Independent Component Analysis

Required Packages: Python(2.7+), NumPy, SciPy Toolkit, Matplotlib

In this tutorial we will learn how to solve the Cocktail Party Problem using Independent Component Analysis(ICA). We will first take a look at Principle Component Analysis(PCA). The limitations of PCA will naturally lead to an understanding of what ICA does.

Overview

The Cocktail Party Problem(CPP)

So what is the Cocktail Party Problem? Imagine you are at a party where a lot of different conversations are happening in different parts of the room. As a listener in the room, you are receiving sound from all of these conversations at the same time. And yet, as humans, we possess the ability to identify different threads of conversation and to focus on any conversation of our choice. How do we do that? And how can we program a computer to do that? So this is essentially the Cocktail Party Problem: Given **m** sources(conversations at the party for example), and some number of sound receivers, separate out the different signals. (We will talk about how many receivers we need later.)

We need to make some mathematical assumptions and also phrase the problem more formally.

The data

So first of all, our signals here are the sounds coming from different sources. At every (uniformly spaced) discrete interval of time we record **m** samples, one at each of our **m** microphones.

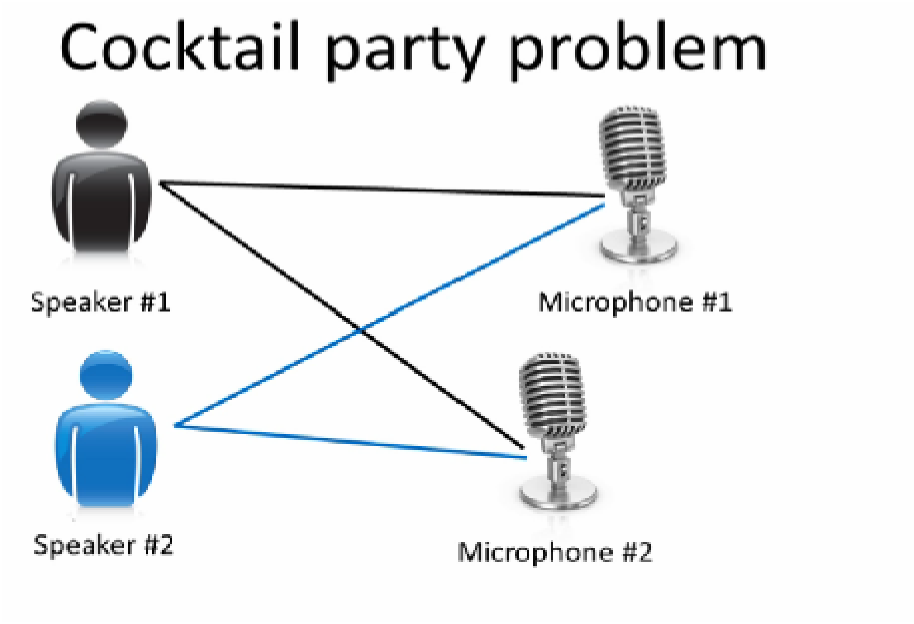
Note the implicit assumptions that we have made here:

1) There are as many microphoneses as there are independent conversations(sources) going on in the room. This assumption allows us to come up with a method to retrieve all the m independent signals. We can say that our system is **critically determined**(and is not under- or over-determined). Henceforth, we shall only consider this case in the tutorial.

2) Each microphone records a reasonably distinct combination of the independent signals. This simply amounts to not keeping two microphones too close to each other. Due to practical computational limits (see floating point math), it is always best to have easily distinguishable recordings.

How are we recording this data? We simply record the amplitude of the sound at each instant. Recording the pressure amplitude is a convenient thing to do(and is what a microphone does. A transducer then converts the pressure amplitude to a voltage). Note that we are recording the signals at discrete intervals of time (at a rate assumed to be greater than the Shannon Sampling rate)and will be working only in the time domain with these discrete signals.

One very important thing: We assume that the sound that any receiver records is a **linear combination** of sounds from the different sources. This is a reasonable assumption to make as pressure adds linearly. Each receiver will receive a different linear combination: If the first receiver is closer to a particular speaker than the second receiver, then the linear weight of this speaker will be proportionately higher for the first receiver.



Courtesy: http://blog.csdn.net/muzilanlan/article/details/45917627 (http://blog.csdn.net/muzilanlan/article/details/45917627)

We further assume that each source is **statistically independent** with respect to all the other sources. We will look at a mathematical interpretation of statistical independence of two signals later. Within the context of the Cocktail Party parable an intuitive understanding of this assumption follows naturally, as the conversations happening in different parts of the room are independent of each other. Hence, knowing the signal at a particular instant from one source does not allow us to predict the value of the signal from any other source at that instant. They are independent variables.

This is the key assumption in Blind Source separation that allows to solve the problem.

We are also making one vital assumption about the sources of the signals: that they are non-Gaussian. We will look at what that means and why it matters in the section on Statistical Independence.

The math

We will index our microphones from **1** to **m**.

The signal received by the microphone labelled **i** over the entire time of recording will be denoted by X_i. A particular sample of this recorded signal, recorded at the time index **j** will then be denoted by X_i^j.

Hence, if the samples of the signals recorded over time be **N**, then X_i can be seen to be a row vector in **N**-dimensional space. It's jth element is given by X_i^j.

We had said that we have **m** microphones. Hence **i** in the above description ranges from **1** to **m**.

If we stack up these row vectors, we will get an **m x N** matrix whose ith row corresponds to the samples recorded by a particular microphone. A 'vertical slice' of this matrix, i.e a column corresponds to all the samples recorded at a given instant of time, indexed by the indices of the corresponding microphone.

Let us call this data matrix **X**.

To reiterate, X_i^j corresponds to the sound sample recorded by the **i**th mike at the time (indexed by) **j**.

Let us now similarly define matrices corresponding to the sources that we wish to finally recover. The indices for the independent sources also go from **1** to **m**.

Let S_i denote the signal generated by the **i**th independent source that we wish to recover (the **i**th conversation in the room). It is defined as a row vector.

S_i^j is then the **j**th time sample of this signal.

Again, we vertically stack up these row vectors to get a **m x N** matrix denoted by **S**.

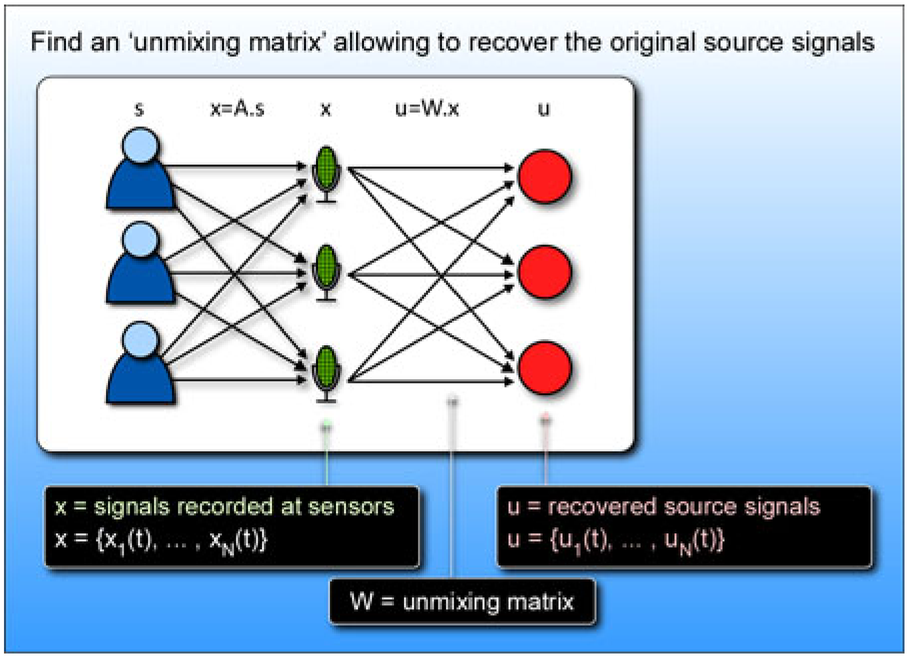
Now that we have defined our data and the signals that we wish to retrieve, we will describe the (assumed) relationship between the two. Note that we had assumed that the independent sources add **linearly** to give the recorded signals.

This means that each X_i is some linear combination of the vectors S_1 through S_m.

Putting it all together, we conclude that $X = AS$; for some **m x m** matrix **A**, called the mixing matrix.

Our objective is to then find an "un-mixing" matrix **W** that satisfies $S = WX$.

If we know this **W**, as we already have **X**, we can calculate **S** by a direct multiplication.


Find an 'unmixing matrix' allowing to recover the original source signals
s
x=A.s
x
u=W.x
u
x = signals recorded at sensors
x = {x₁(t), ... , x_N(t)}
u = recovered source signals
u = {u₁(t), ... , u_N(t)}
W = unmixing matrix

Courtesy: http://www.nocions.org/research/independent-component-analysis-ica-/ (http://www.nocions.org/research/independent-component-analysis-ica-/)

Outline of solution

Our objective is to find the matrix **W**. As we have assumed that the number of microphones is equal to the number of independent conversations, it turns out that the matrix **A** is invertible and hence **W** is just the inverse of **A**.

Hence, it suffices to find **A**. We will employ Singular Value Decomposition(SVD) (https://www.wikiwand.com/en/Singular_value_decomposition) on the matrix **A**.

Courtesy: Wikipedia

Hence, $A = UDV^*$ for orthogonal matrices **U**, **V*** and diagonal matrix **D**.

Note that **V*** is the conjugate transpose of **V**. Here the conjugate transpose is equivalent to the transpose because we are only dealing with real signals and their linear combinations.

We will then determine each of **U**, **D**, **V*** by considering the covariance matrix of **x** and exploiting the independence of the source signals.

The details follow.

Covariance

An important term in the concept of statistics is *covariance*, which is a measure of how much two random variables change *together*. Covariance provides a measure of the strength of the correlation between two or more sets of random variates. The covariance for two random variates X and Y, each with sample size N, is defined by the expectation value:

$$cov(X, Y) = \langle (X - \mu_x)(Y - \mu_y) \rangle = \langle X \rangle \langle Y \rangle - \mu_x \mu_y$$

where $\mu_x = \langle X \rangle$ and $\mu_y = \langle Y \rangle$ are the respective means.
For uncorrelated variates, $\langle XY \rangle = \langle X \rangle \langle Y \rangle$ and hence,

$$cov(X, Y) = \langle XY \rangle - \mu_x \mu_y = \langle X \rangle \langle Y \rangle - \mu_x \mu_y = 0$$

However, if the variables are correlated in some way, then their covariance will be nonzero. In fact, if $cov(X, Y) > 0$, then Y tends to increase as X increases, and if $cov(X, Y) < 0$, then Y tends to decrease as X increases. Note that while statistically independent variables are always uncorrelated, the converse is not necessarily true.

As you can see, covariance can be a good metric to analyse the dependence of two datasets. To read more about covariance, follow this link. (http://mathworld.wolfram.com/Covariance.html)

As a special case, substituting $X = Y$ gives

$$cov(X, X) = \langle X^2 \rangle - \langle X \rangle^2 = \sigma_X^2$$

where σ_X denotes the *standard deviation*. Thus, $cov(X, Y)$ reduces to the statistical variance for this case.

