided by *Jason Palmer*, is the basis for the implementation of AMICA in Julia. Important variables are provided in table 4.1. Because many of them have been renamed, the variable names of both implementations are provided.

Julia	Matlab	Definition
data	x	Input data
A	A	Mixing matrix
W	W	Unmixing matrix
centers	c	Model centers
source_signals	b	unmixed signals
mindll	mindll	Termination criterion
iterwin	iterwin	Window to calculate average likelihood
LL	LL	Log-likelihood for whole mixture model
Lt	Lt	Likelihood for each time point per model
dLL	dLL	LL difference between iters
sdll	sdll	average dLL over iterwin iterations
ica_weights_per_sample	v	ICA weights for each sample
ica_weights	vsum	ICA weights
normalized_ica_weights	gm	Normalized ICA weights
proportions	alpha	source density mixture proportion
scale	beta	source density scale
location	mu	source density location
shape	rho	source density shape
y	y	source signals
Q	Q	Densities for each sample per Gaussian
z	z	Normalized Densities for each sample per Gaussian

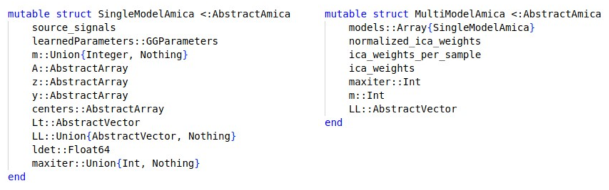
Table 4.1: Important variables of AMICA.

4.1.1 Code examples

One major feature of *Amica.jl* are the mutable structure data types, which are used to define multiple different types of AMICA algorithms. Those structures define which parameters are needed for a certain type of AMICA. When the algorithm is executed, an object of the associated data type gets created. This has multiple advantages:

1. Thanks to *Multiple Dispatch*, a core feature of the Julia programming language, the same function can be defined multiple times depending on the data type of a given parameter [BEKS17]. This allows functions to behave differently if needed, depending on which type of AMICA object they are given. If a new variation of AMICA gets defined, new versions of the functions in the algorithm can be created easily without breaking the previous implementation.
2. The code gets significantly more readable, because the functions don't need to be called with a dozen parameters. For many functions it is sufficient to call them with no parameters besides the AMICA object.

Therefore this feature helps to achieve both goals of readability and extensibility.

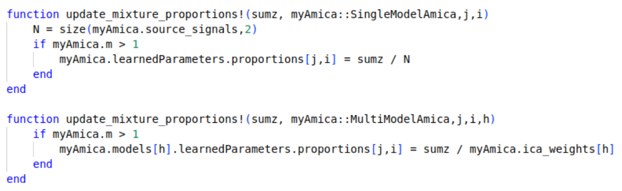


```julia
mutable struct SingleModelAmica <:AbstractAmica
    source_signals
    learnedParameters::GGParameters
    m::Union{Integer, Nothing}
    A::AbstractArray
    z::AbstractArray
    y::AbstractArray
    centers::AbstractArray
    Lt::AbstractVector
    LL::Union{AbstractVector, Nothing}
    ldet::Float64
    maxiter::Union{Int, Nothing}
end
```

```julia
mutable struct MultiModelAmica <:AbstractAmica
    models::Array{SingleModelAmica}
    normalized_ica_weights
    ica_weights_per_sample
    ica_weights
    maxiter::Int
    m::Int
    LL::AbstractVector
end
```

Figure 4.1: The SingleModelAmica and MultiModelAmica data type.

Figure 4.1 shows the *SingleModelAmica* and MultiModelAmica data type. *SingleModelAmica* is supposed to be used when only one ICA model is needed instead of a mixture model. *Multi-ModelAmica* contains an array of *SingleModelAmica* objects and additional parameters. Note that *SingleModelAmica* itself contains an object of another custom data type: *GGParameters*. This type contains the parameters which define a GGMM. A possible use case of this is shown in figure 4.2.



```julia
function update_mixture_proportions!(sumz, myAmica::SingleModelAmica,j,i)
    N = size(myAmica.source_signals,2)
    if myAmica.m > 1
        myAmica.learnedParameters.proportions[j,i] = sumz / N
    end
end

function update_mixture_proportions!(sumz, myAmica::MultiModelAmica,j,i,h)
    if myAmica.m > 1
        myAmica.models[h].learnedParameters.proportions[j,i] = sumz / myAmica.ica_weights[h]
    end
end
```

Figure 4.2: *Multiple Dispatch* allows to define a function multiple times for different types of AMICA.

