Independent Component Analysis

In this exercise, you will implement the FastICA algorithm, and apply it to model independent components of a distribution of image patches. The description of the fastICA method is given in the paper *"A. Hyvärinen and E. Oja. 2000. Independent component analysis: algorithms and applications"* linked from ISIS, and we frequently refer to sections and equations in that paper.

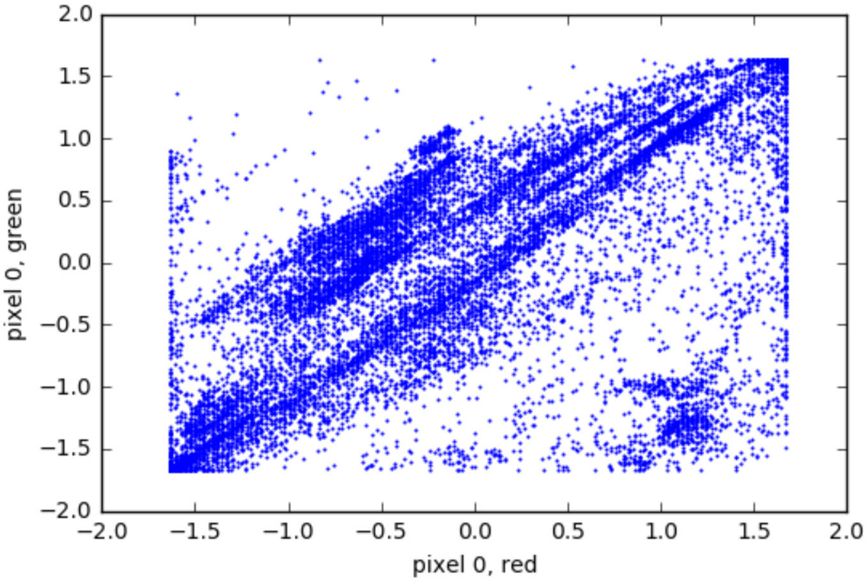
Three methods are provided for your convenience:

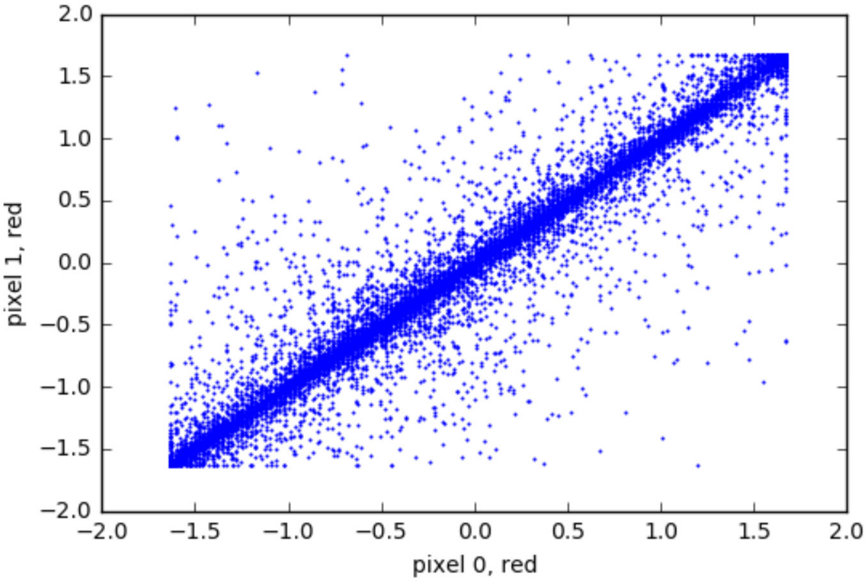
- **`utils.load()`** extracts a dataset of image patches from an collection of images (contained in the folder `images/` that can be extracted from the `images.zip` file). The method returns a list of RGB image patches of size 12×12, presented as a matrix of size $\#patches \times 432$. (Note that $12 \cdot 12 \cdot 3 = 432$).
- **`utils.scatterplot(...)`** produces a scatter plot from a two-dimensional data set. Each point in the scatter plot represents one image patch.
- **`utils.render(...)`** takes a matrix of size $\#patches \times 432$ as input and renders these patches in the IPython notebook.