Given a dataset vector X *(nx1 vector)*, the covariance of X is given by

$$C_x = (X - \mu_x)(X - \mu_x)^T$$

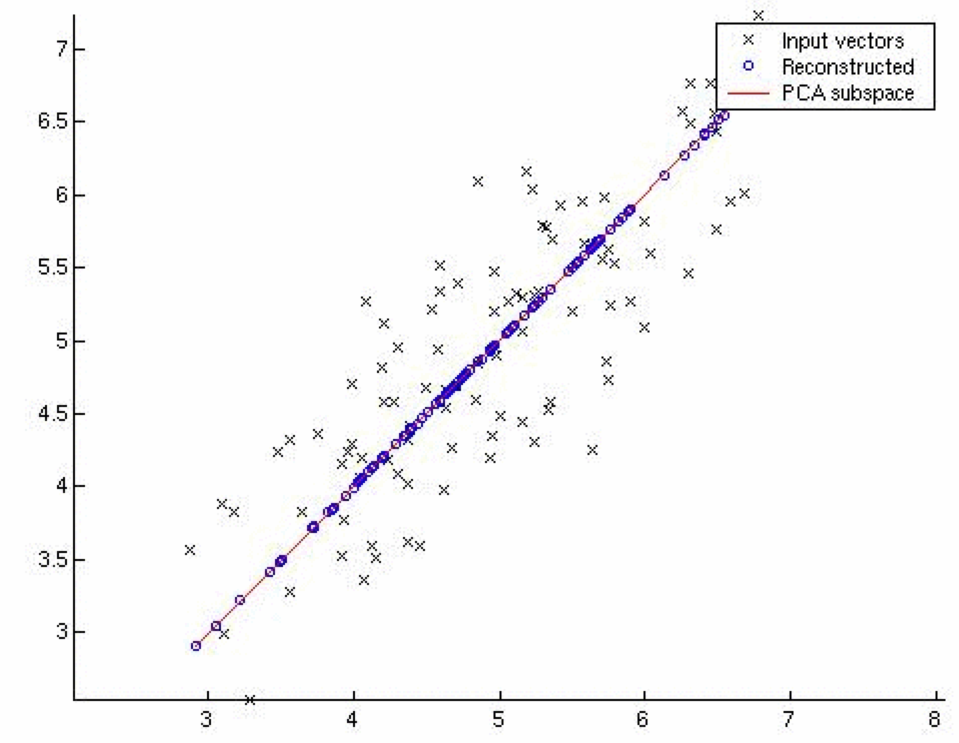
This matrix C_x has very interesting properties, which we will be exploiting in the upcoming sections. This document (http://www.robots.ox.ac.uk/~davidc/pubs/tt2015_dac1.pdf) would be an interesting read.

Now that we have the necessary tools required, let us dive into some algorithms that involve manipulating the information matrix and separating them into components. The first such algorithm is the PCA.

Principle Component Analysis

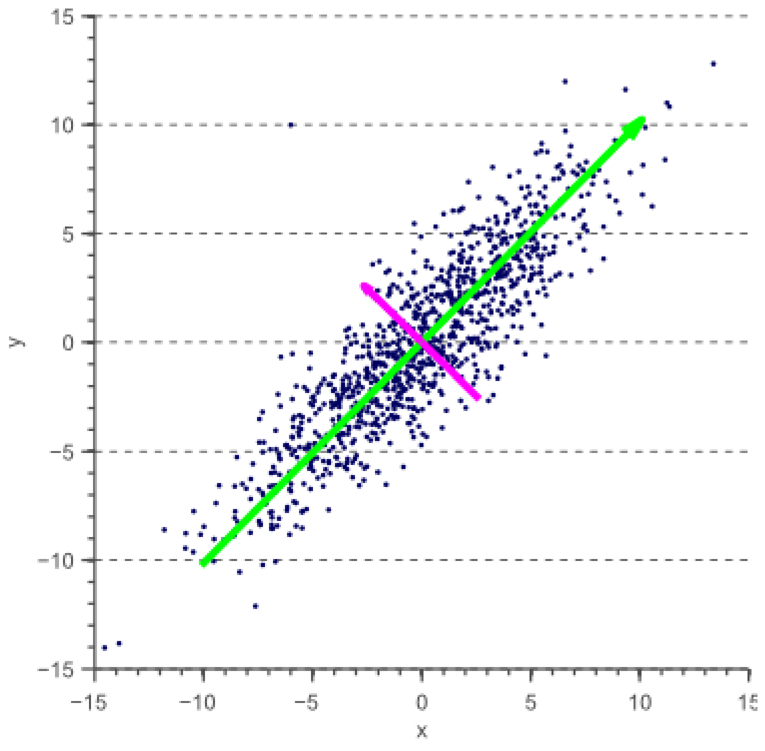
Principal Component Analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. The number of principal components is less than or equal to the number of original variables. This transformation is defined in such a way that the first principal component has the **largest possible variance** (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components.

What does this mean? Let's visualise the problem at hand.
Let us say you are given a dataset of point in the two-dimensional plane like the plot given below.



Courtesy: Vision Dummy

The given dataset is two-dimensional and has a set of points arranged in a roughly elliptical form. What the PCA algorithm does is that it identifies the first principle component of the data, that is, the direction along which the *variance* of the data is maximum. This is the direction indicated by the red line. The second principle component is the one orthogonal to this, along the approximate minor axis of the dataset. Another such example of PCA on a 2D dataset is shown below.



Courtesy: Center for Machine Perception, CTU, Prague (http://cmp.felk.cvut.cz/)

Here, the green line shows the vector corresponding to the first principle component, and pink shows the second component. To visualise how PCA can be applied on higher dimensional datasets, check out this page (http://setosa.io/ev/principal-component-analysis/).

The PCA has various applications, some of which have been listed below.

- To identify the most important feature(s) of a multivariate function.
- Reduce dimensionality of data for saving memory, preserving maximum information.
- Speed up processes (regression, for example) by ignoring the features with low variance.
- As a pre-processing technique for further statistical methods, increasing efficiency.

Before we go ahead with understanding PCA, it is important to consider the following **assumptions and limitation** of PCA:

1. PCA assumes the dataset to be a linear combination of the variables.
2. There is no guarantee that the directions of maximum variance will contain good features for discrimination.
3. PCA assumes that components with larger variance correspond to interesting dynamics and lower ones correspond to noise.
4. Output vectors of PCA are orthogonal, which means that the principal components are *orthogonal* to each other.
5. PCA requires the data to be *mean normalized* and *univariate*, as it is highly susceptible to unscaled variables.
6. Since it assumes data to be uncorrelated and accurate, it is *vulnerable to outliers* and hence may produce incorrect results.

Here, we see that PCA breaks down the dataset into uncorrelated (hence, orthogonal) components.

The Algorithm

Principle component analysis relies on the property that the eigenvectors of the covariance matrix for a dataset X represent a new set of orthogonal components that are the principle components of the dataset. Mathematically, given the covariance matrix C_x, the matrix $U = eig(C_x)$ which returns the eigenvectors is the desired matrix.
Another method to do this is *singular value decomposition* of the covariance matrix. In linear algebra, the singular value decomposition (SVD) is a factorization of a real or complex matrix. It is the generalization of the eigendecomposition of a positive semidefinite normal matrix (for example, a symmetric matrix with positive eigenvalues) to any $m \times n$ matrix via an extension of polar decomposition. The theory behind SVD is exhaustive and beyond the scope of this tutorial; you can find some interesting stuff here. (http://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm) For our use, SVD is called in a programming language like Matlab or Python as

$$[U, S, V] = svd(C_x)$$

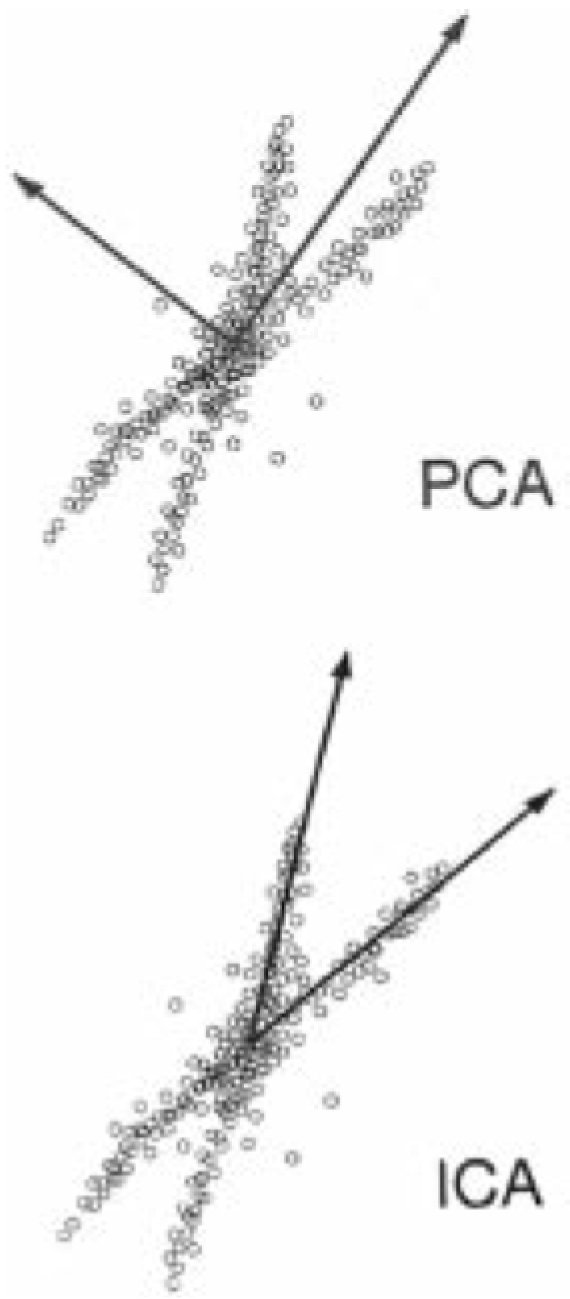
Such that C_x satisfies $C_x = USV^T$. Note that U here refers to the same matrix of orthogonal eigenvectors.

Thus, U here is an $n \times n$ matrix with its column vectors as the eigenvectors of C_x.

- If we aim to perform data compression, define a matrix $U_{reduced} = U[:, 1:k]$ which stores the first k principle components. Thus, the required dataset with reduced dimensionality k can be given by $X_{new} = U^T_{reduced}X$.
- If we do not want to compress data but simply express it in terms of principle/orthogonal components, we have $X_{new} = U^TX$

PCA vs ICA

Now that we have seen what PCA is, let us see how it can be applied in our case - if at all.
We have seen that the basis on which PCA separates out the components is *orthogonality* and *variance*. Drawing parallels to the Cocktail Party Problem, there seems to be an issue. Orthogonality sure is a measure of independence of two datasets (vectors), but it is not the only measure and as it turns out, it is not applicable in the case of CPP. Two different sound signals need not be orthogonal, even if they are independent and this can be seen by simply generating a counter example. *(Here, orthogonal refers to **zero** inner product.)* The very fact that independent signals needn't be orthogonal eliminate the possibility of using PCA. Further, PCA identifies the separated components on the basis of *variance*, which according to Information Theory is a very shallow metric for decorrelating components. Statistical independence of the third or fourth order is generally preferred as a metric - what this means is that we must study deeper features and analyse correlations, but let's leave that for now.
Given below is the simplest way to illustrate how PCA fails due to the orthogonality condition on principle components.



Courtesy: University of Colorado Lectures

Despite the two branches being independent, PCA would not identify them so because it doesn't separate out *independent* components, but *principle* ones based on orthogonality and variance measures. ICA, as the name suggests, looks for *independent* components and not any dependent variable - hence avoiding side-effects! How it does this is by using a metric for ***statistical independence***, described later, and then picking a model that minimizes this dependence. Before we can get to that, let's look at the pre-processing to be done on the data to ensure optimal results in ICA.

