A Purely Fixed-Point Implementation of the FastICA Algorithm

Nikola Rank
March 31, 2006
ELG6163 Project report
Presented to Dr. M. Bolic

1. Background on FastICA

The Fast ICA algorithm was introduced in 1997 by Aapo Hyvarinen and Erkki Oja, from the Helsinki University of Technology. The basic concept is to take a neural network leaning rule and convert it into a fixed-point iteration. This yields "an algorithm that is very simple, does not depend on any user-defined parameters, and is fast to converge to the most accurate solution allowed by the data."[1]

Two important applications of ICA are blind source separation, and feature extraction. This includes some very focused, interesting applications such as analysis of fMRI data, and fetal heart monitoring. ICA is still in its infancy and its applications are still growing.

This particular algorithm can be used in batch mode (processing all the data at once) or in a semi-adaptive manner, working with subsets of the data at a time. In one experiment comparing it to a neural network algorithm, FastICA required 10% of the floating point operations used by the neural algorithm.[2] In the same paper, the convergence of the FastICA algorithm is proven to be cubic, much faster than other similar algorithms. Another important aspect is that FastICA can be used to only extract desired components, instead of having to extract them all at once(though this requires proper initialization of the unmixing matrix).

With these desirable features, FastICA is a good candidate for porting to a fixed-point implementation. This report will first focus on the original floating-point method, and provide a set of tests/results for it. Then a modified version of the algorithm for fixed-point computation will be tested.

2. Floating point FastICA

Operational Summary

The FastICA package comes with a floating point implementation of the FastICA algorithm. The algorithm can be broken down into several software functional units, as depicted in figure 2.0.

[1] Reference 4, Page 2
[2] Reference 4, Page 8

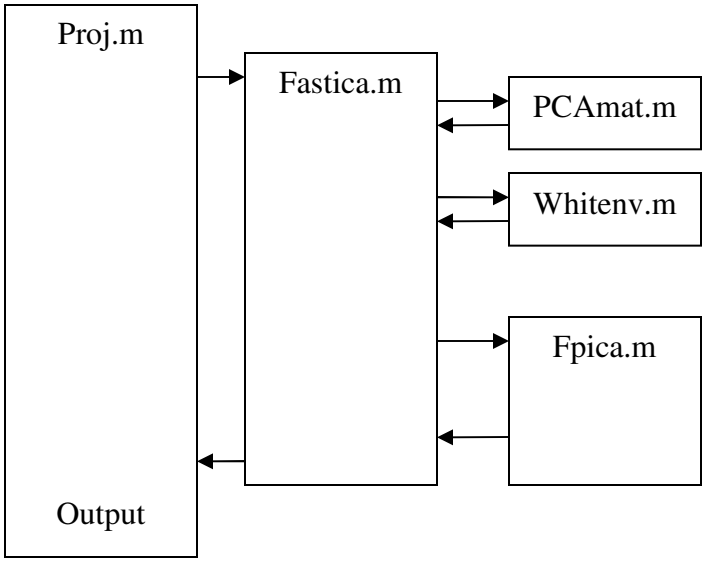


Figure 2.0: Software functional units of Floating point FastICA

The module Proj is a driver module that had to be created to test the algorithm. It is responsible for reading the input files, creating artificial mixtures, calling the ICA algorithm, and computing accuracy of results. It also provides functionality for running experiments over a range of input mixtures.

The module Fastica is the top-level ICA function (note that there is also a fastICAg a GUI version of this). It is responsible for accepting/setting parameters, mean extraction/retoration, calling the PCA, whitening, and ICA functions described below. The purpose of mean extraction is to simplify the ICA algorithm. If the mean of the mixtures is zero, then the mean of the source signals is zero as well (and the mean can be added back later after ICA is computed)[3].

The module PCAmat is responsible for computing the PCA(Principal component analysis) of the data. This technique was added to the package because it is a common practice to reduce the dimension of the input data before doing ICA. In the test cases implemented here, it does not do any reduction. However, some of its output parameters are used in the next module.

The module Whitenv whitens the input data. This is another preprocessing technique that simplifies the ICA algorithm. It is a linear transform on the vector of input mixtures(X) that yields a whitened input vector (X') whose components are uncorrelated and have variance equal to unity. This is tested in the algorithm by computing E[X * X'] = I (identity matrix). This whitening operation causes the mixing matrix to change, and implies it will be orthogonal. The gains here come from the fact that an orthogonal

[3] Reference 2, Page 12

matrix has only N(N-1)/2 degrees of freedom, while a general 2D matrix has N^2 degrees of freedom.[4]

The core module that is actually doing the ICA processing is called fpica (fixed-point ICA, even though it is using floating-point numbers). This module provides two main approaches, Deflation and Symmetric. For the purpose of implementation, the Deflation approach was chosen. For each approach, there are quite a few parameters that can be set, such as which nonlinearity to use, how many input sample to observe at a time, fine tuning of the output once the IC (independent components) have been estimated. For this implementation, the cube nonlinearity (x^3) was chosen for its flexibility of implementation (either by multipliers, or look-up table). There are other nonlinearities available such as hyperbolic tangent (tanh) which were tested and functional, but whose results were not considered for this report.

