An Introduction to ICA and ICA Applications

By: Chris Bodden and Svyat Vergun

ECE 532 – Fall 2014

CONTENTS

INTRODUCTION

Independent component analysis (ICA) is a technique to decompose an observed signal, made up of several unknown source signals, into the original source signals or subcomponents. ICA requires that the subcomponents be statistically independent from one another (non-Gaussian).

In this lab we apply ICA towards the cocktail party problem, which involves separating individual speech measured in a room with many speakers. These signals are taken to be independent. More generally this problem is called blind source separation because we don't know the original source signals, only some mixing of the sources.

For parts of this lab we use the popular FastICA algorithm package that is freely available from: http://research.ics.aalto.fi/ica/fastica/.

In-Depth ICA Reading:
http://www.stat.ucla.edu/~yuille/courses/Stat161-261-Spring14/HyvO00-icatut.pdf

ICA OVERVIEW

Assume that we observe n different linear mixtures $x_1, \dots, x_n$ of the same n original independent source signals. For example, n original audio sources have been combined into n mixtures of the original source. This can be expressed as:

$$x_j = a_{j1}s_1 + a_{j2}s_2 + \cdots a_{jn}s_n \text{ , for } j = 1 \dots n \text{ .}$$

This can also be expressed in Matrix notation. Let $\boldsymbol{x}$ be the vector whose elements are the mixed signals $x_1, \dots, x_n$, and let $\boldsymbol{s}$ be the vector whose elements are the original source signals $s_1, \dots, s_n$. Let $\boldsymbol{A}$ be the square mixing matrix with elements a_{ij} for $i, j = 1 \dots n$. By convention, bold lower case letters indicate vectors and bold upper case letters indicate matrices. Using this notation, the mixing model above can be written as

$$\boldsymbol{x} = \boldsymbol{A}\boldsymbol{s}.$$

In other words, the mixed signal, $\boldsymbol{x} \in \mathbb{R}^n$, where n is the number of sources is some mixing of the original source signals, $\boldsymbol{s} \in \mathbb{R}^n$, produced by combining the original sources using the mixing matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$. Note that if $\boldsymbol{x}$ and $\boldsymbol{s}$ are sampled with some sample number p, then the equation becomes $\boldsymbol{X} = \boldsymbol{A}\boldsymbol{S}$ with all terms being matrices ($\boldsymbol{X} \in \mathbb{R}^{n \times p}$, $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, $\boldsymbol{S} \in \mathbb{R}^{n \times p}$).

Denoting the columns of $\boldsymbol{A}$ as $\boldsymbol{a_j}$, the model above can be expressed as a summation:

$$\boldsymbol{x} = \sum_{i=1}^{n} \boldsymbol{a_i s_i}$$

The model given by $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{s}$ is called the independent component analysis (ICA) model. This is called a *generative model*, which means the observed signals are generated by mixing the original components (sources) $\boldsymbol{s_i}$. These independent components, $\boldsymbol{s_i}$, are not directly observed and the mixing matrix $\boldsymbol{A}$ is also assumed to be unknown. We only observe the mixed signal, $\boldsymbol{x}$, and estimate both $\boldsymbol{A}$ and $\boldsymbol{s}$ using the model.

The major assumption of the ICA model is that the components $\boldsymbol{s_i}$ are statistically independent, or simply have non-Gaussian distributions. Here we also assume that the mixing matrix is square, but this need not be the case in other problems. We can compute the inverse of the mixing matrix, $\boldsymbol{A^{-1}} = \boldsymbol{W}$, and then compute the independent components by:

$$\boldsymbol{s} = \boldsymbol{W}\boldsymbol{x}.$$

In the Introduction *blind source separation (BSS)* was mentioned. The problem of BSS is essentially identical to ICA. In BSS, a *source* is one of the original signals (independent components), such as a single person in the cocktail party problem. The problem is called *blind* because we do not have information about the mixing matrix, meaning we don't know exactly how the individual signals have been mixed.

There are ambiguities in the ICA model. A source can be recovered, but its sign is not certain. Additionally, the order of components cannot be determined. For many problems, including BSS, these are not important.

For ICA there are two unknowns (the mixing matrix, $\boldsymbol{A}$, and the original sources, $\boldsymbol{s}$) and only one known (the mixed signal, $\boldsymbol{x}$). We can solve for $\boldsymbol{A}$ and $\boldsymbol{s}$ by utilizing the assumption that the sources are statistically independent. This is an optimization problem that minimizes the measure of mutual information between the sources (components). Mutual information is defined as:

$$\sum_{c \in C} \sum_{d \in D} p(c,d) \log\left(\frac{p(c,d)}{p(c)p(d)}\right)$$

Where $p(c,d)$ is the joint probability of variables c and d, and $p(c)$ and $p(d)$ is the probability of c and d respectively. The sum is taken over the values of the possible c and d variables.

The FastICA algorithm implements this optimization and outputs the mixing matrix as well as the components.

1) WARM-UP

Important: ICA implementations are very tedious and can be very inefficient if done improperly. For several portions of this lab we will use the FastICA package from the Laboratory of Information and Computer Science at the Helsinki University of Technology. The package is available at: http://research.ics.aalto.fi/ica/fastica/. Please download and extract the FastICA package into your lab folder. Make sure the FastICA '.m' files are in the same directory as your lab scripts.

We will now look at 2 simple examples of ICA, one with components that are non-Gaussian and another with Gaussian components.

1.1) ICA of 2 non-Gaussian signals

First we will define a mixing matrix. To keep things simple we will make our mixing matrix square. In MATLAB, assign the following mixing matrix to A:

```
A = [1.4142, 0.7071; -0.7071, 1.4142]; %mixing matrix
```

Now create two independent signals, a sine wave and a sawtooth over the range 0 to 2π:

```
N = 2*pi;
x = 0:0.1:N;
sig = nan(2,size(x,2));
sig(1,:)=sin(x); %sinusoid
sig(2,:)=(rem(x,0.25)); %saw-tooth
```

Now mix the signals using our mixing matrix:

```
mixed = A * sig;
```

Finally, use the FastICA algorithm on the mixed signals to attempt to recover the original source signals and output the estimated mixing matrix:

```
[ICA_sig, A_est, W_est] = fastica(mixed,'verbose','off');
A_est
```

(Note: the option `'verbose','off'` simply suppresses status messages. If you'd like to see these, just use `fastica(mixed)`)

You can plot the original, mixed, and estimated signals using the subplot command:

```
% plot original
subplot(3,2,1)
plot(x,sig(1,:),'r');
title('Signal 1 Original');
axis tight
```