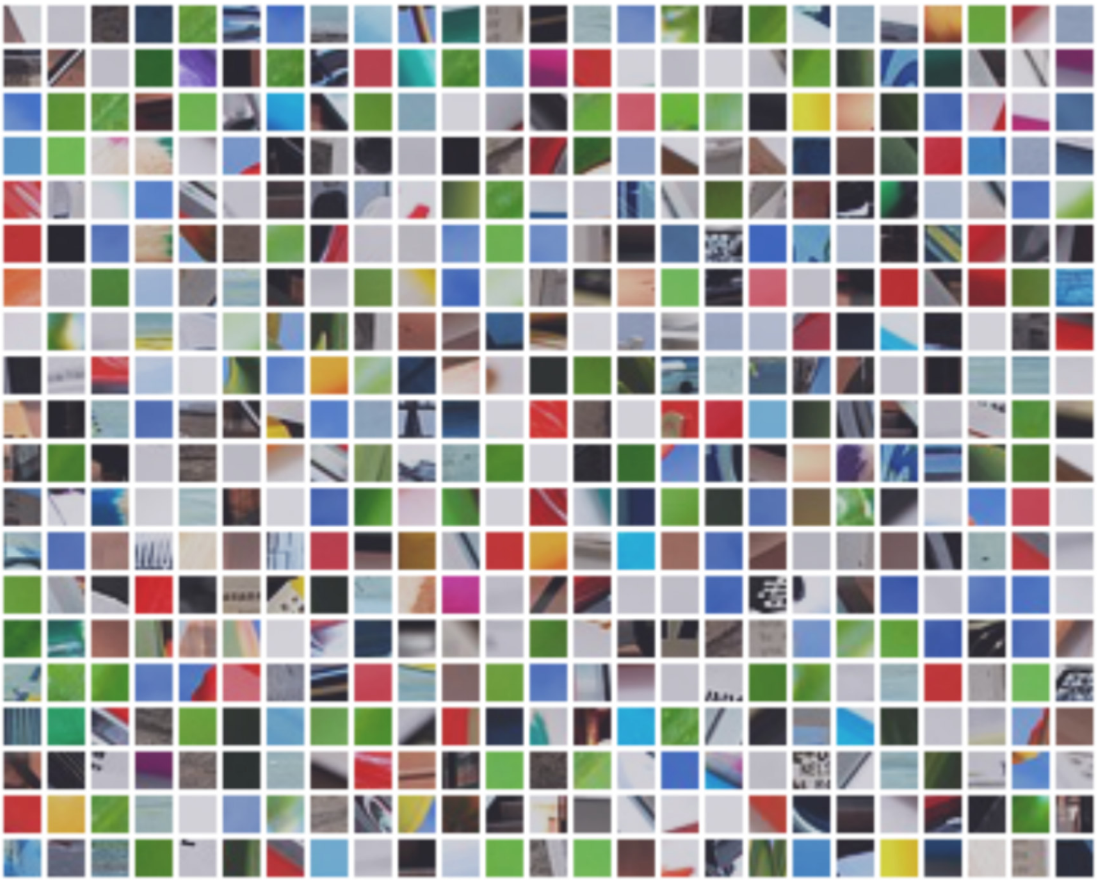
Demo code

A demo code that makes use of these three methods is given below. The code performs basic analysis such as loading the data, plotting correlations between neighboring pixels, or different color channels of the same pixel, and rendering some image patches.

In [57]:
```
import utils
%matplotlib inline

# Load the dataset of image patches
X = utils.load()

# Plot the red vs. green channel of the first pixel
utils.scatterplot(X[:,0],X[:,1],xlabel='pixel 0, red',ylabel='pixel 0, green')

# Plot the red channel of the first and second pixel
utils.scatterplot(X[:,0],X[:,3],xlabel='pixel 0, red',ylabel='pixel 1, red')

# Visualize 500 image patches from the image
utils.render(X[:500])
```


pixel 0, green
pixel 0, red
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0


pixel 1, red
pixel 0, red
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0



Whitening (10 P)

Independent component analysis applies whitening to the data as a preprocessing step. The whitened data matrix $\tilde{X}$ is obtained by linear projection of X, such data such that $\mathrm{E}[\tilde{x}\tilde{x}^\top] = I$, where $\tilde{x}$ is a row of the whitened matrix $\tilde{X}$. See Section 5.2 of the paper for a complete description of the whitening procedure.