The function *update_mixture_proportions!* behaves differently depending on whether it was called with a *SingleModelAmica* or a *MultiModelAmica object*. The additional *MultiModelAmica* function is required because it needs the ICA model weights. This parameter is obviously not needed when performing AMICA with just one ICA model. In some cases however this is not needed. If a function only requires *SingleModelAmica* parameters, then it can be used by *MultiModelAmica* without the need for overloading it. This is shown in figure 4.3. This code is actually part of a *MultiModelAmica* function. But because the *SingleModelAmica* versions of the functions are sufficient, the *SingleModelAmica* objects which are stored in the *MultiModelAmica* are given as a parameter instead.

```
for h in 1:M
    Threads.@threads for i in 1:n
        Q = calculate_Q(myAmica.models[h],Q,i)
        calculate_u!(myAmica.models[h],Q,i)
        calculate_Lt!(myAmica.models[h],Q)
    end
end
```

Figure 4.3: Multiple *SingleModelAmica* functions being called inside a *MultiModelAmica* function.

4.2 Evaluation: Correctness

The MATLAB implementation provided by *Jason Palmer* was used as a the reference implementation to evaluate the correctness of the Julia implementation. The first test consisted of running both implementations on a small synthetic data set with 4 channels and 1000 time samples, which was a mixture of sinus functions and pink Gaussian noise. Usually the model parameters A, β and μ are initialized with random values. To make the results comparable, they were set to the same fixed initial values in both implementations during the tests. Because bad initial parameter values can cause AMICA to terminate early due to NaN or infinite values, the initial values were chosen such that the algorithm could run until convergence. This was done with a single ICA model and with a mixture of two ICA models. The amount of generalized Gaussians was set to 3. The iteration window was set to 10, therefore the algorithm terminated if the average likelihood increase over 10 iterations dropped below $1e^{-8}$. No data pre-processing was performed on the synthetic data. All plots were created with CairoMakie.jl [DK21].

4.2.1 Single Model

As seen in figure 4.4, both implementations were able to identify the the original functions as independent components. The data was not perfectly recovered and appears to be out of order and scaled, but that was expected due to ICA ambiguities (see chapter 2.2.2).

The MATLAB implementation terminated after 544 iterations due to convergence with a final log-likelihood of -1.639571. The Julia implementation terminated after 527 iterations with a log-likelihood of -1.639574.

4.2.2 Multi Model

For the evaluation of the AMICA algorithm with two ICA models, half of the synthetic data was mixed with one mixing matrix and the other half with another. Figure 4.5 shows the first and last 100 data samples, both in their original form and mixed.

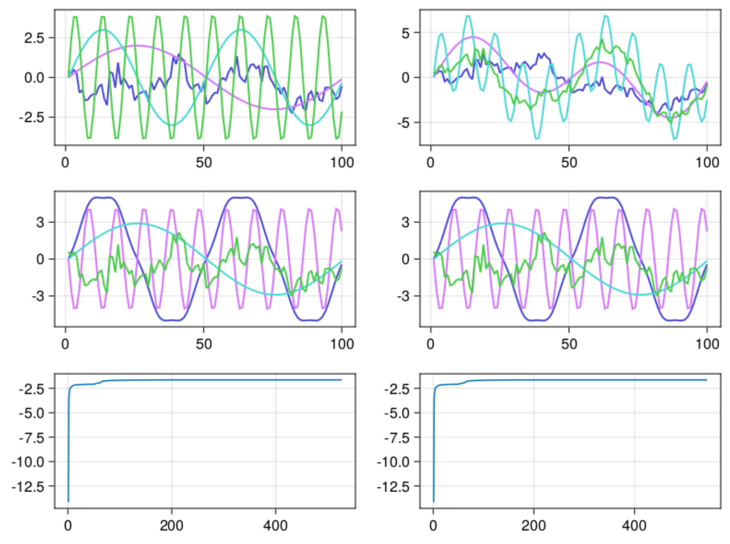


Figure 4.4: Julia and Matlab unmixing of sinus data after 500 iterations. **Top left**: Original source signals; **Top right**: Mixed source signals. **Middle left**. Unmixed signals (Julia). **Middle right**: Unmixed signals (MATLAB). **Bottom left**: Likelihood (Julia). **Bottom right**: Likelihood (MATLAB).