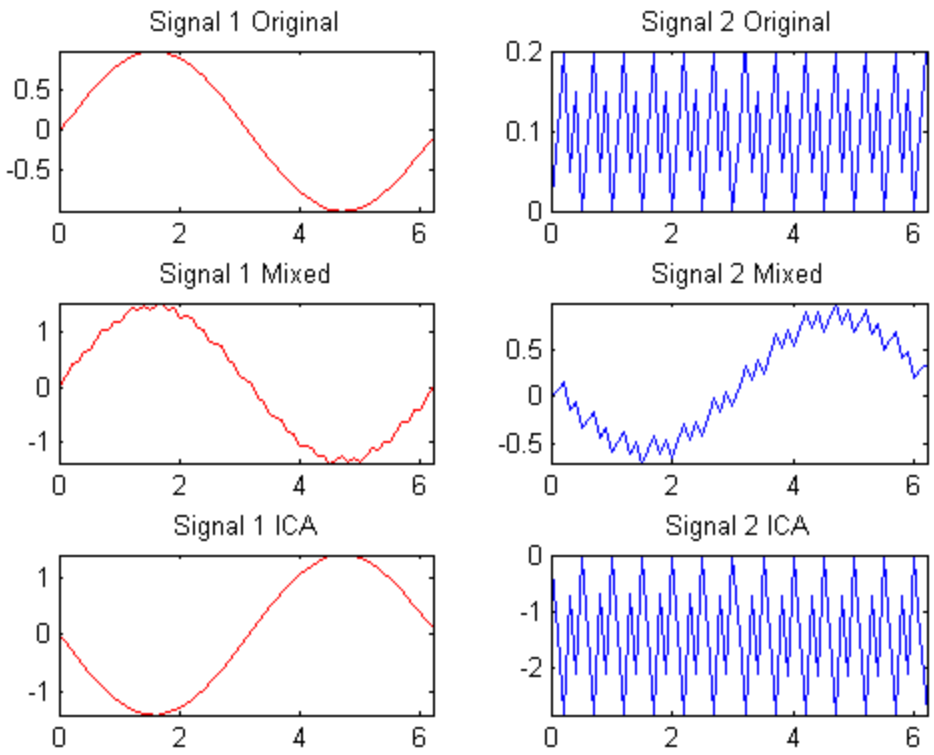
```
subplot(3,2,2)
plot(x,sig(2,:),'b');
title('Signal 2 Original');
axis tight
% plot mixed
subplot(3,2,3)
plot(x,mixed(1,:),'r');
title('Signal 1 Mixed');
axis tight
subplot(3,2,4)
plot(x,mixed(2,:),'b');
title('Signal 2 Mixed');
axis tight
% plot ICA
subplot(3,2,5)
plot(x,ICA_sig(1,:),'r');
title('Signal 1 ICA');
axis tight
subplot(3,2,6)
plot(x,ICA_sig(2,:),'b');
title('Signal 2 ICA');
axis tight
```

QUESTIONS

1. Are the recovered signals the same as the original signals? Remember that ICA has the limitations that it cannot reproduce the order of the signals and that the estimated signals may be off by a scaling factor including the sign.
2. How about the estimated mixing matrix, is it similar to the original?
3. Save your resulting plots for your report.

SOLUTION

The original signals should be recovered and the estimated $\boldsymbol{A}$ matix should be close to the original. The order and sign of the plots and columns of $\boldsymbol{A}$ may be off, however. The plots should look similar to:



1.2) ICA of 2 Gaussian signals

Now let's replace the sine wave and sawtooth with 2 random signals that have a Gaussian distribution.

(***Note***: the MATLAB function, `randn()`,produces vectors of randomly generated numbers that have a Gaussian (or normal) distribution.)

Rerun your code from above, but replace the following 2 lines:

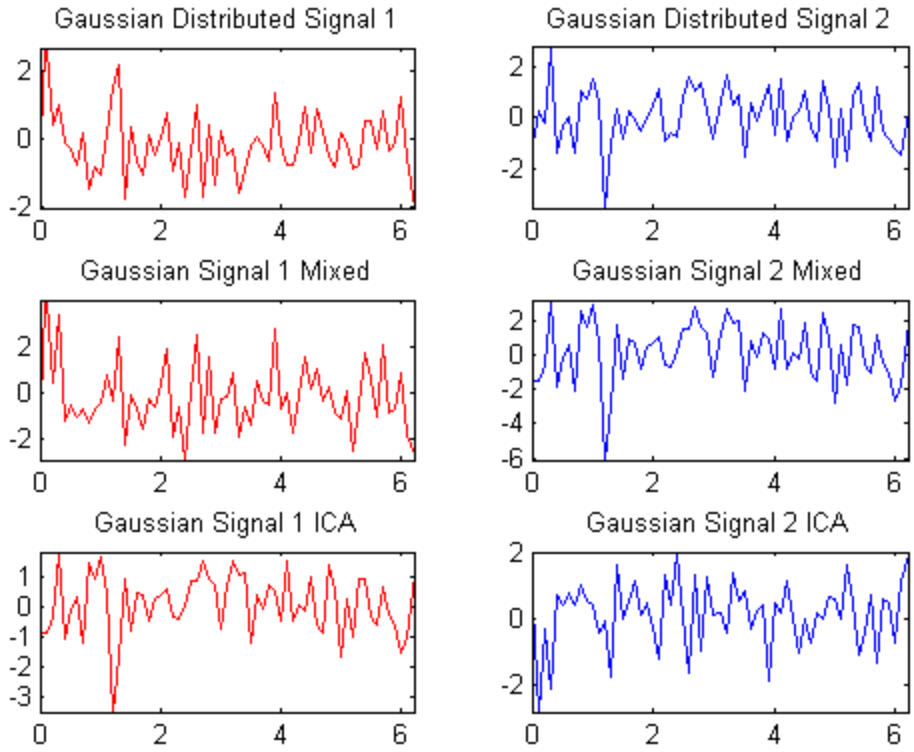
```
sig(1,:) = randn(1,size(x,2));
sig(2,:) = randn(1,size(x,2));
```

QUESTIONS

1. Are the recovered signals the same as the original signals?
2. How about the estimated mixing matrix, is it similar to the original?
3. Save your resulting plots for your report.

SOLUTION

The original signals should not be recovered and the estimated $\boldsymbol{A}$ matix should not be similar to the original. (**Note**: In some odd cases the original sources can be recovered due the random generation of the signals. In this case rerun the script to be convinced that recovering Gaussian distributed signals is highly unlikely). The plots should look similar to:



2) IN-DEPTH ICA: BUILDING YOUR OWN ICA ALGORITHM

Now that we've seen that ICA can recover mixed signals as long as they are independent (non-Gaussian), we'll look further into the mechanics of ICA. In section 2.1 we will try to recover the same source signals as in the Warm-Up by implementing our own ICA algorithm. In section 2.2 we will examine signals that have a strong directional structure.

2.1) Iterative ICA implementation

Here we implement our own ICA algorithm by closely following the implementation of ICA as given in section 6 of the In-Depth ICA Reading reference (which describes the FastICA algorithm).

First generate an example dataset like the one in the Warm-Up:

```
clear all; close all;
xx = 0:0.05:10;      % 201 points

s1 = sin(xx);
s2 = rem(xx,0.25);

figure; plot(s1); hold on; plot(s2); hold off;
title('Original source signals');      % sin and sawtooth

S = [s1; s2];          % source signal matrix

A = [0.3 0.7; ...
```

```
    0.8 0.3];           % mixing the source signals

X = A*S;                % measured mixed signal matrix
```

The first step in the ICA algorithm is to zero-mean (subtract the respective mean of each signal) and whiten the data (enforce unit variance of each signal).

```
X_mean = mean(X,2);         % take mean along rows (remove cols)

X_zerod = zeros(size(X));
for i = 1:size(X,2)
    X_zerod(:,i) = X(:,i) - X_mean(:,1);     % zero mean each signal
end

Cov = cov(X_zerod');     % covariance of the two variables (x1, x2)
[U D V] = svd(Cov);      % svd to get eigen values and eigen vectors

d2 = diag(diag(D.^(-1/2)));     % diagonal matrix

X_zerod2 = (U*d2*U')*X_zerod;  % whiten data with matrix transformation
```

Next we use equations 45 and 46 from the In-Depth ICA Reading reference (pgs 20-21) to build an iterative estimation of the matrix $\boldsymbol{W}$, which contains the independent component directions $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$.

