Assignment1

Optimal Transport with Linear Programming

Important: Please read the installation page for details about how to install the toolboxes.

This numerical tours details how to solve the discrete optimal transport problem (in the case of measures that are sums of Diracs) using linear programming.

In [30]:

```
from __future__ import division

import numpy as np
import scipy as scp
import pylab as pyl
import matplotlib.pyplot as plt
from sklearn import datasets


import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Optimal Transport of Discrete Distribution

We consider two dicretes distributions

$$\forall k = 0, 1, \quad \mu_k = \sum_{i=1}^{n_k} p_{k,i} \delta_{x_{k,i}}$$

where n_0, n_1 are the number of points, δ_x is the Dirac at location $x \in \mathbb{R}^d$, and $X_k = (x_{k,i})_{i=1}^{n_k} \subset \mathbb{R}^d$ for $k = 0, 1$ are two point clouds.

We define the set of couplings between μ_0, μ_1 as

$$\mathcal{P} = \{(\gamma_{i,j})_{i,j} \in (\mathbb{R}^+)^{n_0 \times n_1}, \forall i, \sum_j \gamma_{i,j} = p_{0,i}, \ \forall j, \sum_i \gamma_{i,j} = p_{1,j}\}$$

The Kantorovitch formulation of the optimal transport reads

$$\gamma^\star \in \underset{\gamma \in \mathcal{P}}{argmin} \sum_{i,j} \gamma_{i,j} C_{i,j}$$

where $C_{i,j} \geq 0$ is the cost of moving some mass from $x_{0,i}$ to $x_{1,j}$.

The optimal coupling $\gamma^\star$ can be shown to be a sparse matrix with less than $n_0 + n_1 - 1$ non zero entries. An entry $\gamma^\star_{i,j} \neq 0$ should be understood as a link between $x_{0,i}$ and $x_{1,j}$ where an amount of mass equal to $\gamma^\star_{i,j}$ is transfered.

In the following, we concentrate on the L^2 Wasserstein distance.

$$C_{i,j} = \|x_{0,i} - x_{1,j}\|^2.$$

The L^2 Wasserstein distance is then defined as

$$W_2(\mu_0, \mu_1)^2 = \sum_{i,j} \gamma^\star_{i,j} C_{i,j}.$$

The coupling constraint

$$\forall i, \sum_j \gamma_{i,j} = p_{0,i}, \ \forall j, \sum_i \gamma_{i,j} = p_{1,j}$$

can be expressed in matrix form as

$$\Sigma(n_0, n_1)\gamma = [p_0; p_1]$$

where $\Sigma(n_0, n_1) \in \mathbb{R}^{(n_0+n_1)\times(n_0 n_1)}$.

In [32]:

```
from scipy import sparse

Rows  = lambda n0,n1: sparse.coo_matrix((np.ones(n0*n1),
                                         (np.ravel(np.tile(np.arange(0,n0),(n1,1)),order='C'),
                                          np.arange(0,n0*n1))))
Cols  = lambda n0,n1: sparse.coo_matrix((np.ones(n0*n1),
                                        (np.ravel(np.tile(np.arange(0,n1),(n0,1)),order='F'),
                                         np.arange(0,n0*n1))))
Sigma = lambda n0,n1: sparse.vstack((Rows(n0,n1),Cols(n0,n1)))
```

In [ ]:

```

```

We use a simplex algorithm to compute the optimal transport coupling $\gamma^\star$.

In [33]:

```
from nt_toolbox.perform_linprog import *
maxit = 1e4
tol = 1e-9
otransp = lambda C,p0,p1: np.reshape(perform_linprog(Sigma(len(p0),len(p1)).toarray(),np.vstack((p0
,p1)),C,maxit,tol),(n0,n1),order="F")
```

Dimensions n_0, n_1 of the clouds.

In [34]:

```
n0 = 60
n1 = 120
```

In [35]:

```
np.random.normal(0,0.1,10)
```

Out[35]:

```
array([-0.19344265, -0.10928863, -0.07254971, -0.2227023 , -0.12938163,
       -0.1227497 , -0.02444418,  0.18020032, -0.1077458 ,  0.14179392])
```

Compute a first point cloud X_0 that is Gaussian, and a second point cloud X_1 that is Gaussian mixture.

In [36]:

```
from numpy import random
gauss = lambda q,a,c: a*random.randn(2, q) + np.transpose(np.tile(c, (q,1)))
def circle(n, r, noise):
    theta = np.random.rand(n)*2*np.pi
    a, b = r * np.cos(theta) + np.random.normal(0,noise,n), r * np.sin(theta) + np.random.normal(0,n
oise,n)
    return np.stack((a, b))

X0 = random.randn(2,n0)*.1
X1 = circle(n1,1,0.05)
```

Density weights p_0, p_1.

In [37]:

```
normalize = lambda a: a/np.sum(a)
p0 = normalize(random.rand(n0, 1))
p1 = normalize(random.rand(n1, 1))
```

Shortcut for display.

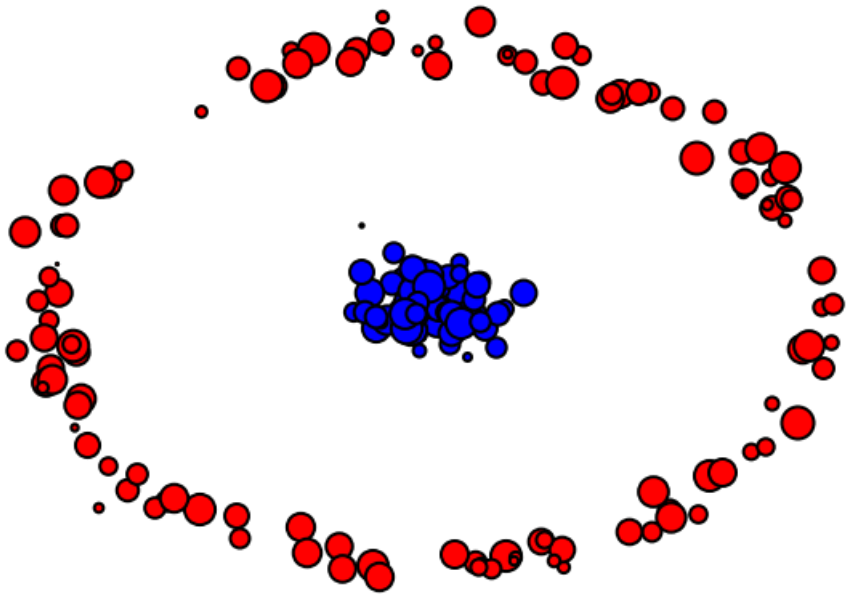
In [46]:

```
myplot = lambda x,y,ms,col: plt.scatter(x,y, s=ms*20, edgecolors="k", c=[col], linewidths=2)
```

Display the point clouds. The size of each dot is proportional to its probability density weight.

In [39]:

```
plt.figure(figsize = (10,7))
plt.axis("off")

for i in range(len(p0)):
    myplot(X0[0,i], X0[1,i], p0[i]*len(p0)*10, 'b')

for i in range(len(p1)):
    myplot(X1[0,i], X1[1,i], p1[i]*len(p1)*10, 'r')

plt.xlim(np.min(X1[0,:])-.1,np.max(X1[0,:])+.1)
plt.ylim(np.min(X1[1,:])-.1,np.max(X1[1,:])+.1)

plt.savefig("Data")
plt.show()
```



Compute the weight matrix $(C_{i,j})_{i,j}$.

In [40]:

```
C = np.transpose(np.tile(np.transpose(np.sum(X0**2,0)),(n1,1))) + np.tile(np.sum(X1**2,0),(n0,1)) -
2*np.dot(np.transpose(X0),X1)
```

Compute the optimal transport plan.

In [41]:

```
gamma = otransp(C, p0, p1)
```

Check that the number of non-zero entries in $\gamma^\star$ is $n_0 + n_1 - 1$.

In [42]:

```
print("Number of non-zero: %d (n0 + n1-1 = %d!)" %(len(gamma[gamma>0]), n0 + n1-1))
```

```
Number of non-zero: 179 (n0 + n1-1 = 179!)
```

Check that the solution satifies the constraints $\gamma \in \mathcal{C}$.

In [43]:

```
from numpy import linalg
print("Constraints deviation (should be 0): %.2f, %.2f" %(linalg.norm(np.sum(gamma,1)-np.ravel(p0))
,linalg.norm(np.transpose(np.sum(gamma, 0))-np.ravel(p1))))
```

```
Constraints deviation (should be 0): 0.00, 0.00
```

Displacement Interpolation

For any $t \in [0,1]$, one can define a distribution μ_t such that $t \mapsto \mu_t$ defines a geodesic for the Wasserstein metric.

Since the W_2 distance is a geodesic distance, this geodesic path solves the following variational problem

$$\mu_t = \underset{\mu}{argmin}\ (1-t)W_2(\mu_0,\mu)^2 + tW_2(\mu_1,\mu)^2.$$

This can be understood as a generalization of the usual Euclidean barycenter to barycenter of distribution. Indeed, in the case that $\mu_k = \delta_{x_k}$, one has $\mu_t = \delta_{x_t}$ where $x_t = (1-t)x_0 + tx_1$.

Once the optimal coupling $\gamma^\star$ has been computed, the interpolated distribution is obtained as

$$\mu_t = \sum_{i,j} \gamma^\star_{i,j} \delta_{(1-t)x_{0,i}+tx_{1,j}}.$$

Find the i,j with non-zero $\gamma^\star_{i,j}$.

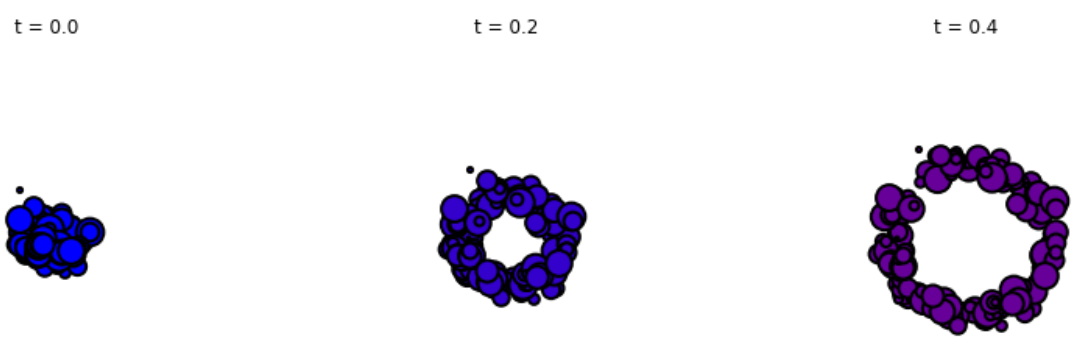
In [44]:

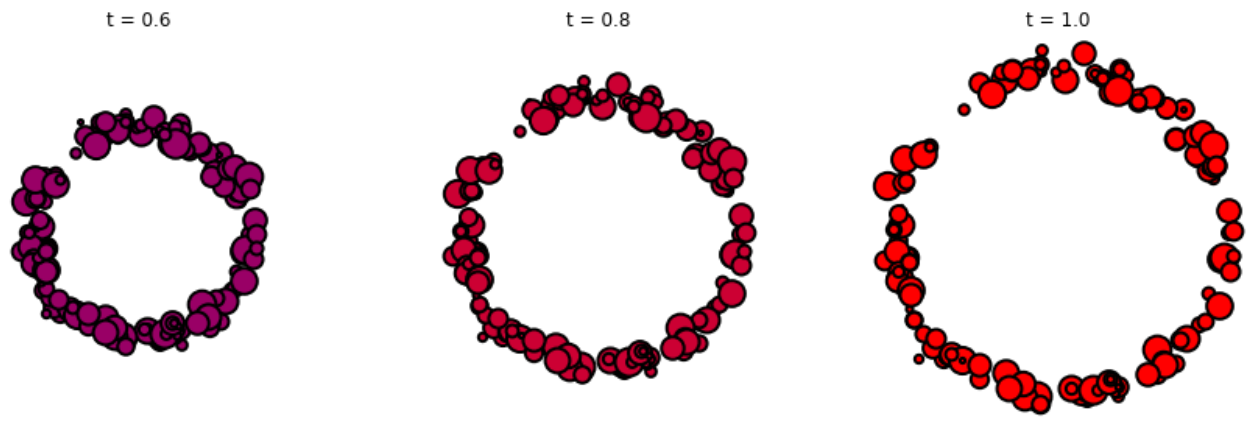
```
I,J = np.nonzero(gamma)
gammaij = gamma[I,J]
```

Display the evolution of μ_t for a varying value of $t \in [0,1]$.

In [47]:

```
plt.figure(figsize =(15,10))
tlist = np.linspace(0, 1, 6)

for i in range(len(tlist)):
    t = tlist[i]
    Xt = (1-t)*X0[:,I] + t*X1[:,J]
    plt.subplot(2,3,i+1)
    plt.axis("off")
    for j in range(len(gammaij)):
        myplot(Xt[0,j],Xt[1,j],gammaij[j]*len(gammaij)*6,[t,0,1-t])

    plt.title("t = %.1f" %t)
    plt.xlim(np.min(X1[0,:])-.1,np.max(X1[0,:])+.1)
    plt.ylim(np.min(X1[1,:])-.1,np.max(X1[1,:])+.1)
plt.savefig("Interpolation.png")
plt.show()
```





Optimal Assignement

In the case where the weights $p_{0,i} = 1/n, p_{1,i} = 1/n$ (where $n_0 = n_1 = n$) are constants, one can show that the optimal transport coupling is actually a permutation matrix. This properties comes from the fact that the extremal point of the polytope $\mathcal{C}$ are permutation matrices.

This means that there exists an optimal permutation $\sigma^\star \in \Sigma_n$ such that

$$\gamma^\star_{i,j} = \begin{cases} 1 & \text{if} \quad j = \sigma^\star(i), \\ 0 & \text{otherwise.} \end{cases}$$

where Σ_n is the set of permutation (bijections) of $\{1, \ldots, n\}$.

This permutation thus solves the so-called optimal assignement problem

$$\sigma^\star \in \underset{\sigma \in \Sigma_n}{argmin} \sum_i C_{i,\sigma(j)}.$$

Same number of points.

In [19]:

```
n0 = 50
n1 = n0
```

Compute points clouds.

In [20]:

```
X = datasets.make_moons(n_samples=n0 + n1, noise = 0.1)
X0, X1 = X[0][X[1] == 1].T*0.1, X[0][X[1] == 0].T*0.1
```

Constant distributions.

In [21]:

```
p0 = np.ones([n0,1])/n0
p1 = np.ones([n1,1])/n1
```

Compute the weight matrix $(C_{i,j})_{i,j}$.

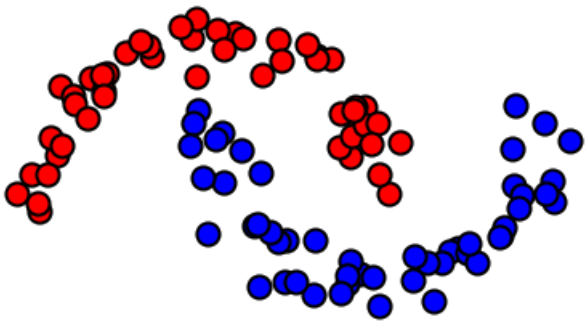
In [22]:

```
C = np.transpose(np.tile(np.transpose(np.sum(X0**2,0)),(n1,1))) + np.tile(np.sum(X1**2,0),(n0,1)) -
2*np.dot(np.transpose(X0),X1)
```

Display the coulds.

In [23]:

```
plt.figure(figsize = (10,7))
plt.axis('off')

myplot(X0[0,:],X0[1,:],10,'b')
myplot(X1[0,:],X1[1,:],10,'r')

plt.xlim(min(np.min(X1[0,:]), np.min(np.min(X0[0,:]) ))-.1,max(np.max(X1[0,:]), np.max(np.max(X0[0,
:])))+.1)
plt.ylim(min(np.min(X1[1,:]), np.min(np.min(X0[1,:]) ))-.1,max(np.max(X1[1,:]), np.max(np.max(X0[1,
:])))+.1)

plt.savefig("DataCoupling.png")
plt.show()
```



Solve the optimal transport.

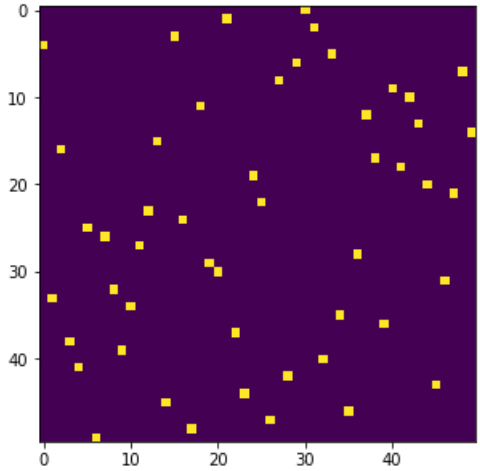
In [24]:

```
gamma = otransp(C, p0, p1)
```

Show that γ is a binary permutation matrix.

In [25]:

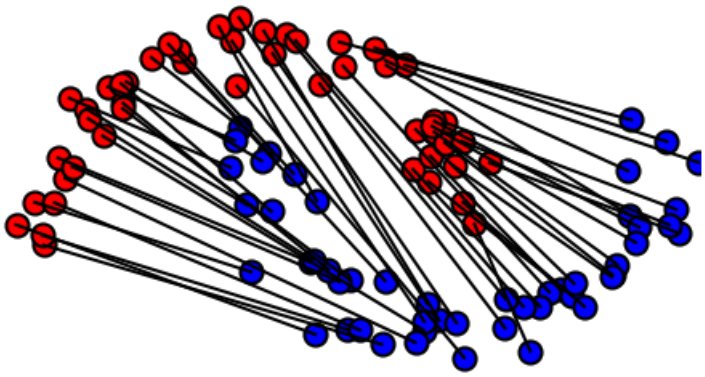
```
plt.figure(figsize = (5,5))
plt.imshow(gamma);
plt.savefig("SparseMatrixCoupling")
```



Display the optimal assignement.

In [26]:

```
I,J = np.nonzero(gamma)

plt.figure(figsize = (10,7))
plt.axis('off')

for k in range(len(I)):
    h = plt.plot(np.hstack((X0[0,I[k]],X1[0,J[k]])),np.hstack(([X0[1,I[k]], X1[1,J[k]]])),'k', lw =
2)

myplot(X0[0,:], X0[1,:], 10, 'b')
myplot(X1[0,:], X1[1,:], 10, 'r')

plt.xlim(np.min(X1[0,:])-.1,np.max(X1[0,:])+.1)
plt.ylim(np.min(X1[1,:])-.1,np.max(X1[1,:])+.1)
plt.savefig("ResultCoupling.png")
plt.show()
```



Assignment2

Entropic Regularization of Optimal Transport

Important: Please read the installation page for details about how to install the toolboxes.

This numerical tours exposes the general methodology of regularizing the optimal transport (OT) linear program using entropy. This allows to derive fast computation algorithm based on iterative projections according to a Kulback-Leiber divergence.

In [47]:

```
from __future__ import division

import numpy as np
import matplotlib.pyplot as plt
import scipy as scp
import pylab as pyl
from sklearn import datasets
```

```
import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Entropic Regularization of Optimal Transport

We consider two input histograms $a, b \in \Sigma_n$, where we denote the simplex in $\mathbb{R}^n$

$$\Sigma_n \equiv \{a \in \mathbb{R}_+^n, \sum_i a_i = 1\}.$$

We consider the following discrete regularized transport

$$W_\varepsilon(a,b) \equiv \min_{P \in U(a,b)} \langle C, P \rangle - \varepsilon E(P).$$

where the polytope of coupling is defined as

$$U(a,b) \equiv \{P \in (\mathbb{R}^+)^{n \times m}, P\mathbb{1}_m = a, P^\top \mathbb{1}_n = b\},$$

where $\mathbb{1}_n \equiv (1, \ldots, 1)^\top \in \mathbb{R}^n$, and for $P \in \mathbb{R}_+^{n \times m}$, we define its entropy as

$$E(P) \equiv -\sum_{i,j} P_{i,j}(\log(P_{i,j}) - 1).$$

When $\varepsilon = 0$ one recovers the classical (discrete) optimal transport. We refer to the monograph Villani for more details about OT. The idea of regularizing transport to allows for faster computation is introduced in Cuturi.

Here the matrix $C \in (\mathbb{R}^+)^{n \times m}$ defines the ground cost, i.e. $C_{i,j}$ is the cost of moving mass from a bin indexed by i to a bin indexed by j.

The regularized transportation problem can be re-written as a projection

$$W_\varepsilon(a,b) = \varepsilon \min_{P \in U(a,b)} \mathrm{KL}(P|K) \quad \text{where} \quad K_{i,j} \equiv e^{-\frac{C_{i,j}}{\varepsilon}}$$

of the Gibbs kernel K according to the Kullback-Leibler divergence. The Kullback-Leibler divergence between $P, K \in \mathbb{R}_+^{n \times m}$ is

$$\mathrm{KL}(P|K) \equiv \sum_{i,j} P_{i,j} \left(\log\left(\frac{P_{i,j}}{K_{i,j}} \right) - 1 \right).$$

This interpretation of regularized transport as a KL projection and its numerical applications are detailed in BenamouEtAl.

Given a convex set $\mathcal{C} \subset \mathbb{R}^N$, the projection according to the Kullback-Leiber divergence is defined as

$$\mathrm{Proj}_{\mathcal{C}}^{\mathrm{KL}}(\xi) = \underset{\pi \in \mathcal{C}}{argmin}\ \mathrm{KL}(\pi|\xi).$$

Iterative Bregman Projection Algorithm

Given affine constraint sets $(\mathcal{C}_1, \mathcal{C}_2)$, we aim at computing

$$\mathrm{Proj}_{\mathcal{C}}^{\mathrm{KL}}(K) \quad \text{where} \quad \mathcal{C} = \mathcal{C}_1 \cap \mathcal{C}_2$$

(this description can of course be extended to more than 2 sets).

This can be achieved, starting by $P_0 = K$, by iterating $\forall \ell \geq 0$,

$$P_{2\ell+1} = \mathrm{Proj}_{\mathcal{C}_1}^{\mathrm{KL}}(P_{2\ell}) \quad \text{and} \quad P_{2\ell+2} = \mathrm{Proj}_{\mathcal{C}_2}^{\mathrm{KL}}(P_{2\ell+1}).$$

One can indeed show that $P_\ell \to \mathrm{Proj}_{\mathcal{C}}^{\mathrm{KL}}(K)$. We refer to BauschkeLewis for more details about this algorithm and its extension to compute the projection on the intersection of convex sets (Dikstra algorithm).

Sinkhorn's Algorithm

A fundamental remark is that the optimality condition of the entropic regularized problem shows that the optimal coupling P_ε necessarily has the form

$$P_\varepsilon = diag(u)\, K\, diag(v)$$

where the Gibbs kernel is defined as

$$K \equiv e^{-\frac{C}{\varepsilon}}.$$

One thus needs to find two positive scaling vectors $u \in \mathbb{R}_+^n$ and $v \in \mathbb{R}_+^m$ such that the two following equality holds

$$P\mathbb{1} = u \odot (Kv) = a \quad \text{and} \quad P^\top \mathbb{1} = v \odot (K^\top u) = b.$$

Sinkhorn's algorithm alternate between the resolution of these two equations, and reads

$$u \longleftarrow \frac{a}{Kv} \quad \text{and} \quad v \longleftarrow \frac{b}{K^\top u}.$$

This algorithm was shown to converge to a solution of the entropic regularized problem by Sinkhorn.

Transport Between Point Clouds

We first test the method for two input measures that are uniform measures (i.e. constant histograms) supported on two point clouds (that do not necessarily have the same size).

We thus first load two points clouds $x = (x_i)_{i=1}^n, y = (y_i)_{i=1}^m$, where $x_i, y_i \in \mathbb{R}^2$.

Number of points in each cloud, $N = (n, m)$.

In [48]:

```
N = [200,150]
```

Dimension of the clouds.

In [49]:

```
d = 2
```

Point cloud x, of n points inside a square.

In [50]:

```
x = np.random.rand(2,N[0])-.5
```

Point cloud y, of m points inside an anulus.

In [51]:

```
theta = 2*np.pi*np.random.rand(1,N[1])
r = .8 + .2*np.random.rand(1,N[1])
y = np.vstack((np.cos(theta)*r,np.sin(theta)*r))
```

In [52]:

```
X = datasets.make_moons(n_samples=2*N[0], noise = 0.03)
x = np.random.randn(2,N[0])*.1
Y = datasets.make_moons(n_samples=2*N[1], noise = 0.03)
y = np.stack((np.random.rand(N[1])- 1/2, np.ones(N[1])))
```

Shortcut for displaying point clouds.

In [53]:

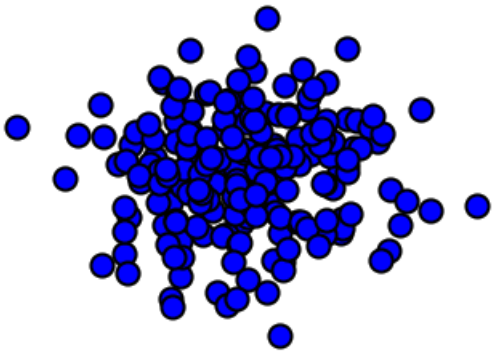
```
plotp = lambda x,col: plt.scatter(x[0,:], x[1,:], s=200, edgecolors="k", c=col, linewidths=2)
```

Display of the two clouds.

In [54]:

```
plt.figure(figsize=(10,10))

plotp(x, 'b')
plotp(y, 'r')

plt.axis("off")
plt.xlim(np.min(y[0,:])-.1,np.max(y[0,:])+.1)
plt.ylim(np.min(y[1,:])-.1,np.max(y[1,:])+.1)
```

```
plt.xlim(min(np.min(x[0,:]), np.min(np.min(y[0,:]) ))-.1,max(np.max(x[0,:]), np.max(np.max(y[0,:])))
+.1)
plt.ylim(min(np.min(x[1,:]), np.min(np.min(y[1,:]) ))-.1,max(np.max(x[1,:]), np.max(np.max(y[1,:])))
+.1)
plt.savefig("Ass2Ex1.png")
plt.show()
```



Cost matrix $C_{i,j} = \|x_i - y_j\|^2$.

In [55]:

```
x2 = np.sum(x**2,0)
y2 = np.sum(y**2,0)
C = np.tile(y2,(N[0],1)) + np.tile(x2[:,np.newaxis],(1,N[1])) - 2*np.dot(np.transpose(x),y)
```

Target histograms (a, b), here uniform histograms.

In [56]:

```
a = np.ones(N[0])/N[0]
b = np.ones(N[1])/N[1]
```

Regularization strength $\varepsilon > 0$.

In [57]:

```
epsilon = .01;
```

Gibbs Kernel K.

In [58]:

```
K = np.exp(-C/epsilon)
```

Initialization of $v = \mathbb{I}_m$ (u does not need to be initialized).

In [59]:

```
v = np.ones(N[1])
```

One sinkhorn iterations.

In [60]:

```
u = a / (np.dot(K,v))
v = b / (np.dot(np.transpose(K),u))
```

Exercise 1

Implement Sinkhorn algorithm. Display the evolution of the constraints satisfaction errors

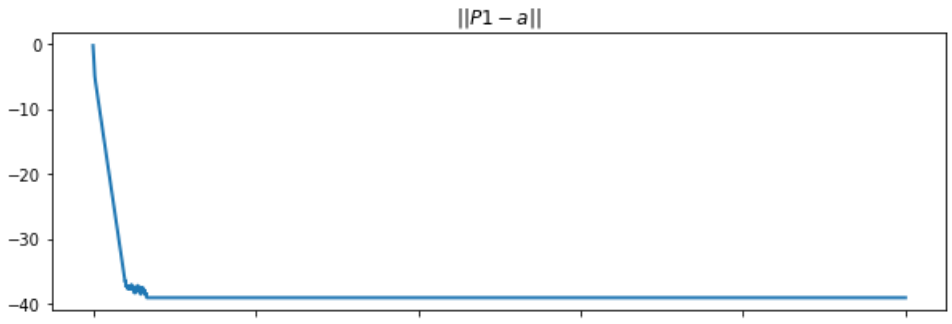
$$\|P\mathbb{I} - a\|_1 \quad \text{and} \quad \|P^\top \mathbb{I} - b\|$$

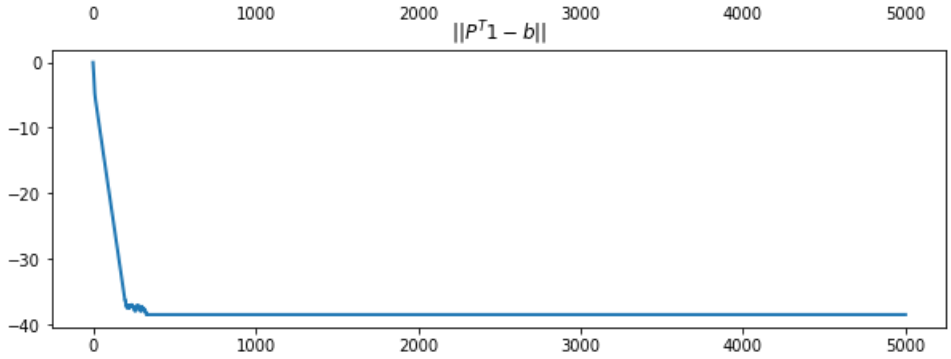
(you need to think about how to compute these residuals from (u, v) alone). isplay the violation of constraint error in log-plot.

In [61]:

```
from numpy import linalg

epsilon = .01;
K = np.exp(-C/epsilon)

v = np.ones(N[1])
print(v.shape)
niter = 5000
Err_p = []
Err_q = []

for i in range(niter):
    # sinkhorn step 1
    u = a / (np.dot(K,v))
    # error computation
    r = v*np.dot(np.transpose(K),u)
    Err_q = Err_q + [linalg.norm(r - b, 1)]
    # sinkhorn step 2
    v = b /(np.dot(np.transpose(K),u))
    s = u*np.dot(K,v)
    Err_p = Err_p + [linalg.norm(s - a,1)]

plt.figure(figsize = (10,7))

plt.subplot(2,1,1)
plt.title("$||P 1 - a||$")
plt.plot(np.log(np.asarray(Err_p)), linewidth = 2)

plt.subplot(2,1,2)
plt.title("$||P^T 1 - b||$")
plt.plot(np.log(np.asarray(Err_q)), linewidth = 2)

plt.savefig("CurveError.png")
plt.show()
```

```
(150,)
```





Compute the final matrix P.

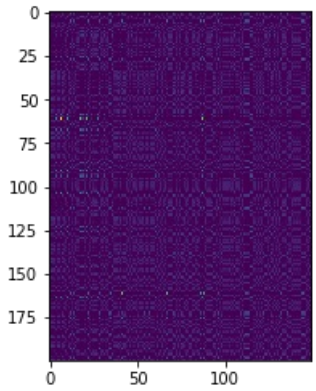
In [62]:

```
P = np.dot(np.dot(np.diag(u),K),np.diag(v))
```

Display it.

In [63]:

```
plt.imshow(P);
```

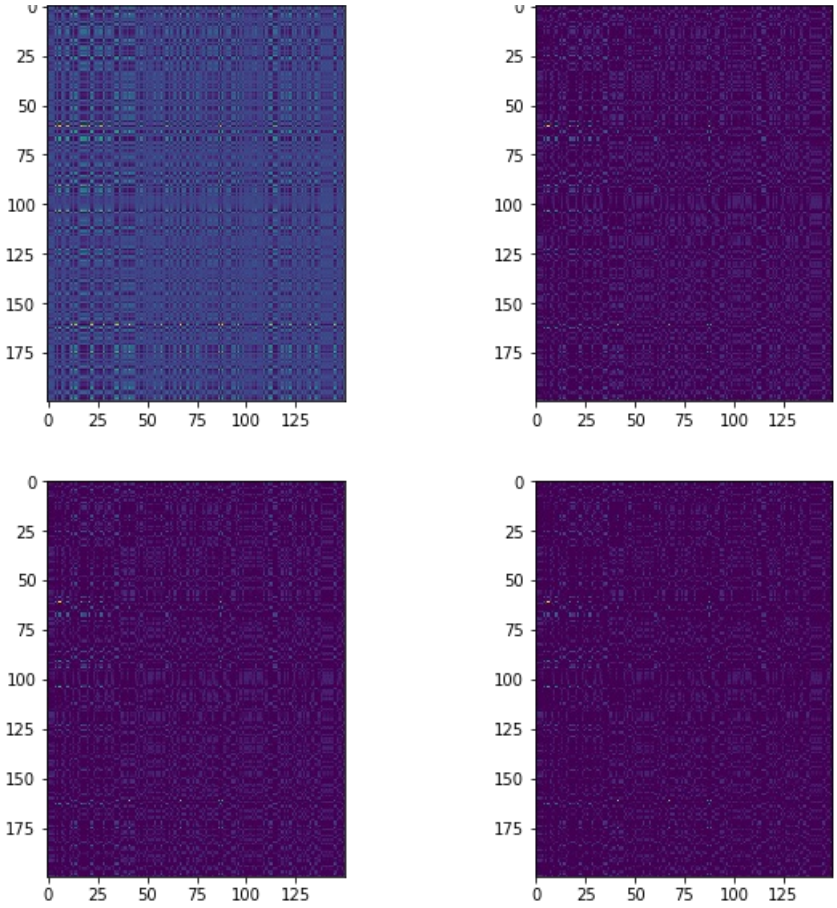


Exercise 2

Display the regularized transport solution for various values of ε. For a too small value of ε, what do you observe ?

In [64]:

```
plt.figure(figsize = (10,10))
glist = [.1,.01,.008,.004]
niter = 300

clamp = lambda x,a,b: min(max(x,a),b)

for k in range(len(glist)):
    epsilon = glist[k]
    K = np.exp(-C/epsilon)
    v = np.ones(N[1])

    for i in range(niter):
        u = a / (np.dot(K,v))
        v = b /(np.dot(np.transpose(K),u))

    P = np.dot(np.dot(np.diag(u),K),np.diag(v))
    #imageplot(clamp(Pi,0,np.min(1/np.asarray(N))*.3),"$\gamma=$ %.3f" %gamma, [2,2,k+1])
    plt.subplot(2,2,k+1)
    plt.savefig("curve.png")
    plt.imshow(np.clip(P,0,np.min(1/np.asarray(N))*.3));
    #"$\gamma=$ %.3f" %gamma, [2,2,k+1])
```



When ε decreases, the solution become "Sparser" and closer to a solution that would have been given by the unrelaxed Kantorovich solution.