Preprocessing for ICA

First, let us consider the basic statement of ICA.

$$x = Ass = Wx$$

Where s refers to the source signals, A, the *mixing matrix* and x, the signal we receive at microphones(say) and $W = A^{-1}$ Given below are the pre-processing stages performed:

- **Centering**: The most basic and necessary preprocessing is to center $\mathbf{x}$, i.e. subtract its mean vector $\mathbf{m} = \mathbf{E}\{\mathbf{x}\}$ so as to make $\mathbf{x}$ a zero-mean variable. This also implies that $\mathbf{s}$ is a zero-mean variable, as can be seen by taking expectation on both sides, above. This preprocessing is made solely to simplify the ICA algorithms: It does not mean that the mean could not be estimated. After estimating the mixing matrix $\mathbf{A}$ with centered data, we can complete the estimation by adding the mean vector of $\mathbf{s}$ back to the centered estimates of $\mathbf{s}$. The mean vector of $\mathbf{s}$ is given by $\mathbf{A}^{-1}\mathbf{m}$, where $\mathbf{m}$ is the mean that was subtracted in the preprocessing.

- **Whitening**: Another useful preprocessing strategy in ICA is to first whiten the observed variables. This means that before the application of the ICA algorithm (and after centering), we transform the observed vector $\mathbf{x}$linearly so that we obtain a new vector $\tilde{\mathbf{x}}$ which is white, i.e. its components are uncorrelated and their variances equal unity. In other words, the covariance matrix of $\tilde{\mathbf{x}}$ equals the identity matrix. A *whitening transformation* is a linear transformation that transforms a vector of random variables with a known covariance matrix into a set of new variables whose **covariance is the identity matrix** meaning that they are uncorrelated and all have *variance unity*.

  $$C_{\tilde{x}} = \tilde{x}\tilde{x}^T = I$$

  The math behind whitening involves a greater understanding of eigenvectors and matrices, which we shall ignore for the purpose of this tutorial. Let us fast-forward to how whitening can be done on a given dataset. The whitening transformation is always possible. One popular method for whitening is to use the eigen-value decomposition (EVD) (http://mathworld.wolfram.com/EigenDecomposition.html) of the covariance matrix $C_{\tilde{x}} = \mathbf{EDE}^T$, where $\mathbf{E}$ is the orthogonal matrix of eigenvectors of $C_{\tilde{x}}$ and $\mathbf{D}$ is the diagonal matrix of its eigenvalues, $\mathbf{D} = \mathrm{diag}(d_1, \ldots, d_n)$. Note that $E\{\mathbf{xx}^T\}$can be estimated in a standard way from the available sample $x(1), \ldots, x(T)$. Whitening can now be done by

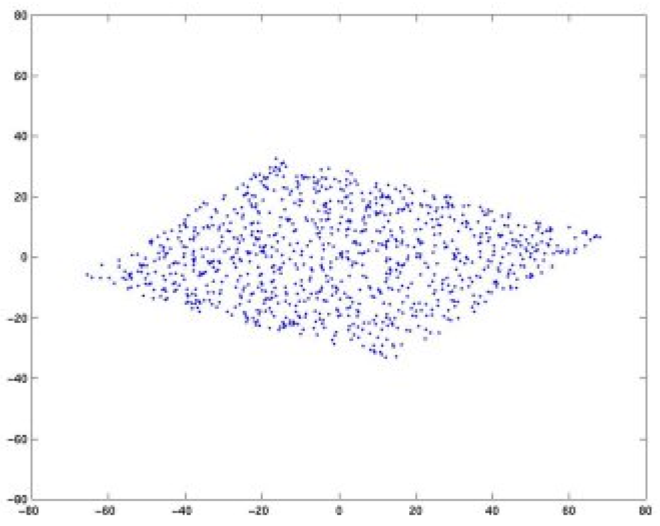
  $$\tilde{x} = ED^{-1/2}E^T x$$

  where the matrix $\mathbf{D}^{-1/2}$is computed by a simple component-wise operation as $\mathbf{D}^{-1/2} = \mathrm{diag}(d_1^{-1/2}, \ldots, d_n^{-1/2})$. It is easy to check that now $C_{\tilde{x}} = \mathbf{I}$.

It is important to note that the whitening transformation changes the matrix A corresponding to the x, and hence

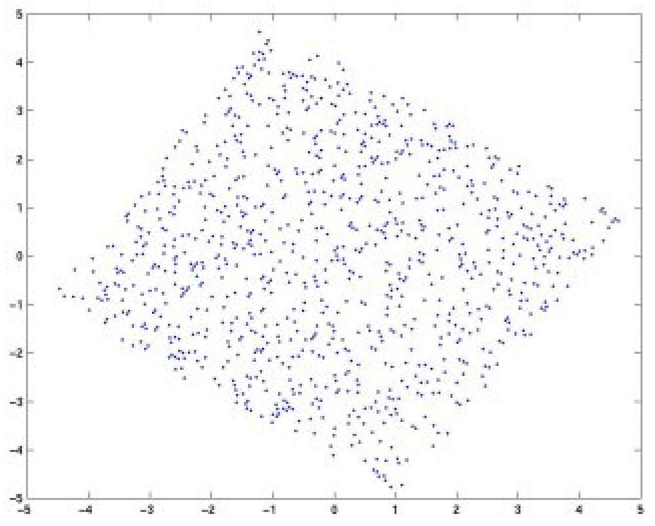
$$\tilde{x} = ED^{-1/2}E^T As = \tilde{A}s$$

The utility of whitening resides in the fact that the new mixing matrix $\tilde{\mathbf{A}}$ is orthogonal. How this affects the data is a little complicated to explain, but I will attempt to illlustrate it below.
Consider a two-dimensional dataset as given below, with *uncorrelated/independent* variables, along the sides of the parallelogram.



Swartz Center for Computational Neuroscience (http://sccn.ucsd.edu)

Evidently, this data has the two independent vectors as the sides of the parallelogram, but when passed into the algorithm, since the vectors are not orthogonal, they might end up showing unnatural dependencies which may not really exist. Hence, we perform whitening of the data to give something that looks like this:



Courtesy: Swartz Center for Computational Neuroscience (http://sccn.ucsd.edu)

Note that we haven't lost any **information** in the above transformation, nor have we created or destroyed any existing correlations. But this new dataset ($\tilde{x}$) is bound to perform better on ICA algorithms. In the rest of this tutorial, we assume that the data has been preprocessed by centering and whitening. For simplicity of notation, we denote the preprocessed data just by $\mathbf{x}$, and the transformed mixing matrix by $\mathbf{A}$, omitting the tildes.

Statistical Independence

This section gives a brief description of statistical independence and it's interpretation in the context of this problem. Recall how we looked at an intuitive explanation. We said that voices are independent because listening to one won't allow us to predict the other.

To exploit this property mathematically, we construct the **probability density functions(pdf)** for each of the signals that our **m** microphones have recorded.

Why would we ever think of doing that? Well, for a number of pragmatic reasons. Statistical methods that work with density functions are very well developed and extremely powerful.

It also makes the math simpler. How you ask?

Well, notice that if we are representing a signal by it's probability density function *alone*, we are saying that at any given time, the value of **the signal is simply a draw of a random variable with this particular pdf.** (What this means is that we are basically discarding/not keeping track of the local correlation of the signal's contiguous values.)

In the context of the Cocktail Party Problem, we use pdfs to quantify statistical independence. There are many ways of quantifying the independence of two random variables if their pdfs are known. This is why we are working with probability density functions.

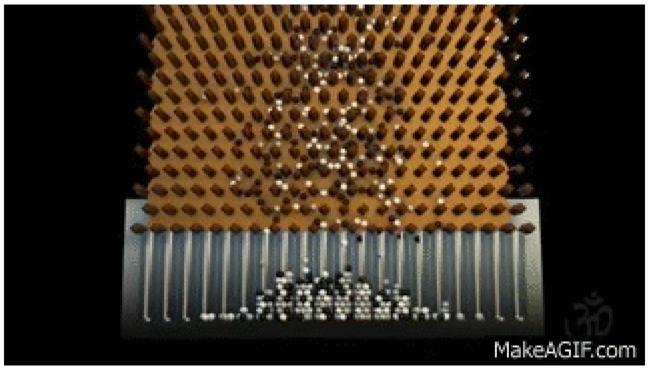
Here comes an extremely important point: While solving the Cocktail party problem, we assume that all the source signals that we wish to recover are non-Gaussian.

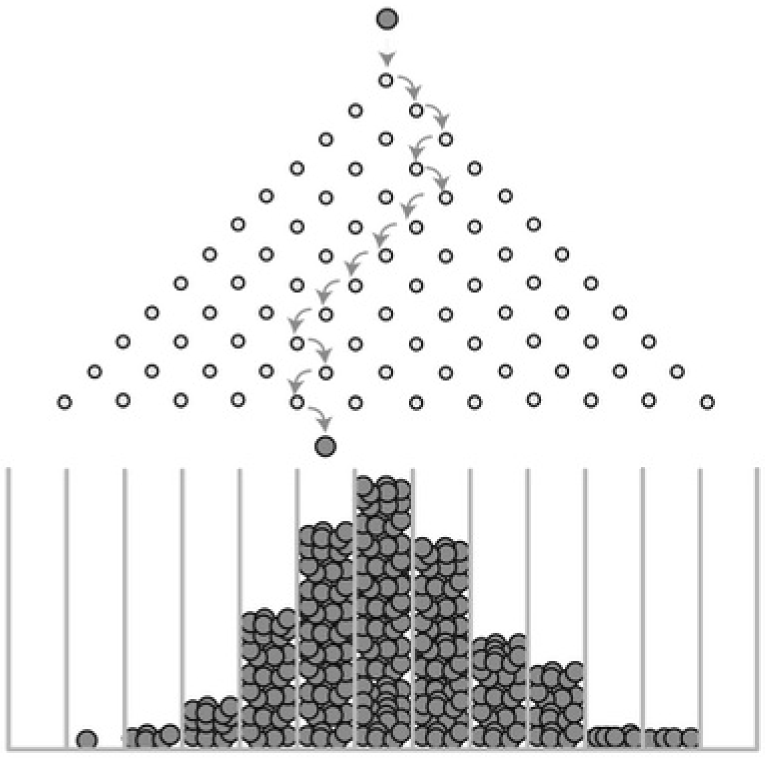
First of all, what do we mean by that? We mean that the pdf of each source signal is not a Gaussian function/distribution.

This might seem like an arbitrary and questionable assumption at first.

We invoke Information theory to justify our assumption. Simply put, there is a result in Information theory that says that the **Gaussian distribution has the greatest possible entropy (for a fixed variance of the distribution).** Entropy in our context is just disorder. The source signals do have order, as they contain lots of information. If they were purely Gaussian, then the signals would just sound like (Gaussian) noise.

We are not going to prove this result here. The mathematics is very involved and doesn't really shed any light on the present problem. However, an intuitive understanding of why this result *should be right* can be acquired by considering the famous Galton box(bean machine).





Courtesy:http://tyulpinova.ru/normal_distribution/# (http://tyulpinova.ru/normal_distribution/)
This is the epitome of a random process(that contains no information) and the fact that it has a Gaussian distribution is confirmation(if not justification) of the fact that the Guassian distribution has the greatest entropy.

Hence we assume that the original source signals have non-Gaussian distributions.

This revelation about the nature of the pdfs of the source signals actually allows us to solve the entire problem! We reiterate that each of the recorded signals is a linear sum of the original source signals. We now invoke the **Central Limit Theorem**.

The central limit theorem says that the pdf (of the average) of the sum of independent random variables, tends to a Gaussian distribution, as the number of random variables tend to infinity. Why is this important here? Well, we have assumed that the source signals are non-Gaussian independent variables and that the recorded signals are linear weighted sums of these non-Gaussian variables. By the Central Limit Theorem, the sum of non-Gaussian variables is more Gaussian than the individual variables. Hence, the recorded signals are more Gaussian than the source signals!

We finally have a way to recover our original signals. Remember that we are trying to *linearly transform* the recorded signals back to the source signals. Thus, now we have to just find the linear transformation that minimises the "Gaussian nature" of the transformed signals. The signals that have the least "Gaussian nature" simply correspond to our source signals.