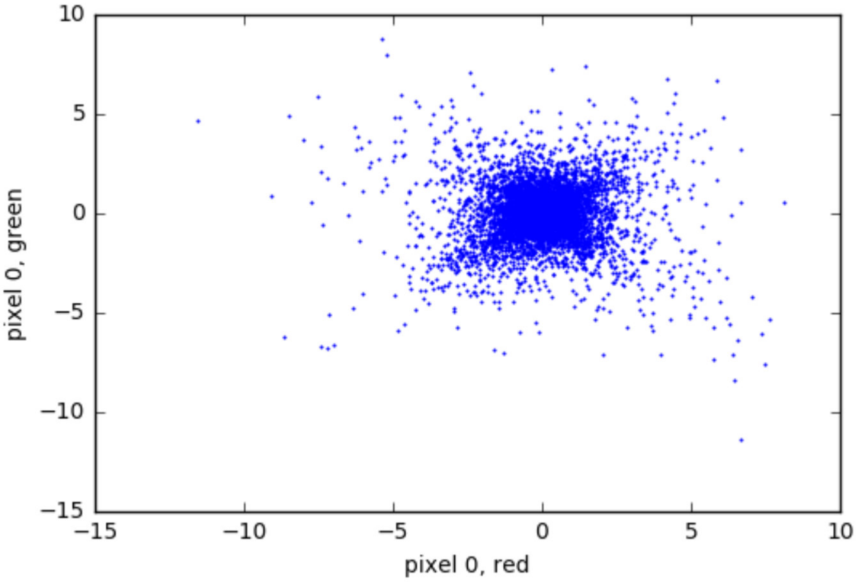
Tasks:

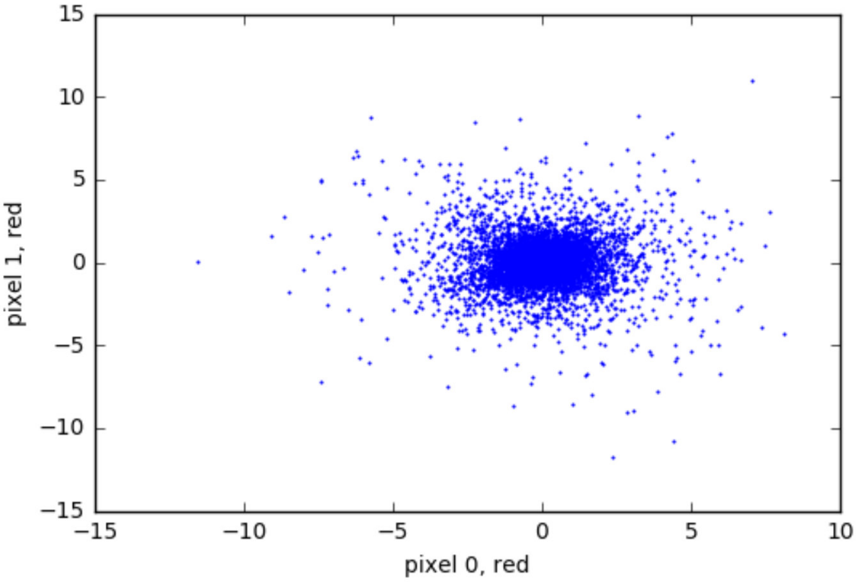
- **Implement a function that returns a whitened version of the data given as input.**
- **Add to this function a test that makes sure that $\mathrm{E}[\tilde{x}\tilde{x}^\top] \approx I$ (up to numerical accuracy).**
- **Reproduce the scatter plots of the demo code, but this time, using the whitened data.**
- **Render 500 whitened image patches.**