The algorithm in deflation mode works by estimating one IC at a time. Within each IC estimation sequence, there are multiple iterations as well. After each iteration, the results (unmixing vector w) are projected onto the previously calculated unmixing vectors, and are decorrelated from them using a Gram-Schmidt-like procedure. The equations for this are shown below(from the reference), with w_{p+1} being the unmixing vector being estimated, and w_j being the previously calculated unmixing vector belonging to IC *j*.[5]

1. Let $\mathbf{w}_{p+1} = \mathbf{w}_{p+1} - \sum_{j=1}^{p} \mathbf{w}_{p+1}^T \mathbf{w}_j \mathbf{w}_j$
2. Let $\mathbf{w}_{p+1} = \mathbf{w}_{p+1} / \sqrt{\mathbf{w}_{p+1}^T \mathbf{w}_{p+1}}$

Within the estimation sequence of a single IC, the following steps occur (taken from reference).

1. Choose an initial (e.g. random) weight vector $\mathbf{w}$.
2. Let $\mathbf{w}^+ = E\{\mathbf{x}g(\mathbf{w}^T\mathbf{x})\} - E\{g'(\mathbf{w}^T\mathbf{x})\}\mathbf{w}$
3. Let $\mathbf{w} = \mathbf{w}^+ / \|\mathbf{w}^+\|$
4. If not converged, go back to 2.

Convergence is tested by comparing the direction of the old w estimate to the current one. When the direction match to a certain (specifiable) tolerance, then convergence is achieved and iterations can stop for this IC.[6] In this formula, the function g is the nonlinearity, and it is chosen to be the cube function. The flowchart below, figure 2.1, illustrates the whole process.

---

[4] Reference 2, Page 12-13
[5] Reference 2, Page 15
[6] Reference 2, Page 14

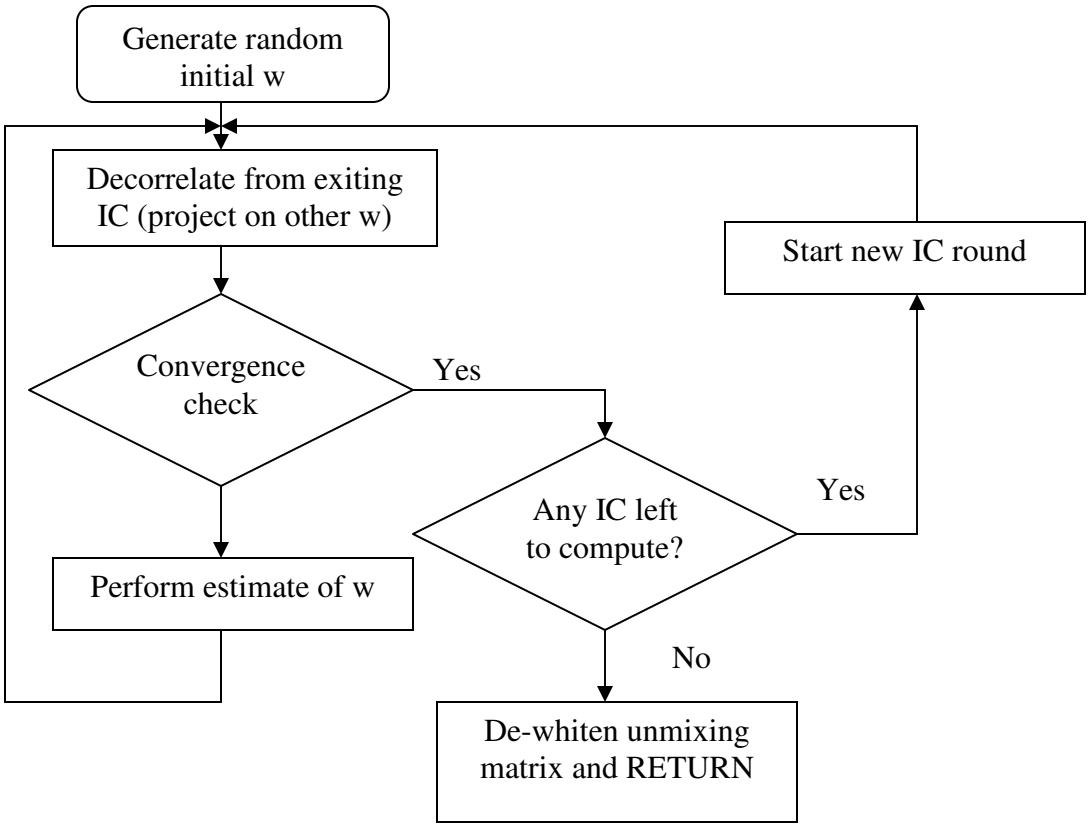


Figure 2.1: Operation of fpica (Floating point ICA algorithm)

Performance measures

In order to test the algorithm, some performance tests had to be established. The driver module is able to generate signal mixtures given the mixture ratios, and some input source signals. The input source signal length was set to 64K samples (65536) in order to provide a compromise between computation time, and sufficient data to extract components. The method used to generate the signal mixtures is illustrated below.