Compute the obtained optimal P.

In [65]:

```
P
```

Out[65]:

```
array([[5.26685866e-05, 9.41543200e-09, 6.47204748e-08, ...,
        1.39237522e-05, 4.34754959e-06, 5.48378962e-10],
       [1.53776916e-21, 2.87871168e-07, 5.50079818e-08, ...,
        1.65982092e-11, 5.02427648e-26, 1.91866825e-06],
       [9.60054490e-16, 2.87819197e-05, 1.16970339e-05, ...,
        6.03283567e-08, 1.63356043e-19, 7.06862776e-05],
       ...,
       [1.42034786e-16, 1.62718315e-05, 5.89005956e-06, ...,
        1.96541993e-08, 1.87629283e-20, 4.65760745e-05],
       [3.87302577e-30, 1.03971199e-10, 7.12668666e-12, ...,
        4.54593519e-17, 1.34456047e-35, 2.69021263e-09],
       [5.46657899e-10, 1.75183777e-04, 1.91286614e-04, ...,
        4.06232619e-05, 8.07569071e-13, 1.16366936e-04]])
```

In [66]:

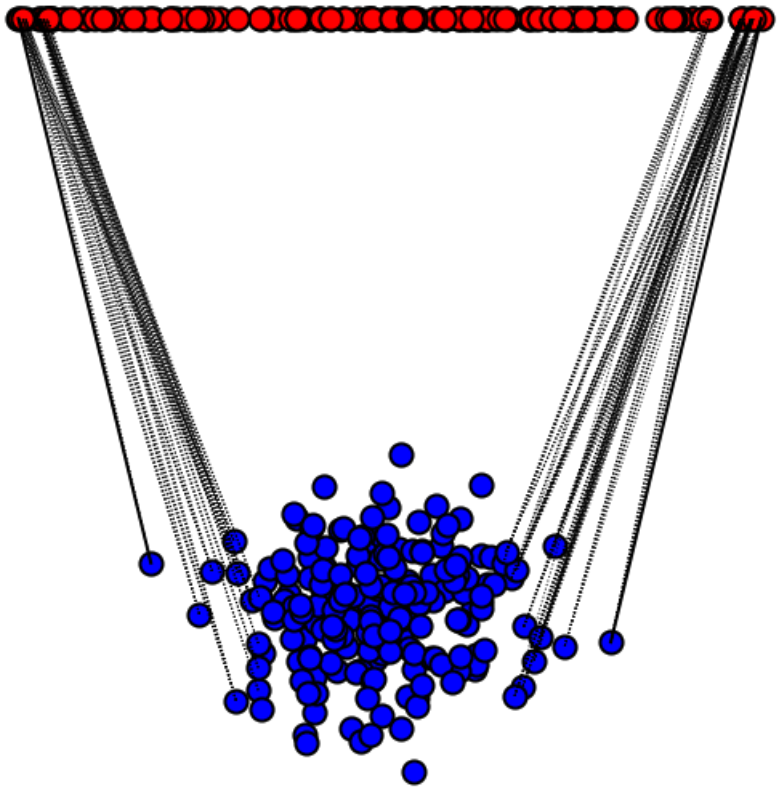
```
P = np.dot(np.dot(np.diag(u),K),np.diag(v))
```

Keep only the highest entries of the coupling matrix, and use them to draw a map between the two clouds. First we draw "strong" connexions, i.e. linkds (i, j) corresponding to large values of $P_{i,j}$. We then draw weaker connexions.

In [67]:

```
plt.figure(figsize=(10,10))

plotp(x, 'b')
```

```
plotp(y, 'r')

A = P * (P > np.max(P)*.8)
i,j = np.where(A != 0)
plt.plot([x[0,i],y[0,j]],[x[1,i],y[1,j]],'k',lw = 2)

A = P * (P > np.max(P)*.2)
i,j = np.where(A != 0)
plt.plot([x[0,i],y[0,j]],[x[1,i],y[1,j]],'k:',lw = 1)

plt.axis("off")
plt.xlim(min(np.min(x[0,:]), np.min(np.min(y[0,:]) ))-.1,max(np.max(x[0,:]), np.max(np.max(y[0,:])))
+.1)
plt.ylim(min(np.min(x[1,:]), np.min(np.min(y[1,:]) ))-.1,max(np.max(x[1,:]), np.max(np.max(y[1,:])))
+.1)

plt.savefig("ResultPartition.png")
plt.show()
```



Transport Between Histograms

We now consider a different setup, where the histogram values a, b are not uniform, but the measures are defined on a uniform grid $x_i = y_i = i/n$. They are thue often refered to as "histograms".

Size n of the histograms.

In [256]:

```
N = 200
#N = 10
```

We use here a 1-D square Euclidean metric.

In [257]:

```
t = np.arange(0,N)/N
```

Define the histogram a, b as translated Gaussians.

In [258]:

```
Gaussian = lambda t0,sigma: np.exp(-(t-t0)**2/(2*sigma**2))
normalize = lambda p: p/np.sum(p)
def Uniform(t0, l):
    res = np.zeros(t.shape[0])
    for i,el in enumerate(t):
        if el < t0 - l or el > t0 + l:
            res[i] = 0
        else:
            res[i] = 1
    return res

sigma = .06;
a = Gaussian(.25,sigma)
b = Uniform(.8,0.2)
```

In [259]:

```
b
```

Out[259]:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Add some minimal mass and normalize.

In [260]:

```
vmin = .02;
a = normalize( a+np.max(a)*vmin)
b = normalize( b+np.max(b)*vmin)
```

In [261]:

```
b
```

Out[261]:

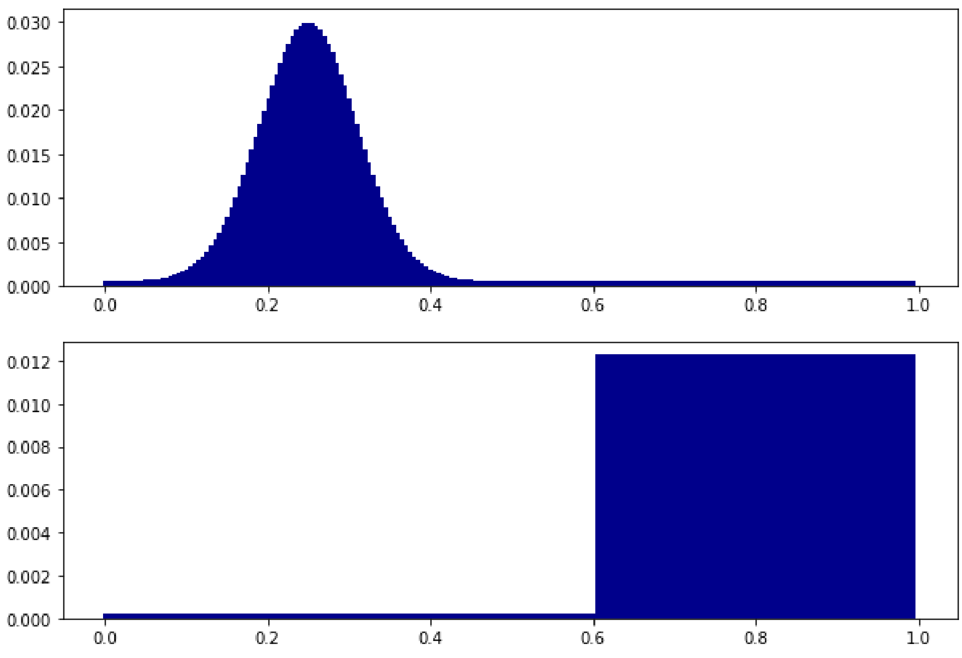
```
array([0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
```

```
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.00024096, 0.00024096, 0.00024096, 0.00024096,
       0.00024096, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916,
       0.01228916, 0.01228916, 0.01228916, 0.01228916, 0.01228916])
```

Display the histograms.

In [262]:

```
plt.figure(figsize = (10,7))

plt.subplot(2, 1, 1)
plt.bar(t, a, width = 1/len(t), color = "darkblue")
plt.subplot(2, 1, 2)
plt.bar(t, b, width = 1/len(t), color = "darkblue")
plt.savefig("dataHistogram.png")
plt.show()
```



Regularization strength γ.

In [263]:

```
epsilon = (.04)**2
```

The Gibbs kernel is a Gaussian convolution,

$$K_{i,j} \equiv e^{-(i/N-j/N)^2/\varepsilon}.$$

In [264]:

```
[Y,X] = np.meshgrid(t,t)
K = np.exp(-(X-Y)**2/epsilon)
```

Initialization of $v = \mathbb{I}_N$.

In [265]:

```
v = np.ones(N)
```

One sinkhorn iteration.

In [266]:

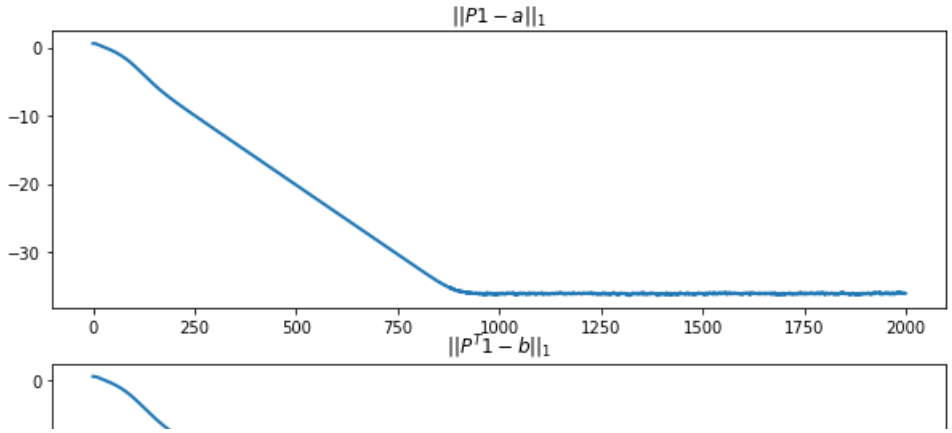
```
u = a / (np.dot(K,v))
v = b / (np.dot(np.transpose(K),u))
```

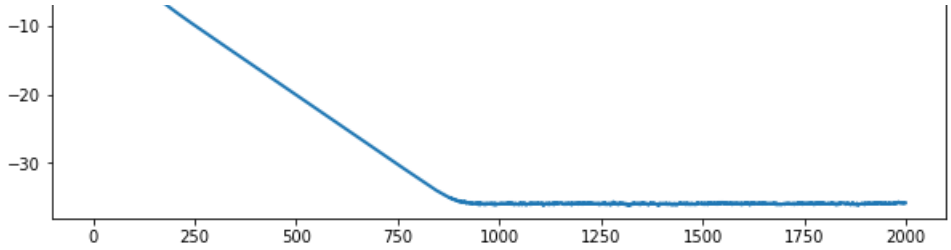
Exercise 3

Implement Sinkhorn algorithm. Display the evolution of the constraints satisfaction errors $\|P\mathbb{I} - a\|_1$, $\|P^\top \mathbb{I} - b\|_1$. You need to think how to compute it from (u, v). Display the violation of constraint error in log-plot.

In [267]:

```
from numpy import linalg

v = np.ones(N)
niter = 2000
Err_p = []
Err_q = []

for i in range(niter):
    # sinkhorn step 1
    u = a / (np.dot(K,v))
    # error computation
    r = v*np.dot(np.transpose(K),u)
    Err_q = Err_q + [linalg.norm(r - b, 1)]
    # sinkhorn step 2
    v = b /(np.dot(np.transpose(K),u))
    s = u*np.dot(K,v)
    Err_p = Err_p + [linalg.norm(s - a,1)]


plt.figure(figsize = (10,7))

plt.subplot(2,1,1)
plt.title("$||P1 -a||_1$")
plt.plot(np.log(np.asarray(Err_p)), linewidth = 2)

plt.subplot(2,1,2)
plt.title("$||P^T 1 -b||_1$")
plt.plot(np.log(np.asarray(Err_q)), linewidth = 2)

plt.savefig("ConstraintsHistograms003.png")
plt.show()
```

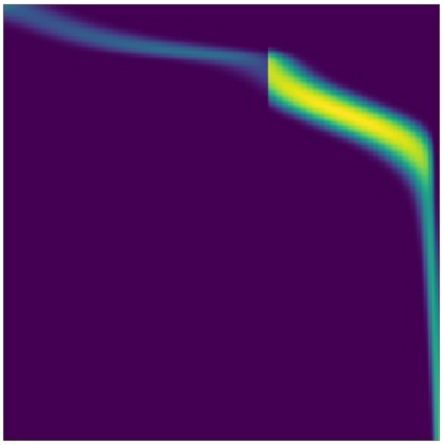




Display the coupling. Use a log domain plot to better vizualize it.

In [268]:

```
P = np.dot(np.dot(np.diag(u),K),np.diag(v))
plt.figure(figsize=(5,5))
plt.imshow(np.log(P+1e-5))
plt.savefig("CouplingHistograms.png")
plt.axis('off');
```



One can compute an approximation of the transport plan between the two measure by computing the so-called barycentric projection map

$$t_i \in [0,1] \longmapsto s_j \equiv \frac{\sum_j P_{i,j} t_j}{\sum_j P_{i,j}} = \frac{[u \odot K(v \odot t)]_j}{a_i}.$$

where $\odot$ and $\frac{\cdot}{\cdot}$ are the enry-wise multiplication and division.

This computation can thus be done using only multiplication with the kernel K.

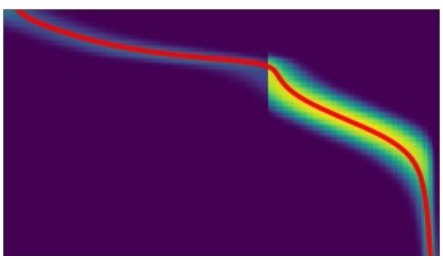
In [269]:

```
s = np.dot(K,v*t)*u/a
```

Display the transport map, super-imposed over the coupling.

In [270]:

```
plt.figure(figsize=(5,5))
plt.imshow(np.log(P+1e-5))
plt.plot(s*N,t*N, 'r', linewidth=3);
plt.savefig("Map01.png")
plt.axis('off');
```



Exercise (bonus)

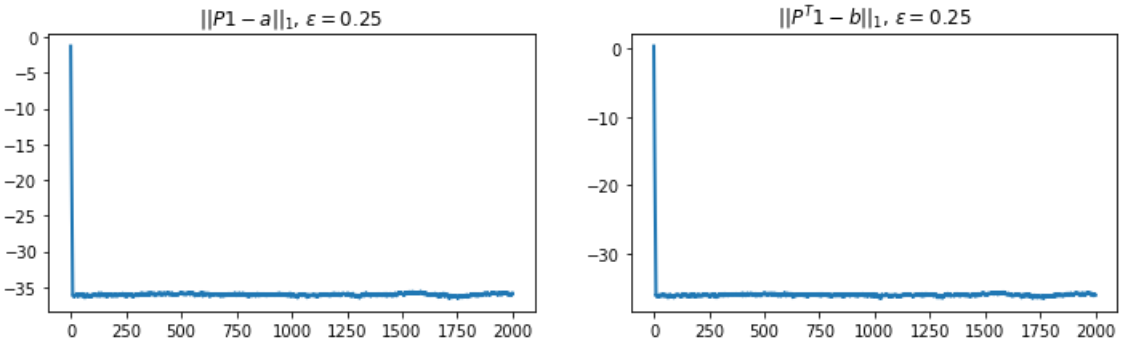
Try different regularization strength ϵ.