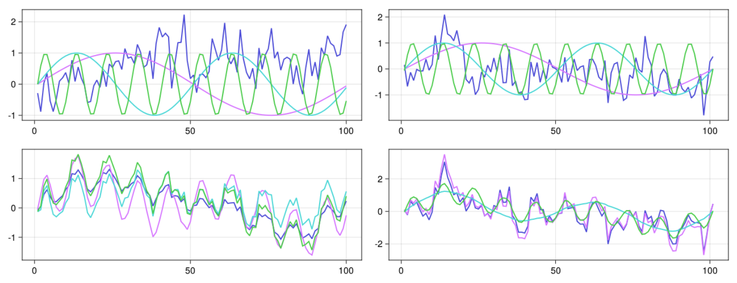


Figure 4.5: Top: First and last 100 samples of the original sources. **Bottom**: First and last 100 samples of the mixed data, mixed with two separate mixing matrices.

Since the data was mixed by using two different matrices, *MultiModel* AMICA should be able to identify the sources by training both it's ICA mixture models. As shown in figure 4.6, this is exactly what happened. Applying both unmixing matrices to their respective parts of the mixed data makes the source signals visible again.

The Julia implementation terminated after 978 iterations due to convergence, the MATLAB implementation after 758. Their final log-likelihoods were 0.0151703 and -0.006776 respectively. The log-likelihood functions can both be seen in figure 4.7.

In order to test if both generated unmixing matrices are only specialized for part of the data, they were also applied in the other way around. Figure 4.8 shows what happens if the matrix which was trained for the first half of the data is applied to the second half and vice versa.

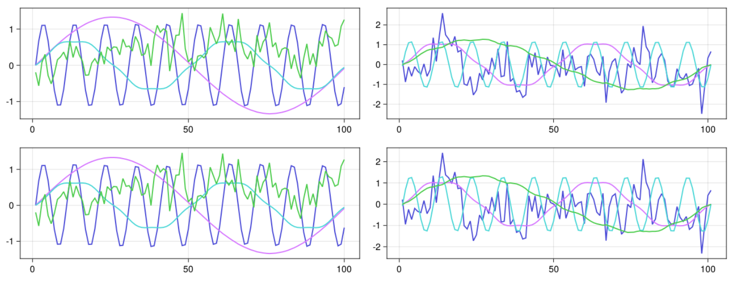


Figure 4.6: Top: First and last 100 samples of the data unmixed by the MultiModel Julia implementation. using both generated unmixing matrices. **Bottom**: First and last 100 samples of the data unmixed by the MATLAB implementation.

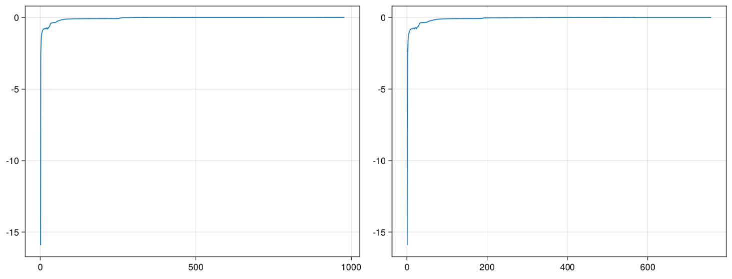


Figure 4.7: Log-likelihood functions of MultiModel Julia (left) and MATLAB (right) when performed on the synthetic data set.

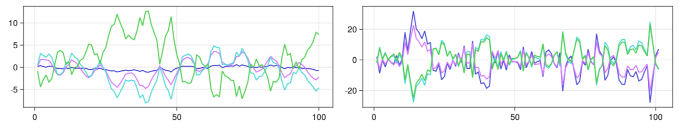


Figure 4.8: Unmixing matrices applied in reverse. The components do not get properly unmixed.