1. Read in two 64K sample vectors s1, s2
2. Generate a mixing matrix A, with the following values:

(Mixture ratio 1)	1-(Mixture ratio 1)
(Mixture ratio 2)	1- (Mixture ratio 2)

3. Generate the signal mixtures x by multiplying A with the source signals s1,s2
4. Feed the input vector x into the ICA algorithm

By using a known mixing matrix, and known source signals, it is possible to compare these with the estimates generated by ICA, allowing one to measure the quality.

The first experiment was to test how close the mixtures could be while still getting convergence of the algorithm. Results show that even with very similar mixtures, the algorithm is able to separate the IC's

A = 0.4950 0.5050
0.5000 0.5000

However, this requires an unmixing matrix with very high values (approaching singularity) W_est = 1.0e+003 *
1.5691 -1.5853
0.8061 -0.7984

The errors associated with the extracted signals are still comparable to ones where the mixing ratios were farther apart:

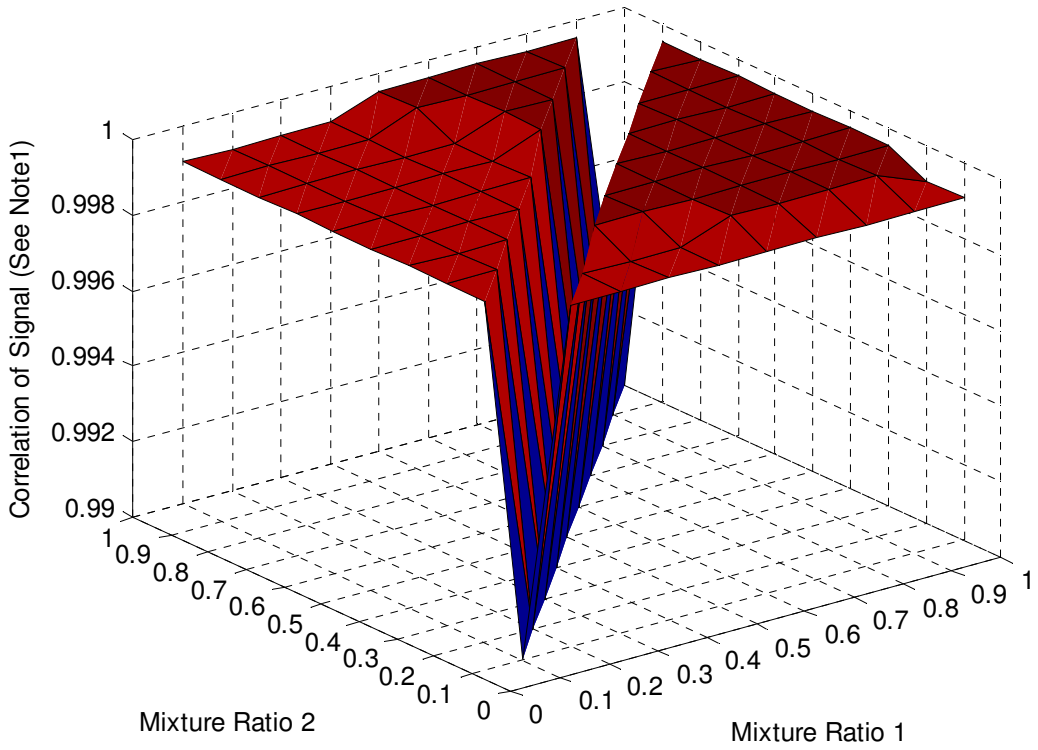
E1 = 0.9995 E2 = 0.9999

Experiment 1

These results prompted interest in testing the algorithm over a wide range of mixture ratios, yielding the following experiment. Mixture ratios 1 and 2 were tested for all values between 0 and 1, with increments of 0.1. This yielded 100 different data points, which are illustrated in the following graphical form.