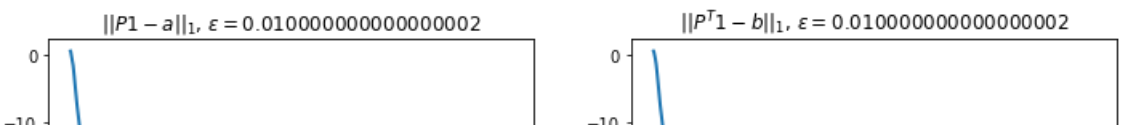
In [271]:

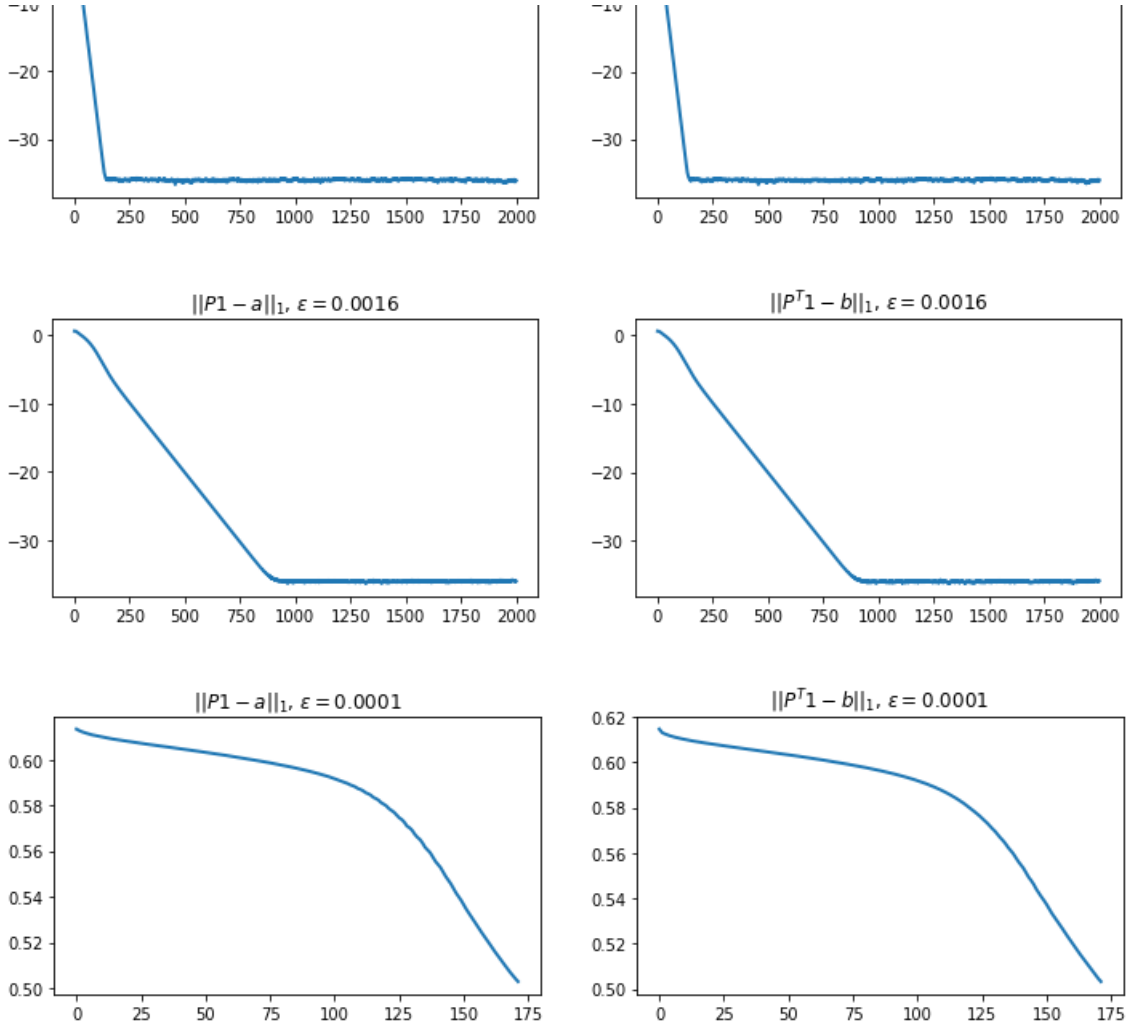
```
epsilons = [(.5)**2, (.1)**2, (.04)**2, (.01)**2]
```

In [272]:

```
[Y,X] = np.meshgrid(t,t)
K = np.exp(-(X-Y)**2/epsilon)
v = np.ones(N)
u = a / (np.dot(K,v))
v = b / (np.dot(np.transpose(K),u))


for index, epsilon in enumerate(epsilons):
    K = np.exp(-(X-Y)**2/epsilon)
    v = b / (np.dot(np.transpose(K),u))

    v = np.ones(N)
    niter = 2000
    Err_p = []
    Err_q = []

    for i in range(niter):
        # sinkhorn step 1
        u = a / (np.dot(K,v))
        # error computation
        r = v*np.dot(np.transpose(K),u)
        Err_q = Err_q + [linalg.norm(r - b, 1)]
        # sinkhorn step 2
        v = b /(np.dot(np.transpose(K),u))
        s = u*np.dot(K,v)
        Err_p = Err_p + [linalg.norm(s - a,1)]
    plt.figure(figsize = (25,7))

    plt.subplot(2,4,2*index + 1)
    plt.title("$||P1 -a||_1$, $\epsilon = $" + str(epsilon))
    plt.plot(np.log(np.asarray(Err_p)), linewidth = 2)

    plt.subplot(2, 4, 2*index + 2)
    plt.title("$||P^T 1 -b||_1$, $\epsilon = $" + str(epsilon))
    plt.plot(np.log(np.asarray(Err_q)), linewidth = 2)
```







As with all the previous cases, the convergence is faster as epsilon grows. For Epsilon too small, (last case), the elements become Nan as we divide by zero in the Sinkhorn algorithm.

Using GPUs

We will use here Pytorch to implement Sinkhorn on the GPU. If you are running the code on Google Colab, this means you need to switch on in the preferences the use of a GPU.

In [273]:

```
import torch
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print( device )
```

```
cuda:0
```

Since CUDA uses float number on 32 bits, one needs to use a quite large value for ε to avoid overflow.

In [274]:

```
epsilon = (.06)**2
K = np.exp(-(X-Y)**2/epsilon)
```

Convert Sinkohrn variables and host them on GPU (if available).

In [275]:

```
u = torch.ones(N);
v = torch.ones(N);
K1 = torch.from_numpy(K).type(torch.FloatTensor);
```

```
a1 = torch.from_numpy(a).type(torch.FloatTensor);
b1 = torch.from_numpy(b).type(torch.FloatTensor);
K1.to(device);
u.to(device); v.to(device);
a1.to(device); b1.to(device);
```

When using Pytorch, it is good practice to implement matrix operation as summation and dummy variables. We show here how to implement one iteration of Sinkhorn this way.

In [276]:

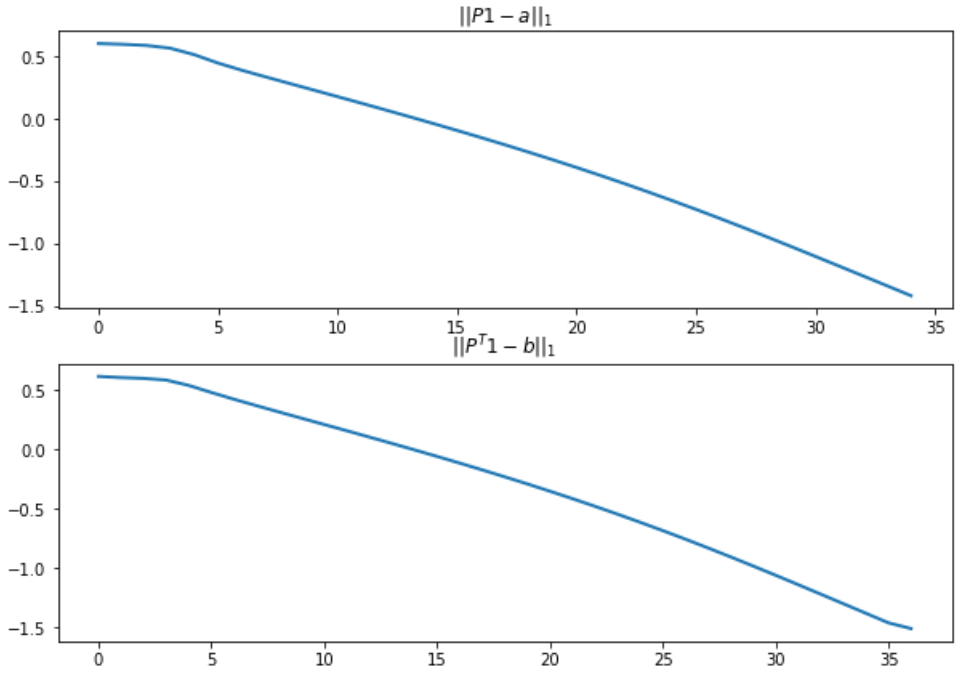
```
u = a1 / (K1 * v[None,:]).sum(1)
v = b1 / (K1 * u[:,None]).sum(0)
```

Exercise:

Implement the full algorithm.

In [277]:

```
v = torch.ones(N)
niter = 2000
Err_p = torch.zeros(niter)
Err_q = torch.zeros(niter)
for i in range(niter):
    # sinkhorn step 1
    u = a1 / (K1 * v[None,:]).sum(1)
    # error computation
    r = v*(K1 * u[:,None]).sum(0)
    Err_q[i] = torch.norm(r - b1, p=1)
    # sinkhorn step 2
    v = b1 / (K1 * u[:,None]).sum(0)
    s = u*(K1 * v[None,:]).sum(1)
    Err_p[i] = torch.norm(s - a1,p=1)
plt.figure(figsize = (10,7))
plt.subplot(2,1,1)
plt.title("$||P1 -a||_1$")
plt.plot(np.log(np.asarray(Err_p)), linewidth = 2)
plt.subplot(2,1,2)
plt.title("$||P^T 1 -b||_1$")
plt.plot(np.log(np.asarray(Err_q)), linewidth = 2)
plt.show()
```



For the epsilon selected, we encounter underflow. In order to cope with this issue, we use the log-sum-exp stabilization trick.

$\frac{n}{}$ $\frac{n}{}$

$$\forall c \in \mathbb{R},\ log(\sum_{i=1} exp(x_i)) = c + log(\sum_{i=1} exp(x_i - c))$$

Exercice To avoid underflow, replace the matrix/vector multiplication in a log-sum-exp style, and use the log-sum-exp stabilization trick.

We have the following equations for the Sinkhorn's algorithm :

$$log(u) \longleftarrow log(a) - log(\sum_i \exp(-\frac{(X_{:,i} - Y_{:,i})^2}{\varepsilon} + log(v)_i))$$

$$log(v) \longleftarrow log(b) - log(\sum_i \exp(-\frac{(X_{i,:} - Y_{i,:})^2}{\varepsilon} + log(u)_i))$$

We then apply the trick with

$$(c_u)_j = max_{1 \leq i \leq n}(-\frac{(X_{j,i} - Y_{j,i})^2}{\varepsilon} + log(v)_i)$$

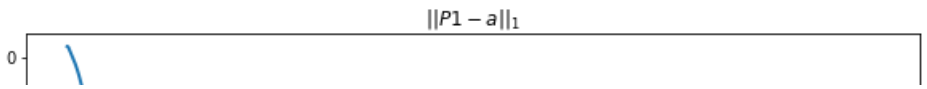
$$(c_v)_j = max_{1 \leq i \leq n}(-\frac{(X_{i,j} - Y_{i,j})^2}{\varepsilon} + log(u)_i)$$

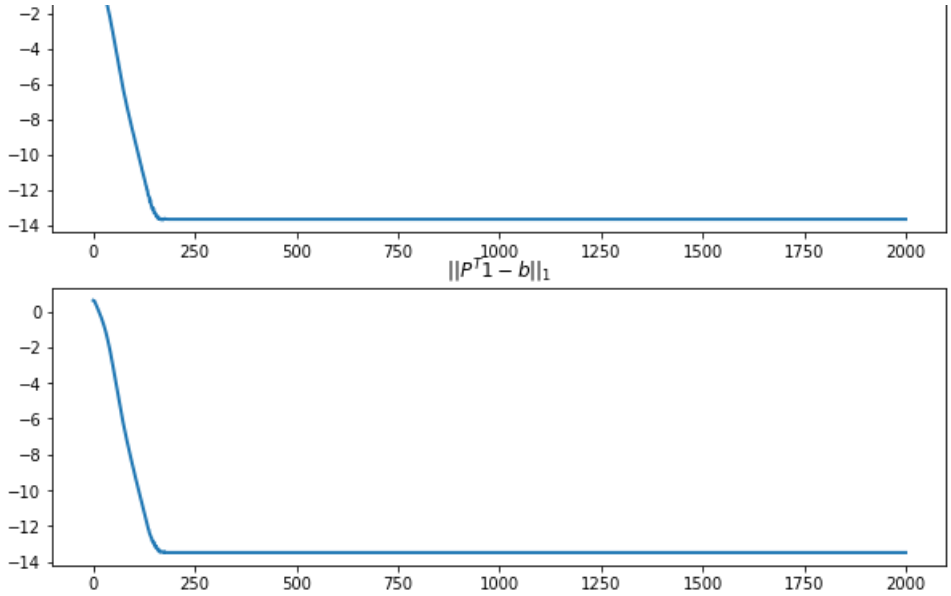
In [278]:

```
## For the log-sum-exp trick we need the maximum of the exp over line and column
la1 = np.log(a1)
lb1 = np.log(b1)
lK = torch.from_numpy(-(X-Y)**2/epsilon).type(torch.FloatTensor)


m_column = torch.max(lK, axis = 1)[0]
m_rows = torch.max(lK, axis = 0)[0]
lK_NormalizedCol = lK - m_column[:,None]
lK_NormalizedRow = lK - m_rows[None, :]
```

In [279]:

```
### We use log formulation
lv = torch.zeros(N)
niter = 2000
Err_p = torch.zeros(niter)
Err_q = torch.zeros(niter)
for i in range(niter):
    # sinkhorn step 1
    m_column = torch.max(lK + lv[None,:], axis = 1)[0]
    lK_NormalizedCol = lK + lv[None,:] - m_column[:,None]
    lu = la1 - m_column - np.log(np.exp(lK_NormalizedCol).sum(1))
    # error computation
    m_rows = torch.max(lK + lu[:,None], axis = 0)[0]
    lK_NormalizedRow = lK + lu[:,None] - m_rows[None,:]
    lr = lv + m_rows + np.log(np.exp(lK_NormalizedRow).sum(0))
    Err_q[i] = torch.norm(np.exp(lr) - b1, p=1)

    # sinkhorn step 2
    m_rows = torch.max(lK + lu[:,None], axis = 0)[0]
    lK_NormalizedRow = lK + lu[:,None] - m_rows[None,:]
    lv = lb1 - m_rows - np.log(np.exp(lK_NormalizedRow).sum(0))
    # error computation
    m_column = torch.max(lK + lv[None,:], axis = 1)[0]
    lK_NormalizedCol = lK + lv[None,:] - m_column[:,None]
    ls = lu + m_column + np.log(np.exp(lK_NormalizedCol).sum(1))
    Err_p[i] = torch.norm(np.exp(ls) - a1,p=1)
plt.figure(figsize = (10,7))
plt.subplot(2,1,1)
plt.title("$||P1 -a||_1$")
plt.plot(np.log(np.asarray(Err_p)), linewidth = 2)
plt.subplot(2,1,2)
plt.title("$||P^T 1 -b||_1$")
plt.plot(np.log(np.asarray(Err_q)), linewidth = 2)
plt.show()
```





We are happy to see that the trick works and makes the algorithm converge, contrary to when we don't use it where at $n = 35$ we already obtain numerical instability.

Wasserstein Barycenters

Instead of computing transport, we now turn to the problem of computing barycenter of R input measures $(a_k)_{k=1}^R$. A barycenter b solves

$$\min_b \sum_{k=1}^R W_\gamma(a_k, b)$$

where λ_k are positive weights with $\sum_k \lambda_k = 1$. This follows the definition of barycenters proposed in AguehCarlier.

Dimension (width of the images) N of the histograms.

In [280]:

```
N = 70
```

You need to install imageio, for instance using

> conda install -c conda-forge imageio

If you need to rescale the image size, you can use

> skimage.transform.resize

Load input histograms $(a_k)_{k=1}^R$, store them in a tensor A.

In [281]:

```
import imageio
rescale = lambda x: (x-x.min())/(x.max()-x.min())
names = ['disk','twodisks','letter-x','letter-z']
vmin = .01
A = np.zeros([N,N,len(names)])
for i in range(len(names)):
    a = imageio.imread("nt_toolbox/data/" + names[i] + ".bmp") # ,N)
    a = normalize(rescale(a)+vmin)
    A[:,:,i] = a
R = len(names)
```

In [282]:

```
A[:,:,1] = A[:,:,1] + np.flip(A[:,:,1], axis = 0)
```

In [293]:

```
A[:,:,0].max()
```

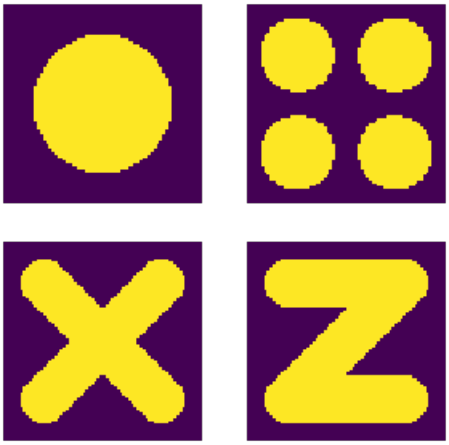
Out[293]:

```
0.0005335446381405178
```

Display the input histograms.

In [283]:

```
plt.figure(figsize=(5,5))
for i in range(R):
    plt.subplot(2,2,i+1)
    plt.imshow(A[:,:,i])
    plt.axis('off');
plt.savefig("DataBaricenter.png")
```



In this specific case, the kernel K associated with the squared Euclidean norm is a convolution with a Gaussian filter

$$K_{i,j} = e^{-\|i/N - j/N\|^2/\varepsilon}$$

where here (i,j) are 2-D indexes.

The multiplication against the kernel, i.e. $K(a)$, can now be computed efficiently, using fast convolution methods. This crucial points was exploited and generalized in SolomonEtAl to design fast optimal transport algorithm.