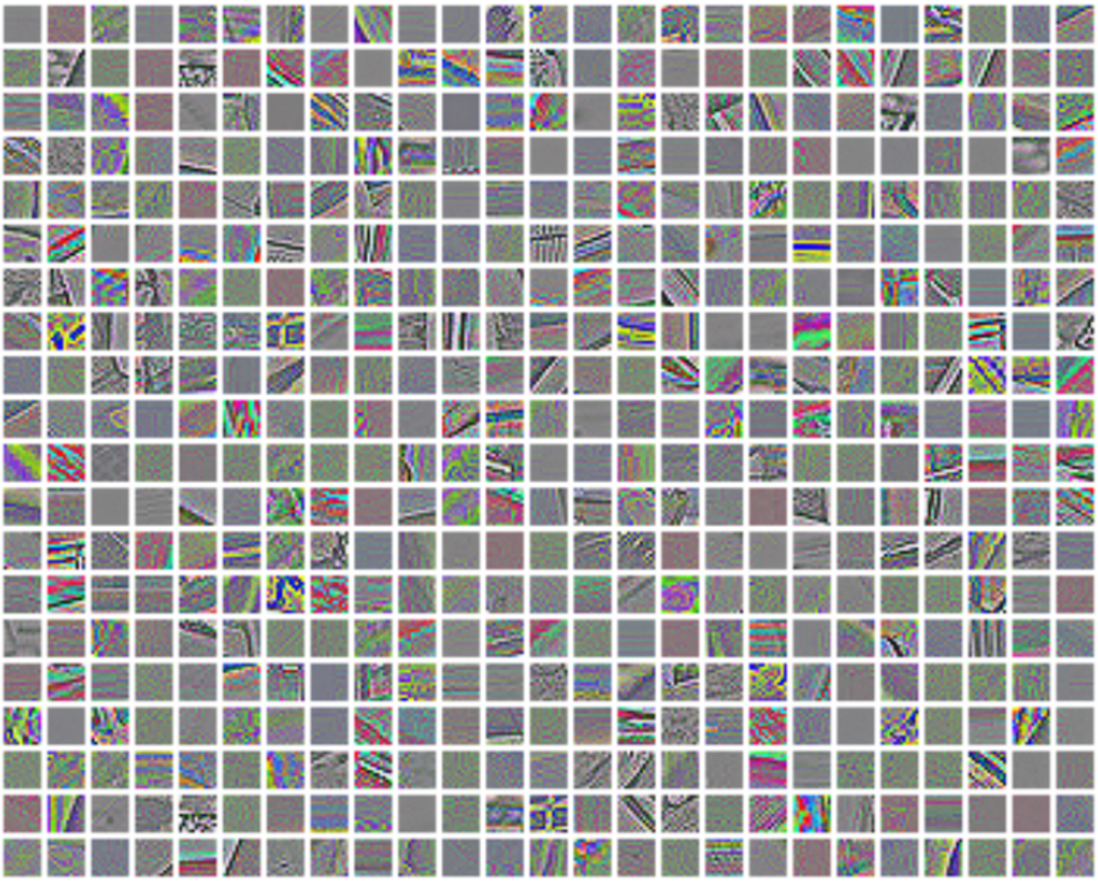
In [45]:
```
##### REPLACE BY YOUR CODE
#import solutions
#solutions.whitening()
#####
import numpy as np

def centering(X):
    X = np.array(X)
    mean = X.mean(axis=0)
    X -= mean
    return X


def whiten(X):
    n,d = X.shape
    X = centering(X)
    C = 1.0/(n-1) *np.dot(X.T,X)
    d, E = np.linalg.eigh(C)
    D = np.diag(1./np.sqrt(abs(d)))
    XW = np.dot(E,np.dot(D, np.dot(E.T,X.T)))

    # test
    CW = np.cov(XW)
    d = CW.shape[0]
    print('test if whitened covariance equals identity matrix: ', np.sum(CW - np.identit
y(d)))

    return XW.T
```

In [46]:
```
%matplotlib inline

# Load the dataset of image patches
XW = whiten(X)

# Plot the red vs. green channel of the first pixel
utils.scatterplot(XW[:,0],XW[:,1],xlabel='pixel 0, red',ylabel='pixel 0, green')

# Plot the red channel of the first and second pixel
utils.scatterplot(XW[:,0],XW[:,3],xlabel='pixel 0, red',ylabel='pixel 1, red')

# Visualize 500 image patches from the image
utils.render(XW[:500])
```

```
('test if whitened covariance equals identity matrix: ', -1.5862959034199535e-13)
```







Implementing FastICA (20 P)

We now would like to learn 100 independent components of the distribution of whitened image patches. For this, we follow the procedure described in the Chapter 6 of the paper. Implementation details specific to this exercise are given below:

- **Nonquadratic function G**: In this exercise, we will make use of the nonquadratic function $G(x) = \frac{1}{a} \log\cosh(ax)$, proposed in Section 4.3.2 of the paper, with $a = 1.5$. This function admits as a derivative the function $g(x) = \tanh(ax)$, and as a double derivative the function $g'(x) = a \cdot (1 - \tanh^2(ax))$.

- **Number of iterations**: The FastICA procedure will be run for 64 iterations. Note that the training procedure can take a relatively long time (up to 5 minutes depending on the system). Therefore, during the developement phase, it is advised to run the algorithm for a fraction of the total number of iterations.

- **Objective function**: The objective function that is maximized by the ICA training algorithm is given in Equation 25 of the paper. Note that since we learn 100 independent components, the objective function is in fact the *sum* of the objective functions of each independent components.

- **Finding multiple independent components**: Conceptually, finding multiple independent components as described in the paper is equivalent to running multiple instances of FastICA (one per independent component), under the constraint that the components learned by these instances are decorrelated. In order to keep the learning procedure computationally affordable, the code must be parallelized, in particular, make use of numpy matrix multiplications instead of loops whenever it is possible.