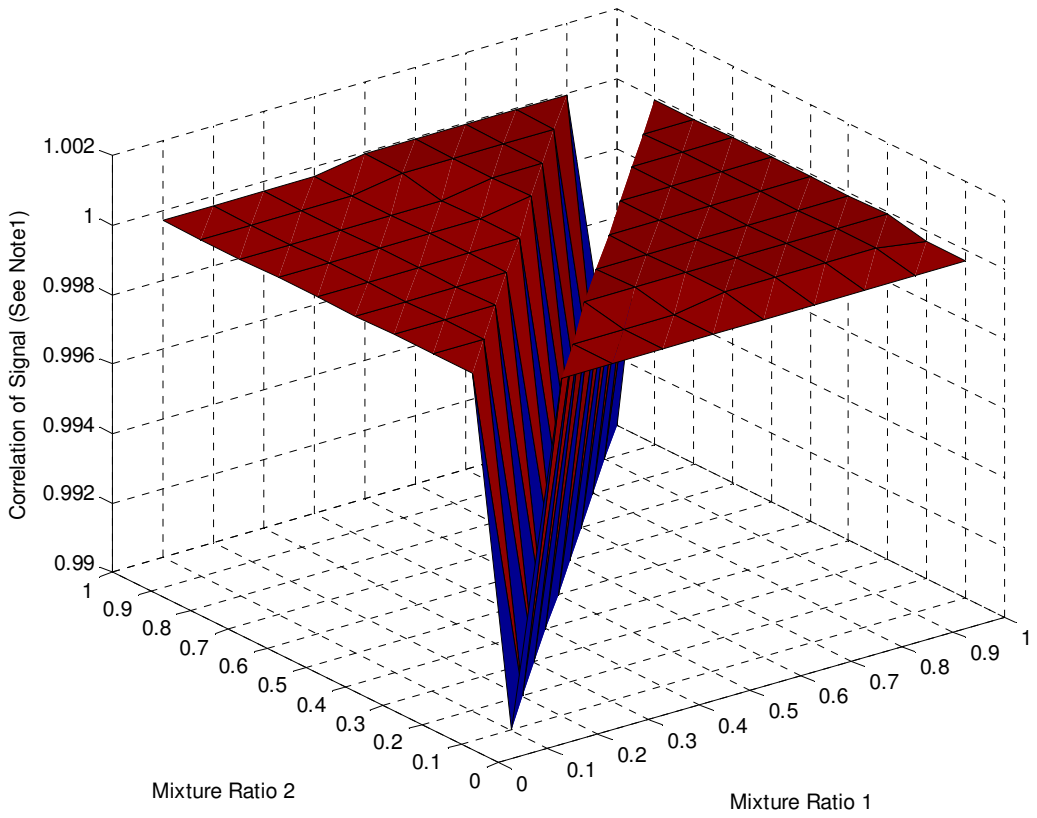
In this experiment, the source signals consisted of 2 different voices, and two linear mixtures were generated from these. As described, both mixture ratios were varied between 0 and 1. The results are given in graphical form below.

Effect of Mixture Ratio on Estimated Source Signal 1 Quality



Note 1: The estimated source signal was correlated (with lag 0) to the input source signal, and then normalized. Higher correlation values indicated a better quality signal(more closely matching the original). The valley in the middle of the graph has correlation values around 0, not 0.99. The values were padded to 0.99 to make the graph more visually clear.

Effect of Mixture Ratio on Estimated Source Signal 2 Quality



Note 1: The estimated source signal was correlated (with lag 0) to the input source signal, and then normalized. Higher correlation values indicated a better quality signal(more closely matching the original). The valley in the middle of the graph has correlation values around 0, not 0.99. The values were padded to 0.99 to make the graph more visually clear.

From this experiment, one can conclude that the algorithm is quite capable of separating mixtures with varying ratios of input signals. Performance of the algorithm is similar in all combinations. The exception occurs when the mixture ratios of both input mixtures are equal (basically giving the same mixture twice). Obviously, ICA is not able to find two independent components, and the correlation is assigned a value of 0.

Another interesting outcome from this experiment is the convergence speed of the algorithm. It usually takes between 5-13 steps for IC#1 to be extracted, while IC#2 always takes 2 steps. Since the initial guess is based on a random number generator, the seed was kept the same for all experiments in this report. This was an effort to compare convergence speed of various algorithms. However, in the experiment where the mixture ratios are varied, the convergence speed depends completely on how close the true unmixing matrix is to the initial guess. Since the true unmixing matrix depends

completely on the mixture ratios, then it is not meaningful to compare the speeds for the various mixture ratio combinations.

3. Fixed point FastICA

Operational summary

In order to obtain something that could be implemented in hardware, or mixed hardware-software systems, it is necessary to have an algorithm that can work with purely fixed-point numbers. Converting a complex algorithm such as ICA into pure fixed point proved quite challenging, since fixed point numbers may not only degrade output quality, but also prevent(or slow down) convergence. For this reason, several parts of the algorithm had to be changed, and these are outlined below.

The algorithm was developed in MATLAB, using the fixed point toolbox (whose basic numerical representation is the fi object). The first step was to quantize the input data. This was done in the proj module, with the following line (generate 16 bit numbers with optimal fraction length):

```
x = fi(x,1,16);
```

Various other parts of the proj module (such as error calculation) had to contain conversions from fi to double since the MATLAB functions used did not allow many operations on fi objects. However, both input and output to the fastica2 was done with fixed point numbers.