In brief, the algorithm initializes random directions to $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$, then iteratively updates the estimate of $\boldsymbol{W}$ (equation 46) while symmetrically decorrelating (orthogonalizing) $\boldsymbol{W}$ at each iteration.

The updating estimate of $\boldsymbol{W}$ is given by:

$$\boldsymbol{W}^{+} = \boldsymbol{W} + \Gamma * [\mathrm{diag}(-\beta_i) + E\{g(\boldsymbol{y})\boldsymbol{y}^T\}] * \boldsymbol{W}.$$

Where $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x}$, $\beta_i = E\{y_i g(y_i)\}$, $\Gamma = \mathrm{diag}(1/(-\beta_i - \mathrm{E}\{\mathrm{g}'(\mathrm{y_i})\}))$, and $g = \tanh(2\boldsymbol{y})$. E is the expectation value; here we use the mean of our signals to estimate expectation values.

Decorrelating $\boldsymbol{W}$ is accomplished by repeating (until convergence):

$$1.\ \boldsymbol{W} = \boldsymbol{W}/\sqrt{\|\boldsymbol{W}\boldsymbol{W}^T\|_2}$$

$$2.\ \boldsymbol{W} = \frac{3}{2}\boldsymbol{W} - \frac{1}{2}\boldsymbol{W}\boldsymbol{W}^T\boldsymbol{W}.$$

A shell code of the iteration is provided. The term Γ will be called 'gamma,' $\mathrm{diag}(-\beta_i)$ will be called 'diag term' and $E\{g(\boldsymbol{y})\boldsymbol{y}^T\}$ will be called 'term 2'. The student is to complete the ICA implementation code where indicated.

```
x = X_zerod2;       % x variable as used in the reference paper

W = rand(2,2)-0.5;  % initialize with 2 random vectors, -0.5 to 0.5 range

% FAST ICA iteration
diff_outer = 5;         % outer loop W+ norm difference
counter_out = 0;        % outer loop W+ counter
```

```
while (diff_outer > 1e-7)  % while not converged

y = W*x;
g_y = tanh(2*y);        % 2xN matrix
gy_mult = y.*g_y;       % y*g_y component wise multiplication
gy_expect = mean(gy_mult,2); % mean of components (remove cols), get row vect

diag_term = -1*diag(gy_expect);    % ***TO BE USED BELOW***

%term2 = g_y*y'   2x2 matrix
term2_temp = zeros(size(x,2),size(diag_term(:),1)); % N x 4 matrix

for i = 1:size(x,2)   %loop through data pts
    term2_temp(i,:) = reshape(g_y(:,i)*y(:,i)', 1, 4);
    % store each 2x2 matrix
end

% mean of matrix elements
mean_temp = mean(term2_temp,1);    % 1x4 matrix
mean_temp_mat = reshape(mean_temp, 2, 2);    % reshape back to 2x2

term2 = mean_temp_mat;             % ***TO BE USED BELOW***


% gamma term
g_prime = 1-(tanh(2*y)).^2;              % 2xN matrix

g_prime_expect = mean(g_prime,2);        % 2x1 matrix

term_g = gy_expect - g_prime_expect;
term_g = term_g.^(-1);               % -1 power   2x1 matrix

gamma = diag(term_g);               % ***TO BE USED BELOW***



% update W *** fill in below ***
W_plus = W + ;      % update W (using updating estimate)
%          *** fill in ***


% orthogonalize
W_plus = W_plus/sqrt(norm(W_plus*W_plus',2));


W_old = W_plus;                % store W_plus
diff = 5;                      % inner loop norm difference
counter = 0;                   % inner loop counter

while (diff > 1e-7)           % orthogonalize W estimate
    % update W *** fill in below ***
W_plus =   ;                        % use decorrelating step 2
```

```
    %          *** fill in ***

    diff = norm(W_old-W_plus,2);        % norm of successive estimates

    W_old = W_plus;                      % reset W_old
    counter = counter+1;
end  % inner iteration loop

if (counter_out == 1000)      % break out of loop for 1000 iterations
    break
end

diff_outer = norm(W-W_plus,2);  % norm of successive outer estimates

W = W_plus;          % update W to be orthogonalized W_plus

counter_out = counter_out+1;

end  % outer iteration loop
```

Now that we have the iteration algorithm complete, run the algorithm to get a $\boldsymbol{W}$ estimate.

Next we undo the whitening on the output $\boldsymbol{W}$ matrix to get back component directions that are meaningful in our original space.

```
% dewhiten (rotate and scale matrix)
W_dewhit = (U*(D.^(1/2))*U')*W;

% W here has component directions in rows (by construction in the
algorithm), they should be in columns. Since W is orthogonal, taking
the transpose gives us column directions, and also since W is
orthogonal, inv(W) = W' = A. In this manner we found A and the columns
of W_dewhit give us the component directions.
```

Now to recover our original signals we multiply $\boldsymbol{X}$ by $\boldsymbol{A^{-1}}$:

```
S_back = inv(W_dewhit)*X;      % take the inverse of 'A'
% (A has component directions in columns)

% Plot the recovered original signals
figure; plot(S_back(1,:)); hold on; plot(S_back(2,:)); hold off;
title('W+ Implemented ICA recovered signals');
```

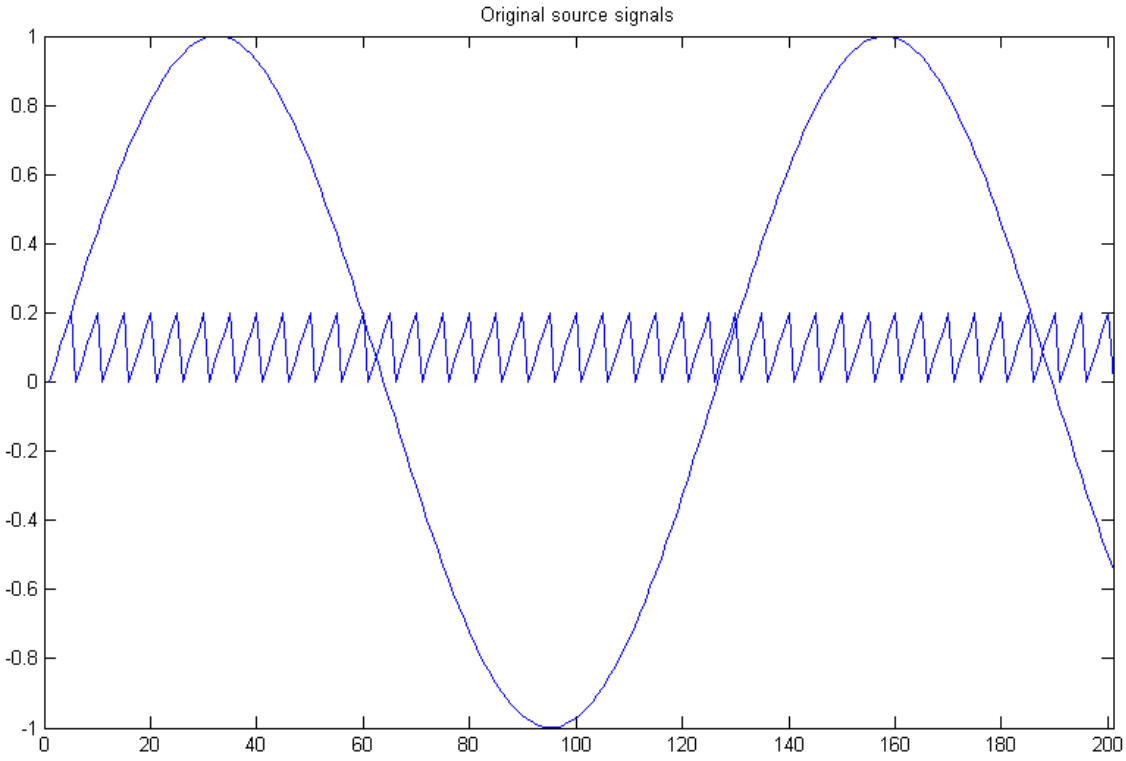
If the original signals are not recovered, rerun the ICA implementation code and repeat the de-whitening and plotting steps above. Sometimes the $\boldsymbol{W}$ matrix does not converge to the correct components and rerunning with another random initialization can solve this. Note that the recovered signals may differ from the originals by a sign and scaling factor.

