CSCD 539/599 Scientific Computing with Matlab/Octave
Project: Projection Pursuit

Please do your own work on this project. I encourage you to ask any questions that arise in class so that others can benefit from the discussion. Your task is to implement a Projection Pursuit algorithm that is able to separate mixtures of sound files into the sources. I will provide you with a set of mixed signals in an m x n data file and will demonstrate the un-mixing that is expected in class, along with details on how to proceed. Vectors containing sound samples may be played back in either Octave or Matlab using the soundsc() function. The sample frequency of the sound files I provide will be recorded in the .mat file in a variable called Fs.

As discussed in class, Principal Component Analysis suffers from the restrictive requirement that it is only able to separate mixtures onto orthogonal component axes. Another way to state this is that it is only able to remove first-order, or linear, dependencies amongst the data variables. Yet another way to state this is that the output variables it produces are de-correlated, but not statistically independent.

There is a class of algorithms called ICA (Independent Component Analysis) that can separate signal mixtures into Independent (as opposed to Principal) components. Projection Pursuit is a closely related approach that is somewhat easier to understand. It searches for components by iteratively maximizing the (numerator of) the Kurtosis of the unmixed signals. The underlying idea is that the Central Limit Theorem ensures that the distribution (histogram) of a mixture of signals is more Gaussian than the distribution of the original signals. This does not guarantee that the source signals themselves are non-Gaussian, but, in nature, signal sources tend to be non-Gaussian, and more specifically, super-Gaussian (peaky). It has been proven that ICA and Projection Pursuit will produce the same results if the signal sources are independent and super-Gaussian (and Projection Pursuit can also be easily extended to search for sub-Gaussian signals if desired). When the sources are not independent, some differences between the Projection Pursuit and ICA will occur because they pursue different optimization measures. Some algorithmic details of Projection Pursuit are:

1) We assume that the signals we are seeking to separate from the mixtures are independent. If they are independent, then they are also uncorrelated. Thus you should start by casting the data onto a basis system that de-correlates the variables (i.e., PCA). This will not, in general, produce the signals we seek, but makes it possible to extract components, as we find them, via orthogonal projection. Once you have your code working, I encourage you to try skipping this step to see what happens. Recall that the principal components can easily be obtained from the left or right singular vectors of an SVD of the data matrix, depending on the orientation of the data matrix. After producing your de-correlated dataset, scale each mixture (row) to have unit variance. Note that you do not typically use the var() function to scale a signal to unit variance.
2) I will henceforth refer to the signal mixtures as (matrix) x, to the de-correlated mixtures as z, and to the estimated (unmixed) signals as y. The signal mixtures that I give you will be (m x n), where m is the number of dimensions (signals = variables = microphones) and n is the number of measurements taken (samples of the variables, which in this case happen to be time series samples from each microphone). You should write your code such that it can be run on an arbitrary number of dimensions. I do intend to test it for a different number of mixes (m) than I give you.
3) For each dimension, m, generate a random un-mixing vector (trial axis), which I will refer to as w. Make sure it has unit norm so that projecting z onto it doesn't change the scaling. Then refine that guess iteratively, as follows:
    a. Compute a trial projection onto w. I call the result y. It has dimension (1 x n) because we are un-mixing one dimension at a time.
    b. Measure the numerator of the kurtosis in y. If you scaled the rows of z to unit variance, and w has unit magnitude, then any projection onto w should also have unit variance. Note that you are required to store the history of your kurtosis climb, for each dimension, into an output variable, so that the caller of your function can track the convergence of the algorithm and possibly tune some of the parameters as a result. I defined an anonymous function to calculate the numerator of the kurtosis, and henceforth refer to that function as K. Using the built-in kurtosis function of Matlab might get you into trouble, so I recommend avoiding that.
    c. Either compute or estimate the direction of the gradient in K with respect to w (delta K/delta w). If you estimate it, do so by probing K for small steps in the components of w, as discussed in class. The step size for this probing will be a parameter to your function that I call h. Thus, for each dimension of w, you want to add h to that component (and only that component) of w, then recalculate y and K(y). The change in K, divided by h, gives you an estimate of the component of the gradient in that direction. When you do this for all dimensions of w, you have an estimate of the gradient (which is a vector).
    d. Then modify w by one step in the direction of the gradient. This is done by adding some fraction of the gradient to w. That fraction will be defined by another parameter that I call eta. Thus w will get its current value plus eta times the gradient vector. Force w to unit norm again, so that y will continue to have unit variance.

e. You have now taken one step uphill on K. Keep iterating on the refinement of this w until the *relative* change in K is less than the parameter called tol.
f. Once you are no longer significantly improving K, you can save the most recent estimate of y for passing back out of the function.
g. Before moving on to the next dimension, you need to subtract that y from the signal set in z. Note that y is a 1-dimensional un-mixing of multiple signals (m x n) in z, obtained by scalar projection of each signal sample. In order to subtract that from z you need to make sure that y is expressed in the multidimensional basis that defines the z-space. This is simply scalar projection vs. vector projection.
h. If you look at the SVD or the covariance matrix after subtracting out your signal, you should be able to see that the rank is reduced by one. That means that you could reduce the dimensionality of z by one at this point, but you should also be able to safely skip that.

4) Your projection pursuit must be implemented in an m-file function with the following signature. If you use an analytic expression for the gradient, simply ignore h, but it must still appear in the parameter list for my testing:

```
function [y, K] = ppursuit(h, eta, tol, mxi, x)
%
% This function uses projection pursuit to demix m
% signal mixtures into m estimated source signals.
%
% h   Step size for probing the Kurtosis (K) of the
%     demixed signals. K is probed by looking at
%     K(w + h) in the m dimensions of w, one at a time.
% eta Distance that the demixing vector w is adjusted
%     in the direction of the estimated gradient of
%     K. IOW, wnext = w + eta*g where g is the
%     (perhaps estimated) gradient of K.
% tol Stopping criterion for gradient ascent. The
%     ascent terminates when the relative change in
%     abs(K) is < tol (abs change in K divided by K).
% mxi Maximum iterations to execute on the ascent
%     of K for each recovered signal.
% x   (m x n) matrix of signal mixtures. Each row
%     is a 1 x n mixture of m source signals. The
%     source signals may be anything, and there is
%     no assumed relationship between the n samples
%     of a given mixture.
% y   The (m x n) source estimates passed out.
% K   The (m x ?) history of the ascent of Kurtosis
%     for each recovered source. Having this history
%     helps the user tune the search parameters.
```