The module fastica2 was renamed from fastica, in order to enhance it for fixed point operation. In this module, the only changes made were to the mean extraction/replacement. An algorithm remmean2 was developed that would model a fixed-point method for averaging numbers. The mean restoration after the ICA algorithm was also modeled in fixed point. It was decided not to model the PCA and whitening modules in fixed point, as neither are necessary to ICA. An analysis of the gains obtained by the whitening of the data would also have to be performed to determine if it is worthwhile for a fixed point system to compute it. Instead, the data was converted to double for input to both whitening and PCA, and the output whitened data was converted to the same 16 bit numbers as above before input to the ICA algorithm.

The module fpica2 is where the majority of changes occurred. The Decorrelation of ICs was changed to fixed point. This involved several multiplications and additions of arrays of numbers. This required a normalization operation of a 2 variable vector. Typically normalization requires taking a square root, and a division: both operations are costly in hardware. However, there exist methods for fast square root algorithms of integers, which could be applied to these fixed point numbers. The division could also be implemented by an embedded CPU which would be running the system (for instance, NIOS II hardware divide). In order to keep MATLAB simulations speeds reasonable, the

normalization operation was simulated in floating point and then the outputs converted to fixed point format. Since the Decorrelation occurs only once per iteration, is may be acceptable to use a slower implementation. Up to this point, it is reasonable to think that everything could be implemented in software on an embedded CPU.

The next major change to fpica2 was in the estimation of the new w values. This required a complex series of array operations, and accessing a nonlinear function (cube). These operations are the bottleneck of the algorithm, and would gain significantly from implementation in pure hardware. For this reason, this section of the algorithm is modeled purely in fixed point. Considerations had to be made for overflow, truncation, averaging of long sequences, and the nonlinear operation. Overflow bits were added in areas where long summations (over the 64K samples) were done. The input and output to this whole sequence was kept at a standard 32 bit format to model interfacing to a 32 bit data bus. The key difficulty was modeling the nonlinearity, since it is called upon 2 * 64K times during each iteration. It was modeled in MATLAB as 2 fixed point multiplications by the same number (keeping data width constant). In a real implementation, it is likely that a look-up table would be used (which would allow flexibility to implement other nonlinearities like tanh).

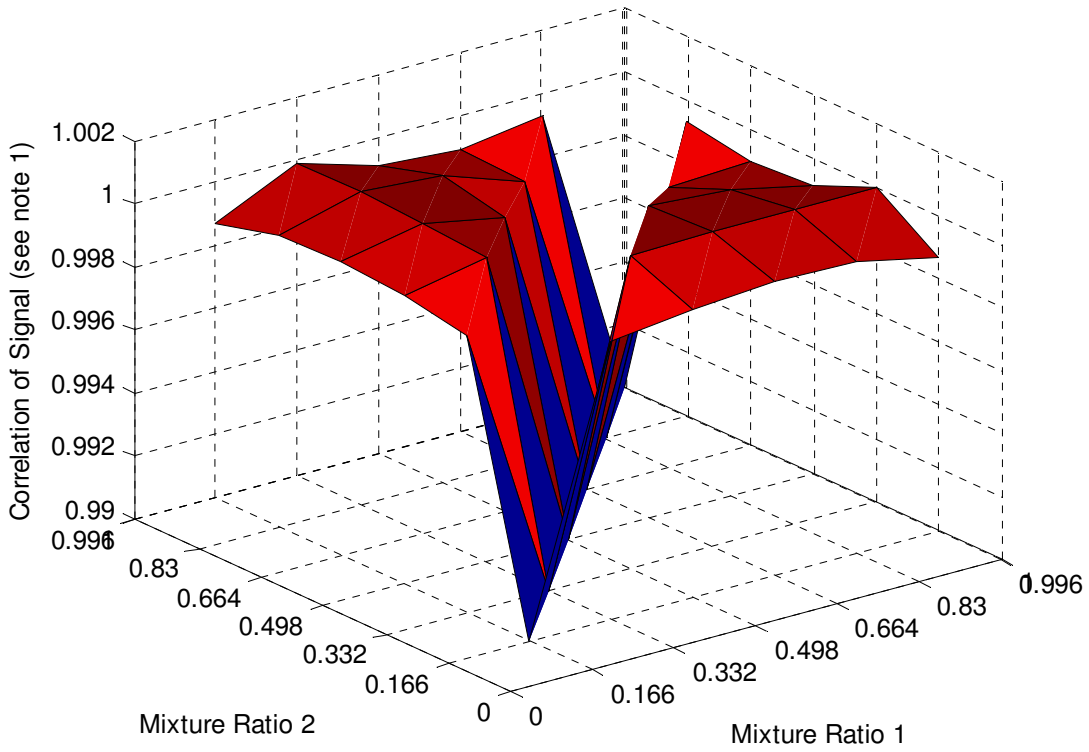
Performance measures

Experiment 2

As with the floating point algorithm, tests were done over a wide range of mixture ratios. The difference is that due to increased MATLAB overhead in handling fi objects, the increments were increased from 1/10 to 1/6, yielding 36 data points. Mixture ratios 1 and 2 were tested for values between 0 and 1, with increments of 0.166. The results are illustrated below in graphical form.