4.3 Evaluation: Performance

4.3.1 Execution Time

In order to evaluate the performance, the implementations of AMICA in Julia, MATLAB and Fortran have all been run on the *SSVEP* data set from MNE [GLL+13]. This data set contains 32-channeled EEG recordings from two participants who were experiencing visual stimulation in the form of checkerboard patterns. The data was loaded with a 1Hz highpass filter and 128 Hz resampling. Mean removal and sphering were applied by the AMICA algorithm. The implementations were tested with one, two and four ICA mixture models, both with a single active thread and with 64 threads. Every run lasted for 100 iterations and the average time per iteration was calculated. The initial parameters were set on the same values for Julia and Matlab, but were randomized in Fortran. Because of this, the Fortran implementation was executed 3 times for each experiment and the average time was taken.

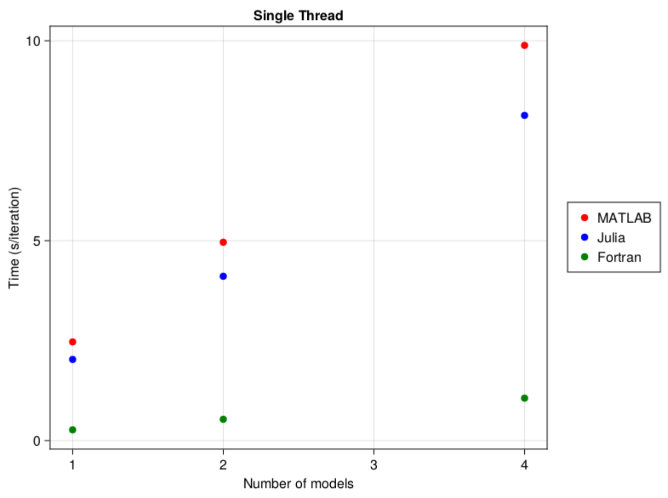


Figure 4.9: AMICA execute time for 100 iterations with one thread.

As seen in figure 4.9, the Matlab and Julia implementations performed similarly for one model. Julia took 2.03 seconds per iteration and MATLAB 2.4 seconds. While this is a slight win for the Julia implementation, both were beaten by the Fortran implementation with 0.26 seconds per iteration. With an increasing amount of models, the difference in performance grows relatively smaller. The Fortran implementation was 6.1 times faster than Matlab and 5.08 times faster than Julia (previously 9.2 times and 7.8 times respectively). With four ICA models, the MATLAB implementation reached 9.88 seconds per iteration, the Julia implementation reached 8.13 and the Fortran implementation 1.6 seconds per iteration. Figure 4.10 shows the same experiment performed with 64 active threads.

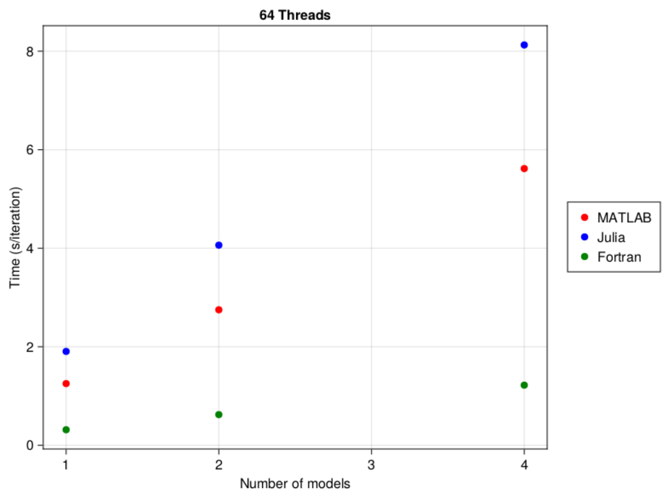


Figure 4.10: AMICA execute time for 100 iterations on 64 threads.

In this setup, the MATLAB implementation got ahead of Julia for one, two and four models. This was expected, as there was not much optimization done on the Julia implementation so far. Another interesting observation is the Fortran implementation not getting any speed up from being executed with multiple threads. Quite the opposite, it even got slightly slower, going from 0.27 seconds to 0.315 for one model A possible reason for this is an error in the experimental setup.

