Lab 1: Independent Component Analysis

Machine Learning: Principles and Methods, November 2013

- The lab exercises should be made in groups of three people, or at least two people.
- The deadline is Sunday, Nov 10, 23:59.
- Assignment should be sent to T.S.Cohen at uva dot nl (Taco Cohen). The subject line of your email should be "[MLPM2013] lab#_lastname1_lastname2_lastname3".
- Put your and your teammates' names in the body of the email
- Attach the .IPYNB (IPython Notebook) file containing your code and answers. Naming of the file follows the same rule as the subject line. For example, if the subject line is "[MLPM2013] lab01_Kingma_Hu", the attached file should be "lab01_Kingma_Hu.ipynb". Only use underscores ("_") to connect names, otherwise the files cannot be parsed.

Notes on implementation:

- You should write your code and answers in an IPython Notebook: http://ipython.org/notebook.html. If you have problems, please contact us.
- Among the first lines of your notebook should be "%pylab inline". This imports all required modules, and your plots will appear inline.
- NOTE: test your code and make sure we can run your notebook / scripts!

Literature

In this assignment, we will implement the Independent Component Analysis algorithm as described in chapter 34 of David MacKay's book "Information Theory, Inference, and Learning Algorithms", which is freely available here: http://www.inference.phy.cam.ac.uk/mackay/itila/book.html

Read the ICA chapter carefuly before you continue!

Notation

$\mathbf{X}$ is the $M \times T$ data matrix, containing M measurements at T time steps.

$\mathbf{S}$ is the $S \times T$ source matrix, containing S source signal values at T time steps. We will assume $S = M$.

$\mathbf{A}$ is the mixing matrix. We have $\mathbf{X} = \mathbf{AS}$.

$\mathbf{W}$ is the matrix we aim to learn. It is the inverse of $\mathbf{A}$, up to indeterminacies (scaling and permutation of sources).

ϕ is an elementwise non-linearity or activation function, typically applied to elements of $\mathbf{WX}$.

Code

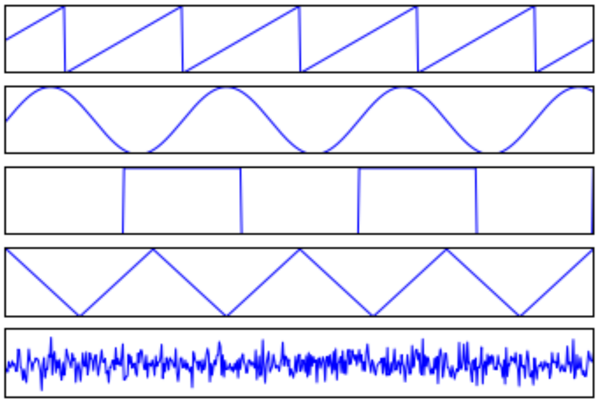
In the following assignments, you can make use of the signal generators listed below.

In [1]:
```
%pylab inline

# Signal generators
def sawtooth(x, period=0.2, amp=1.0, phase=0.):
    return (((x / period - phase - 0.5) % 1) - 0.5) * 2 * amp

def sine_wave(x, period=0.2, amp=1.0, phase=0.):
    return np.sin((x / period - phase) * 2 * np.pi) * amp

def square_wave(x, period=0.2, amp=1.0, phase=0.):
    return ((np.floor(2 * x / period - 2 * phase - 1) % 2 == 0).astype(float) - 0.5)

def triangle_wave(x, period=0.2, amp=1.0, phase=0.):
    return (sawtooth(x, period, 1., phase) * square_wave(x, period, 1., phase) + 0.5

def random_nonsingular_matrix(d=2):
    """
    Generates a random nonsingular (invertible) matrix if shape d*d
    """
    epsilon = 0.1
    A = np.random.rand(d, d)
    while abs(np.linalg.det(A)) < epsilon:
        A = np.random.rand(d, d)
    return A

def plot_signals(X):
    """
    Plot the signals contained in the rows of X.
    """
    figure()
    for i in range(X.shape[0]):
        ax = plt.subplot(X.shape[0], 1, i + 1)
        plot(X[i, :])
        ax.set_xticks([])
        ax.set_yticks([])
```

```
Welcome to pylab, a matplotlib-based Python environment [backend:
module://IPython.zmq.pylab.backend_inline].
For more information, type 'help(pylab)'.
```