Regularization strength $\varepsilon > 0$.

In [294]:

```
epsilon = (.06)**2
```

Define the K kernel. We use here the fact that the convolution is separable to implement it using only 1-D convolution, which further speeds up computations.

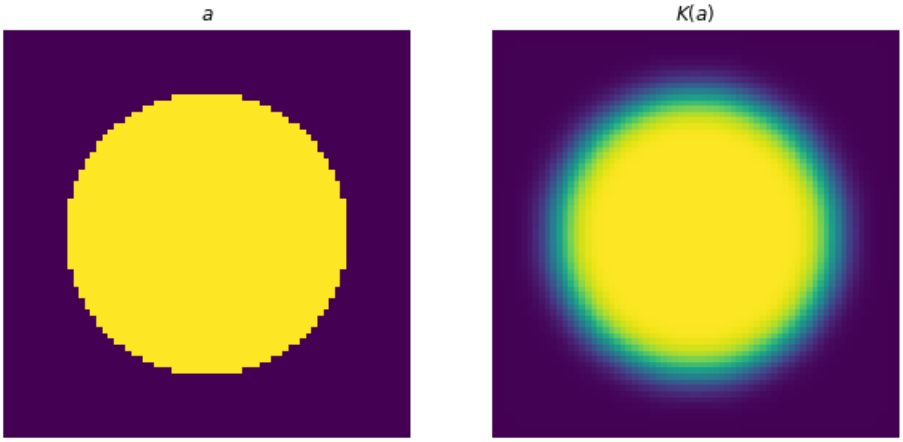
In [295]:

```
t = np.linspace(0,1,N)
[Y,X] = np.meshgrid(t,t)
K1 = np.exp(-(X-Y)**2/epsilon)
K = lambda x: np.dot(np.dot(K1,x),K1)
```

Display the application of the K kernel on one of the input histogram.

In [296]:

```
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.imshow(A[:,:,0])
plt.title("$a$")
plt.axis('off');
plt.subplot(1,2,2)
plt.imshow(K(A[:,:,0]))
plt.title("$K(a)$")
plt.axis('off');
```



Weights λ_k for isobarycenter.

In [297]:

```
lambd = np.ones(R)/R
```

It is shown in BenamouEtAl that the problem of Barycenter computation boilds down to optimizing over couplings $(P_k)_{k=1}^R$, and that this can be achieved using iterative a Sinkhorn-like algorithm, since the optimal coupling has the scaling form

$$P_k = diag\,(u_k)\,K\,diag\,(v_k)$$

for some unknown positive weights (u_k, v_k).

Initialize the scaling factors $(u_k, v_k)_k$, store them in matrices.

In [298]:

```
v = np.ones([N,N,R])
u = np.copy(v)
```

The first step of the Bregman projection method corresponds to the projection on the fixed marginals constraints $P^k\mathbb{I} = a_k$. This is achieved by updating

$$\forall k = 1, \ldots, R, \quad u_k \longleftarrow \frac{a_k}{K(v_k)}.$$

In [299]:

```
for k in range(R):
    u[:,:,k] = A[:,:,k]/K(v[:,:,k])
```

The second step of the Bregman projection method corresponds to the projection on the equal marginals constraints $\forall k, P_k^\top \mathbb{I} = b$ for a common barycenter target b. This is achieved by first computing the target barycenter b using a geometric means

$$\log(b) \equiv \sum_k \lambda_k \log(u_k \odot K(v_k)).$$

In [300]:

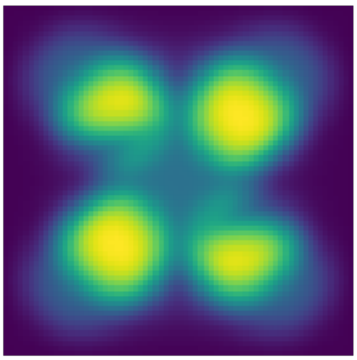
```
b = np.zeros(N)
for k in range(R):
```

```
    b = b + lambd[k] * np.log(np.maximum(1e-19*np.ones(len(v[:,:,k])), v[:,:,k]*K(u[:,:,k])))
b = np.exp(b)
```

Display b.

In [301]:

```
plt.imshow(b);
plt.axis('off');
plt.savefig("BarycenterofData.png")
```



And then one can update the scaling by a Sinkhorn step using this newly computed histogram b as follow (note that $K = K^\top$ here):

$$\forall k = 1, \ldots, R, \quad v_k \longleftarrow \frac{b}{K(u_k)}.$$

In [302]:

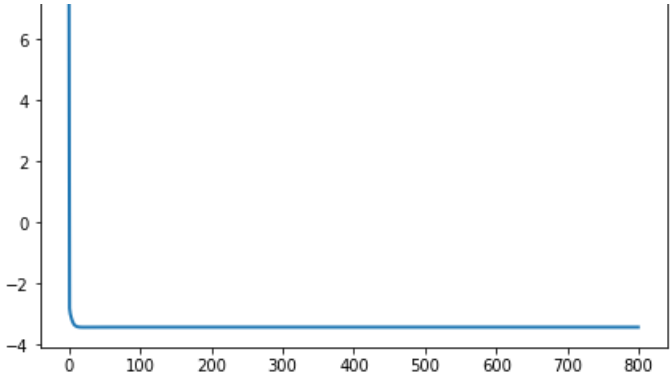
```
for k in range(R):
    v[:,:,k] = b/K(u[:,:,k])
```

Exercise 4

Implement the iterative algorithm to compute the iso-barycenter of the measures. Plot the decay of the error $\sum_k \|P_k \mathbb{I} - a_k\|$.

In [303]:

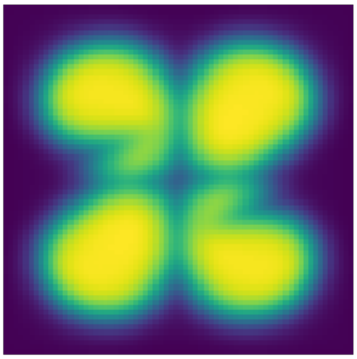
```
from numpy import linalg

niter = 800
v = np.ones([N,N,R])
u = np.copy(v)
Err_q = np.zeros(niter)

for i in range(niter):

    for k in range(R):
        Err_q[i] = Err_q[i] + linalg.norm(u[:,:,k]*K(v[:,:,k]) - A[:,:,k], 1)
        u[:,:,k] = A[:,:,k]/K(v[:,:,k])

    b = np.zeros(N)
    for k in range(R):
        b = b + lambd[k] * np.log(np.maximum(1e-19*np.ones(len(v[:,:,k])), v[:,:,k]*K(u[:,:,k])))
    b = np.exp(b)

    for k in range(R):
        v[:,:,k] = b/K(u[:,:,k])

plt.figure(figsize=(7,5))
plt.plot(np.log(Err_q),linewidth = 2)
plt.show()
```



Display the barycenter.

In [304]:

```
plt.imshow(b)
plt.axis('off');
```



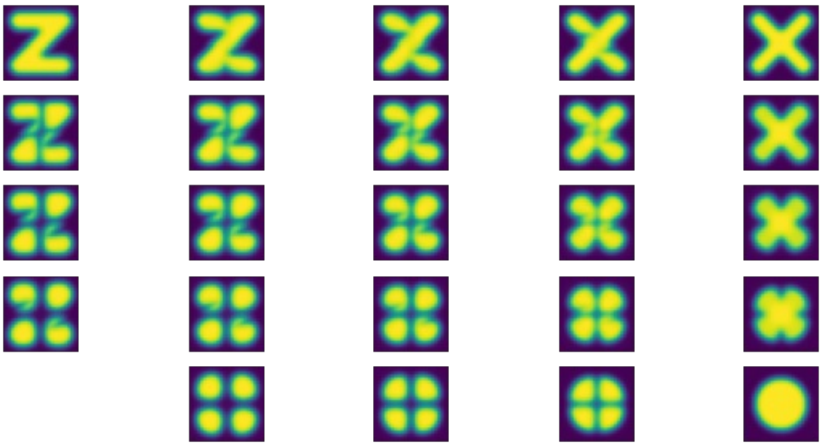
Exercise 5

Compute barycenters for varying weights λ corresponding to a bilinear interpolation inside a square.

In [305]:

```
# Insert your code here.

m = 5
[T,S] = np.meshgrid(np.linspace(0,1,m), np.linspace(0,1,m))
T = np.ravel(T,order="F")
S = np.ravel(S,order="F")
niter = 1000

plt.figure(figsize=(10,5))
for j in range(m**2):
    # weights
    lambd = np.hstack((S[j]*T[j], (1-S[j])*T[j], S[j]*(1-T[j]), (1-S[j])*(1-T[j])))
    # computation
    v = np.ones([N,N,R])
    u = np.copy(v)

    for i in range(niter):
        for k in range(R):
            u[:,:,k] = A[:,:,k]/K(v[:,:,k])
        b = np.zeros(N)
        for k in range(R):
            b = b + lambd[k] * np.log(np.maximum(1e-19*np.ones(len(v[:,:,k])), v[:,:,k]*K(u[:,:,k]))
)
        b = np.exp(b)
        for k in range(R):
            v[:,:,k] = b/K(u[:,:,k])

    # display
    plt.subplot(m,m,j+1)
    plt.imshow(b)
    plt.axis('off')
```

```
plt.savefig("barycenterSquare.png")
```



Bibliography

- [Villani] C. Villani, (2009). Optimal transport: old and new, volume 338. Springer Verlag.
- [Cuturi] M. Cuturi, (2013). Sinkhorn distances: Lightspeed computation of optimal transport. In Burges, C. J. C., Bottou, L., Ghahramani, Z., and Weinberger, K. Q., editors, Proc. NIPS, pages 2292-2300.
- [AguehCarlier] M. Agueh, and G Carlier, (2011). Barycenters in the Wasserstein space. SIAM J. on Mathematical Analysis, 43(2):904-924.
- [CuturiDoucet] M. Cuturi and A. Doucet (2014). Fast computation of wasserstein barycenters. In Proc. ICML.
- [BauschkeLewis] H. H. Bauschke and A. S. Lewis. Dykstra's algorithm with Bregman projections: a convergence proof. Optimization, 48(4):409-427, 2000.
- [Sinkhorn] R. Sinkhorn. A relationship between arbitrary positive matrices and doubly stochastic matrices. Ann. Math. Statist., 35:876-879, 1964.
- [SolomonEtAl] J. Solomon, F. de Goes, G. Peyr , M. Cuturi, A. Butscher, A. Nguyen, T. Du, and L. Guibas. Convolutional Wasserstein distances: Efficient optimal transportation on geometric domains. Transaction on Graphics, Proc. SIGGRAPH, 2015.
- [BenamouEtAl] J-D. Benamou, G. Carlier, M. Cuturi, L. Nenna, G. Peyr . Iterative Bregman Projections for Regularized Transportation Problems. SIAM Journal on Scientific Computing, 37(2), pp. A1111-A1138, 2015.

Assignment3

Advanced Topics on Sinkhorn Algorithm

This numerical tour explore several extensions of the basic Sinkhorn method.

In [8]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
```

Log-domain Sinkhorn

For simplicity, we consider uniform distributions on point clouds, so that the associated histograms are $(a, b) \in \mathbb{R}^n \times \mathbb{R}^m$ being constant $1/n$ and $1/m$.

In [9]:

```
n = 100
m = 200
```

```
a = np.ones((n,1))/n
b = np.ones((1,m))/m
```

Point clouds x and y.

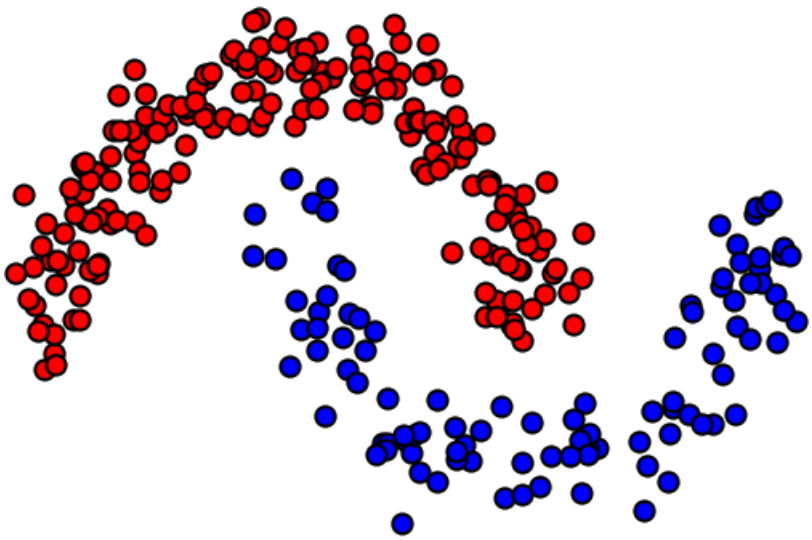
In [10]:

```
x = np.random.rand(2,n)-.5
theta = 2*np.pi*np.random.rand(1,m)
r = .8 + .2*np.random.rand(1,m)
y = np.vstack((np.cos(theta)*r,np.sin(theta)*r))

X = datasets.make_moons(n_samples=2*max(m,n), noise = 0.1)
x, y = (X[0][X[1] == 1])[:n].T*0.1, (X[0][X[1] == 0])[:m].T*0.1
```

Display of the two clouds.

In [11]:

```
plotp = lambda x,col: plt.scatter(x[0,:], x[1,:], s=150, edgecolors="k", c=col, linewidths=2)
plt.figure(figsize=(10,7))
plotp(x, 'b')
plotp(y, 'r')
plt.axis("off");
```



Cost matrix $C_{i,j} = \|x_i - y_j\|^2$.

In [12]:

```
def distmat(x,y):
    return np.sum(x**2,0)[:,None] + np.sum(y**2,0)[None,:] - 2*x.transpose().dot(y)
C = distmat(x,y)
```

Sinkhorn algorithm is originally implemented using matrix-vector multipliciation, which is unstable for small epsilon. Here we consider a log-domain implementation, which operates by iteratively updating so-called Kantorovitch dual potentials $(f,g) \in \mathbb{R}^n \times \mathbb{R}^m$.

The update are obtained by regularized c-transform, which reads

$$f_i \leftarrow \min_\varepsilon^b (C_{i,\cdot} - g)$$
$$g_j \leftarrow \min_\varepsilon^a (C_{\cdot,j} - f),$$

where the regularized minimum operator reads

$$\min_\varepsilon^a(h) \equiv -\varepsilon \log \sum_i a_i e^{-h_i/\varepsilon}.$$

In [13]:

```
def mina_u(H,epsilon): return -epsilon*np.log( np.sum(a * np.exp(-H/epsilon),0) )
def minb_u(H,epsilon): return -epsilon*np.log( np.sum(b * np.exp(-H/epsilon),1) )
```

The regularized min operator defined this way is non-stable, but it can be stabilized using the celebrated log-sum-exp trick, wich relies on the fact that for any constant $c \in \mathbb{R}$, one has

$$\min_\varepsilon^a(h+c) = \min_\varepsilon^a(h) + c,$$

and stabilization is achieved using $c = \min(h)$.

In [14]:

```
def mina(H,epsilon): return mina_u(H-np.min(H,0),epsilon) + np.min(H,0);
def minb(H,epsilon): return minb_u(H-np.min(H,1)[:,None],epsilon) + np.min(H,1);
```

Value of ε.

In [15]:

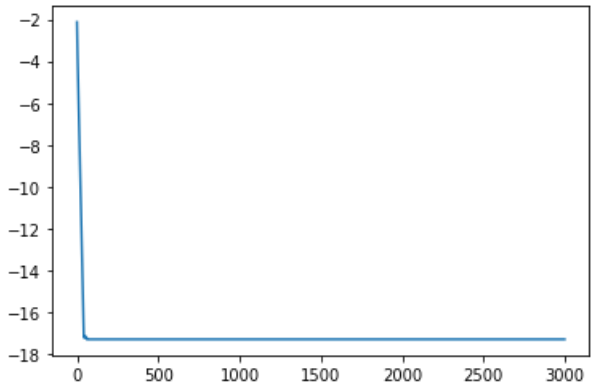
```
epsilon = .01
```

Exercise 1

Implement Sinkhorn in log domain.

In [16]:

```
def Sinkhorn(C,epsilon,f,niter = 500):
    Err = np.zeros(niter)
    for it in range(niter):
        g = mina(C-f[:,None],epsilon)
        f = minb(C-g[None,:],epsilon)
        # generate the coupling
        P = a * np.exp((f[:,None]+g[None,:]-C)/epsilon) * b
        # check conservation of mass
        Err[it] = np.linalg.norm(np.sum(P,0)-b,1)
    return (P,Err)
# run with 0 initialization for the potential f
(P,Err) =  Sinkhorn(C,epsilon,np.zeros(n),3000)
plt.plot(np.log10(Err));
```



Exercise 2

Study the impact of ε on the convergence rate of the algorithm. Comparison woth normal

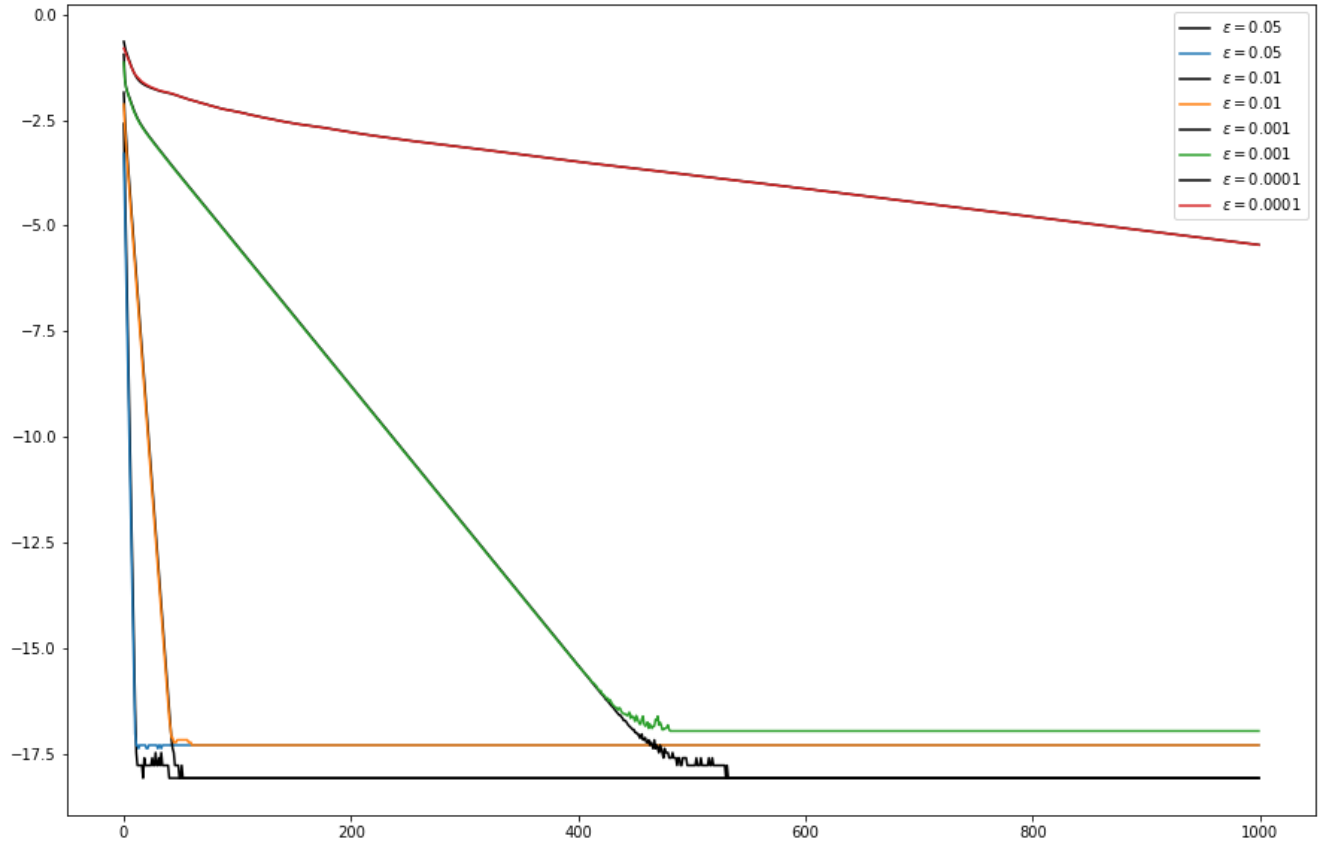
In [17]:

```
## Iterative Bragman
from numpy import linalg

def iterativeBregman(C,epsilon, N, niter = 5000):
    Err_p = []
    Err_q = []
```

```
    K = np.exp(-C/epsilon)
    v = np.ones(N[1])
    for i in range(niter):
        # sinkhorn step 1
        u = a[:,0] / (np.dot(K,v))
        # error computation
        r = (v*np.dot(np.transpose(K),u))[None, :]
        Err_q = Err_q + [linalg.norm(r - b, 1)]
        # sinkhorn step 2
        v = b[0,:] /(np.dot(np.transpose(K),u))
        s = u*np.dot(K,v)
    return np.dot(np.dot(np.diag(u),K),np.diag(v)), Err_q

    plt.plot(np.log(np.asarray(Err_q)), linewidth = 2)
```

In [18]:

```
plt.figure(figsize=(15,10))
for epsilon in (.05, .01, .001, .0001):
    (P,Err) =  Sinkhorn(C,epsilon,np.zeros(n),1000)
    (Pbre, ErrBret) = iterativeBregman(C, epsilon, [n,m], niter = 1000)
    plt.plot(np.log10(np.asarray(ErrBret)), 'k-', label='$\epsilon=$' + str(epsilon) )
    plt.plot(np.log10(Err), label='$\epsilon=$' + str(epsilon))
plt.legend();
```



There is hardly any difference between the Sinkhorn log-domain and the Bregman classic algorithm. Although we have seen in our previous assignment that log-domain algorithm significantly improve the results our histogram matching example.

Wasserstein Flow for Matching

We aim at performing a "Lagrangian" gradient (also called Wasserstein flow) descent of Wasserstein distance, in order to perform a non-parametric fitting. This corresponds to minimizing the energy function

$$\mathcal{E}(z) \equiv W_\varepsilon \left(\frac{1}{n}\sum_i \delta_{z_i}, \frac{1}{m}\sum_i \delta_{y_i} \right).$$

Here we have denoted the Sinkhorn score as

$$W_\varepsilon(\alpha,\beta) \equiv \langle P, C\rangle - \varepsilon \mathrm{KL}(P|ab^\top)$$

where $\alpha = \frac{1}{\sum_i} \delta_{\cdot}$ and $\beta = \frac{1}{\sum_i} \delta_{\cdot}$ are the measures (beware that C depends on the points positions).

$n \sum_i a_i$ $m \sum_i b_i$

In [19]:

```
z = x # initialization
```

The gradient of this energy reads

$$(\nabla \mathcal{E}(z))_i = \sum_j P_{i,j}(z_i - y_j) = a_i z_i - \sum_j P_{i,j} y_j,$$

where P is the optimal coupling. It is better to consider a renormalized gradient, which corresponds to using the inner product associated to the measure a on the deformation field, in which case

$$(\bar{\nabla} \mathcal{E}(z))_i = z_i - \bar{y}_i \quad \text{where} \quad \bar{y}_i \equiv \frac{\sum_j P_{i,j} y_j}{a_i}.$$

Here $\bar{y}_i$ is often called the "barycentric projection" associated to the coupling matrix P.

First run Sinkhorn, beware you need to recompute the cost matrix at each step.

In [20]:

```
epsilon = .01
niter = 300
(P,Err) = Sinkhorn(distmat(z,y), epsilon,np.zeros(n),niter);
```

Compute the gradient

In [21]:

```
G = z - ( y.dot(P.transpose()) ) / a.transpose()
```

In [22]:

```
np.linalg.norm(G)
```

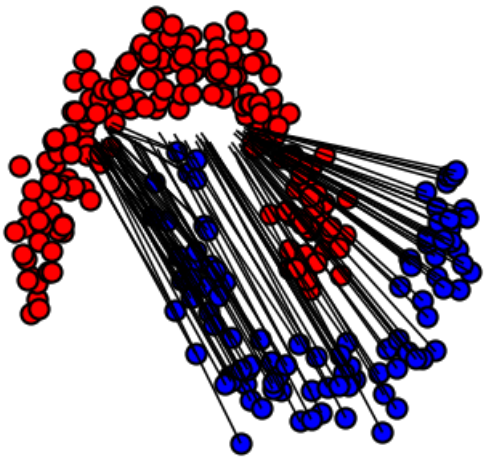
Out[22]:

```
1.3457298358293273
```

Display the gradient field.

In [23]:

```
plt.figure(figsize=(6,6))
plotp(x, 'b')
plotp(y, 'r')
for i in range(n):
    plt.plot([x[0,i], x[0,i]-G[0,i]], [x[1,i], x[1,i]-G[1,i]], 'k')
plt.axis("off");
```



Set the descent step size.

In [24]:

```
tau = 10  #.1
```

Update the point cloud.

In [25]:

```
z = z - tau * G
```

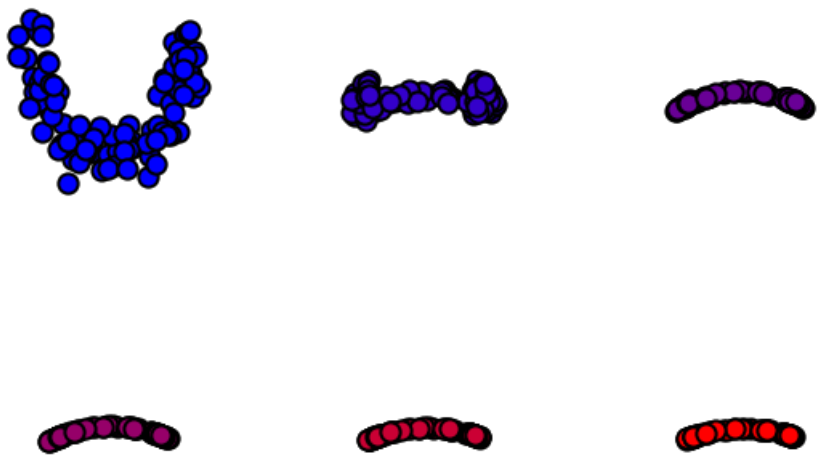
Exercise 3

Implement the gradient flow.

In [26]:

```
plt.figure(figsize=(10,7))
z = x; # initialization
tau = 0.3; # step size for the descent
giter = 40; # iter for the gradient descent
ndisp = np.round( np.linspace(0,giter-1,6) )
kdisp = 0
f = np.zeros(n) # use warm restart in the following
for j in range(giter):
    # drawing
    if ndisp[kdisp]==j:
        plt.subplot(2,3,kdisp+1)
        s = j/(giter-1)
        col = np.array([s,0,1-s])[None,:]
        plotp(z, col )
        plt.axis("off")
        kdisp = kdisp+1
        print(np.linalg.norm(G))
    # Sinkhorn
    (P,Err) = Sinkhorn(distmat(z,y), epsilon,f,niter)
    # gradient
    G = z - ( y.dot(P.transpose()) ) / a.transpose()
    z = z - tau * G;
```

```
1.3457298358293273
0.12706849666125747
0.02577454619712953
0.014616561505810083
0.009039252619095484
0.006064376436220336
```



Exercise 4

Show the evolution of the fit as ε increases. What do you observe. Replace the Sinkhorn score $W_\varepsilon(\alpha,\beta)$ by the Sinkhorn divergence $W_\varepsilon(\alpha,\beta) - W_\varepsilon(\alpha,\alpha)/2 - W_\varepsilon(\beta,\beta)/2$

Evolution with ε

In [27]:

```
epsilons = [.05,0.01,0.001, 0.0001]
for index, epsilon in enumerate(epsilons):
    z = x; # initialization
    tau = 0.3; # step size for the descent
    giter = 50; # iter for the gradient descent
    ndisp = np.round( np.linspace(0,giter-1,6) )
    kdisp = 0
    f = np.zeros(n) # use warm restart in the following
    for j in range(giter):
        # drawing
        if giter - 1 == j:
            plt.subplot(2,2,index+1)
            s = index/(len(epsilons)-1)
            col = np.array([s,0,1-s])[None,:]
            plotp(z, col )
            plt.axis("off")
            kdisp = kdisp+1
            print(np.linalg.norm(G))
        # Sinkhorn
        (P,Err) = Sinkhorn(distmat(z,y), epsilon,f,niter)
        # gradient
        G = z - ( y.dot(P.transpose()) ) / a.transpose()
        z = z - tau * G;
```

```
1.304128507577922e-06
0.0039592977062334975
0.003083999966884972
0.0024529858120507166
```

We can see that, as epsilon decreases, the fitting becomes more faithful. The norm of the gradient shows that the state attained is final.

Generative Model Fitting

The Wasserstein is a non-parametric idealization which does not corresponds to any practical application. We consider here a simple toy example of density fitting, where the goal is to find a parameter θ to fit a deformed point cloud of the form $(g_\theta(x_i))_i$ using a Sinkhorn cost. This is ofen called a generative model in the machine learning litterature, and corresponds to the problem of shape registration in imaging.

The matching is achieved by solving

$$\min_\theta \mathcal{F}(\theta) \equiv \mathcal{E}(G_\theta(z)) = W_\varepsilon\left(\frac{1}{n}\sum_i \delta_{g_\theta(z_i)}, \frac{1}{m}\sum_i \delta_{y_i}\right),$$

where the function $G_\theta(z) = (g_\theta(z_i))_i$ operates independently on each point.

The gradient reads

$$\nabla \mathcal{F}(\theta) = \sum_i \partial g_\theta(z_i)^* [\nabla \mathcal{E}(G_\theta(z))_i],$$

where $\partial g_\theta(z_i)^*$ is the adjoint of the Jacobian of g_θ.

We consider here a simple model of affine transformation, where $\theta = (A, h) \in \mathbb{R}^{d \times d} \times \mathbb{R}^d$ and $g_\theta(z_i) = A z_i + h$.

Denoting $v_i = \nabla \mathcal{E}(G_\theta(z))_i$ the gradient of the Sinkhorn loss (which is computed as in the previous section), the gradient with respect to the parameter reads

$$\nabla_A \mathcal{F}(\theta) = \sum_i v_i z_i^\top \quad \text{and} \quad \nabla_h \mathcal{F}(\theta) = \sum_i v_i.$$

Generate the data.

In [108]:

```
z = np.random.randn(2,n)*.2
y = np.random.randn(2,m)*.2
y[0,:] = y[0,:]*.05 + 1
```

Initialize the parameters.

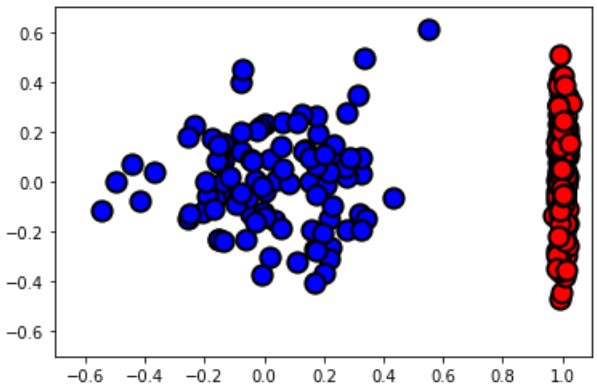
In [109]:

```
A = np.eye(2)
h = np.zeros(2)
```

Display the clouds.

In [110]:

```
plotp(A.dot(z)+h[:,None], 'b')
plotp(y, 'r')
plt.xlim(-.7,1.1)
plt.ylim(-.7,.7);
```



Run Sinkhorn.

In [111]:

```
x = A.dot(z)+h[:,None]
f = np.zeros(n)
(P,Err) = Sinkhorn(distmat(x,y), epsilon,f,niter)
```

In [113]:

```
y.shape
```

Out[113]:

```
(2, 200)
```

Compute gradient with respect to positions.

In [32]:

```
v = a.transpose() * x - y.dot(P.transpose())
```

gradient with respect to parameters

In [33]:

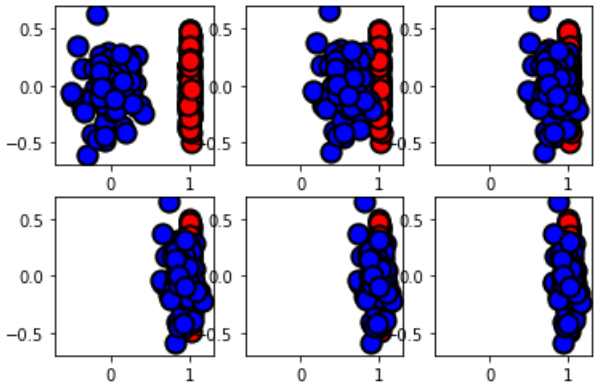
```
nabla_A = v.dot(z.transpose())
nabla_h = np.sum(v,1)
```

Exercise 5

Implement the gradient descent.

In [34]:

```
A = np.eye(2)
h = np.zeros(2)
# step size for the descent
tau_A = .8
tau_h = .1
# #iter for the gradient descent
giter = 40
ndisp = np.round( np.linspace(0,giter-1,6) )
kdisp = 0
f = np.zeros(n)
for j in range(giter):
    x = A.dot(z)+h[:,None]
    if ndisp[kdisp]==j:
        plt.subplot(2,3,kdisp+1)
        plotp(y, 'r')
        plotp(x, 'b')
        kdisp = kdisp+1
        plt.xlim(-.7,1.3)
        plt.ylim(-.7,.7)
    (P,Err) = Sinkhorn(distmat(x,y), epsilon,f,niter)
    v = a.transpose() * x - y.dot(P.transpose())
    nabla_A = v.dot(z.transpose())
    nabla_h = np.sum(v,1)
    A = A - tau_A * nabla_A
    h = h - tau_h * nabla_h
```



Exercise 5

Test using a more complicated deformation (for instance a square being deformed by a random A.

In [132]:

```
n = 100
m = 100
```

```
a = np.ones((n,1))/n
b = np.ones((1,m))/m
z = (np.random.random((2,n)) - 0.5)*0.7
Atrue = np.random.randn(2,2)
htrue = np.random.randn(2)*0.3
y = Atrue.dot(z) + htrue[:,None]
```

In [133]:

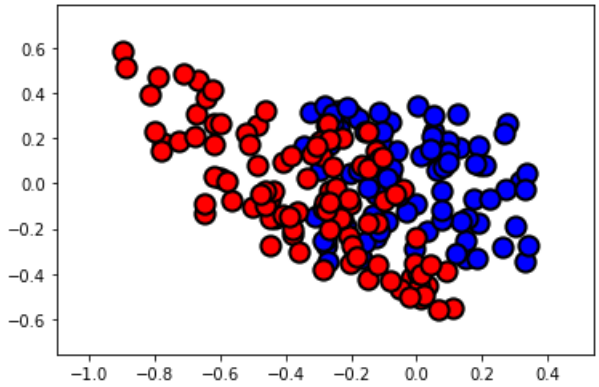
```
A = np.eye(2)
h = np.zeros(2)
```

In [134]:

```
plotp(A.dot(z)+h[:,None], 'b')
plotp(y, 'r')
plt.xlim(min(np.min(z[0]),np.min(y[0])) - 0.2,  max(np.max(z[0]),np.max(y[0])) + 0.2)
plt.ylim(min(np.min(z[1]),np.min(y[1])) - 0.2,  max(np.max(z[1]),np.max(y[1])) + 0.2)
```

Out[134]:

```
(-0.75610311884502, 0.7892239605285916)
```



In [135]:

```
x = A.dot(z)+h[:,None]
```

In [136]:

```
y.shape
```

Out[136]:

```
(2, 100)
```

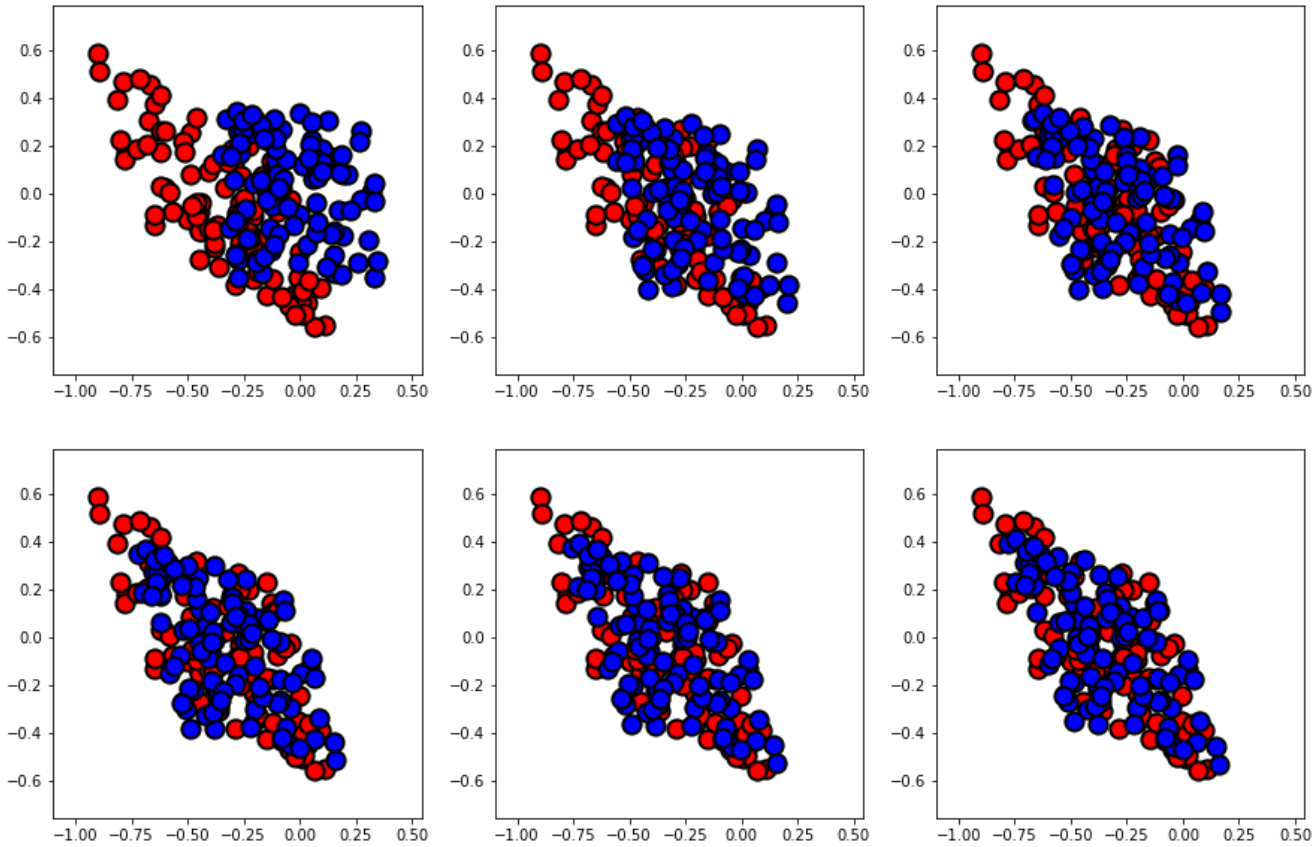
In [137]:

```
x = A.dot(z)+h[:,None]
f = np.zeros(n)
(P,Err) = Sinkhorn(distmat(x,y), epsilon,f,niter)
```

In [139]:

```
plt.figure(figsize=(15,10))
A = np.eye(2)
h = np.zeros(2)
xm, xM = min(np.min(z[0]),np.min(y[0])) - 0.2, max(np.max(z[0]),np.max(y[0])) + 0.2
ym, yM = min(np.min(z[1]),np.min(y[1])) - 0.2,  max(np.max(z[1]),np.max(y[1])) + 0.2
# step size for the descent
tau_A = .8
tau_h = .1
# #iter for the gradient descent
giter = 80
ndisp = np.round( np.linspace(0,giter-1,6) )
kdisp = 0
```

```
kdisp = 0
f = np.zeros(n)
for j in range(giter):
    x = A.dot(z)+h[:,None]
    if ndisp[kdisp]==j:
        plt.subplot(2,3,kdisp+1)
        plotp(y, 'r')
        plotp(x, 'b')
        kdisp = kdisp+1
        plt.xlim(xm,xM)
        plt.ylim(ym,yM)
    (P,Err) = Sinkhorn(distmat(x,y), epsilon,f,niter)
    v = a.transpose() * x - y.dot(P.transpose())
    nabla_A = v.dot(z.transpose())
    nabla_h = np.sum(v,1)
    A = A - tau_A * nabla_A
    h = h - tau_h * nabla_h
```



Assignment4

Semi-discrete Optimal Transport

This numerical tour studies semi-discrete optimal transport, i.e. when one of the two measure is discrete.

The initial papers that proposed this approach are [Oliker89,Aurenhammer98]. We refer to [Mérigot11,Lévy15] for modern references and fast implementations.

This tour is not inteded to show efficient algorithm but only conveys the main underlying idea (c-transform, Laguerre cells, connexion to optimal quantization). In the Euclidean case, there exists efficient algorithm to compute Laguerre cells leveraging computational geometry algorithm for convex hulls [Aurenhammer87].

In [2]:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

Dual OT and c-transforms

The primal Kantorovitch OT problem reads

$$W_c(\alpha,\beta) = \min_{\pi} \{ \int_{\mathcal{X}\times\mathcal{Y}} c(x,y)\mathrm{d}\pi(x,y), \pi_1 = \alpha, \pi_2 = \beta \}.$$

It dual is

$$W_c(\alpha,\beta) = \max_{f,g} \{ \int_{\mathcal{X}} f\mathrm{d}\alpha + \int_{\mathcal{Y}} g\mathrm{d}\beta, f(x)+g(y) \leq c(x,y) \}.$$

We consider the case where $\alpha = \sum_i a_i \delta_{x_i}$ is a discrete measure, so that the function $f(x)$ can be replaced by a vector $(f_i)_{i=1}^n \in \mathbb{R}^n$. The optimal $g(y)$ function can the be replaced by the c-transform of f

$$f^c(y) \equiv \min_i \; c(x_i,y) - f_i.$$

The function to maximize is then

$$W_c(\alpha,\beta) = \max_{f\in\mathbb{R}^n} \; \mathcal{E}(f) \equiv \sum_i f_i a_i + \int f^c(y)\mathrm{d}\beta(y).$$

Semi-discret via Gradient Ascent

We now implement a gradient ascent scheme for the maximization of $\mathcal{E}$. The evaluation of $\mathcal{E}$) can be computed via the introduction of the partition of the domain in Laguerre cells

$$\mathcal{Y} = \bigcup_i L_i(f) \quad \text{where} \quad L_i(f) \equiv \{y, \forall j, c(x_i,y) - f_i \leq c(x_j,y) - f_j\}.$$

When $f=0$, this corrsponds to the partition in Voronoi cells.

One has that $\forall y \in L_i(f)$, $f^c(y) = c(x_i,y) - f_i$, i.e. f^c is piecewise smooth according to this partition.

The grid for evaluation of the "continuous measure".

In [3]:

```
p = 300  # size of the image for sampling, m=p*p
t = np.linspace(0, 1, p)
[V, U] = np.meshgrid(t, t)
Y = np.concatenate((U.flatten()[None, :], V.flatten()[None, :]))
```

First measure, sums of Dirac masses $\alpha = \sum_{i=1}^n a_i \delta_{x_i}$.

In [4]:

```
n = 30
X = .5+.5j + np.exp(1j*np.pi/4) * 1 * \
    (.1*(np.random.rand(1, n)-.5)+1j*(np.random.rand(1, n)-.5))
X = np.concatenate((np.real(X), np.imag(X)))

X = np.random.rand(2, n)
a = np.ones(n)/n
```

In [5]:

```
X.shape
```

Out[5]:

```
(2, 30)
```

Second measure β, potentially a continuous one (i.e. with a density), mixture of Gaussians. Here we discretize $\beta = \sum_{j=1}^m b_j \delta_{y_j}$ on a very fine grid.

In [6]:

```
def Gauss(mx, my, s): return np.exp((-(U-mx)**2-(V-my)**2)/(2*s**2))
```

```
Mx = [.6, .4, .1, .8]  # means
My = [.9, .1, .4,.5]
S = [.07, .09, .09, .07]  # variance
W = [.25, .25, .25, .25]  # weights
b = W[0]*Gauss(Mx[0], My[0], S[0]) + W[1]*Gauss(Mx[1], My[1], S[1]) + W[2]*Gauss(Mx[2], My[2], S[2])
+ W[3]*Gauss(Mx[3], My[3], S[3])
b = b/np.sum(b.flatten())
```

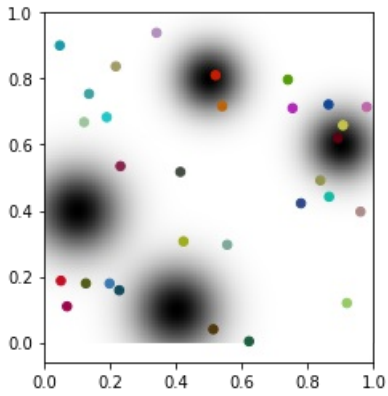
Display the two measures.

In [7]:

```
Col = np.random.rand(n, 3)
plt.imshow(-b[::-1, :], extent=[0, 1, 0, 1], cmap='gray')
plt.scatter(X[1, :], X[0, :], s=30, c=.8*Col)
```

Out[7]:

```
<matplotlib.collections.PathCollection at 0x26cfc0f7f08>
```



Initial potentials.

In [8]:

```
f = np.zeros(n)
```

compute Laguerre cells and c-transform

In [9]:

```
def distmat(x, y): return np.sum(
    x**2, 0)[:, None] + np.sum(y**2, 0)[None, :] - 2*x.transpose().dot(y)


D = distmat(Y, X) - f[:].transpose()
fC = np.min(D, axis=1)
I = np.reshape(np.argmin(D, axis=1), [p, p])
```

Dual value of the OT, $\langle f, a \rangle + \langle f^c, \beta \rangle$.

In [10]:

```
OT = np.sum(f*a) + np.sum(fC*b.flatten())
print(OT)
```

```
0.012268007300070116
```

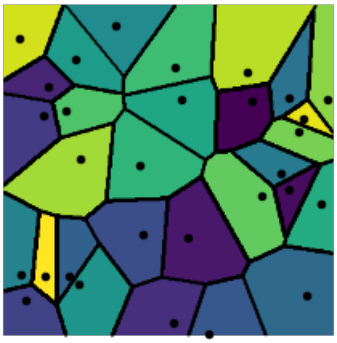
Display the Laguerre call partition (here this is equal to the Vornoi diagram since $f = 0$).

In [11]:

```
plt.imshow(I[::-1, :], extent=[0, 1, 0, 1])
plt.scatter(X[1, :], X[0, :], s=20, c='k')
plt.contour(t, t, I, np.linspace(-.5, n-.5, n), colors='k')
plt.axis('off')
```

Out[11]:

```
(0.0, 1.0, -0.05674338979111575, 1.0)
```



Where β has a density with respect to Lebesgue measure, then $\mathcal{E}$ is smooth, and its gradient reads

$$\nabla\mathcal{E}(f)_i = a_i - \int_{L_i(f)} \mathrm{d}\beta(x).$$

sum area captured

Exercise 1

Implement a gradient ascent

$$f \leftarrow f + \tau\nabla\mathcal{E}(f).$$

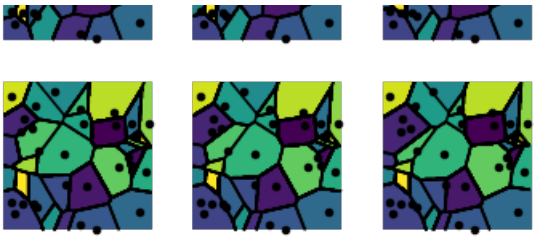
Experiment on the impact of τ, display the evolution of the OT value $\mathcal{E}$ and of the Laguerre cells.

Example with $\tau = 0.02$

In [12]:

```
tau = .02  # step size
niter = 200  # iteration for the descent
q = 6  # number of displays
ndisp = np.unique(np.round(1 + (niter/4-1)*np.linspace(0, 1, q)**2))
kdisp = 0
f = np.zeros(n)
E = np.zeros(niter)
for it in range(niter):
    # compute Laguerre cells and c-transform
    D = distmat(Y, X) - f[:].transpose()
    fC = np.min(D, axis=1)
    I = np.reshape(np.argmin(D, axis=1), [p, p])
    E[it] = np.sum(f*a) + np.sum(fC*b.flatten())
    # display
    if (kdisp < len(ndisp)) and (ndisp[kdisp] == it):
        plt.subplot(2, 3, kdisp+1)
        plt.imshow(I[::-1, :], extent=[0, 1, 0, 1])
        plt.scatter(X[1, :], X[0, :], s=20, c='k')
        plt.contour(t, t, I, np.linspace(-.5, n-.5, n), colors='k')
        plt.axis('off')
        kdisp = kdisp+1
    # gradient
    R = (I[:, :, None] == np.arange(0, n)[None, None, :]) * b[:, :, None]
    nablaE = a-np.sum(R, axis=(0, 1)).flatten()
    f = f+tau*nablaE
```





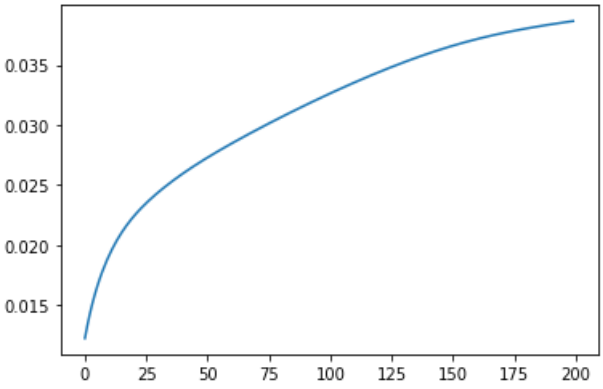
Display the evolution of the estimated OT distance.

In [13]:

```
plt.plot(E, '-')
```

Out[13]:

```
[<matplotlib.lines.Line2D at 0x26c803c8c88>]
```

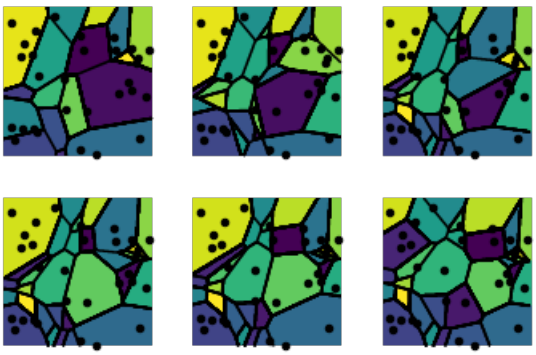


Experiment with different τ

In [23]:

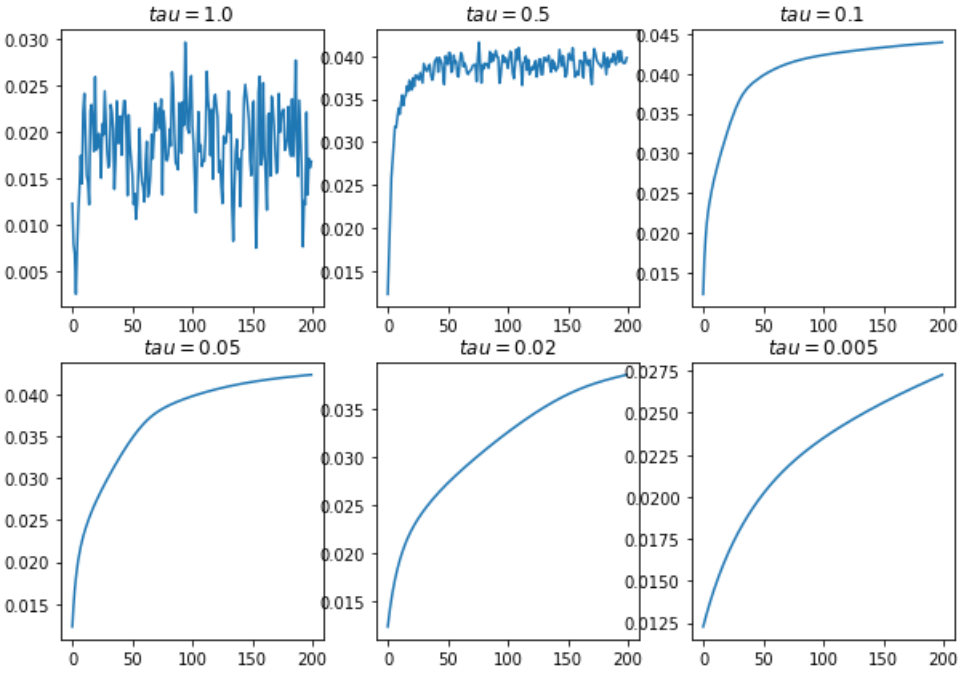
```
taus = [1., .5, .1, .05, .02, .005]  # step size
#taus = [.02]
E = [np.zeros(niter) for i in range(len(taus))]
for index, tau in enumerate(taus):
    print("Computation for tau = ", str(tau), "...")
    niter = 200  # iteration for the descent
    f = np.zeros(n)
    for it in range(niter):
        # compute Laguerre cells and c-transform
        D = distmat(Y, X) - f[:].transpose()
        fC = np.min(D, axis=1)
        I = np.reshape(np.argmin(D, axis=1), [p, p])
        E[index][it] = np.sum(f*a) + np.sum(fC*b.flatten())
        # display
        if it == niter - 1:
            plt.subplot(2, 3, index+1)
            plt.imshow(I[::-1, :], extent=[0, 1, 0, 1])
            plt.scatter(X[1, :], X[0, :], s=20, c='k')
            plt.contour(t, t, I, np.linspace(-.5, n-.5, n), colors='k')
            plt.
            plt.axis('off')
        # gradient
        R = (I[:, :, None] == np.arange(0, n)[None, None, :]) * b[:, :, None]
        nablaE = a-np.sum(R, axis=(0, 1)).flatten()
        f = f+tau*nablaE
```

```
Computation for tau =  1.0 ...
Computation for tau =  0.5 ...
Computation for tau =  0.1 ...
Computation for tau =  0.05 ...
Computation for tau =  0.02 ...
Computation for tau =  0.005 ...
```



In [29]:

```
plt.figure(figsize=(10,7))
for index, i in enumerate(taus):
    plt.subplot(2,3,index+1)
    plt.plot(E[index], '-')
    plt.title("$tau = $" + str(i))
```



The previous results recover the clasical results of the impact of the step size on the performance of a gradient descent algorithm. For τ too large, the algorithm fails to converge, and for τ too small, the convergence is very slow. In our experiments, the optimal τ is $\tau = 0.1$

Stochastic Optimization

The function $\mathcal{E}$ to minimize can be written as an expectation over a random variable $Y \sim \beta$

$$\mathcal{E}(f) = \mathbb{E}(E(f, Y)) \quad \text{where} \quad E(f, y) = \langle f, a \rangle + f^c(y).$$

As proposed in [Genevay16], one can thus use a stochastic gradient ascent scheme to minimize this function, at iteration ℓ

$$f \leftarrow f + \tau_\ell \nabla E(f, y_\ell)$$

where $y_\ell \sim Y$ is a sample drawn according to β and the step size $\tau_\ell \sim 1/\ell$ should decay at a carefully chosen rate.

The gradient of the integrated functional reads

$$\nabla E(f, y)_i = a - 1_{L_i(f)}(y),$$

where 1_A is the binary indicator function of a set A.

Initialize the algorithm.

In [30]:

```
f = np.zeros(n)
```

Draw the sample.

In [31]:

```
k = np.int(np.random.rand(1) < W[1])  # select one of the two Gaussian
y = np.array((S[k] * np.random.randn(1) + Mx[k],
              S[k] * np.random.randn(1) + My[k]))
```

Compute the randomized gradient: detect Laguerre cell where y is.

In [32]:

```
R = np.sum(y**2) + np.sum(X**2, axis=0) - 2*y.transpose().dot(X) - f[:]
i = np.argmin(R)
```

Randomized gradient.

In [33]:

```
a = np.ones(n)/n
nablaEy = a.copy()
nablaEy[i] = nablaEy[i] - 1
```

Exercise 2

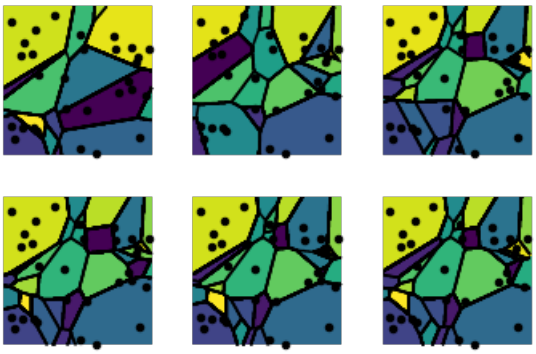
Implement the stochastic gradient descent. Test various step size selection rule.

In [47]:

```
niter = 400
taus = [1., .5, .1, .05, .02, .005]  # step size
#taus = [.02]
E = [np.zeros(niter) for i in range(len(taus))]
for index, subtau in enumerate(taus):
    print("Computation for tau = ", str(subtau), "...")
    for it in range(niter):
        # sample
        k = np.where(np.random.multinomial(1,W) == 1)[0][0] ## Select one of the gaussians
        y = np.array((S[k] * np.random.randn(1) + Mx[k],
                      S[k] * np.random.randn(1) + My[k]))
        # detect Laguerre cell where y is
        R = np.sum(y**2) + np.sum(X**2, axis=0) - 2*y.transpose().dot(X) - f[:]
        i = np.argmin(R)
        # gradient
        nablaEy = a.copy()
        nablaEy[i] = nablaEy[i] - 1
        # gradient ascent
        l0 = 50  # warmup phase.
        tau = subtau/(1 + it/l0)
        f = f + tau*nablaEy
        # compute Laguerre cells and c-transform
        D = distmat(Y, X) - f[:].transpose()
        fC = np.min(D, axis=1)
        I = np.reshape(np.argmin(D, axis=1), [p, p])
        E[index][it] = np.sum(f*a) + np.sum(fC*b.flatten())
        # display
        if it == niter - 1:
            plt.subplot(2, 3, index+1)
            plt.imshow(I[::-1, :], extent=[0, 1, 0, 1])
            plt.scatter(X[1, :], X[0, :], s=20, c='k')
            plt.contour(t, t, I, np.linspace(-.5, n-.5, n), colors='k')
            plt.axis('off')
```

```
Computation for tau =  1.0 ...
Computation for tau =  0.5 ...
Computation for tau =  0.1 ...
Computation for tau =  0.05 ...
```

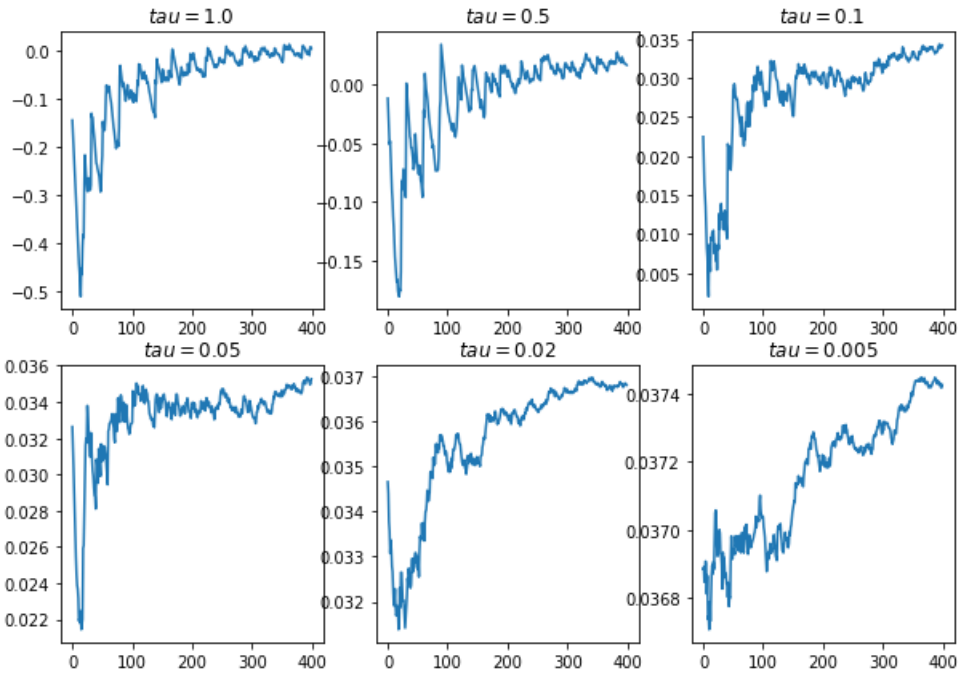
```
Computation for tau =  0.02 ...
Computation for tau =  0.005 ...
```



Display the evolution of the estimated OT distance (warning: recording this takes lot of time).

In [48]:

```
plt.figure(figsize=(10,7))
for index, i in enumerate(taus):
    plt.subplot(2,3,index+1)
    plt.plot(E[index], '-')
    plt.title("$tau = $" + str(i))
```



Optimal Quantization and Lloyd Algorithm

We consider the following optimal quantization problem [Gruber02]

$$\min_{(a_i)_i,(x_i)_i} W_c\left(\sum_i a_i\delta_{x_i},\beta\right).$$

This minimization is convex in a, and writing down the optimality condition, one has that the associated dual potential should be $f=0$, which means that the associated optimal Laguerre cells should be Voronoi cells $L_i(0)=V_i(x)$ associated to the sampling locations

$$V_i(x)=\{y,\forall j,c(x_i,y)\leq c(x_j,y)\}.$$

This problem is tightly connected to semi-discrete OT, and this connexion and its implications are studied in [Canas12].

The minimization is non-convex with respect to the positions $x=(x_i)_i$ and one needs to solve

$$\min_x \mathcal{F}(x)\equiv\sum_{i=1}^n\int_{V_i(x)}c(x_i,y)\mathrm{d}\beta(y).$$

For the sake of simplicity, we consider the case where $c(x,y) = \frac{1}{2}\|x-y\|^2$.

The gradient reads

$$\nabla \mathcal{F}(x)_i = x_i \int_{V_i(x)} \mathrm{d}\beta - \int_{V_i(x)} y \mathrm{d}\beta(y).$$

The usual algorithm to compute stationary point of this energy is Lloyd's algorithm [Lloyd82], which iterate the fixed point

$$x_i \leftarrow \frac{\int_{V_i(x)} y \mathrm{d}\beta(y)}{\int_{V_i(x)} \mathrm{d}\beta},$$

i.e. one replaces the centroids by the barycenter of the cells.

Intialize the centroids positions.

In [49]:

```
X1 = X.copy()
```

Compute the Voronoi cells $V_i(x)$.

In [50]:

```
D = D = distmat(Y, X1)
fC = np.min(D, axis=1)
I = np.reshape(np.argmin(D, axis=1), [p, p])
```

Update the centroids to the barycenters.

In [51]:

```
A = (I[:, :, None] == np.arange(0, n)[None, None, :]) * b[:, :, None]
B = (I[:, :, None] == np.arange(0, n)[None, None, :]) * \
    b[:, :, None] * (U[:, :, None] + 1j*V[:, :, None])
X1 = np.sum(B, axis=(0, 1)) / np.sum(A, axis=(0, 1))
X1 = np.concatenate((np.real(X1)[None, :], np.imag(X1)[None, :]))
```

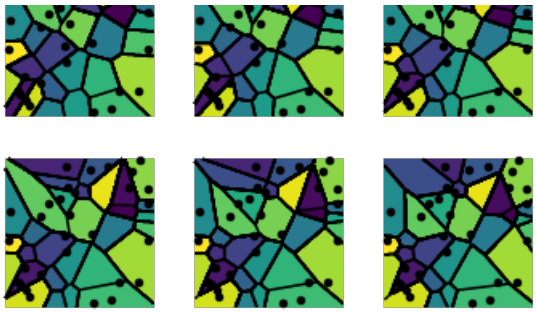
Exercise 3

Implement Lloyd algortihm.

In [52]:

```
niter = 60
q = 6
ndisp = np.unique(np.round(1 + (niter/4-1)*np.linspace(0, 1, q)**2))
kdisp = 0
E = np.zeros(niter)
X1 = X.copy()
for it in range(niter):
    # compute Voronoi cells
    D = D = distmat(Y, X1)
    fC = np.min(D, axis=1)
    I = np.reshape(np.argmin(D, axis=1), [p, p])
    E[it] = np.sum(fC*b.flatten())
    # display
    if (kdisp < len(ndisp)) and (ndisp[kdisp] == it):
        plt.subplot(2, 3, kdisp+1)
        plt.imshow(I[::-1, :], extent=[0, 1, 0, 1])
        plt.scatter(X[1, :], X[0, :], s=20, c='k')
        plt.contour(t, t, I, np.linspace(-.5, n-.5, n), colors='k')
        plt.axis('off')
        kdisp = kdisp+1
    # update barycenter
    A = (I[:, :, None] == np.arange(0, n)[None, None, :]) * b[:, :, None]
    B = (I[:, :, None] == np.arange(0, n)[None, None, :]) * \
        b[:, :, None] * (U[:, :, None] + 1j*V[:, :, None])
    X1 = np.sum(B, axis=(0, 1)) / np.sum(A, axis=(0, 1))
    X1 = np.concatenate((np.real(X1)[None, :], np.imag(X1)[None, :]))
```





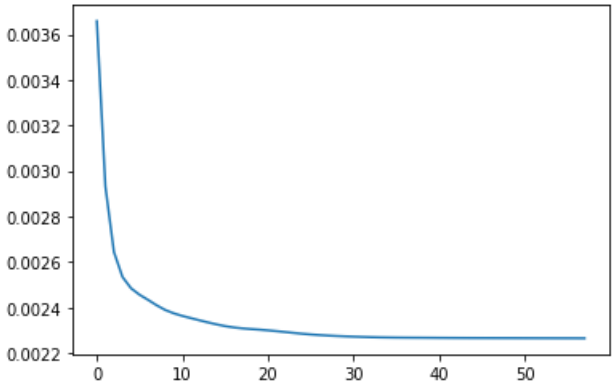
Display the evolution of the estimated OT distance.

In [53]:

```
plt.plot(E[1:-1])
```

Out[53]:

```
[<matplotlib.lines.Line2D at 0x2d7d165a708>]
```



References

- [Oliker89] Vladimir Oliker and Laird D Prussner. *On the numerical solution of the equation* $\frac{\partial^2 z}{\partial x^2}\frac{\partial^2 z}{\partial y^2} - \left(\frac{\partial^2 z}{\partial x \partial y}\right)^2 = f$ *and its discretizations*, I. Numerische Mathematik, 54(3):271-293, 1989.
- [Aurenhammer98] Franz Aurenhammer, Friedrich Hoffmann and Boris Aronov. *Minkowski-type theorems and least-squares clustering*. Algorithmica, 20(1):61-76, 1998.
- [Mérigot11] Quentin Mérigot. *A multiscale approach to optimal transport*. Comput. Graph. Forum, 30(5):1583-1592, 2011.
- [Lévy15] Bruno Lévy. *A numerical algorithm for l2 semi-discrete optimal transport in 3D*. ESAIM: Mathematical Modelling and Numerical Analysis, 49(6):1693-1715, 2015.
- [Aurenhammer87] Franz Aurenhammer. *Power diagrams: properties, algorithms and applications*. SIAM Journal on Computing, 16(1):78-96, 1987.
- [Canas12] Guillermo Canas, Lorenzo Rosasco, *Learning probability measures with respect to optimal transport metrics*. In Advances in Neural Information Processing Systems, pp. 2492--2500, 2012.
- [Gruber02] Peter M. Gruber. *Optimum quantization and its applications*. Adv. Math, 186:2004, 2002.
- [Lloyd82] Stuart P. Lloyd, *Least squares quantization in PCM*, IEEE Transactions on Information Theory, 28 (2): 129-137, 1982.
- [Genevay16] Aude Genevay, Marco Cuturi, Gabriel Peyré and Francis Bach. *Stochastic oppmization for large-scale optimal transport*. In Advances in Neural Information Processing Systems, pages 3440-3448, 2016.