Compare the recovered signals to output from the FastICA full algorithm.

```
[icasig, A_i, W_i] = fastica(X);      % run FastICA

S_back_i = W_i*X;                     % recover signals

% Plot the FastICA recovered original signals
figure; plot(S_back_i(1,:)); hold on; plot(S_back_i(2,:)); hold off;
title('FASTICA recovered signals');
```

SOLUTION

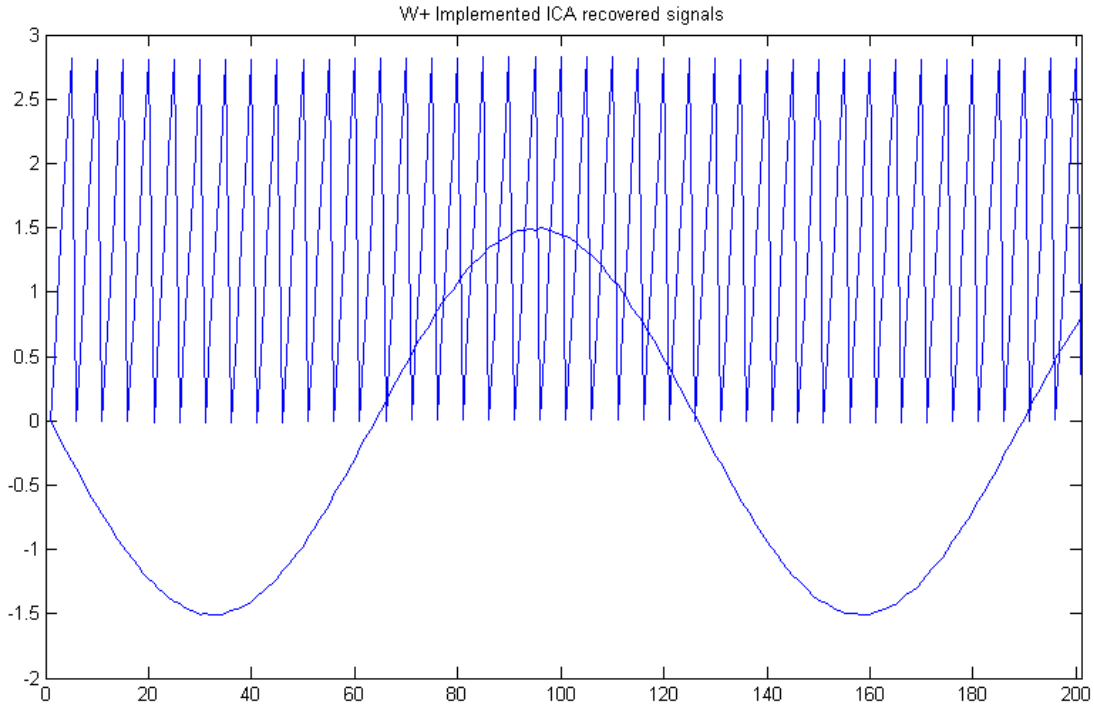
The plot of the original signals should look like:



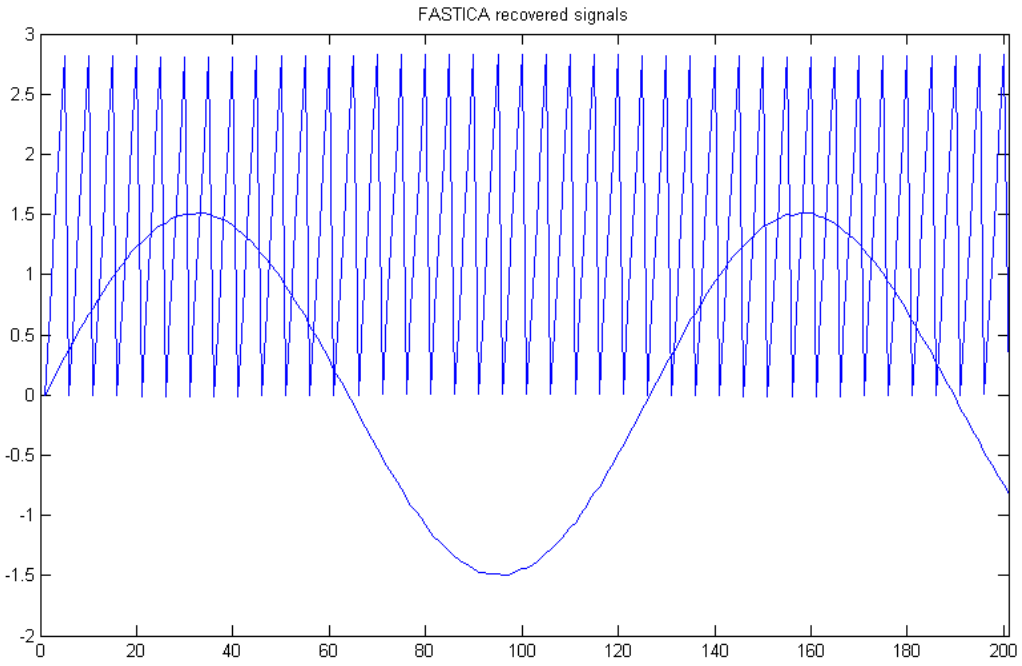
The missing code in the implementation is:

```
W_plus = W + gamma*(diag_term + term2)*W;       % update W

W_plus = (3/2)*W_plus - 0.5*(W_plus)*(W_plus')*W_plus;
```

The plot of the recovered signals (using the iterative implementation) should look like:



The plot of the recovered signals (using FastICA) should look like:



The recovered sine wave is negative and scaled differently for our ICA implementation result. The sawtooth function has the correct sign but is scaled differently than the original for the results of both ICA algorithms.