The following code generates some toy data to work with.

```
In [2]: # Generate data
        num_sources = 5
        signal_length = 500
        t = linspace(0, 1, signal_length)
        S = np.c_[sawtooth(t), sine_wave(t, 0.3), square_wave(t, 0.4), triangle_wave(t, 0.25

        plot_signals(S)
```



1.1 Make mixtures (5 points)

Write a function `make_mixtures(S, A)' that takes a matrix of source signals $\mathbf{S}$ and a mixing matrix $\mathbf{A}$, and generates mixed signals $\mathbf{X}$.

1.2 Histogram (5 points)

Write a function `plot_histograms(X)` that takes a data-matrix $\mathbf{X}$ and plots one histogram for each signal (row) in $\mathbf{X}$. You can use the numpy `histogram()` function.

Plot histograms of the sources and the measurements. Which of these distributions (sources or measurements) tend to look more like Gaussians? Why is this important for ICA? Can you think of an explanation for this phenomenon?

1.3 Implicit priors (20 points)

As explained in MacKay's book, an activation function ϕ used in the ICA learning algorithm corresponds to a prior distribution over sources. Specifically, $\phi(a) = \frac{d}{da} \ln p(a)$. For each of the following activation functions, derive the source distribution they correspond to.

$\phi_0(a) = -\tanh(a)$
$\phi_1(a) = -a + \tanh(a)$
$\phi_2(a) = -a^3$
$\phi_3(a) = -\frac{6a}{a^2 + 5}$

The normalizing constant is not required, so an answer of the form $p(a) \propto$ `[answer]`is ok.

Plot the activation functions and the corresponding prior distributions. Compare the shape of the priors to the histogram you plotted in the last question.

1.4 Whitening (15 points)

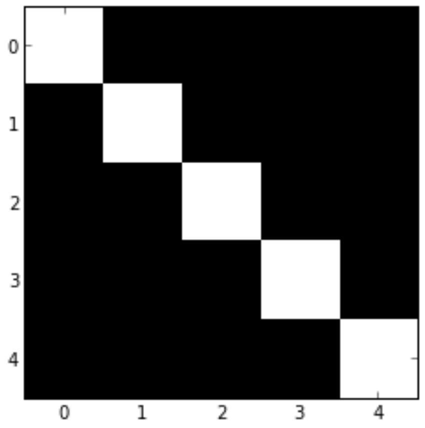
Some ICA algorithms can only learn from whitened data. Write a method `whiten(X)` that takes a $M \times T$ data matrix $\mathbf{X}$ (where M is the dimensionality and T the number of examples) and returns a whitened matrix. If you forgot what whitening is or how to compute it, various good sources are available online, such as http://courses.media.mit.edu/2010fall/mas622j/whiten.pdf

1.5 Interpret results of whitening (10 points)

Make scatter plots of the sources, measurements and whitened measurements. Each axis represents a source/measurement and each time-instance is plotted as a dot in this space. You can use the `np.scatter()` function. Describe what you see.

Now compute the covariance matrix of the sources, the measurements and the whitened measurements. You can visualize a covariance matrix using the line of code below. Are the signals independent after whitening?

In [3]:
```
C = np.eye(5)  # Dummy matrix; compute covariance here
ax = imshow(C, cmap='gray', interpolation='nearest')
```



1.6 Covariance (5 points)

Explain what a covariant algorithm is.

1.7 Independent Component Analysis (25 points)

Implement the covariant ICA algorithm as described in MacKay. Write a function `ICA(X, activation_function, learning_rate)`, that returns the demixing matrix $\mathbf{W}$. The input `activation_function` should accept a function such as `lambda a: -tanh(a)`. Update the gradient in batch mode, averaging the gradients over the whole dataset for each update. Try to make it efficient, i.e. use matrix operations instead of loops where possible (loops are slow in interpreted languages such as python and matlab, whereas matrix operations are internally computed using fast C code).