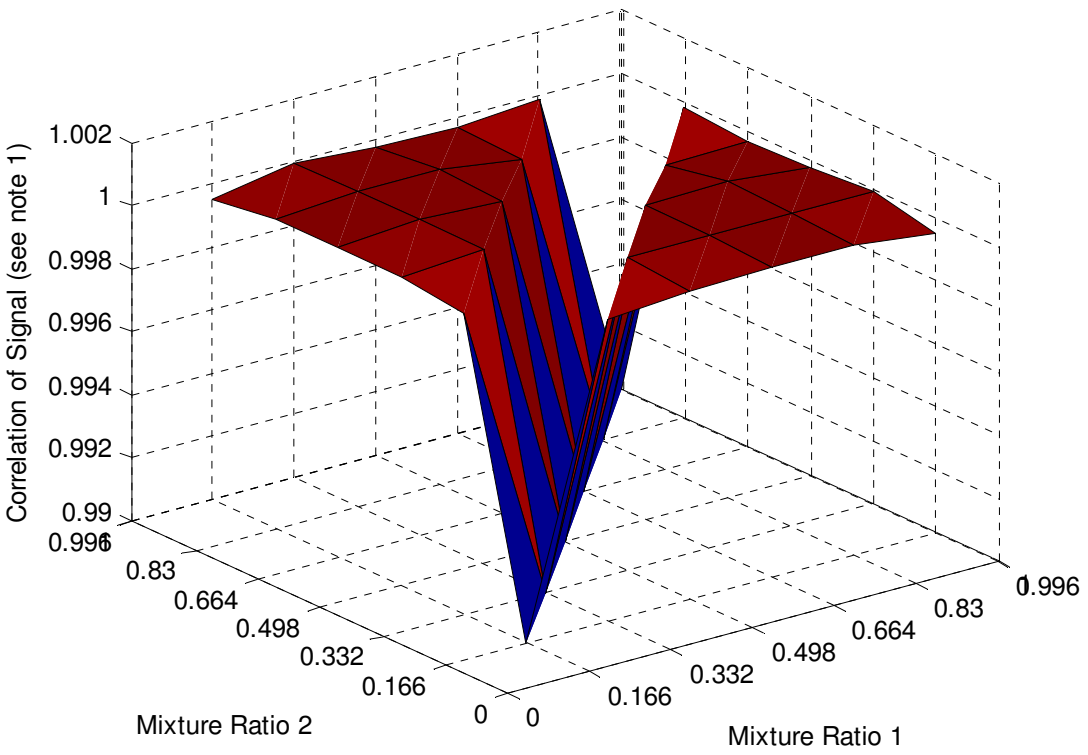
In this experiment, the source signals consisted of 2 different voices, and two linear mixtures were generated from these. As described, both mixture ratios were varied between 0 and 1. The results are given in graphical form below.

Effect of Mixture Ratio on Estimated Source Signal 1 Quality



Note 1: The estimated source signal was correlated (with lag 0) to the input source signal, and then normalized. Higher correlation values indicated a better quality signal(more closely matching the original). The valley in the middle of the graph has correlation values around 0, not 0.99. The values were padded to 0.99 to make the graph more visually clear.

Effect of Mixture Ratio on Estimated Source Signal 2 Quality



(see previous note 1)

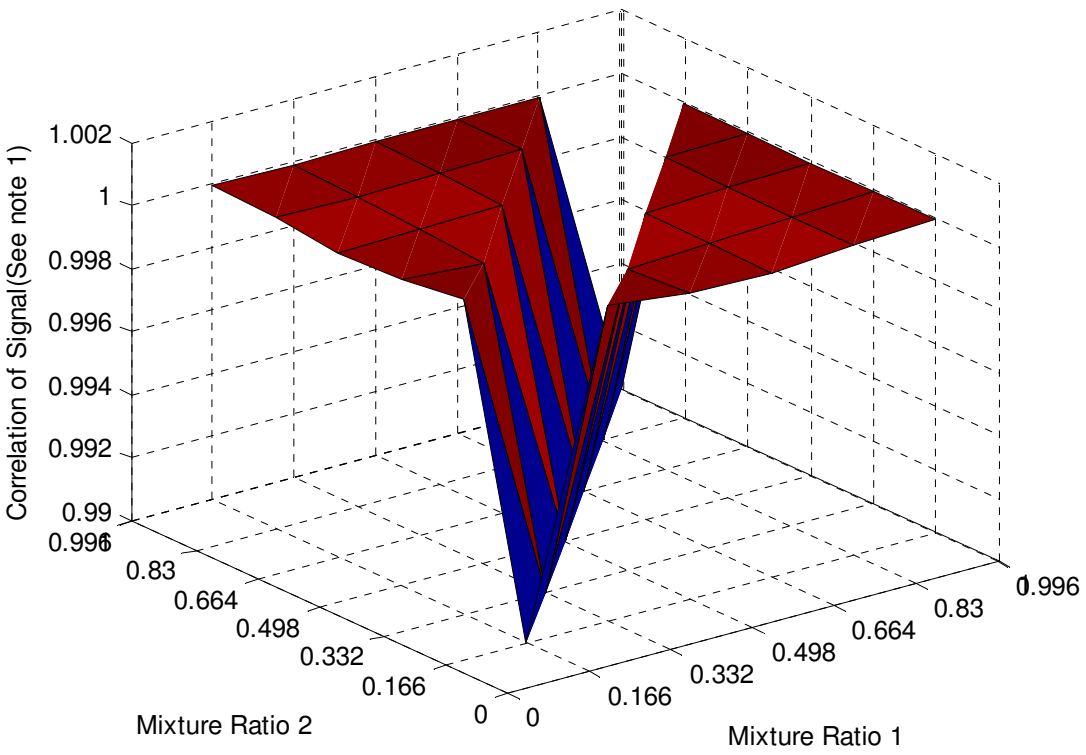
As can be observed from the data, the fixed-point implementation of the algorithm is equally capable of functioning over a variety of mixture ratios, with no significant loss in quality compared to the floating point. These simulations results are quite encouraging for the goal of a pure fixed-point ICA implementation.