- **Weight decorrelation**: To decorrelate outputs, we use the inverse square root method given in Equation 45.

Tasks:

- **Implement the FastICA method described in the paper, and run it for 64 iterations.**
- **Print the value of the objective function at each iteration.**
- **Create a scatter plot of the projection of the whitened data on two distinct independent components after 0, 1, 3, 7, 15, 31, 63 iterations.**
- **Visualize the learned independent components using the function `render(...)`.**

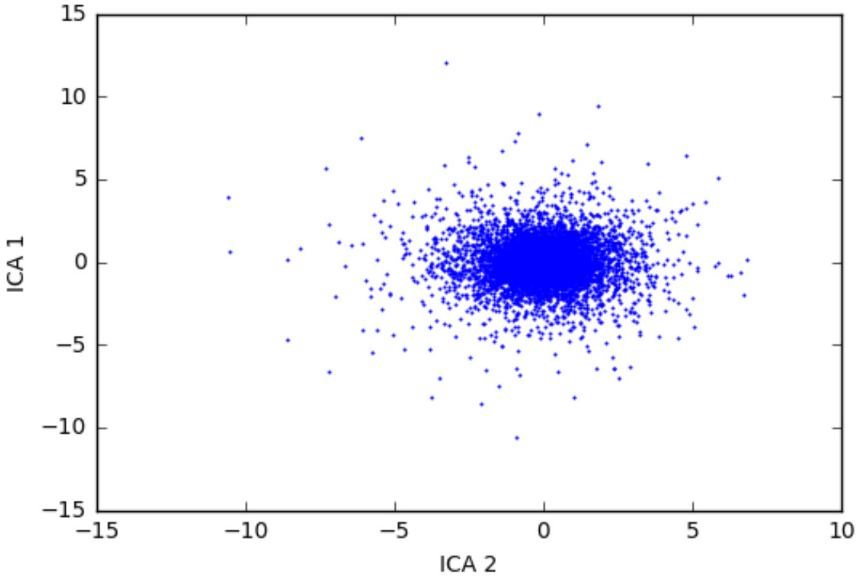
In [47]:
```
def G(x):
    a = 1.5
    return 1.0/a * np.log(np.cosh(a*x))

def g(x):
    a = 1.5
    return np.tanh(a*x)

def gprime(x):
    a = 1.5
    return a*(1-np.tanh(a*x)**2)

def objective(w):
    v = np.mean(G(np.random.normal(0,1,n)))
    J = (np.mean(G(np.dot(X,w))) -v)**2
    return J

def objectiveMulti(W):
    v = np.mean(G(np.random.randn(n,100)), axis =0)
    xtw = np.dot(X,W)
    y = np.mean(G(xtw), axis=0)
    J = np.sum((y-v)**2)
    return J

def decorrelate(W):
    #normalize W
    norms = np.linalg.norm(W, axis=0)
    W = W / norms
    #decorrelate W
    #W = W.T
    W2 = np.dot(W,W.T)
    eig,F = np.linalg.eigh(W2)
    eig = eig + 1.282566102764301e-25
    D = np.diag(1./np.sqrt(abs(eig)))
    Wsqrt = np.dot(F, np.dot(D,F.T))
    W = np.dot(Wsqrt, W)
    return W
```

In [48]:
```
def singleUnitICA(X, iterations):
    it = iterations
    n,d = X.shape
    w = np.random.rand(d,1)
    for i in range(it):
        y = g(np.dot(X,w))
        Y = np.dot(X.T, y).T
        E1 = np.mean(Y, axis=0)
        E1 = E1.reshape(d,1)
        E2 = np.mean(g2(np.dot(X,w))) * w
        w = E1 -E2
        w = w/np.linalg.norm(w)
        print('Jsingle', objective(w))
    return w
```