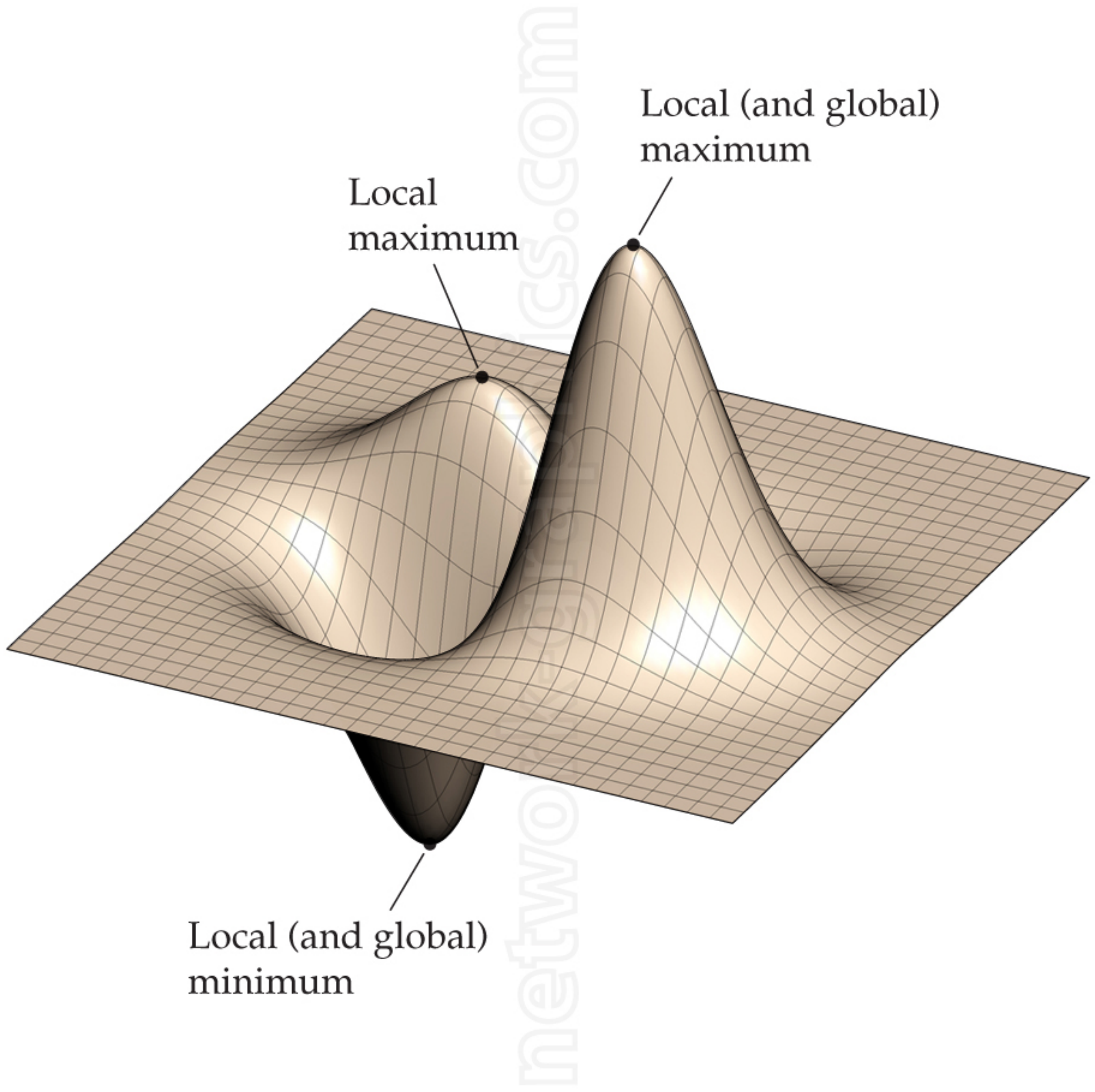
Notice that I have put "Guassian nature" in quotes. This is because we have not yet quantified deviations from a Gaussian distribution. Indeed, this is precisely where FOBI and fastICA (and other ICA implementations) differ from one another. The steps that we have outlined in the previous sections apply to both. But the subsequent sections elucidate two different approaches to quantifying deviations from a Gaussian distribution and the details of the solutions to the Cocktail Party Problem formulated based on them.

fastICA

The first method that we will be discussing is known as the fastICA which uses two important algorithms used widely in numerical analysis - *gradient descent* and *fixed-point iteration*. An understanding of both these is very important and hence we shall be discussing these before we encounter the actual algorithm. As a brief, we must estimate the matrix W such that the source matrix $s = Wx$ has minimum *'statistical dependence'*. To quantify the stastical dependence/independence, we use the concept of a *cost function*. We must choose the matrix W which minimises the cost function, and hence maximises statistical independence. Fasten your seat belts, as we dive into the world of *Machine Learning* and tackle the topics one at a time, eventually solving the Cocktail Party Problem.

Gradient Descent

Let's assume we are given a function $f(x, y)$, like the one given below, and an operation - say, to find the minima (or maxima) of the function over a domain.

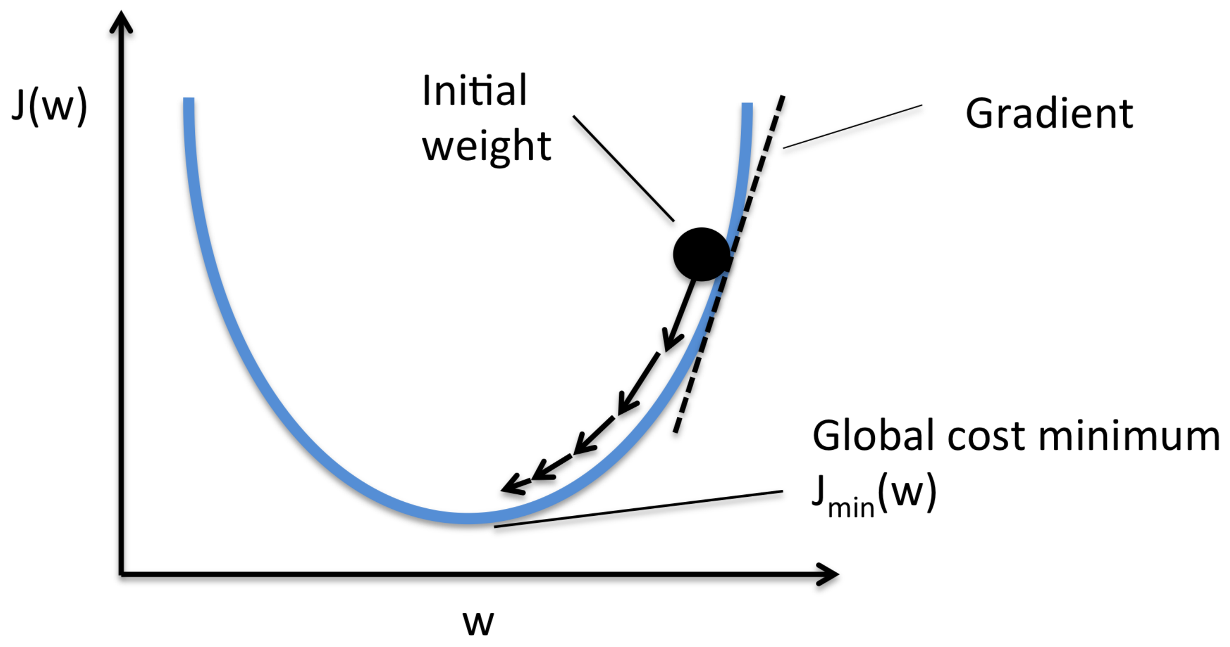


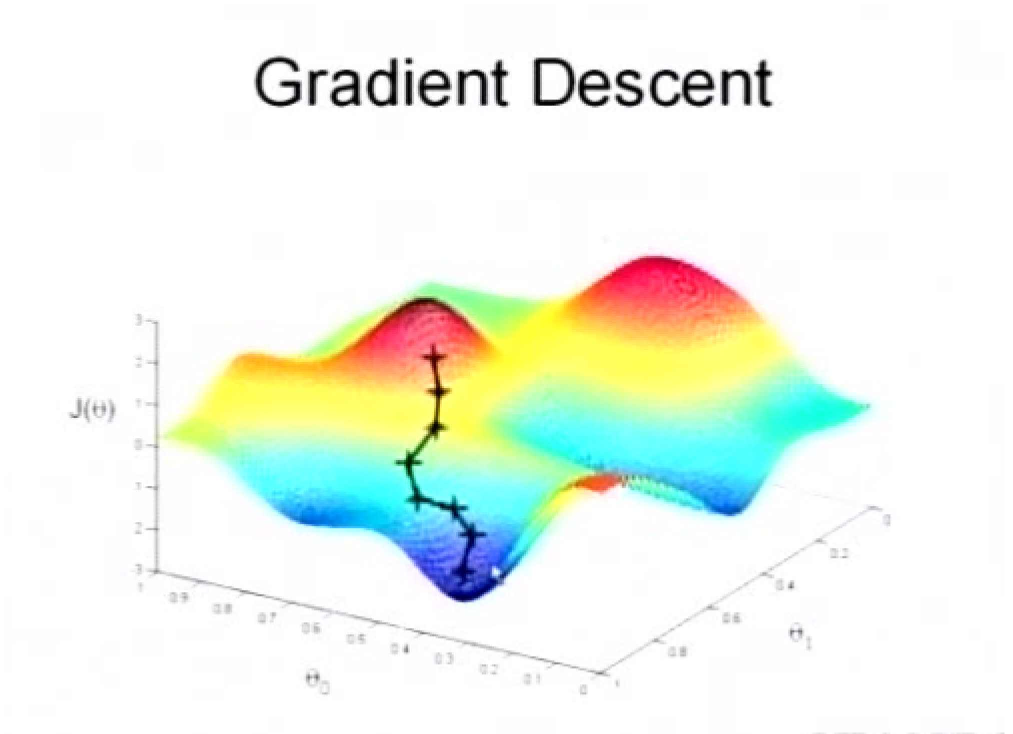
Courtesy: http://www.network-graphics.com/images/calculus (http://www.network-graphics.com/images/calculus)

With a small amount of data, known function f and just two variables (x, y) itself, the problem can become difficult to solve algebraically. Simply computing minimas of a complex function comprising of exponential and trigonometric terms can become highly complex and computationally expensive; also, the solution is dependent on the type of function we wish to optimise, and here comes the need of a more generalised algorithm.

Come in, *gradient descent*! Let us now try to intuitively form an algorithm to solve such optimisation problems without any dependence of the function f, albeit some necessary conditions. Given that the function is continuously differentiable, we can claim that a minima is the point where the function is decreasin, no matter which direction you come from (similarly, maxima is where function is increasing no matter which direction you come from). In the given illustration , given a point p_i or (x, y) in the two-dimensional plane, and its gradient $\nabla f(p_i)$, if we could somehow follow the slope of the function, we could slide down all the way to a minima (or climb up to a maxima) and the value of function f at that point would be the required optimised value.