2.2) Directional Data and ICA

Consider data from two source signals that have two distinct directions:

```
N=250;                          % 250 data points
A = [0.3,0.9;...
    0.8,0.1];                   % two column directions

r = randn(1,N);
X = A*(randn(2,N).*[(r>=1/2);(r<1/2)]);  % generate X

% plot data
figure; scatter(X(1,:)', X(2,:)','.');
title('Original data');
```

Now zero mean and whiten the data:

```
X_mean = mean(X,2);         % take mean along rows (remove cols)

X_zerod = zeros(size(X));
for i = 1:size(X,2)
    X_zerod(:,i) = X(:,i) - X_mean(:,1);      % zero mean each signal
```

```
end

Cov = cov(X_zerod');     % covariance of the two variables (x1, x2)
[U D V] = svd(Cov);      % svd to get eigen values and eigen vectors

d2 = diag(diag(D.^(-1/2)));     % diagonal matrix

X_zerod2 = (U*d2*U')*X_zerod;  % whiten data with matrix transformation
```

Next, run the implemented algorithm from 2.1 on this new data:

```
x = X_zerod2;        % x variable as used in the reference paper

W = rand(2,2)-0.5;  % initialize with 2 random vectors, -0.5 to 0.5
range

% FAST ICA iteration
.
.
.
```

Then de-whiten the output matrix and plot the resulting components:

```
% dewhiten (rotate and scale matrix)
W_dewhit = (U*(D.^(1/2))*U')*W;

% plot components
figure; scatter(X(1,:)', X(2,:)', '.'); hold on;
title('Orig data, W+ iteration components');
plot([0 W_dewhit(1,1)], [0 W_dewhit(2,1)] ,'g', 'Linewidth', 3);
plot([0 W_dewhit(1,2)], [0 W_dewhit(2,2)], 'g', 'Linewidth', 3);
hold off;
```

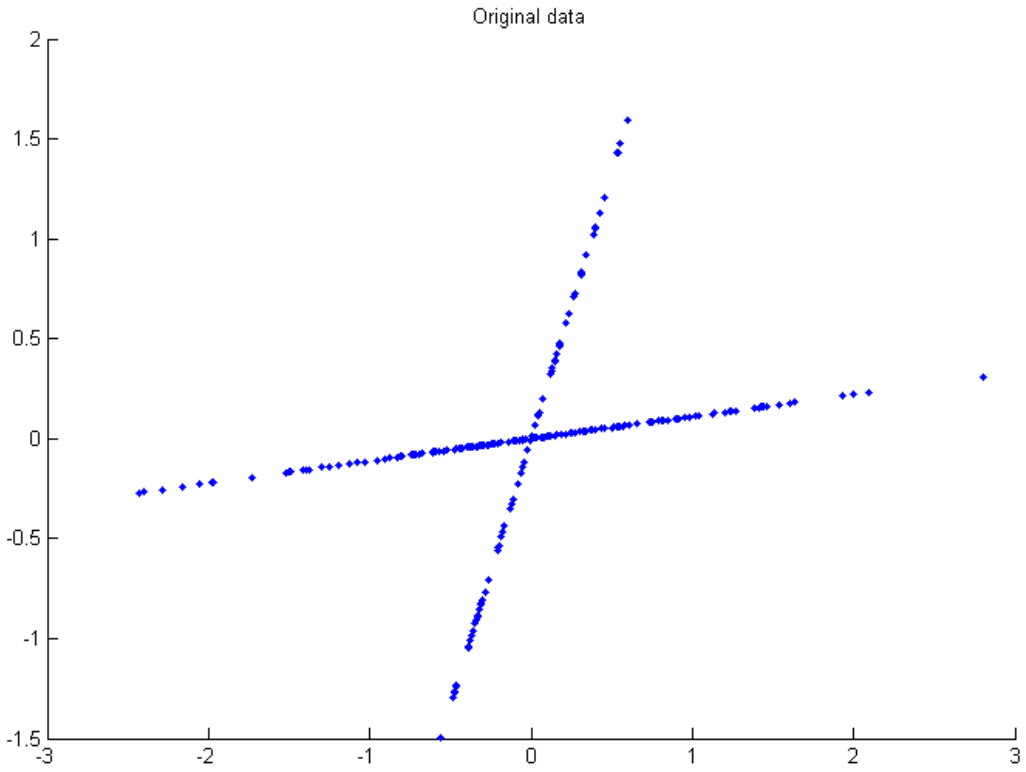
The two directions plotted should match the directions of the data. If they do not match well, rerun the iteration algorithm with a new random vector initialization as was done in 2.1.

Now try FastICA on the dataset and compare the two algorithms' results:

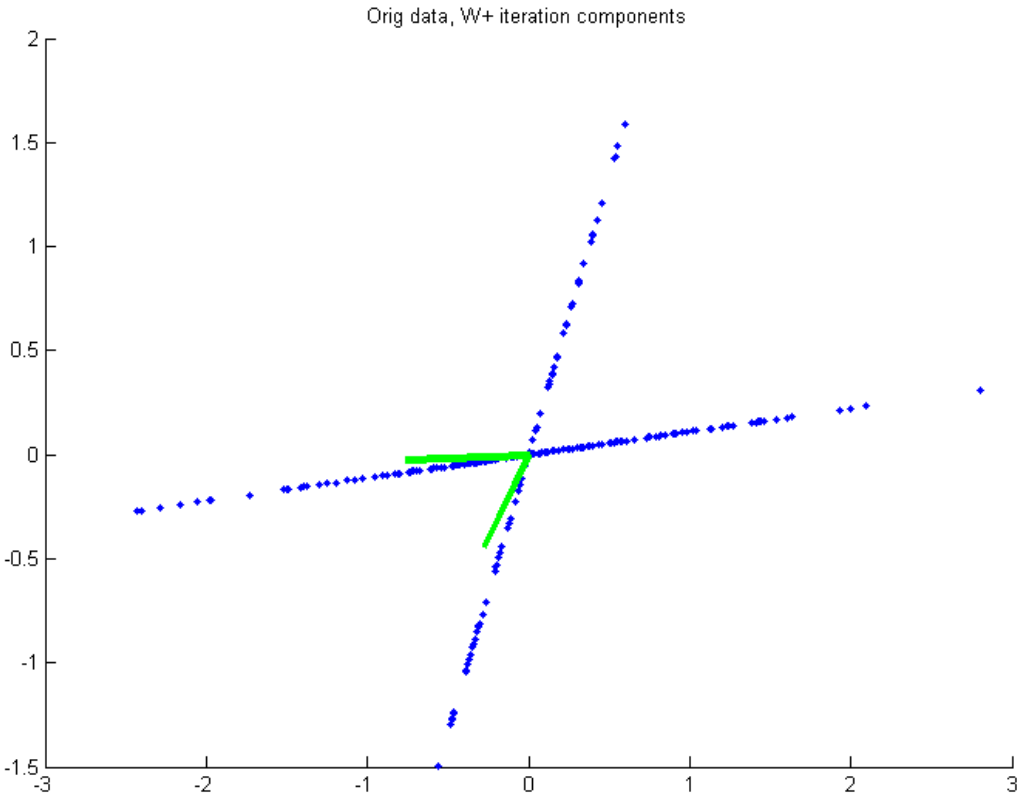
```
[icasig, A_i, W_i] = fastica(X);     % run FastICA

% plot data points and component directions
figure; scatter(X(1,:)', X(2,:)','.'); hold on;
title('Orig data, FASTICA components');

plot([0 A_i(1,1)], [0 A_i(2,1)],'g', 'linewidth', 3);
plot([0 A_i(1,2)], [0 A_i(2,2)],'g', 'linewidth', 3); hold off;
```

SOLUTION

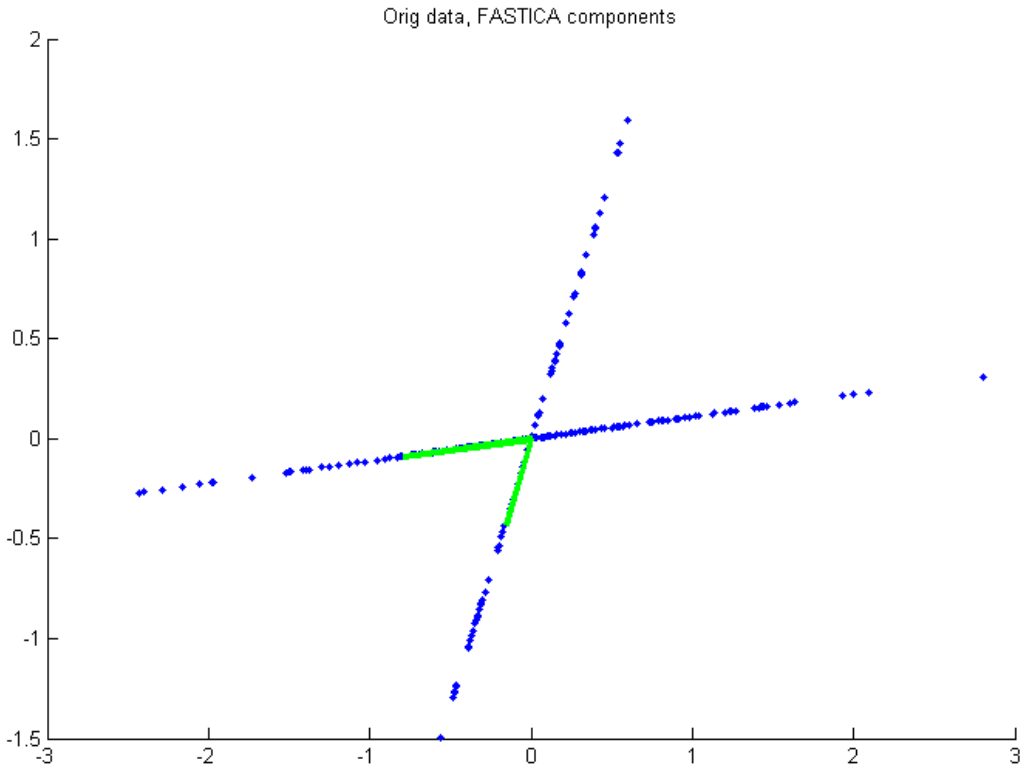
The plotted data should look like:



The implemented algorithm component plot should look like:



The FastICA algorithm component plot should look like:



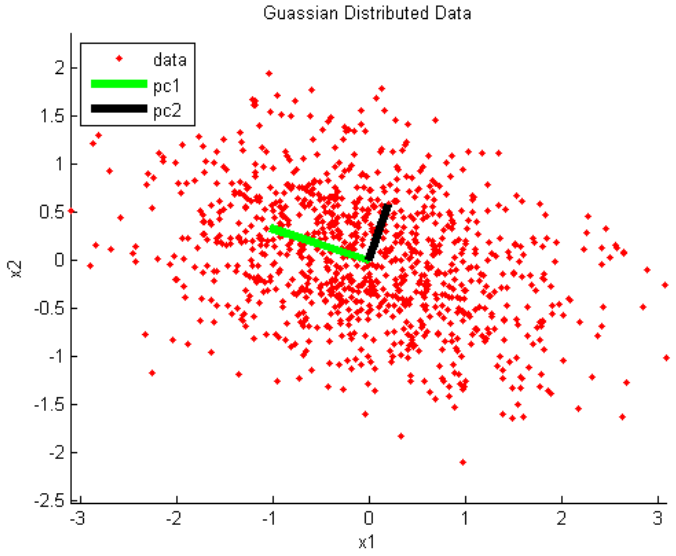
Note that the sign of the component directions can be different for each run of the algorithms. The FastICA algorithm should output components that lie on the directions of the data, whereas the implemented algorithm outputs components that are somewhat misaligned with the directions of the data.

3. PCA VS. ICA: WHAT IS RIGHT FOR YOUR DATA?

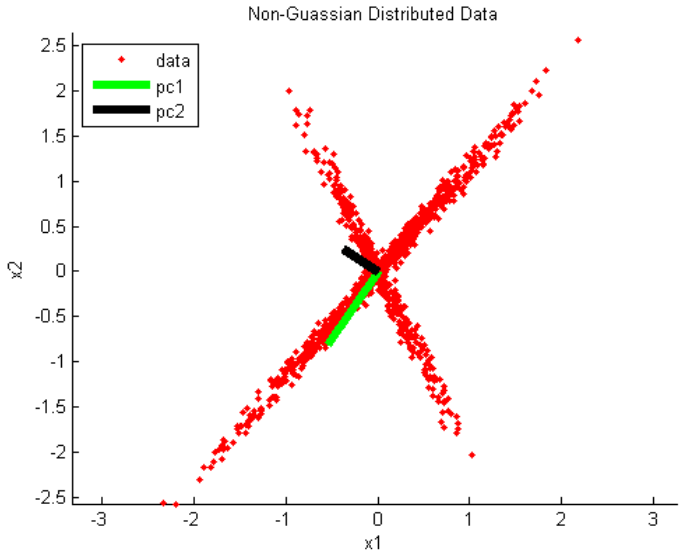
The independent components in a dataset, $\boldsymbol{x} = \boldsymbol{As}$, are the columns of the mixing matrix, $\boldsymbol{A}$. Another technique exists for finding the underlying components of a dataset, *principle components analysis* (PCA). Briefly, PCA is a technique that uses an orthogonal transformation to create a set of orthogonal basis vectors that describe the strongest direction of the data. This set of principle components is less than or equal to the rank of the original data matrix. It turns out that the *singular value decomposition* (SVD) of the data can be used to find the principle components. For example:

Let $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ be a set of observed data where $\boldsymbol{m}$ is the number of features and $\boldsymbol{n}$ is the number of data samples. Using the SVD, there exists a decomposition such that $\boldsymbol{X} = \boldsymbol{U\Sigma V^T}$. Here $\boldsymbol{U}$ and $\boldsymbol{V}$ are the left and right singular vectors and are also orthogonal matrices. $\boldsymbol{\Sigma}$ contains the singular values which describe the weights for the singular vectors. In this case, the columns of $\boldsymbol{U\Sigma}$ represent the weighted principle components for the features of $\boldsymbol{X}$. These principle components are orthogonal.

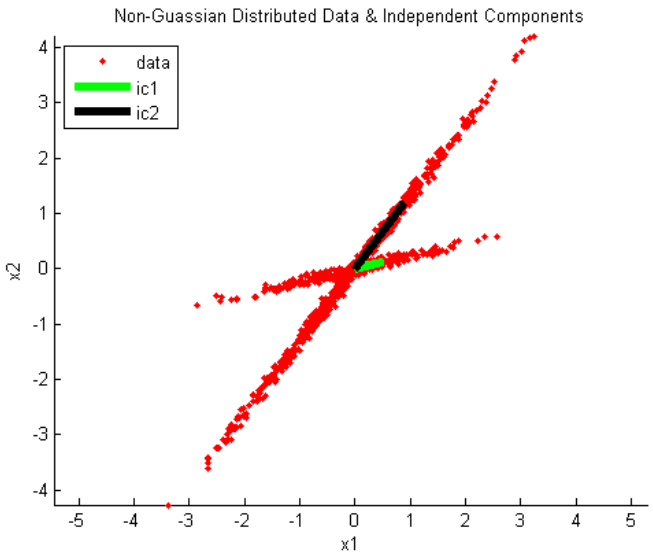
If PCA is so simple, why even bother with ICA? The answer depends on the data in $\boldsymbol{X}$. The components of PCA are orthogonal, so they do very well to describe data that appears as an ellipse (or blob-like). This makes sense because an ellipse can be described in terms of orthogonal axis. It turns out that this sort of distribution is a Gaussian distribution of data. An example of Gaussian data and its principle components is shown below:



But what if the data is not Gaussian? In other words, there is a hidden structure in the data that is not orthogonal? This could pose a problem for PCA because PCA assumes the hidden structure is orthogonal. You can see in the non-Gaussian data below that the principle components miss a strong underlying structure in the data!



ICA does not have the orthogonality assumption of PCA. ICA assumes that the data is independent or not Gaussian. For a distribution such as above, ICA should be able to identify the underlying structure. An example is shown below:



The following code can generate Gaussian and non-Gaussian datasets:

```
% generate n Gaussian data points
n=1000;
A = randn(2,2);
G = A*(randn(2,n));
for i=1:length(G) %add noise
    G(1,i) = G(1,i) + 0.05*randn;
    G(2,i) = G(2,i) + 0.05*randn;
end


% generate n non-Gaussian data points
n=1000;
A = randn(2,2);
r = randn(1,n);
X = A*(randn(2,n).*[(r>=1/2);(r<1/2)]);
for i=1:length(X) %add noise
    X(1,i) = X(1,i) + 0.05*randn;
    X(2,i) = X(2,i) + 0.05*randn;
end
```

QUESTIONS

Note: Use your own ICA implementation from Section 2 for these problems! For the principle components use the built-in MATLAB function `[u,d,v]=svd(X);` to compute the SVD of the data.

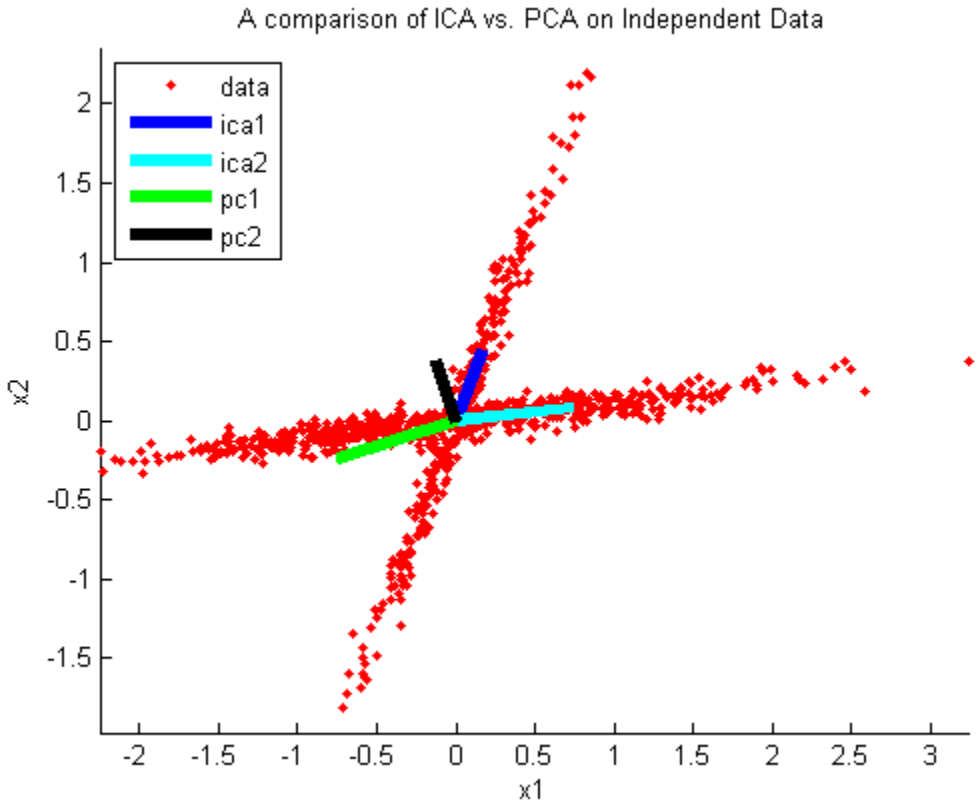
1. Generate a set of non-Gaussian data using the code above in Section 3. Plot the data with the 1st feature on the x-axis and the 2nd feature on the y-axis. On the same plot, show the 2 independent components and the 2 principle components as lines (**Note**: the principle components may need to be scaled by dividing by the square root of the number of samples). Describe the difference between the independent components and the principle components. Which more accurately represents the data? Does this make sense? Why?

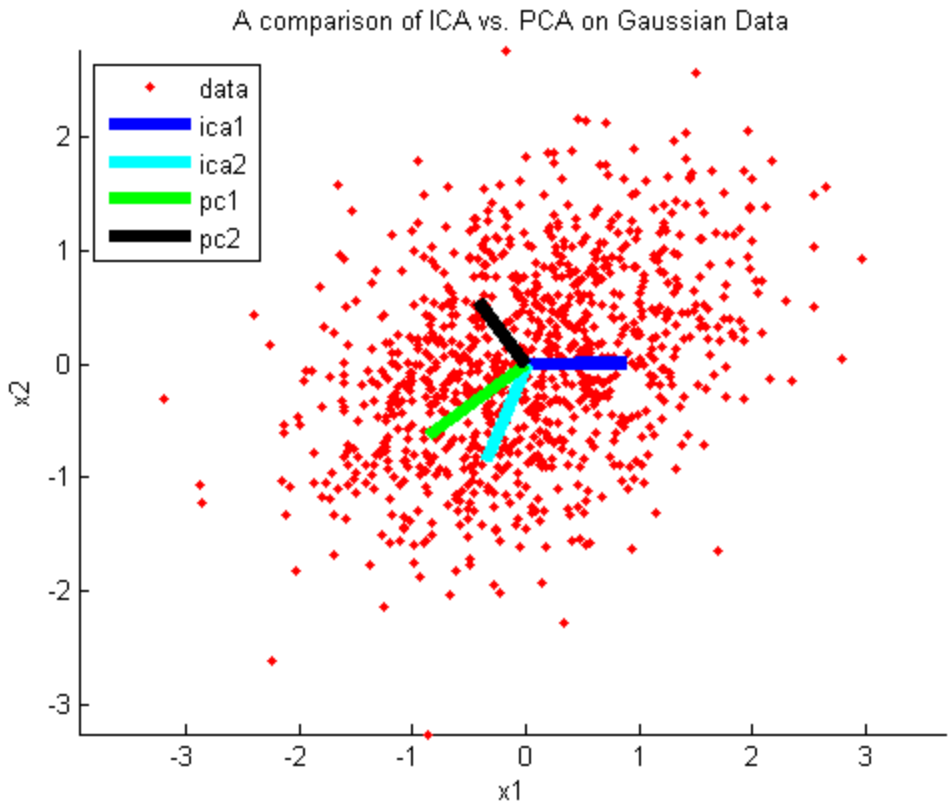
2. Repeat part 1 with a set of Gaussian data using the code in Section 3. Now which set of components makes more sense? Are the independent components meaningful here? Why?
3. What is the most important factor in determining whether to use ICA or PCA?
4. Try part 1 with the FastICA package. How do your results compare?

Feel free to try different settings for `A = [a b; c d];` and observe the results.