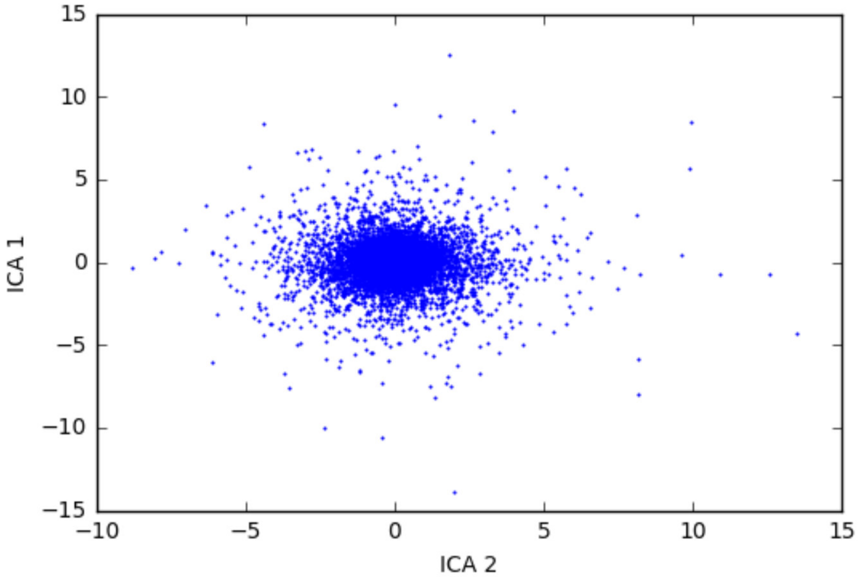
In [77]:
```
def multiUnitICA(X, iterations, scatterplotMilestones):
    n,d = X.shape
    W = np.random.rand(d,100)
    norms = np.linalg.norm(W, axis=0)
    #W = W / norms.astype(float)

    for i in range(iterations):
        wtx = np.dot(X,W)
        gwtx = g(wtx)
        g_wtx = gprime(wtx)
        W1 = np.dot(X.T, gwtx)/float(n-1)
        W2 = np.dot(W, np.diag(g_wtx.mean(axis=0)))
        W = W1- W2
        W = decorrelate(W)
        XW = np.dot(X, W)
        if i in scatterplotMilestones:
            utils.scatterplot(XW[:,0],XW[:,1], xlabel='ICA 2',ylabel='ICA 1')
        print "iteration ", i, ": ", objectiveMulti(W)

    W = decorrelate(W)

    return W
```

In [78]:
```
X = utils.load()
n,d = X.shape
X = whiten(X)
milestonesForScatterPlots = [0,1,3,7,15,31,63]
W = multiUnitICA(X, 64, milestonesForScatterPlots)
```

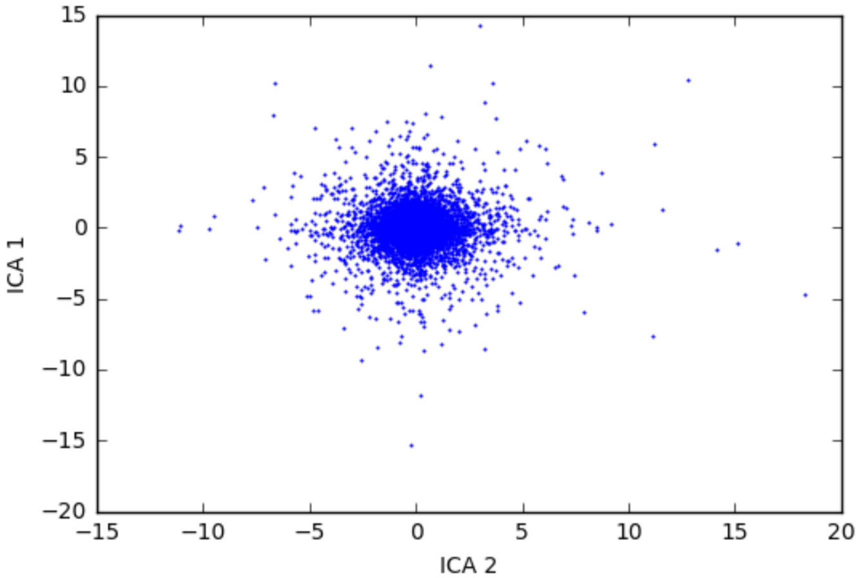
```
('test if whitened covariance equals identity matrix: ', 8.5133294929377542e-14)
```