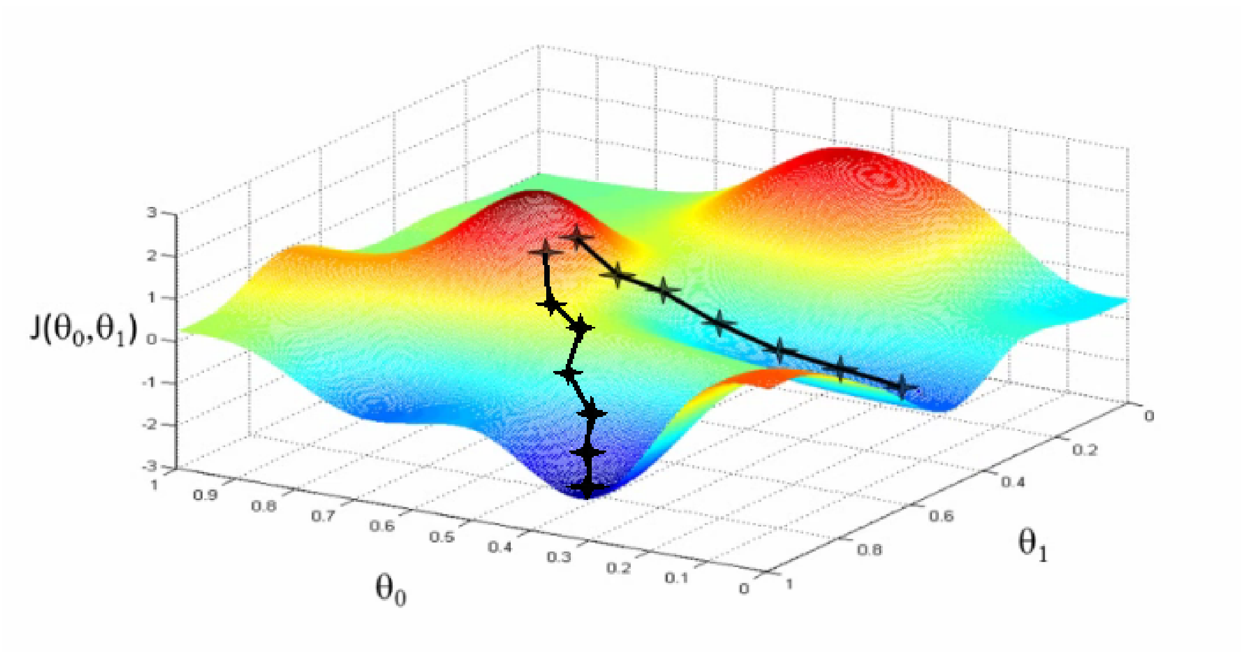
A simple way to look at this is to imagine a hill, or a complex terrain just like the illustration. Say, you are sitting at a point on the terrain and wish to reach the bottommost/topmost point in your *neighbourhood*. For the minima of potential energy, for example, you could simply let gravity do the talking and slide down, in whichever direction it takes you, until you reach *one of the* minimas in your neighbourhood. What you just actuated, is known as gradient descent!





Courtesy: Sachin Joglekar's Blog (https://codesachin.files.wordpress.com)

In practice, it involves moving a small distance in the direction of gradient (inreasing or decreasing, based on the optimisation objective), until convergence. Note that we cannot be certain of landing at the actual minima, but we end up converging in the neighbourhood. The distance is controlled by a parameter α, known as the step-size. Optimising this α is important because a large α can cause overshooting and failure in convergence, and a small α can slow down the process and make it computationally expensive. This can be tackled in two ways. *Running the algorithm multiple times* *and choosing an α that gives good results (a value of 0.3 to 2 should do fine). In fact, a good implementation of learning would be to provide the program with a range of αs, say [0.01:10] on a logarithmic scale and making the system learn the best parameter. (This would require some background in Supervised ML and hence we leave this for you to explore!)* Another smart way to do this efficiently would be to make α dependent on the error of convergence (values of f between two iteration), ideally the difference. This way, α automatically decreases as we get closer to the optimisation point. This *adaptive* model is generally a better way to implement gradient descent as it guarantees fast convergence in a simplistic way. An important feature of gradient descent is that we can extend the algorithm to any n-dimesional space. We use 2-D to visualise it because visualising higher dimesional spaces can be complicated, almost impossible. But on extending the feature vector $X(x_1, \ldots, x_n)$ and a scalar function $f(X^i)$, this algorithm works just as well. **It must be noted that of the various drawbacks gradient descent may have, the most important one is perhaps the fact that it converges to *an* optima, that is, a *local optima* and not necessarily the global optima.** This can become a major issue with complicated functions, which may have multiple possible local optima. It is also possible that the algorithm converges to different optimas on different runs, depending on the starting point.



Courtesy: www.holehouse.org

In advanced machine learning, we use more complex optimisation tools in scientific packages like SciPy, MATLAB etc. (*fminuc, fmincg etc.*) We can ignore such complexities and assume that a run of gradient descent gives us the desired optima because of our choice of *cost function* (Coming Soon!).

That is all you need to know about gradient descent for this tutorial! Feel free to explore this topic further if you are interested. Gradient Descent has a wide range of applications in numerical analysis, the most exciting of which is in Machine Learning and Artificial Neural Networks!.

Fixed-Point Iteration

Next up, we talk about another famous numerical solving technique known as the fixed-point iteration method. Given an expression $f(x) = 0$, where x can be a scalar or a vector, it is not always possible to solve it by standard methods. Consider the following equation.

$$e^{x^2} - tan(x) + x = 0$$

It is not possible to solve such an equation directly, and thus we must use a method that can allow us to find a numeric solution. In general, the given function $f(x)$ can be anything and it would be nice if the method is independent of f. Consider the example of gradient descent above. We talked about what we would have to do, but how really would you take *steps* and *move* along the gradient? This is where the need of a solving method comes in. For now, let us consider solving an equation rather than an optimisation problem.

The FPI aims at solving the problem, as the name suggests, *iteratively*. The first step is to convert an equation of the form $\mathbf{f(x) = 0}$ to the form $\mathbf{x = g(x)}$ so that we can update the parameter $\mathbf{x}$ directly. Let's tackle this with the help of an example. Let's say we want to find the roots of an equation $x^4 - x - 10 = 0$. We must first convert this into the form $x = g(x)$ to apply FPI. Some possible cases are $g_1(x) = x^4 - 10, g_2(x) = (x + 10)^{1/4}, g_3(x) = (x + 10)^{1/2}/x$ etc. It is intuitively evident that since we want convergence and run an iteration, the function $g_i(x)$ better be a converging function, and hence $g_1(x)$ is not a good function to use.

The algorithm next involves iterating upto convergence, the following expression:

$$x_{n+1} = g(x_n)$$

given the function $g(x)$. We run this iteration untill the sequence converges, i.e., $x_{n+1} - x_n < \epsilon$, where ϵ or tolerance is a small number.

An important parameter that we must consider is the extent of convergence, or tolerance ϵ. It defines how close two consecutive values of f must be, before we assume that convergence has occured. Choosing a value between 10^{-4} to 10^{-2} should work fine. Very small ϵ may take too long to converge, or not converge at all, and a large ϵ may give unsatisfactory results.

Let us look at a simple example with the function defined above, and $g(x) = (x + 10)^{1/4}$. We make an initial guess of $x_0 = 1.0$

$$x_1 = (x_0 + 10)^{1/4} = 1.82116 x_2 = (x_2 + 10)^{1/4} = 1.85424 x_2 = (x_1 + 10)^{1/4} = 1.85558 x_3 = (x_2 + 10)^{1/4} = \mathbf{1.85}$$

Similarly, starting with $x_0 = 4.0$, we get:

$$x_1 = (x_0 + 10)^{1/4} = 1.93434 x_2 = (x_2 + 10)^{1/4} = 1.85866 x_2 = (x_1 + 10)^{1/4} = 1.8557 x_3 = (x_2 + 10)^{1/4} = 1.855$$

defined as the normalized form of the fourth central moment of a distribution2

This way, we can solve an equation, indirectly, without worrying about the nature of the roots and the function itself. This is method in some very famous numerical methods like the Newton-Raphson's Method (http://www.sosmath.com/calculus/diff/der07/der07.html). Note that this method again finds *one of the roots* and can find different roots depending on seeding point.

The Cost Function

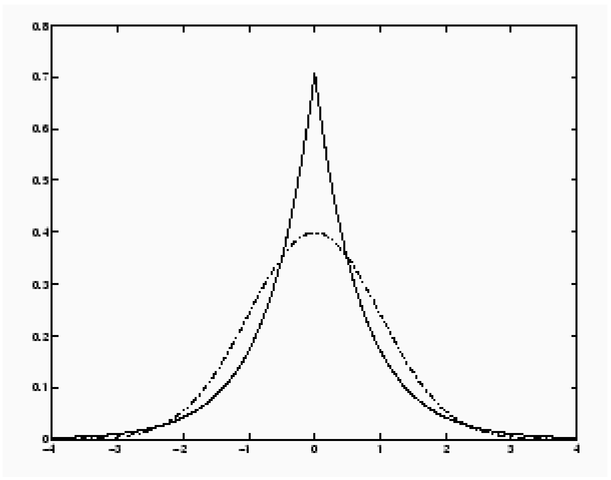
We've talked all about optimising the function and solving for its optimal value and roots etc. *But hey!* In the problem that we were discussing where is this ***function***? In this section we'll see how we can choose a function, whose optimisation gives us the desired result - separated sources.

In any optimisation problem of this sort, we talk about a term called the **cost**, which (in the case of a minimisation) is analogous to the price you are paying by deviating from ideal results. A similar analogy can be thought of for maximisation. Hence, we must quantify the property *statistical dependence* and minimise it for the sources to be best separated. *Note that the ideal case of perfect separation can generally not be attained due to various limitations*. So, how do we quantify this dependence, or *Gaussianity*, as discussed in the last topic?

- **Kurtosis** is defined as the normalized form of the fourth central moment of a distribution.

  $$kurt(x) = E(x^4) - 3(E(x^2))^2$$

  If we assume x to have zero mean $\mu_x = E\{x\} = 0$ and unit variance $\sigma_x^2 = E\{x^2\} - \mu_x^2 = 1$, then $E\{x^2\} = 1$ and $kurt(x) = E\{x^4\} - 3$. Kurtosis measures the degree of peakedness (spikiness) of a distribution and it is zero only for Gaussian distribution. Any other distribution's kurtosis is either positive if it is supergaussian (spikier than Gaussian) or negative if it is subgaussian (flatter than Gaussian). Therefore the absolute value of the kurtosis or kurtosis squared can be used to measure the non-Gaussianity of a distribution. However, kurtosis is very sensitive to outliers, and it is not a robust measurement of non-Gaussianity.

  Courtesy: Harvey Mudd College (https://www.hmc.edu/) Lectures
  Hence, maximising this *kurtosis* function can be our optimisation objective!

- **Differential Entropy *(Negentropy)***: Entropy is a very complex concept, which we wish to skip here, but as a fact, an important result in Information Theory states that the Gaussian Distribution the maximum entropy among all distributions over the entire real axis $(-\infty, \infty)$. Thus, it can be used as a measure of *Gaussianity*. For more information on this, you can refer to this site (http://fourier.eng.hmc.edu/e161/lectures/ica/node4.html).

fastICA Algorithm

Pheww! That was long! Now let's finally take a look at the fastICA algorithm in a brief manner.

1. Obtain the data matrix x.
2. Subtract off the mean to center x.
3. Whiten the matrix x to obtain the matrix x^T.
4. Choose a random guess for the initial value of the de-mixing matrix $\mathbf{w}$.
5. Iterate the following:

$$w \leftarrow E(xg(w^T x)) - E(g'(w^T x))w$$

   You will realise that this step is in fact the implementation of gradient descent using fixed-point iteration, as discussed above! The function g is a function used to capture higher order features of the matrix $w^T x$ and is similar to capturing the negentropy.

6. Normalize $\mathbf{w}$

$$w \leftarrow w/\|w\|$$

7. If not converged, go back to 2.

Convergence can be judged by the fact that an ideal w would be orthogonal, and hence $w_i^T w_{i+1}$ would ≈ 1. We thus use this condition to examine convergence.

The gradient descent step uses a function g which can be a function like $cosh(x)$ or $tanh(x)$ which capture higher order correlations and hence we finally aim to maximise the function $\Sigma E(g(y_i))$ where $y_i = \mathbf{w_i^T x}$ is a component of $\mathbf{y} = \mathbf{Wx}$.