Convergence on EEG Data

The three implementations were executed on the same data set as in the previous section. For Julia and MATLAB, the parameters were initialized with the same fixed values again. The Fortran implementation ran with random initial parameters. The experiment was done with a single ICA model. MATLAB and Julia were executed until convergence and the Fortran implementation was set to 1200 iterations. It terminated with a final log-likelihood of -0.7097. The log-likelihood at 1200 iterations was -1.39657 for Julia and -1.39656 for MATLAB.

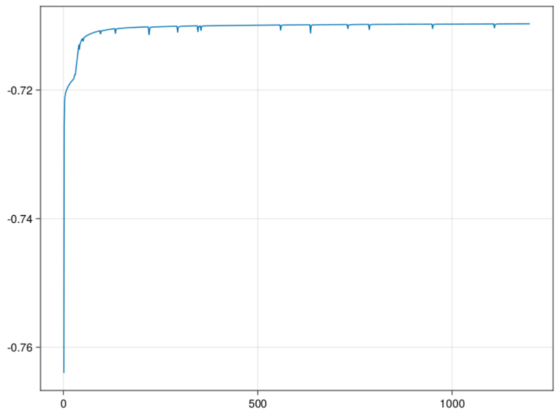


Figure 4.11: Log-likelihood of the Fortran AMICA implementation.

The Julia implementation converged with a final log-likelihood of -1.396543 after 1430 iterations and the MATLAB implementation with -1.3965477 after 1735 iterations. A calculation of the difference in log-likelihood during iteration 1430 in MATLAB showed that the termination criterion of sdll < mindll was just missed by $-2.1866e^{-07}$, with sdll being the avererage difference in log-likelihood for the last 10 iterations and mindll = $1e^{-08}$. This can be due to small differences in how basic math functions are calculated in Julia.

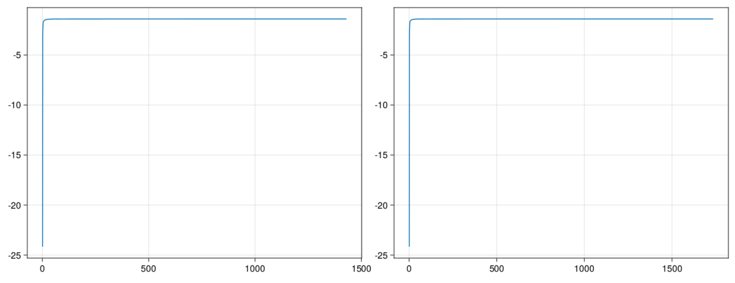


Figure 4.12: Log-likelihood of the Julia implementation (left) and MATLAB (right).

5 Conclusion and Outlook

The results show that multiple goals of this project have been achieved. A working implementation of AMICA has been produced. The evaluation shows that not only is Amica.jl unmixing independent source signals properly, but the results are very close to the reference implementation in MATLAB. Another goal that has been achieved is providing a basis for further development in the future. The code of *Amica.jl* is is more descriptive, with variables and functions providing information about their purpose by their names alone. By spreading the functionality among many small parts, the whole becomes easier to understand. The goal of providing modularity is also achieved. A new type of AMICA algorithm can simply be defined as a new data type and if some of the existing functions need to be adjusted, this can be done easily due to Julia's *Multiple Dispatch*.

Nevertheless there is still huge room for improvement by every metric. The performance of the implementation is clearly lackluster. It can be considered an achievement that *amica.jl* can, at least with a signle thread, perform better than the reference implementation in MATLAB. But when it comes to making use of multiple threads, the implementation needs to be adjusted. The Fortran implementation remains the gold standard when it comes to performance. Regarding readability, there are still multiple functions in the code which have not been untangled properly. This should definitely be done.

Another important point is that Amica.jl is based on the MATLAB version, which lacks in functionality when compared to the Fortran implementation. For example, the Fortran implementation is able to restart the algorithm when newly randomized parameters if it encounters a NaN value early. Amica.jl will just terminate with an exception. Bringing more features like this to Amica.jl should be the goal for the future.

Bibliography

[Aro92] B. Arons. “A review of the cocktail party effect”. In: *Journal of the American Voice I/O society* 12.7 (1992), pp. 35–50 (cit. on p. 13).

[BEKS17] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah. “Julia: A fresh approach to numerical computing”. In: *SIAM review* 59.1 (2017), pp. 65–98 (cit. on pp. 11, 22).

[BPA+19] M. Besançon, T. Papamarkou, D. Anthoff, A. Arslan, S. Byrne, D. Lin, J. Pearson. “Distributions. jl: Definition and modeling of probability distributions in the JuliaStats ecosystem”. In: *arXiv preprint arXiv:1907.08611* (2019) (cit. on p. 19).

[CL96] X.-R. Cao, R.-w. Liu. “General approach to blind source separation”. In: *IEEE Transactions on signal Processing* 44.3 (1996), pp. 562–571 (cit. on pp. 11, 13, 14).

[Com94] P. Comon. “Independent component analysis, a new concept?” In: *Signal processing* 36.3 (1994), pp. 287–314 (cit. on p. 11).

[DK21] S. Danisch, J. Krumbiegel. “Makie. jl: Flexible high-performance data visualization for Julia”. In: *Journal of Open Source Software* 6.65 (2021), p. 3349 (cit. on p. 23).

[DLR77] A. P. Dempster, N. M. Laird, D. B. Rubin. “Maximum likelihood from incomplete data via the EM algorithm”. In: *Journal of the royal statistical society: series B (methodological)* 39.1 (1977), pp. 1–22 (cit. on p. 19).

[GLL+13] A. Gramfort, M. Luessi, E. Larson, D. A. Engemann, D. Strohmeier, C. Brodbeck, R. Goj, M. Jas, T. Brooks, L. Parkkonen, M. S. Hämäläinen. “MEG and EEG Data Analysis with MNE-Python”. In: *Frontiers in Neuroscience* 7.267 (2013), pp. 1–13 (cit. on p. 25).

[HJ17] S.-H. Hsu, T.-P. Jung. “Monitoring alert and drowsy states by modeling EEG source nonstationarity”. In: *Journal of neural engineering* 14.5 (2017), p. 056012 (cit. on p. 17).

[HPP+18] S.-H. Hsu, L. Pion-Tonachini, J. Palmer, M. Miyakoshi, S. Makeig, T.-P. Jung. “Modeling brain dynamic state changes with adaptive mixture independent component analysis”. In: *NeuroImage* 183 (2018), pp. 47–61 (cit. on pp. 11, 17, 18).

[LCG10] D. Langlois, S. Chartier, D. Gosselin. “An introduction to independent component analysis: InfoMax and FastICA algorithms”. In: *Tutorials in Quantitative Methods for Psychology* 6.1 (2010), pp. 31–38 (cit. on pp. 13, 14).

[NK11] G. R. Naik, D. K. Kumar. “An overview of independent component analysis and its applications”. In: *Informatica* 35.1 (2011) (cit. on pp. 14, 15).

[PKM06] J. A. Palmer, K. Kreutz-Delgado, S. Makeig. “Super-Gaussian mixture source model for ICA”. In: *International Conference on Independent Component Analysis and Signal Separation*. Springer. 2006, pp. 854–861 (cit. on p. 17).

[PMKR08] J. A. Palmer, S. Makeig, K. Kreutz-Delgado, B. D. Rao. “Newton method for the ICA mixture model”. In: *2008 IEEE International Conference on acoustics, speech and signal processing*. IEEE. 2008, pp. 1805–1808 (cit. on p. 20).

[SLB05] L. Sun, Y. Liu, P. J. Beadle. “Independent component analysis of EEG signals”. In: *Proceedings of 2005 IEEE International Workshop on VLSI Design and Video Technology, 2005*. IEEE. 2005, pp. 219–222 (cit. on p. 15).

[Tha21] A. Tharwat. “Independent component analysis: An introduction”. In: *Applied Computing and Informatics* 17.2 (2021), pp. 222–249 (cit. on p. 13).

[VDTS10] F. C. Viola, S. Debener, J. Thorne, T. R. Schneider. “Using ICA for the analysis of multi-channel EEG data”. In: *Simultaneous EEG and fMRI: Recording, Analysis, and Application: Recording, Analysis, and Application* (2010), pp. 121–133 (cit. on p. 16).

All links were last followed on September 20, 2023.

Declaration