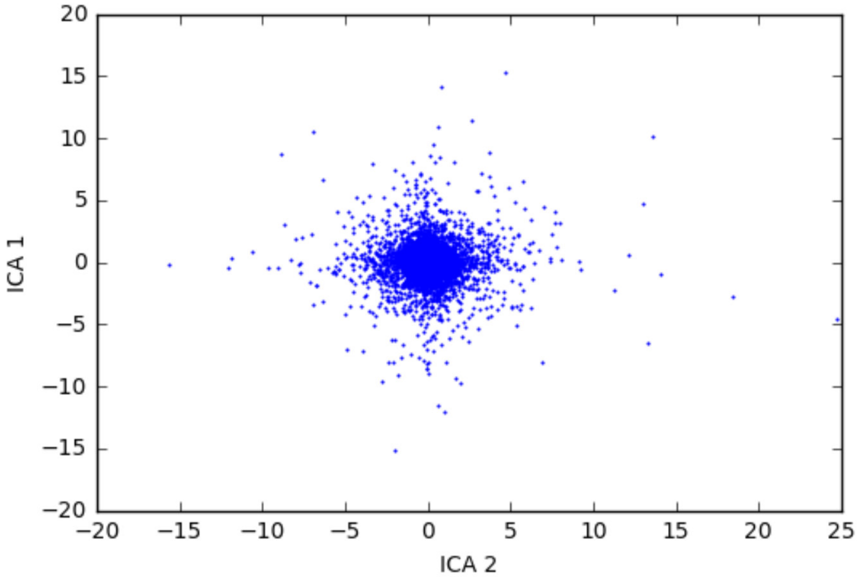

```
iteration  0 :  0.972531737595
```



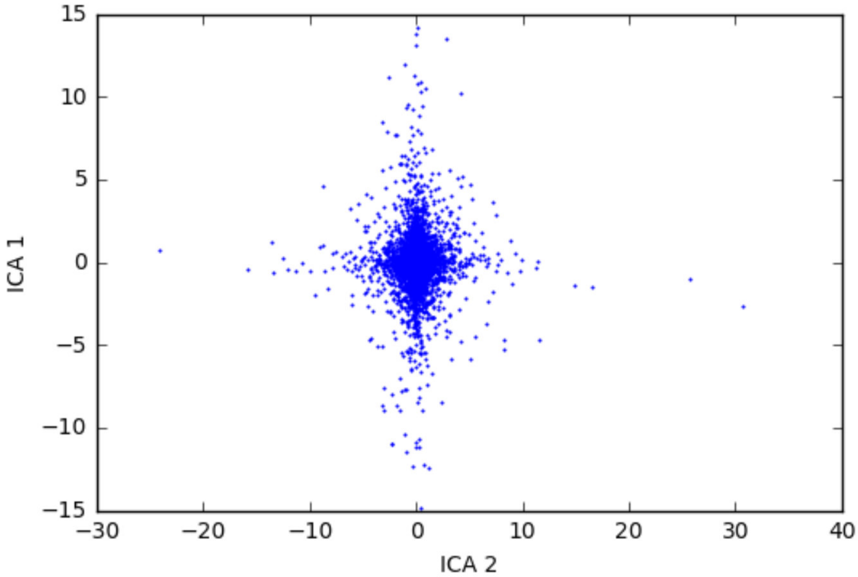

```
iteration  1 :  1.53019880465
iteration  2 :  2.00242203426
```



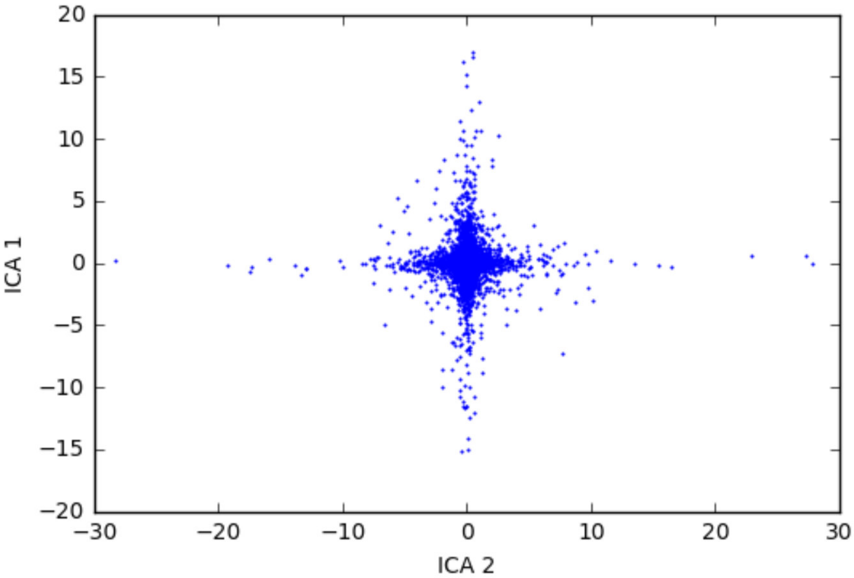
```
iteration  3 :  2.32971881711
iteration  4 :  2.60935723986
iteration  5 :  2.78749240253
iteration  6 :  2.95109591787
```