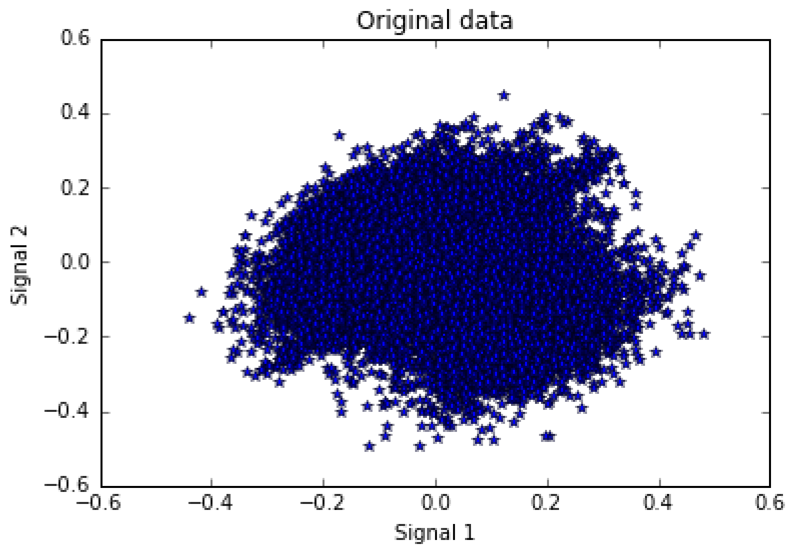
In [17]:

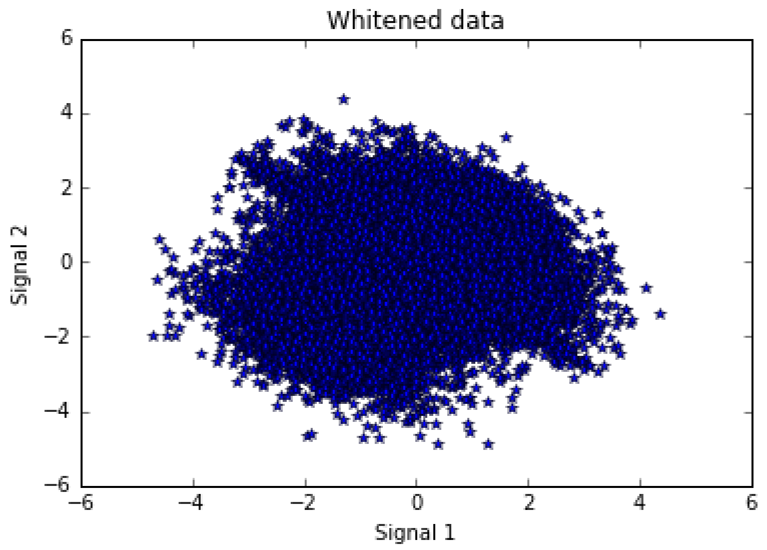
```
%matplotlib inline
"""
Cocktail Party Problem solved via Independent Component Analysis.
The fastICA algorithm is implemented here,
using negentropy as a measure of non-gaussianity.
"""
# Import packages.
import matplotlib.pyplot as plt
from scipy import signal
import numpy as np
from scipy.io import wavfile
from scipy import linalg as LA
from numpy.random import randn as RNDN

def g(x):
    out = np.tanh(x)
    return out


def dg(x):
    out = 1 - g(x) * g(x)
    return out

# Dimension
dim = 2

# Input the data from the first receiver.
samplingRate, signal1 = wavfile.read('fastICA/mic1.wav')
print "Sampling rate= ", samplingRate
print "Data type is ", signal1.dtype

# Convert the signal so that amplitude lies between 0 and 1.
# uint8 takes values from 0 through 255; sound signals are oscillatory
signal1 = signal1 / 255.0 - 0.5

# Output information about the sound samples.
a = signal1.shape
n = a[0]
print "Number of samples: ", n
n = n * 1.0

# Input data from the first receiver and standardise it's amplitude.
samplingRate, signal2 = wavfile.read('fastICA/mic2.wav')
signal2 = signal2 / 255.0 - 0.5

# x is our initial data matrix.
x = [signal1, signal2]

# Plot the signals from both sources to show correlations in the data.
plt.figure()
plt.plot(x[0], x[1], '*b')
plt.ylabel('Signal 2')
plt.xlabel('Signal 1')
plt.title("Original data")

# Calculate the covariance matrix of the initial data.
```

```
cov = np.cov(x)
# Calculate eigenvalues and eigenvectors of the covariance matrix.
d, E = LA.eigh(cov)
# Generate a diagonal matrix with the eigenvalues as diagonal elements.
D = np.diag(d)

Di = LA.sqrtm(LA.inv(D))
# Perform whitening. xn is the whitened matrix.
xn = np.dot(Di, np.dot(np.transpose(E), x))

# Plot whitened data to show new structure of the data.
plt.figure()
plt.plot(xn[0], xn[1], '*b')
plt.ylabel('Signal 2')
plt.xlabel('Signal 1')
plt.title("Whitened data")
```

```
Sampling rate=  8000
Data type is  uint8
Number of samples:  50000
```

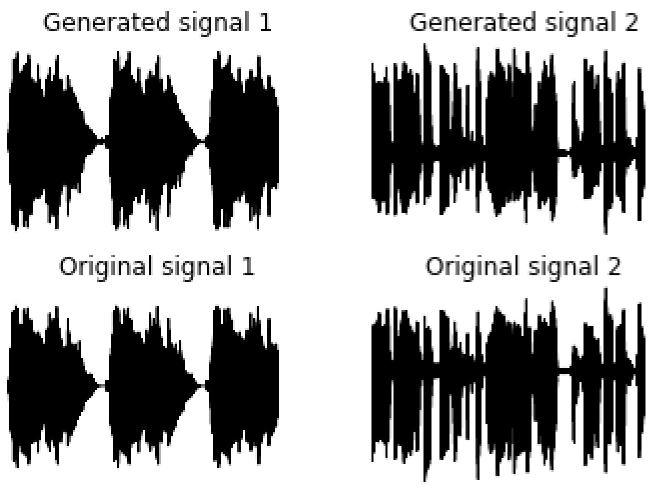
```
Out[17]: <matplotlib.text.Text at 0xdd87240>
```





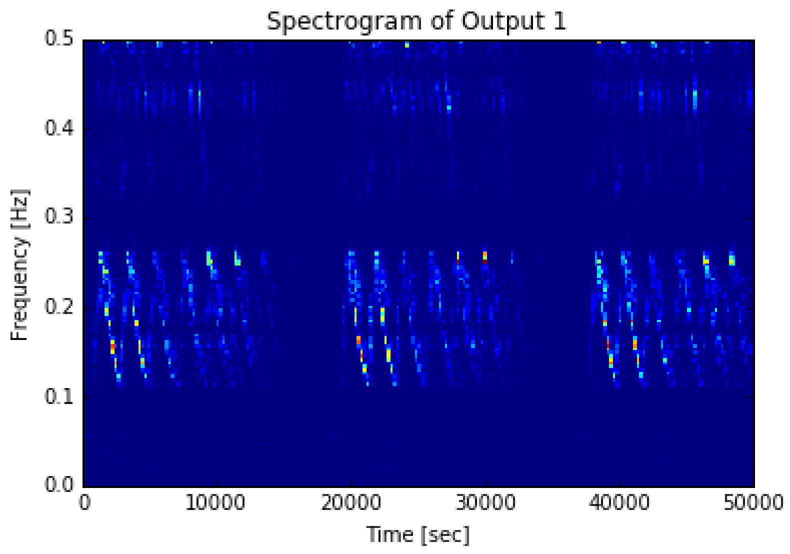
In [18]:
```
# Now that we have the appropriate signal,
# we proceed to implement fastICA on the source signal 'x'

# Creating random weight vector
w1 = RNDN(dim, 1)
w1 = w1 / LA.norm(w1)

w0 = RNDN(dim, 1)
w0 = w0 / LA.norm(w0)


# Running the fixed-point algorithm, with gradient descent
epsilon = 0.01  # Determines the extent of convergence
alpha = 1  # Step-size for gradient-descent

while (abs(abs(np.dot(np.transpose(w0), w1)) - 1) > epsilon):
    w0 = w1
    w1 = np.dot(xn, np.transpose(g(np.dot(np.transpose(w1), xn)))) / \
        n - alpha * \
        np.transpose(np.mean(np.dot(dg(np.transpose(w1)), xn), axis=1)) * w1
    w1 = w1 / LA.norm(w1)

w2 = RNDN(dim, 1)
w2 = w2 / LA.norm(w2)

w0 = RNDN(dim, 1)
w0 = w0 / LA.norm(w0)

while (abs(abs(np.dot(np.transpose(w0), w2)) - 1) > 0.01):
    w0 = w2
    w2 = np.dot(xn, np.transpose(g(np.dot(np.transpose(w2), xn)))) / \
        n - alpha * \
        np.transpose(np.mean(np.dot(dg(np.transpose(w2)), xn), axis=1)) * w2
    w2 = w2 - np.dot(np.transpose(w2), w1) * w1
    w2 = w2 / LA.norm(w2)

# Forming the source signal matrix
w = np.transpose([np.transpose(w1), np.transpose(w2)])
s = np.dot(w, x)

# Plot the separated sources.
time = np.arange(0, n, 1)
time = time / samplingRate
time = time * 1000  # convert to milliseconds

plt.figure()
plt.subplot(2, 2, 1).set_axis_off()
plt.plot(time, s[0][0], color='k')
plt.ylabel('Amplitude')
plt.xlabel('Time (ms)')
plt.title("Generated signal 1")

plt.subplot(2, 2, 2).set_axis_off()
plt.plot(time, s[1][0], color='k')
plt.ylabel('Amplitude')
plt.xlabel('Time (ms)')
plt.title("Generated signal 2")

# Plot the actual sources for comparison.
samplingRate, orig1 = wavfile.read('fastICA/source1.wav')
orig1 = orig1 / 255.0 - 0.5  # uint8 takes values from 0 to 255

plt.subplot(2,2, 3).set_axis_off()
plt.plot(time, orig1, color='k')
plt.ylabel('Amplitude')
plt.xlabel('Time (ms)')
plt.title("Original signal 1")

samplingRate, orig2 = wavfile.read('fastICA/source2.wav')
orig2 = orig2 / 255.0 - 0.5  # uint8 takes values from 0 to 255

plt.subplot(2, 2, 4).set_axis_off()
plt.plot(time, orig2, color='k')
plt.ylabel('Amplitude')
plt.xlabel('Time (ms)')
plt.title("Original signal 2")
```

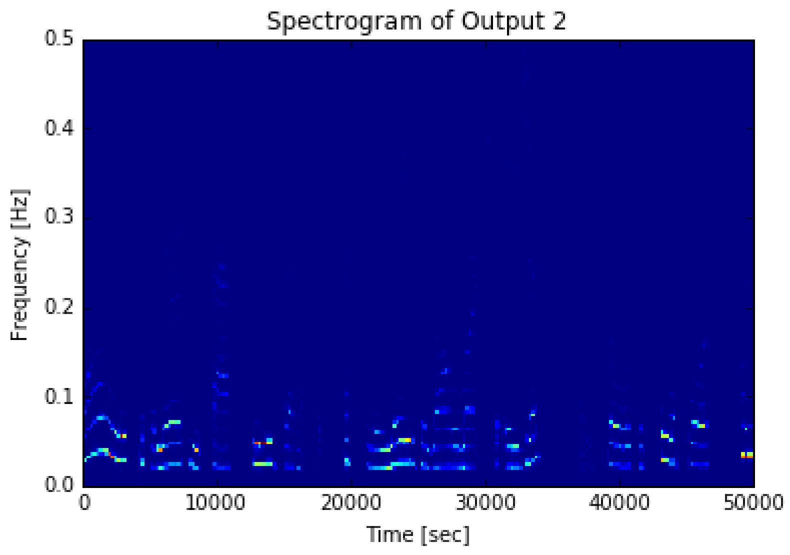
```
Out[18]: <matplotlib.text.Text at 0x9d3bf60>
```



But a good way to represent the audio files is through a spectrogram (short time fourier transform), google this if you dont know what this means. So we plot the spectrograms of our separation.