1.8 Experiments (5 points)

Run ICA on the provided signals using each activation function $\phi_0, \ldots, \phi_3$. Plot the retreived signals for each choice of activation function.

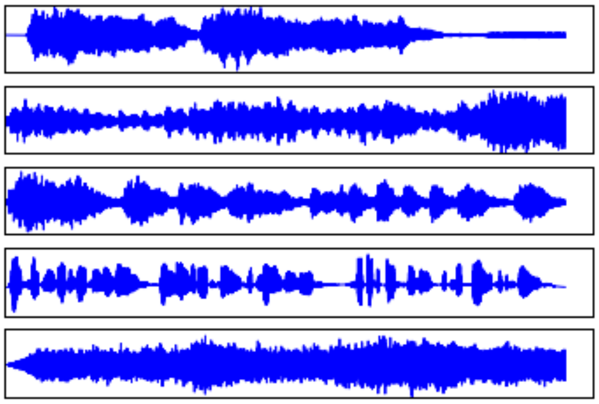
1.9 Audio demixing (5 points)

The 'cocktail party effect' refers to the ability humans have to attend to one speaker in a noisy room. We will now use ICA to solve a similar but somewhat idealized version of this problem. The code below loads 5 sound files and produces 5 mixed sound files, which are saved to disk so you can listen to them. Use your ICA implementation to de-mix these and reproduce the original source signals. As in the previous exercise, try each of the activation functions and report your results.

Keep in mind that this problem is easier than the real cocktail party problem, because in real life there are often more sources than measurements (we have only two ears!), and the number of sources is unknown and variable. Also, mixing is not instantaneous in real life, because the sound from one source arrives at each ear at a different point in time. If you have time left, you can think ways to deal with these issues.

In [3]:
```
import scipy.io.wavfile
def save_wav(data, out_file, rate):
    scaled = np.int16(data / np.max(np.abs(data)) * 32767)
    scipy.io.wavfile.write(out_file, rate, scaled)
```

In [4]:
```
# Load audio sources
source_files = ['beet.wav', 'beet9.wav', 'beet92.wav', 'mike.wav', 'street.wav']
wav_data = []
sample_rate = None
for f in source_files:
    sr, data = scipy.io.wavfile.read(f, mmap=False)
    if sample_rate is None:
        sample_rate = sr
    else:
        assert(sample_rate == sr)
    wav_data.append(data[:190000])  # cut off the last part so that all signals have

# Create source and measurement data
S = np.c_[wav_data]
plot_signals(S)

# Requires your function make_mixtures
#X = make_mixtures(S, make_random_nonsingular(S.shape[0]))
#plot_signals(X)
# Save mixtures to disk, so you can listen to them in your audio player
#for i in range(X.shape[0]):
#    save_wav(X[i, :], 'X' + str(i) + '.wav', sample_rate)
```



1.9 Excess Kurtosis (20 points)

The (excess) kurtosis is a measure of 'peakedness' of a distribution. It is defined as

$$\mathrm{Kurt}[X] = \frac{\mu_4}{\sigma^4} - 3 = \frac{\mathrm{E}[(X-\mu)^4]}{(\mathrm{E}[(X-\mu)^2])^2} - 3$$

Here, μ_4 is known as the fourth moment about the mean, and σ is the standard deviation. The '-3' term is introduced so that a Gaussian random variable has 0 excess kurtosis. We will now try to understand the performance of the various activation functions by considering the kurtosis of the corresponding priors, and comparing those to the empirical kurtosis of our data.

First, compute analytically the kurtosis of the four priors that you derived from the activation functions before. To do this, you will need the normalizing constant of the distribution, which you can either obtain analytically (good practice!), using computer algebra software (e.g. Sage) or by numerical integration (see scipy.integrate).

Now use the `scipy.stats.kurtosis` function, with the `fisher` option set to `True`, to compute the empirical kurtosis of the dummy signals and the real audio signals. Can you use this data to explain the performance of the various activation functions?