SOLUTION

ICA should find the hidden structure of the non-Gaussian data better than PCA. The directions of the principle components do show the overall direction of the data, but they miss the hidden structure. However, for the Gaussian data the ICA results will not have much meaning whereas the PCA results will be very representative of the data structure. How Gaussian the data is determines whether PCA or ICA should be used! If implemented correctly, the student ICA and FastICA should be very similar for this problem. Below are some sample results:

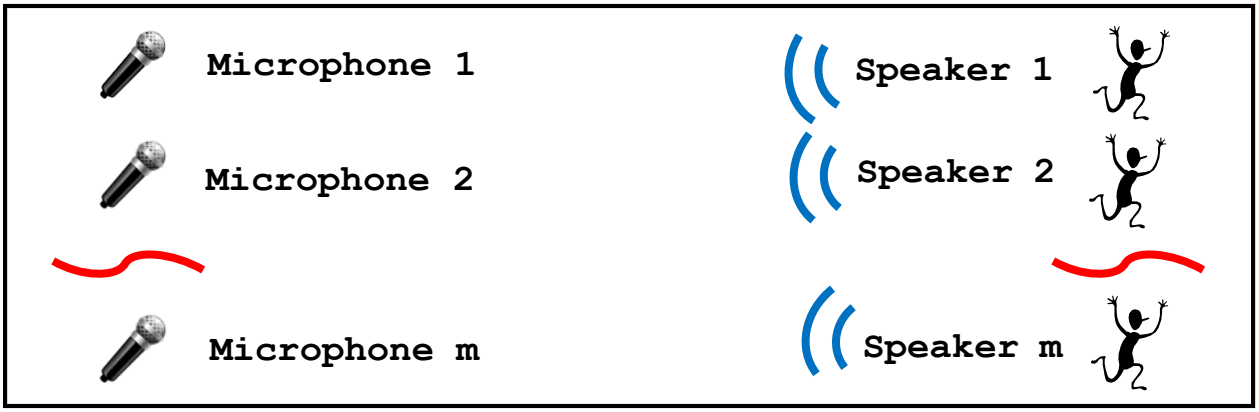




4. PRACTICAL APPLICATION OF ICA: THE COCKTAIL PARTY PROBLEM

The ICA Overview, briefly mentions that *blind source separation* (BSS) is equivalent to the problem solved by ICA. To recap, BSS is the process of extracting the original or source signals from a set of mixed signals. In other words, performing ICA on the mixed signals solves the problem of blind source separation. We have seen many examples of this with simple examples. But now let's look at a real problem, the cocktail party problem.

Consider the situation below:



In this example, $\boldsymbol{m}$ speakers are talking in a room with $\boldsymbol{m}$ microphones. Assume that over some period of time, $\boldsymbol{n}$ samples are collected at each of the $\boldsymbol{m}$ microphones. The recording at each microphone will be a

mixture of each of the speakers, $\boldsymbol{mic_i = a_i^T * Speaker}$. The entire system of microphone recordings, $\boldsymbol{M = AS}, \boldsymbol{M} \in \mathbb{R}^{\boldsymbol{mxn}}$ where $\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{mxm}}$ is some mixing matrix and $\boldsymbol{S} \in \mathbb{R}^{\boldsymbol{mxn}}$ is the matrix of each individual's speech (each row is one individual). Performing ICA on $\boldsymbol{M}$ can therefore recover $\boldsymbol{S}$ and solve this problem. In other words, solving the cocktail party problem is a real application of ICA.

MATLAB has some built in command for loading and playing audio:

- `[y,Fs] = audioread(filename)` reads an audio file and converts it to a signal and sampling rate. **Important**: y in this case is #samples by #signals, so to use FastICA or your ICA, use y'
- `sound(y,Fs)` plays a signal at a given sampling rate over the computer speakers.

QUESTIONS

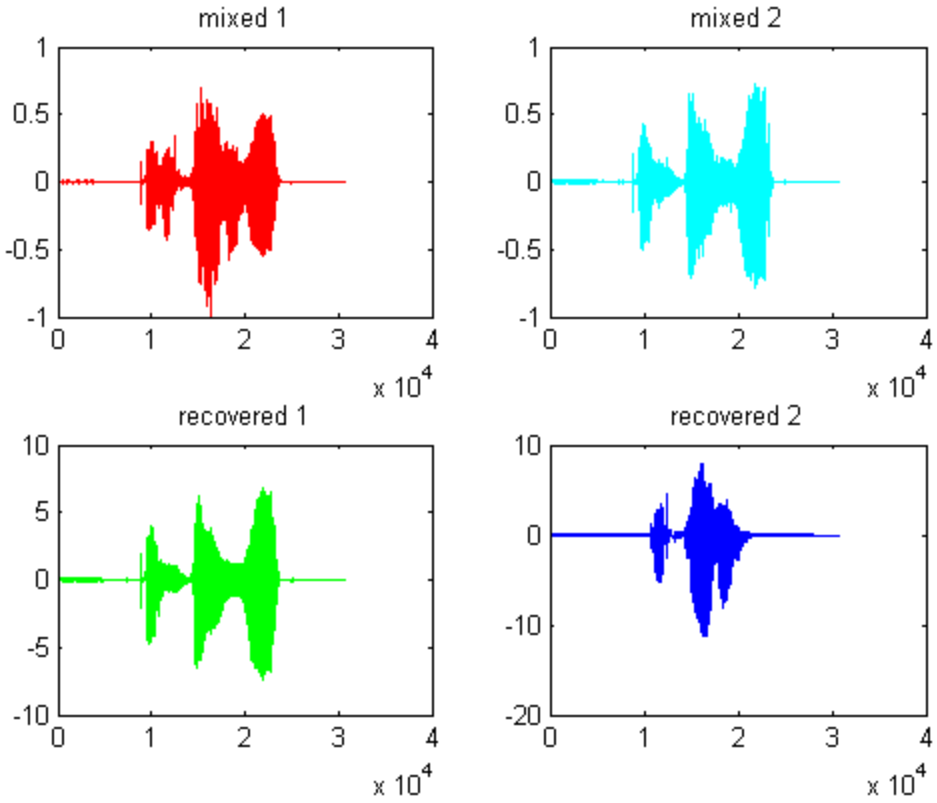
Use the following audio mixture (it contains 2 mixed signals, so 2 columns using `audioread`):

- Mixed Speech: http://www.ism.ac.jp/~shiro/research/sounds/LINEAR/X_linear.wav

1. Use the FastICA package to recover the original source signals. Plot the 2 mixed audio signal (each channel) and the 2 recovered audio signals on the same plot using subplots (similar to Section 1). Does it look like the original audio was recovered?
2. Play the mixed audio and recovered audio using the built-in MATLAB audio commands. Were the 2 speakers separated? Describe the quality of the recovery.

SOLUTION

The graph should show 4 subplots with the recovered audio waveform being 2 distinct waveforms (example below). When played the audio sources should be separated (some distortion may appear in the recovered signals).


mixed 1
mixed 2
recovered 1
recovered 2
x 10^4

REFERENCES

[1] A. Hyvärinen and E. Oja, "Independent Component Analysis: Algorithms and Applications," [Online]. Available: http://www.cs.helsinki.fi/u/ahyvarin/papers/NN00new.pdf.

[2] J. Shlens, "A Tutorial on Principal Component Analysis," 7 April 2014. [Online]. Available: http://arxiv.org/pdf/1404.1100.pdf.

[3] J. Shlens, "A Tutorial on Independent Component Analysis," 14 April 2014. [Online]. Available: http://arxiv.org/pdf/1404.2986.pdf.

[4] S. Ikeda, "Some Trials of Blind Source Separation," [Online]. Available: http://www.ism.ac.jp/~shiro/research/blindsep.html.

[5] J. Särelä, P. Hoyer and E. Bingham, "Cocktail Party Problem," [Online]. Available: http://research.ics.aalto.fi/ica/cocktail/cocktail_en.cgi.