In [19]:
```
plt.figure()
f, t, S = signal.spectrogram(s[0][0])
plt.pcolormesh(t, f, S)
plt.ylabel('Frequency [Hz]')
plt.xlabel('Time [sec]')
plt.title('Spectrogram of Output 1')

plt.figure()
f, t, S = signal.spectrogram(s[1][0])
plt.pcolormesh(t, f, S)
plt.ylabel('Frequency [Hz]')
plt.xlabel('Time [sec]')
plt.title('Spectrogram of Output 2')

# Storing numpy array as audio
wavfile.write('fastICA/out1.wav', samplingRate, np.transpose(s[0][0]))
wavfile.write('fastICA/out2.wav', samplingRate, np.transpose(s[1][0]))
```





In [20]:
```
from IPython.display import Audio
print('Mixed Signal 1')
Audio("fastICA/mic1.wav")
```

```
Mixed Signal 1
```

Out[20]:

0:00

In [21]:
```
print('Mixed Signal 2')
Audio("fastICA/mic2.wav")
```

```
Mixed Signal 2
```

Out[21]:

0:00

In [22]:
```
print('Original separated signal 1')
Audio("fastICA/source1.wav")
```

```
Original separated signal 1
```

Out[22]:

0:00

In [23]:
```
print('Original separated signal 2')
Audio("fastICA/source1.wav")
```

```
Original separated signal 2
```

Out[23]:

0:00

In [24]:
```
print('Separated signal 1 (output)')
Audio("fastICA/out1.wav")
```

```
Separated signal 1 (output)
```

Out[24]:

0:00

In [25]:
```
print('Separated signal 2 (output)')
Audio("fastICA/out2.wav")
```

```
Separated signal 2 (output)
```

Out[25]:

0:00

FOBI

FOBI stands for Fourth Order Blind Identification.

As the name suggests, it is based on considering fourth order moments and using them as a metric for Gaussianity. Covariance, variance and the covariance matrix are second order moments. Fourth order moments are usually referred to by the term kurtosis. There are other ICA methods that explicitly work with kurtosis, but not so with FOBI.

FOBI is a very elegant method that works without the need to search a solution space for the indepenedent components(as is the case with fastICA and other negentropy or maximum-likelihood based ICA approaches). Rather, it provides an explicit algebraic formula for the independent components.

It has some striking and appealing similarities to PCA. FOBI is based on the following amazing fact:

The weights of the independent components are the eigenvectors of the matrix $cov(|X^{'}|X^{'})$.

The statement requires some explanation. $X^{'}$ is the matrix obtained after preprocessing X (mean-centered and whitened).

$|X^{'}|$ needs to be explained. Remember that each column of $X^{'}$ corresponds to the (whitened) set of amplitudes recorded at some instant of time by the **m** microphones. If we take this column to be a vector, then the norm (aka the modulus) of this vector is well defined. We find the norm of each column of **X'** and put these calculated norms in a row matrix. This is the matrix $|X^{'}|$. Hence, $|X^{'}|$ is simply the norm of the matrix X taken along the "column-axis".

But here comes a caveat: The matrix $|X^{'}|X^{'}$ is not formed by the normal matrix multiplication between $|X^{'}|$ and $X^{'}$. We have written it this way only for notational convnenience. Instead, we are using the elements of $|X^{'}|$ as weights for the columns of $X^{'}$. That is, the matrix $|X^{'}|X^{'}$ is formed as follows:

The **i**th column of $|X^{'}|X^{'}$ is formed by a scalar multiplication of the **i**th element of the **1 x m** row matrix $|X^{'}|$ with the **i**th column of the **m x n** matrix $X^{'}$ (each element within this column is thus multiplied by the same scalar value equal to the **i**th element of $|X^{'}$).

Hence, $|X^{'}|X^{'}$ is a weighted version of $X^{'}$. It is easy to think of each column of $X^{'}$ as a vector in an m-dimensional space. Then $|X^{'}|$ contains the norms of the **m** columns of $X^{'}$ i.e vectors. Multiplying each vector (column) by it's norm and storing these column vectors as columns in a new matrix yields $|X^{'}|X^{'}$.

And what does the statement itself mean? Well, remember how we said that $X = AS$. After whitening, we can say that $X^{'} = A^{'}S^{'}$. Here, $S^{'}$ corresponds to the source signals itself, but their amplitude has been normalised. (Recall how ICA can only recover the signals upto to a multiplicative factor.)

The columns of $A^{'}$ are the weights being referred to in the statement above. They are called weights because the each row of $S^{'}$ is a source signal. Upon performing matrix multiplication between $A^{'}$ and $S^{'}$, we can see that $X^{'} = \sum_{1}^{m} A_{i}^{'} S_{i}^{'}$. Note that $A_{i}^{'}$ are column vectors and $S_{i}^{'}$ are row vectors. Hence, the columns of $A^{'}$ act as weights for the source signals.

Hence, FOBI allows us to directly compute the matrix $A^{'}$ which as expected, turns out to be orthogonal. (The result says that the columns of $A^{'}$ are orthogonal eigenvectors of the matrix $cov(|X^{'}|X^{'})$. Hence, $A^{'}$ is an orthogonal matrix.) Finding the signals is now trivial. $S^{'} = A^{'-1}X^{'} = A^{'T}X^{'}$. (The inverse of an orthogonal matrix is simply it's transpose.)

Recall that in PCA, the principal components were eigenvectors of the matrix $cov(X)$ and they were all orthogonal. Here, the columns of $A^{'}$, which are the weights for the source signals are the eigenvectors of the matrix $cov(|X^{'}|X^{'})$. And the weights are orthogonal to one another!

The proof of this result and a good explanation of the FOBI algorithm can be found in the original paper by the inventor of the FOBI algorithm Jean Cardoso, here (http://perso.telecom-paristech.fr/~Cardoso/Papers.PDF/icassp89.pdf). The notation used in the paper is rather different from the one that we have been using so far and can be rather confusing. The proof itself is however extremely elegant.

Also, refer to the paper to better understand why it involves fourth order moments. Basically, if $X^{'}$ were a simple vector, then $cov(|X^{'}|X^{'})$ is fourth order in $X^{'}$. This extends to $X^{'}$ being a matrix.

So the FOBI algorithm in summary is:

1) Obtain the data matrix *X*.

2) Subtract off the mean to center *X*.

3) Whiten the matrix *X* to obtain the matrix *X'*.

4) Compute the matrix $cov(|X^{'}|X^{'})$.

5) Find the eigenvectors of $cov(|X^{'}|X^{'})$.

6) Store the eigenvectors as columns of a matrix Y. (Y is just $A^{'}$)

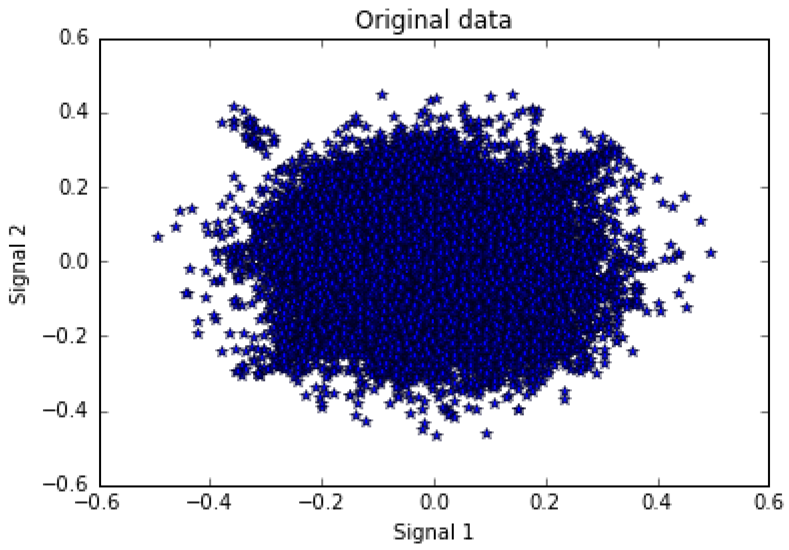
7) Retrieve the original signals as $S^{'} = Y^{T}X^{'}$.

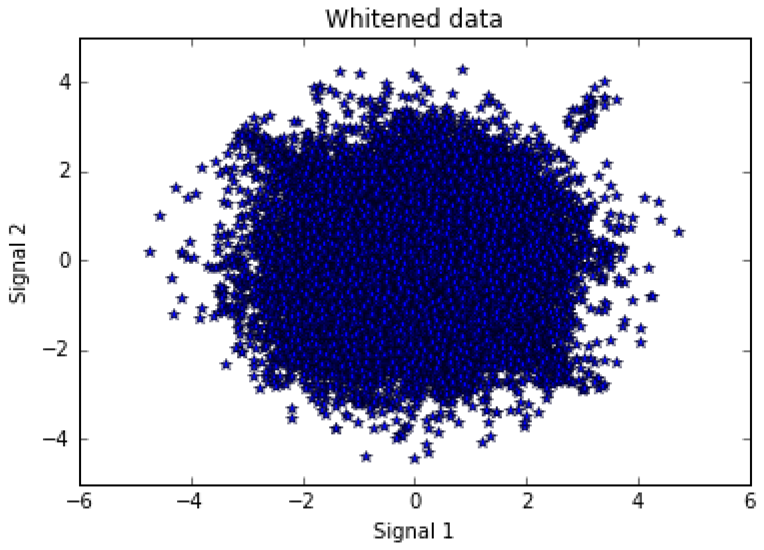
The code for the solvong the Cocktail Party Problem based on the FOBI algorithm is given below. In the code, we have used slightly different notation in some places. For instance, x for X and xn for X'.

In [26]:
```
%matplotlib inline
"""
Cocktail Party Problem solved via Independent Component Analysis.
The Fourth Order Blind Identification(FOBI) ICA is implemented here.
"""
# Import packages.
import matplotlib.pyplot as plt
from scipy import signal
import numpy as np
from scipy.io import wavfile
from scipy import linalg as LA

# Input the data from the first receiver.
samplingRate, signal1 = wavfile.read('FOBI/Sounds/mix1.wav')
print "Sampling rate = ", samplingRate
print "Data type is ", signal1.dtype

# Convert the signal so that amplitude lies between 0 and 1.
signal1 = signal1 / 255.0 - 0.5  # uint8 takes values from 0 to 255

# Output information about the sound samples.
a = signal1.shape
n = a[0]
print "Number of samples: ", n
n = n * 1.0

# Input data from the second receiver and standardise it's amplitude.
samplingRate, signal2 = wavfile.read('FOBI/Sounds/mix2.wav')
signal2 = signal2 / 255.0 - 0.5  # uint8 takes values from 0 to 255

# x is our initial data matrix.
x = [signal1, signal2]

# Plot the signals from both sources to show correlations in the data.
plt.figure()
plt.plot(x[0], x[1], '*b')
plt.ylabel('Signal 2')
plt.xlabel('Signal 1')
plt.title("Original data")

# Calculate the covariance matrix of the initial data.
cov = np.cov(x)
# Calculate eigenvalues and eigenvectors of the covariance matrix.
d, E = LA.eigh(cov)
# Generate a diagonal matrix with the eigenvalues as diagonal elements.
D = np.diag(d)

Di = LA.sqrtm(LA.inv(D))
# Perform whitening. xn is the whitened matrix.
xn = np.dot(Di, np.dot(np.transpose(E), x))

# Plot whitened data to show new structure of the data.
plt.figure()
plt.plot(xn[0], xn[1], '*b')
plt.ylabel('Signal 2')
plt.xlabel('Signal 1')
plt.title("Whitened data")
```

```
Sampling rate =  8000
Data type is  uint8
Number of samples:  50000
```

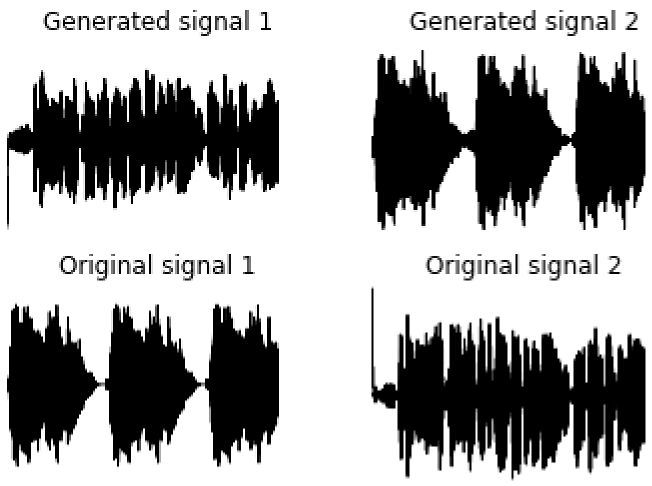
```
Out[26]: <matplotlib.text.Text at 0x11131978>
```





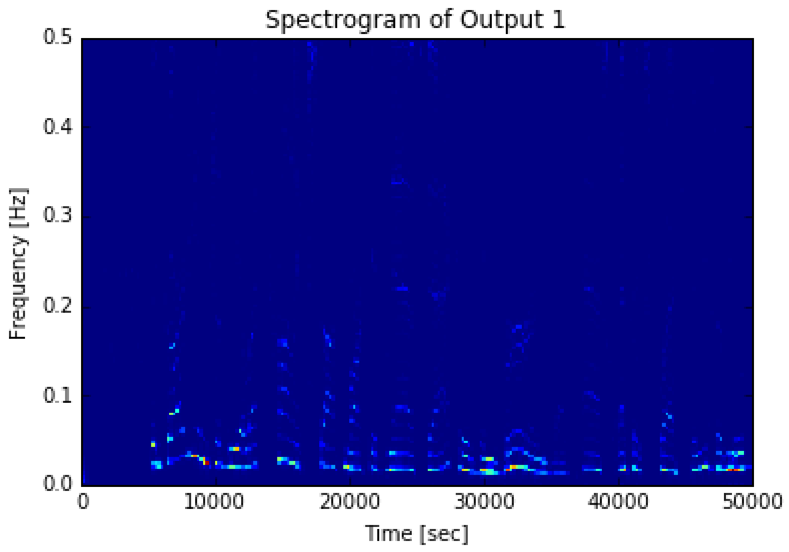
In [27]:
```
# Perform FOBI.
norm_xn = LA.norm(xn, axis=0)
norm = [norm_xn, norm_xn]

cov2 = np.cov(np.multiply(norm, xn))

d_n, Y = LA.eigh(cov2)

source = np.dot(np.transpose(Y), xn)

# Plot the separated sources.
time = np.arange(0, n, 1)
time = time / samplingRate
time = time * 1000  # convert to milliseconds

plt.figure()
plt.subplot(2, 2, 1).set_axis_off()
plt.plot(time, source[0], color='k')
plt.ylabel('Amplitude')
plt.xlabel('Time (ms)')
plt.title("Generated signal 1")

plt.subplot(2, 2, 2).set_axis_off()
plt.plot(time, source[1], color='k')
plt.ylabel('Amplitude')
plt.xlabel('Time (ms)')
plt.title("Generated signal 2")

# Plot the actual sources for comparison.
samplingRate, orig1 = wavfile.read('FOBI/Sounds/source1.wav')
orig1 = orig1 / 255.0 - 0.5  # uint8 takes values from 0 to 255

plt.subplot(2, 2, 3).set_axis_off()
plt.plot(time, orig1, color='k')
plt.ylabel('Amplitude')
plt.xlabel('Time (ms)')
plt.title("Original signal 1")

samplingRate, orig2 = wavfile.read('FOBI/Sounds/source2.wav')
orig2 = orig2 / 255.0 - 0.5  # uint8 takes values from 0 to 255

plt.subplot(2, 2, 4).set_axis_off()
plt.plot(time, orig2, color='k')
plt.ylabel('Amplitude')
plt.xlabel('Time (ms)')
plt.title("Original signal 2")
```

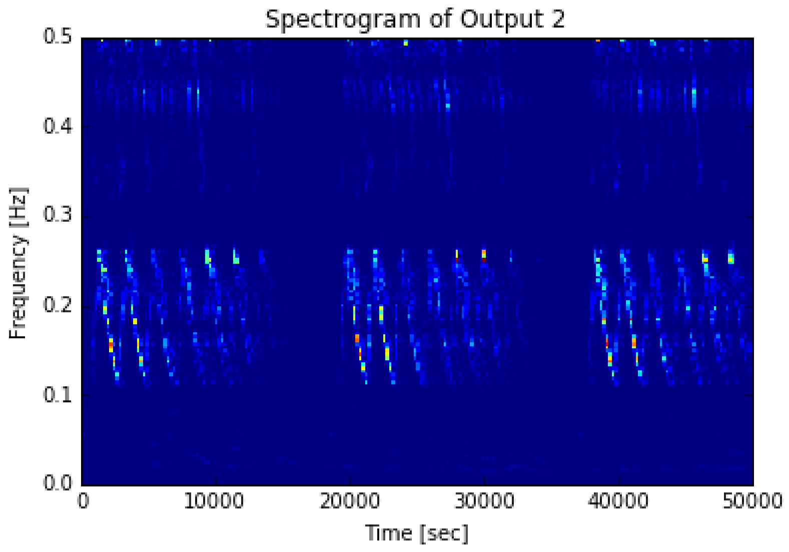
Out[27]:
```
<matplotlib.text.Text at 0x11087cf8>
```



Again we plot the spectrograms

In [28]:
```
plt.figure()
f, t, S = signal.spectrogram(source[0])
plt.pcolormesh(t, f, S)
plt.ylabel('Frequency [Hz]')
plt.xlabel('Time [sec]')
plt.title('Spectrogram of Output 1')

plt.figure()
f, t, S = signal.spectrogram(source[1])
plt.pcolormesh(t, f, S)
plt.ylabel('Frequency [Hz]')
plt.xlabel('Time [sec]')
plt.title('Spectrogram of Output 2')

# Storing numpy array as audio
wavfile.write('FOBI/Sounds/out1.wav', samplingRate, np.transpose(source[0]))
wavfile.write('FOBI/Sounds/out2.wav', samplingRate, np.transpose(source[1]))
```





In [29]:
```
from IPython.display import Audio
print('Mixed Signal 1')
Audio("FOBI/Sounds/mix1.wav")
```

```
Mixed Signal 1
```

Out[29]:

0:00

In [30]:
```
print('Mixed Signal 2')
Audio("FOBI/Sounds/mix2.wav")
```

```
Mixed Signal 2
```

Out[30]:

0:00

In [31]:
```
print('Original separated sound 1')
Audio("FOBI/Sounds/source1.wav")
```

```
Original separated sound 1
```

Out[31]:

0:00

In [35]:
```
print('Original separated sound 2')
Audio("FOBI/Sounds/source2.wav")
```

```
Original separated sound 2
```

Out[35]:

0:00

In [36]:
```
print('Separated signal 1 (output)')
Audio("FOBI/Sounds/out1.wav")
```

```
Separated signal 1 (output)
```

Out[36]:

0:00

In [37]:
```
print('Separated signal 1 (output)')
Audio("FOBI/Sounds/out2.wav")
```

```
Separated signal 1 (output)
```

Out[37]:

0:00

fastICA v/s FOBI

The comparison of different implementations of ICA is very difficult as there are few parameters available to quantify and compare performance, and even then no one algorithm performs better unilaterally in all scenarios. Such metrics are usually thus compiled for data from highly specific domains.

However, some broad conclusions have been drawn based on the asymptotic behaviour of algorithms.

Alogrithms can be categorised on the basis of how of how Gaussianity is quantified.

fastICA is based on negentropy. Maximum-likelihood and negenrtropy are considered optimal in their statistical properties. However, nengentropy is very computationally expensive to compute if the actual formula is used without any approximations. Hence, in practice, approximate formulae for negentropy are used. These approximations are a compromise between computational speed and statistical properties.

FOBI does not explicitly use a cost function at all. It provides a purely algebraic method of implementing ICA. Yet, it's statistical properties are essentially that of Kurtosis and it implicitly uses fourth order moments. Kurtosis suffers from non-optimal dependence on outliers. FOBI in particular also involves a lot of matrix computations that might not scale well with the size of the data.

Limitations of ICA

ICA generates the independent components upto a scaling factor. This means that the output can also be 'flipped' due to scaling by a negative number. This does not matter when it comes to sound signals but may be a problem to be rectified in other applications of ICA.

ICA works very well when there is no noise. Noisy ICA methods have been developed that assume a Gaussian pdf for the noise. This is a reasonabel assumption to make in most cases but does not hold always hold.

Feature extraction using ICA may not give the best results. This is true of face recognition in particular. Real images have a lot of variations in parameters that don't really matter in feature extraction. Like brightness. ICA is not 'smart' and will identify brightness also as an Independent component. Also, in higher dimensional data like faces, independent components may not form the best basis of understanding data. For face-recognition, due to the limitations of ICA, Fischer Linear Discriminant Analysis(LDA) is used instead.

ICA works on linear models. Extending ICA to non-linear systems is much harder to do and is yet to have been solved for a general non-linear system.

Often, the components estimated from data by an ICA algorithm are not independent. While the components are assumed to be independent in the model, the model does not have enough parameters to actually make the components independent for any given random vector **X**. This is because statistical independence is a very strong property with potentially an infinite number of degrees of freedom.

Empirical results tend to show that ICA estimation seems to be rather robust against some violations of the independence assumption. Modelling dependencies of the estimated components is an important extension of the analysis provided by ICA. It can give useful information on the interactions between the components or sources recovered by ICA. Thus, the fact that the components are dependent can be a great opportunity for gaining further insights into the structure of the data.

Conclusion

It is fair to say that ICA has become a standard tool in machine learning and signal processing. The generality and potential usefulness of the model were never in question, but in the early days of ICA, there was some doubt about the adequacy of the assumptions of non-Gaussianity and independence.

It has been realized that non-Gaussianity is in fact quite widespread in any application dealing with scientific measurement devices (as opposed to, for example, data in the social and human sciences). On the other hand, independence is now being seen as a useful approximation that is hardly ever strictly true. Fortunately, it does not need to be strictly true because most ICA methods are relatively robust regarding some dependence of the components.

References

A notebook by many sources, mainly ***Shashwat Shukla*** and ***Dhruv Ilesh Shah*** and below:

A great tutorial on PCA:

https://arxiv.org/pdf/1404.1100.pdf (https://arxiv.org/pdf/1404.1100.pdf)

Quick explanation of ICA:

http://research.ics.aalto.fi/ica/icademo/ (http://research.ics.aalto.fi/ica/icademo/)

Comprehensive introductions to ICA:

http://www.dsp.pub.ro/articles/spie/ICA_course.PDF (http://www.dsp.pub.ro/articles/spie/ICA_course.PDF)

http://www.bsp.brain.riken.jp/ICApub/NN00.pdf (http://www.bsp.brain.riken.jp/ICApub/NN00.pdf)

ICA implemented using Neural Networks:

https://www.cs.purdue.edu/homes/dgleich/projects/pca_neural_nets_website/ (https://www.cs.purdue.edu/homes/dgleich/projects/pca_neural_nets_website/)

http://shulgadim.blogspot.in/search/label/DSP (http://shulgadim.blogspot.in/search/label/DSP)

```
In [ ]:
```