Although the simulations took much longer for the fixed-point numbers(due to MATLAB overhead), the number of iteration steps was similar to the floating point numbers. It typically took between 5-15 steps to extract IC #1, and always 2 steps to extract IC #2. This indicates that a very fast fixed-point implementation is possible.

Experiment 2

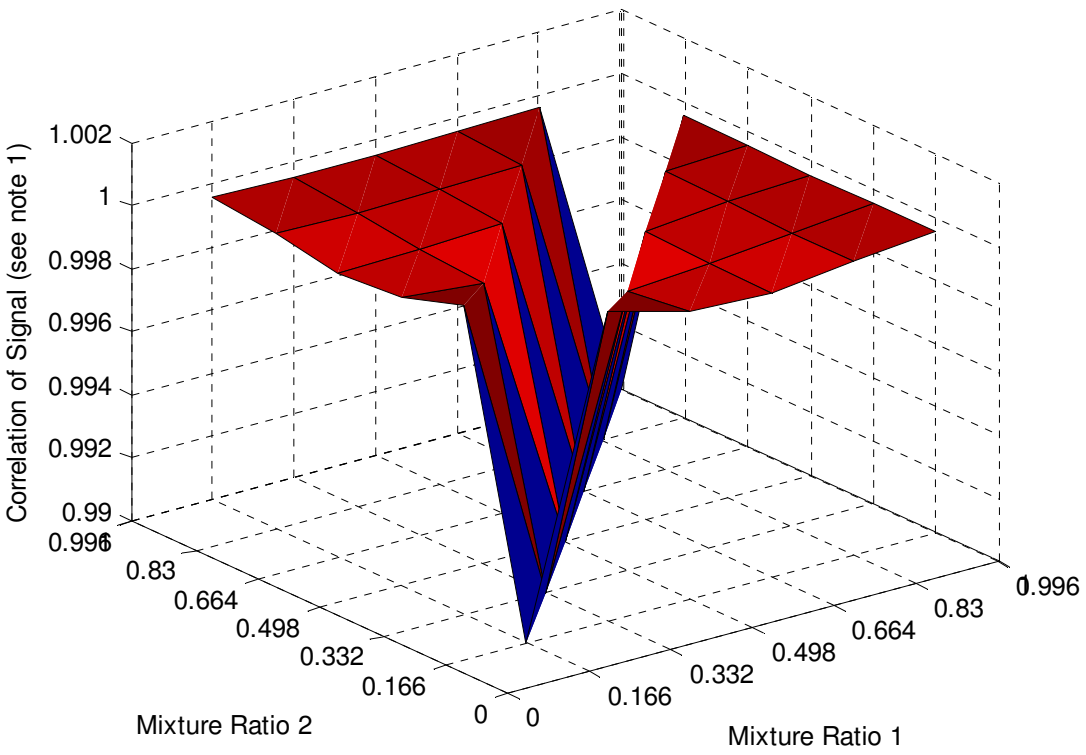
This experiment is identical to the previous one, except that the input data used was different. Instead of using two different voice samples, the same voice sample was used, but the (64K sample) source signals were taken at different (non-overlapping) indexes into the file.

Effect of Mixture Ratio on Estimated Source Signal 1 Quality



(see previous note 1)

Effect of Mixture Ratio on Estimated Source Signal 2 Quality



(see previous note 1)

One can see a performance similar to that in the previous experiment, with convergence times for IC#1 at 4-13 steps, and IC2 at 2 steps. The purpose of this experiment was to test robustness of technique without modifying it to a different set of input data (other undocumented tests were also performed for this purpose).

Experiment 3

The freedom with a fixed point implementation to vary bit widths of various components provides interesting experimental potential. In this case, the input bit width (to the whole system) was varied, simulating varying precision ADCs used. The results are given below for each bit width.

16bit(reference value for other comparisons)
IC 1 .....computed (6 steps)
IC 2 ..computed (2 steps)

E1 = 0.9996 E2 = 0.9989

8 bit
IC 1 .....computed (6 steps)
IC 2 ..computed (2 steps)

E1 = 0.9990 E2 = 0.9975

Once can notice a very slight degradation of the output signals (shown by the decrease in correlation).

4 bit
IC 1 .....computed (6 steps)
IC 2 ..computed (2 steps)

E1 = 0.9244 E2 = 0.7782

At this point the input mixture signals are quite noisy and despite the low correlation values (especially for the second IC) the audio output of the source signals is still understandable.

An important point of note is that the algorithm is still able to separate the ICs in the same amount of steps, regardless of the input data width. This is important for a robust algorithm.

Experiment 4

Reducing various bit widths within the section of code concerned with updating w is of special interest, since this is proposed to be implemented in hardware, and speed is crucial in this section.

Eliminating overflow bits in summation/averaging section of w update(65bits -> 49 bits accumulator/adder)

IC 1 .....computed (6 steps)
IC 2 ..computed (2 steps)

E1 = 0.9996 E2 = 0.9989

This did not make a difference compared to the reference values of experiment 3 (16 bit), suggesting that for this batch of input data, overflow was not an issue.

Reducing multiplier output width before averaging section of w update(49/26 bit to 32/26 bit)

Here the fractional length was kept constant at 26 bits, and the word length decreased to 32 bits. This makes the algorithm not converge within a reasonable time.

On the other hand, if the fraction length is changed to 15 bits with a 32 bit word length, the algorithm functions as well as the reference. This suggests that the integer part of the product here is growing beyond what a 32/26 number can accommodate.

Reducing cube operation(nonlinearity) output from 49/26 to 32/26
This experiment was of special interest since the nonlinearity's speed and precision are crucial to the operation of the algorithm.

IC 1 .....computed (6 steps)
IC 2 ..computed (2 steps)

E1 = 0.9966 E2 = 0.9954

No degradation in convergence time was observed, but a noticeable degradation in quality is present.

This can be remedied by going to a 15 bit fraction length with a 32 bit word:
IC 1 .....computed (6 steps)
IC 2 ..computed (2 steps)

E1 = 0.9996 E2 = 0.9989

Thus, quality is restored to the reference values, suggesting again that the integer portion is being overflowed.

It is even possible to go to a look-up table-friendly output of 12 bit word 4 bit fraction and still get convergence, but a loss of quality.

IC 1 .....computed (6 steps)
IC 2 ..computed (2 steps)

E1 = 0.9986 E2 = 0.9976

Certainly whatever option is chosen for the nonlinearity implementation will be easily testable. A fast nonlinearity is required to make this work, since unlike the normalization operator (once per iteration), the nonlinearity is required c * N times during each iteration (where c = number of mixtures, and N = number of samples).

These experiments show promise in the development of a fast, fixed point algorithm which uses reasonable hardware and software resources.

4. General Conclusions

1. This report proves that it is possible to perform ICA in a purely fixed point environment, without loss of quality or convergence.

2. Although the purely fixed-point version of the FastICA algorithm does not have a fixed execution time (no upper bound to iterations), in all tested cases the algorithm converged within a reasonable amount of time.

3. The results from Experiment 4 indicate that there is potential for area/complexity savings (from reduced word length) in some areas of the update algorithm. Before this could be implemented in hardware, further investigation of these savings is necessary.

4. In this implementation, the whitening transform is performed in floating point. Though this transform is not a necessary part of FastICA, it may be an effective means of reducing the overall cost of the ICA operation. This merits further investigation as well.

5. Possible implementation

Though the focus of this project was not on implementation, it has laid the groundwork and some proofs of concepts necessary to show that FastICA could be performed in a fixed-point system. The next logical step is to consider various implementations of the algorithm. A system combining a flexible embedded processor with a fast hardware block capable of performing the estimation of individual ICs (the iterations in the algorithm) would be a reasonable candidate for this.

References

1.Hyvärinen, A. and Bingham, E.. (2000). A fast fixed-point algorithm for independent component analysis of complex valued signals. *Helsinki University of Technology. http://www.cis.hut.fi/projects/ica*

2.Hyvärinen, A. and Oja, E. (2000). Independent Component Analysis: Algorithms and Applications. *Neural Networks, 13(4-5):411-430, 2000*

3.Hyvärinen,A. (1999a). Fast and robust fixed-point algorithms for independent component analysis. *IEEE Transactions on Neural Networks*, 10(3):626–634.

4.Hyvärinen, A. and Oja, E. (1997). A fast fixed-point algorithm for independent component analysis. *Neural*
Computation, 9(7):1483–1492.

FastICA package available: http://www.cis.hut.fi/projects/ica/fastica/