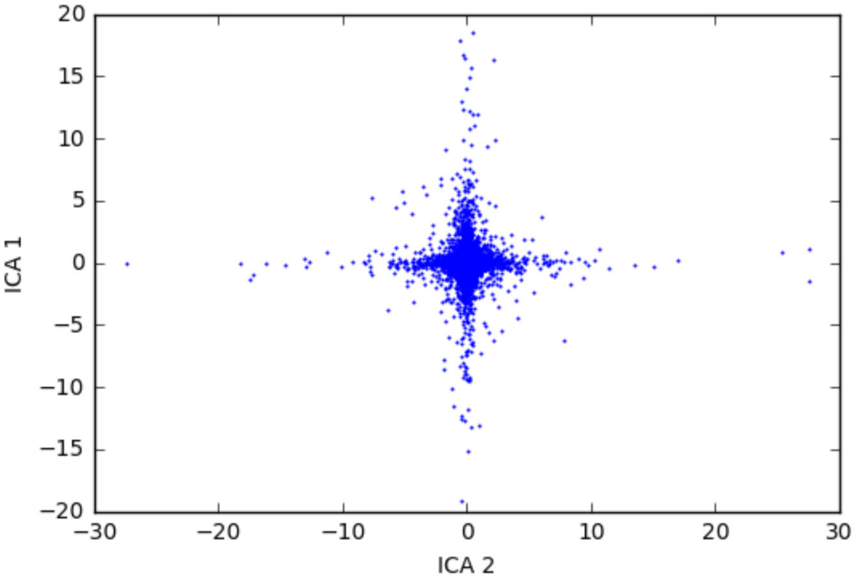

```
iteration  7 :  3.11420842449
iteration  8 :  3.26604584187
iteration  9 :  3.38072392465
iteration  10 :  3.47620969998
iteration  11 :  3.59322484583
iteration  12 :  3.68510431382
iteration  13 :  3.80234135492
iteration  14 :  3.89649381131
```



```
iteration  15 :  4.02079136984
iteration  16 :  4.11622360964
iteration  17 :  4.19222405345
iteration  18 :  4.32706756083
iteration  19 :  4.36870712984
iteration  20 :  4.52511641001
iteration  21 :  4.58230029487
iteration  22 :  4.6930098143
iteration  23 :  4.78843015842
iteration  24 :  4.87176895554
iteration  25 :  4.97705814424
iteration  26 :  5.05459761487
iteration  27 :  5.16069095383
iteration  28 :  5.25257142607
iteration  29 :  5.29443950043
iteration  30 :  5.39636170937
```

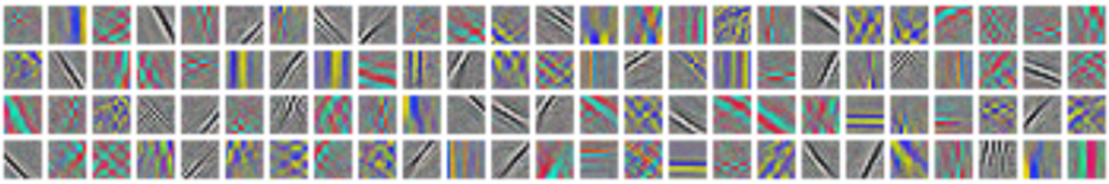



```
iteration  31 :  5.51488819426
iteration  32 :  5.56446585643
iteration  33 :  5.60347612312
iteration  34 :  5.68184504674
iteration  35 :  5.77411803694
iteration  36 :  5.87567867256
iteration  37 :  5.87768483135
iteration  38 :  5.94324647688
iteration  39 :  5.98400709674
iteration  40 :  6.04757874524
iteration  41 :  6.07120041963
iteration  42 :  6.11716132841
iteration  43 :  6.1714447388
iteration  44 :  6.18865145511
iteration  45 :  6.22031085697
iteration  46 :  6.26417194767
iteration  47 :  6.2937735182
iteration  48 :  6.26141274746
iteration  49 :  6.30216268817
iteration  50 :  6.33240700364
iteration  51 :  6.33888803582
iteration  52 :  6.38618593292
iteration  53 :  6.43312657646
iteration  54 :  6.42974250892
iteration  55 :  6.40780532858
iteration  56 :  6.47313753705
iteration  57 :  6.46409171455
iteration  58 :  6.44724623001
iteration  59 :  6.47429812097
iteration  60 :  6.51912822402
iteration  61 :  6.51098958122
iteration  62 :  6.53350123267
```




```
iteration  63 :  6.55844702828
```

In [81]:
```
XW = np.dot(X, W)
utils.render(W.T)
```



In [ ]: