Numerical Experiment 1

January 8, 2021

1 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.

You need to install CVXPY. *Warning:* seems to not be working on Python 3.7, use rather 3.6.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
```

1.1 Optimal Transport of Discrete Distributions

We consider two dicretes distributions

$$= \sum_{i=1}^{n} a_i x_i = \sum_{j=1}^{m} b_j y_j.$$

where n, m are the number of points, $_x$ is the Dirac at location $x \in ^d$, and $(x_i)_i, (y_j)_j$ are the positions of the diracs in ^{d}.

Dimensions (n, m) of the coulds.

[2]:
```
#n = 60
#m = 80

n = 40
m = 60
```

Generate the point clouds $X = (x_i)_i$ and $Y = (y_j)_j$.

[3]:
```
gauss = lambda q,a,c: a*np.random.randn(2, q) + np.transpose(np.tile(c, (q,1)))
X = np.random.randn(2,n)*.4
Y = np.hstack((gauss(int(m/2),.5,[0,1.6]),np.hstack((gauss(int(m/4),.
↪3,[-1,-1]),gauss(int(m/4),.3,[1,-1])))))
```

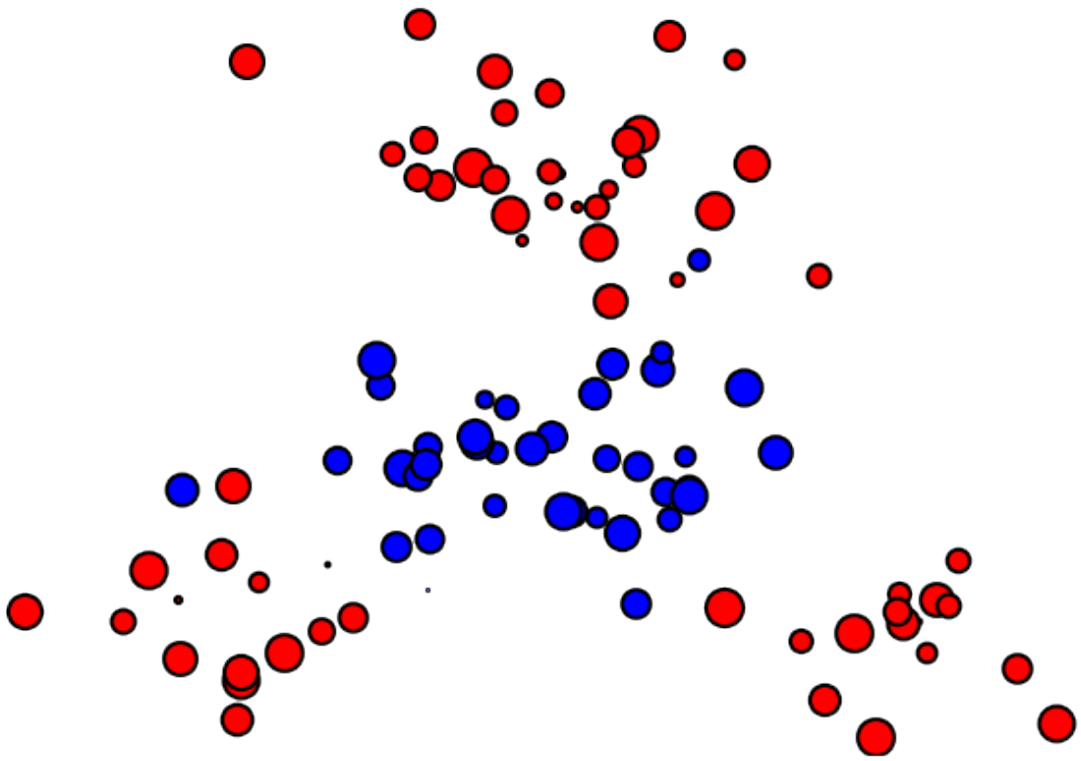
[4]:
```
normalize = lambda a: a/np.sum(a)
a = normalize(np.random.rand(n, 1))
b = normalize(np.random.rand(m, 1))
```

Helper function for display of clouds.

[5]:
```
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.

[6]:
```
plt.figure(figsize = (10,7))
plt.axis("off")
for i in range(len(a)):
    myplot(X[0,i], X[1,i], a[i]*len(a)*10, 'b')
for j in range(len(b)):
    myplot(Y[0,j], Y[1,j], b[j]*len(b)*10, 'r')
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.show()
```



Compute the cost matrix $C_{i,j} := x_i - x_j{}^2$.

[7]:
```
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)
```

Define the optimization variable P.

[8]:
```
P = cp.Variable((n,m))
```

Define the set of discrete couplings between and

$$U(a,b) := P \in_+^{n\times m} \forall i, \sum_j P_{i,j} = a_i,\ \forall j, \sum_i P_{i,j} = b_j.$$

[9]:
```
u = np.ones((m,1))
v = np.ones((n,1))
U = [0 <= P, cp.matmul(P,u)==a, cp.matmul(P.T,v)==b]
```

The Kantorovitch formulation of the optimal transport reads

$$P^\star \in P \in U(a,b) \sum_{i,j} P_{i,j} C_{i,j}.$$

Solve it using CVXPY

[10]:
```
objective = cp.Minimize( cp.sum(cp.multiply(P,C)) )
prob = cp.Problem(objective, U)
result = prob.solve()
```

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

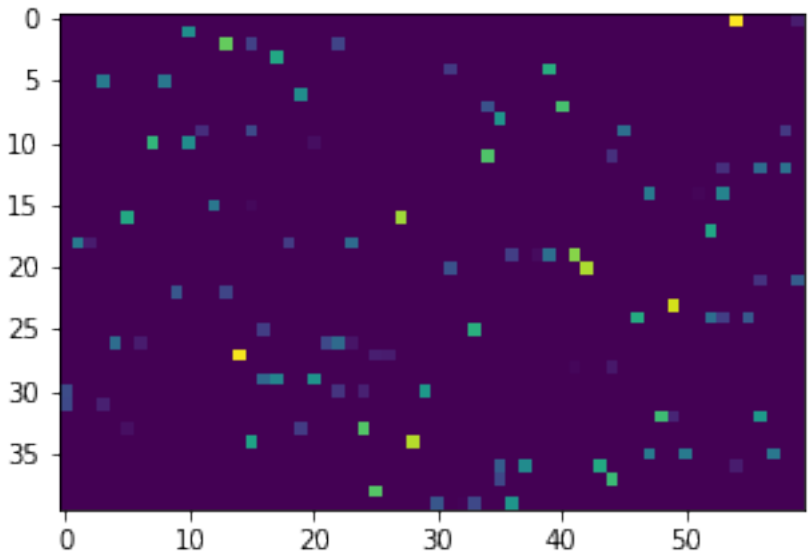
Check that the number of non-zero entries in $P^\star$ is $n+m-1$. Beware that we are using an interior point method here, so that entries of $P^\star$ are nevery exactly 0.

[11]:
```
print("Number of non-zero: %d (n + m-1 = %d)" %(len(P.value[P.value>1e-5]), n +␣
 ↪m-1))
```

```
Number of non-zero: 99 (n + m-1 = 99)
```

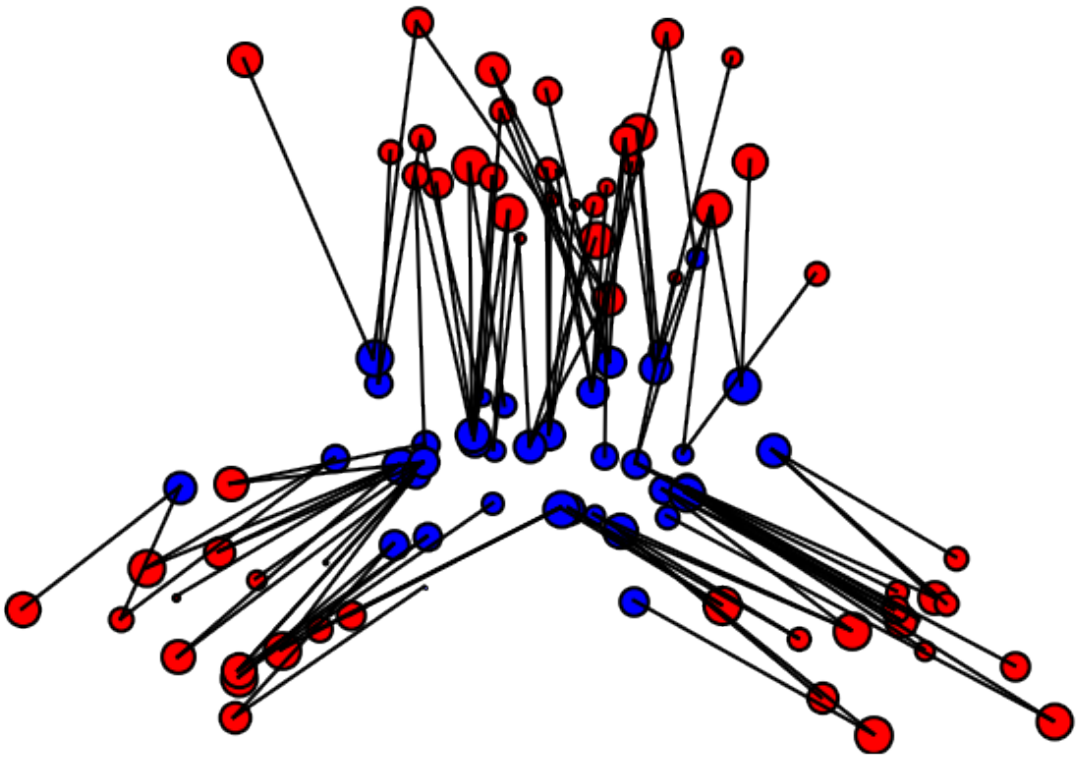
Display the solution coupling.

[12]:
```
plt.figure(figsize = (5,5))
plt.imshow(P.value);
```



Display the connexion defined by the optimal coupling.

[13]:
```
I,J = np.nonzero(P.value>1e-5)
plt.figure(figsize = (10,7))
plt.axis('off')
for k in range(len(I)):
    h = plt.plot(np.hstack((X[0,I[k]],Y[0,J[k]])),np.hstack(([X[1,I[k]],␣
 ↪Y[1,J[k]]])),'k', lw = 2)
for i in range(len(a)):
    myplot(X[0,i], X[1,i], a[i]*len(a)*10, 'b')
for j in range(len(b)):
    myplot(Y[0,j], Y[1,j], b[j]*len(b)*10, 'r')
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.show()
```



1.2 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

$$\mu_t = \mu(1-t)W_2(,\mu)^2 + tW_2(,\mu)^2.$$

This can be understood as a generalization of the usual Euclidean barycenter to barycenter of distribution. Indeed, in the case that $=_x$ and $=_y$, one has $\mu_t =_{x_t}$ where $ x_t = (1-t)x+t y $.

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

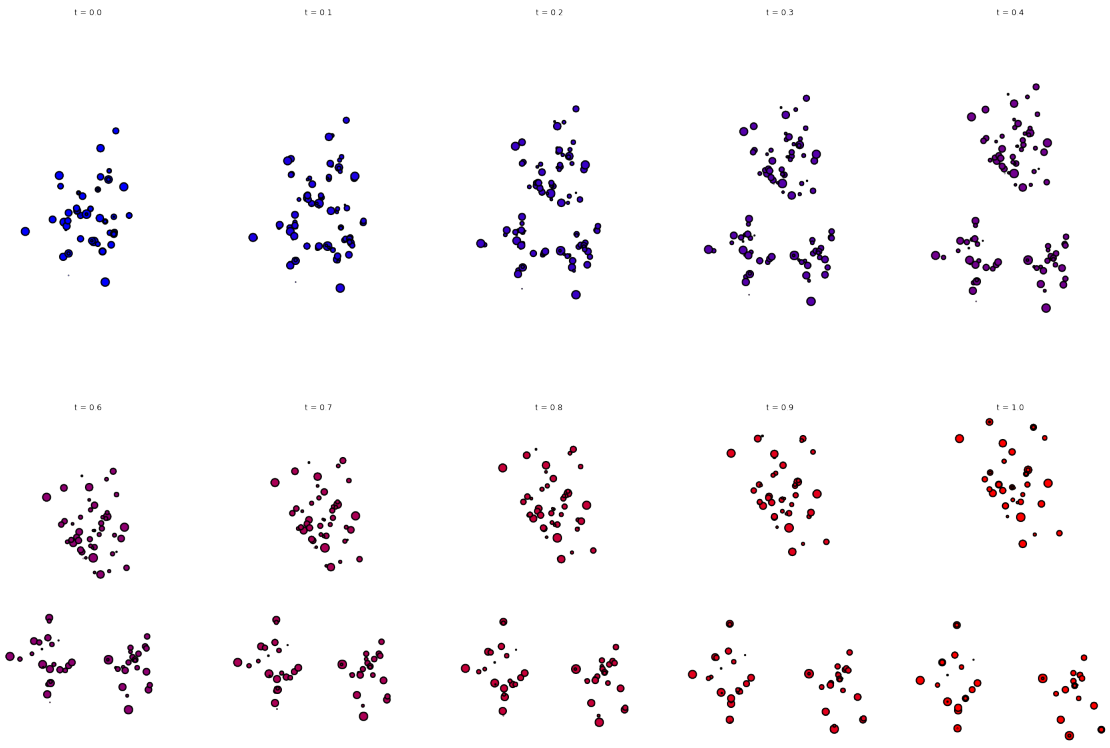
$$\mu_t = \sum_{i,j} P^\star_{i,j(1-t)x_i+ty_j}.$$

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

[14]:
```
I,J = np.nonzero(P.value>1e-5)
Pij = P.value[I,J]
```

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

[15]:
```
plt.figure(figsize =(30,20))
tlist = np.linspace(0, 1, 10)
for i in range(len(tlist)):
    t = tlist[i]
    Xt = (1-t)*X[:,I] + t*Y[:,J]
    plt.subplot(2,5,i+1)
    plt.axis("off")
    for j in range(len(Pij)):
        myplot(Xt[0,j],Xt[1,j],Pij[j]*len(Pij)*3,[[t,0,1-t]])
    plt.title("t = %.1f" %t)
    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.show()
```



1.3 Optimal Assignment

In the case where $n = m$ and the weights are uniform $a_i = 1/n, b_j = 1/n$, one can show that there is at least one optimal transport coupling which is actually a permutation matrix. This properties comes from the fact that the extremal point of the polytope $U(1,1)$ are permutation matrices.

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

$$P^\star_{i,j} = 1j =^\star (i), 0 \quad \text{otherwise.}$$

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

This permutation thus solves the so-called optimal assignement problem

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

Use the same number of points.

[16]:
```
n = 40
m = n
```

Compute points clouds.

[17]:
```
X = np.random.randn(2,n)*.3
Y = np.hstack((gauss(int(m/2),.5,[0,1.6]),np.hstack((gauss(int(m/4),.
 ↪3,[-1,-1]),gauss(int(m/4),.3,[1,-1])))))
```

Constant distributions.

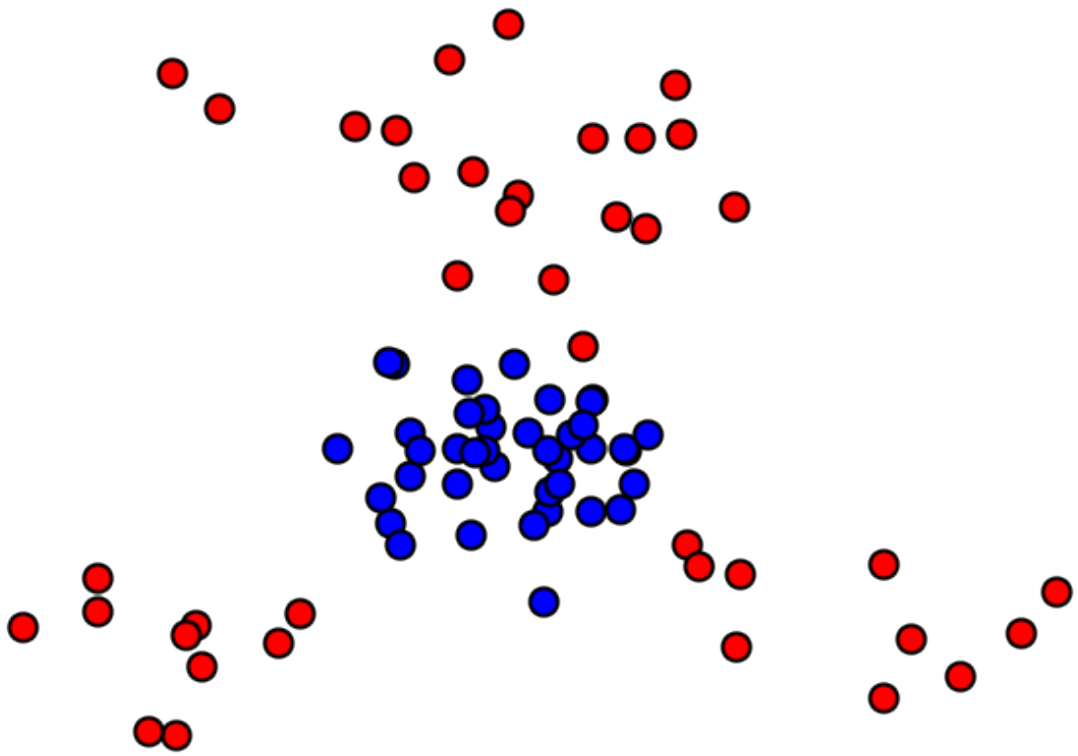
[18]:
```
a = np.ones([n,1])/n
b = a
```

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

[19]:
```
C = distmat(X,Y)
```

Display the coulds.

[20]:
```
plt.figure(figsize = (10,7))
plt.axis('off')
myplot(X[0,:],X[1,:],10,'b')
myplot(Y[0,:],Y[1,:],10,'r')
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.show()
```

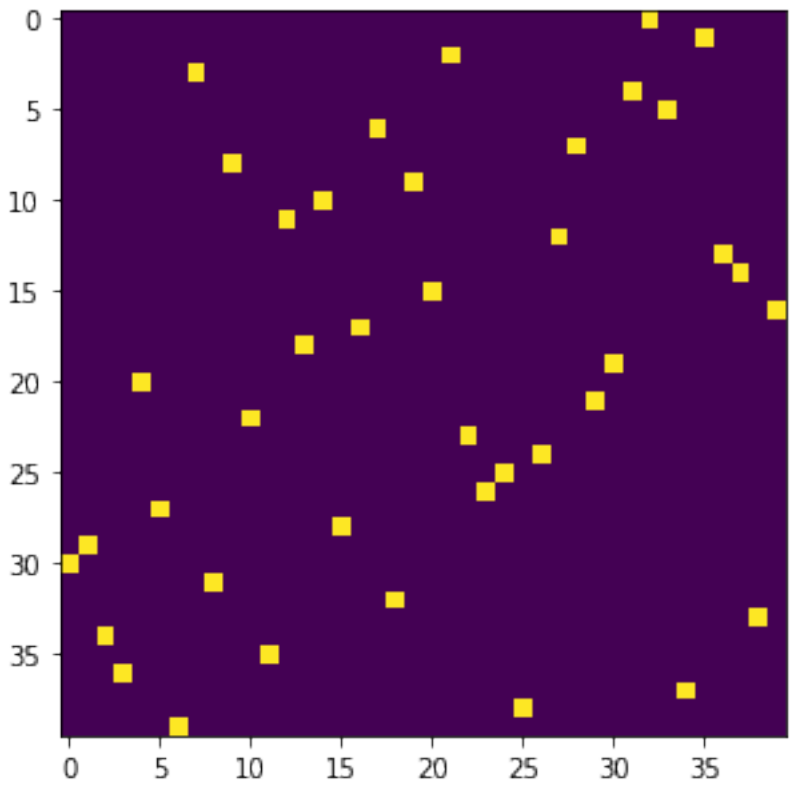


Solve the optimal transport.

[21]:
```
P = cp.Variable((n,m))
u = np.ones((m,1))
v = np.ones((n,1))
U = [0 <= P, cp.matmul(P,u)==a, cp.matmul(P.T,v)==b]
objective = cp.Minimize( cp.sum(cp.multiply(P,C)) )
prob = cp.Problem(objective, U)
result = prob.solve()
```

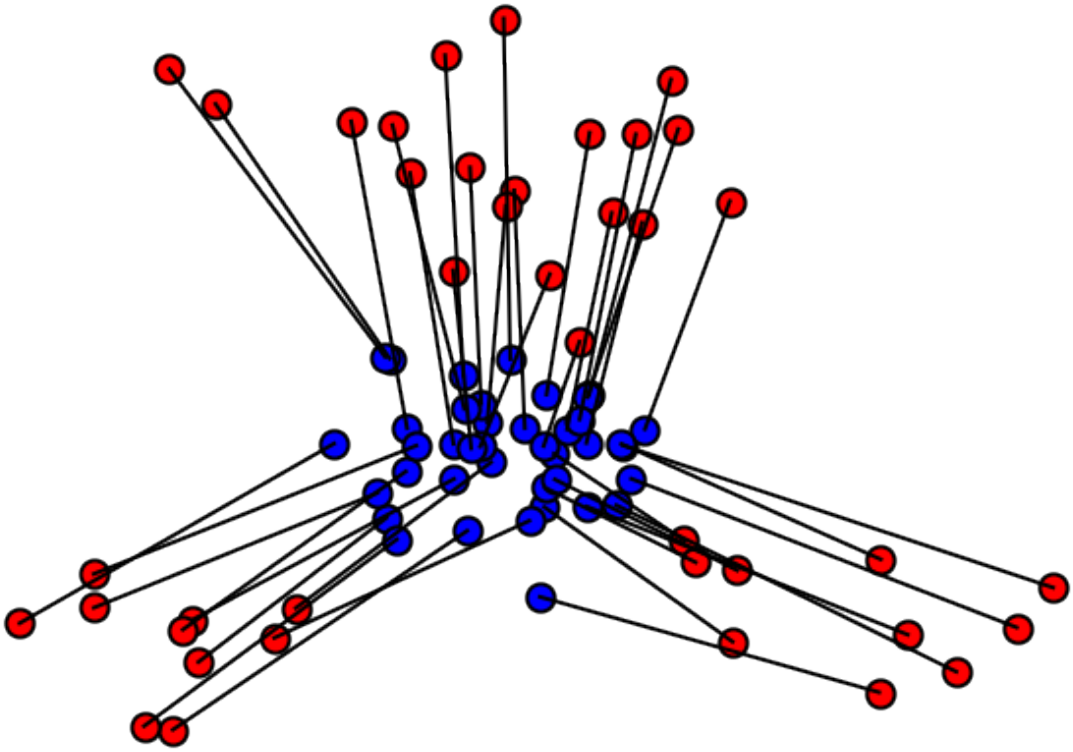
Show that P is a binary permutation matrix.

[22]:
```
plt.figure(figsize = (5,5))
plt.imshow(P.value);
```



Display the optimal assignement.

[23]:
```
I,J = np.nonzero(P.value>1e-5)
plt.figure(figsize = (10,7))
plt.axis('off')
for k in range(len(I)):
    h = plt.plot(np.hstack((X[0,I[k]],Y[0,J[k]])),np.hstack(([X[1,I[k]],␣
 ↪Y[1,J[k]]])),'k', lw = 2)
myplot(X[0,:], X[1,:], 10, 'b')
myplot(Y[0,:], Y[1,:], 10, 'r')
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.show()
```



Numerical Experiment 2

January 8, 2021

1 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.

$$KL$$

[1]:
```
from __future__ import division

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

import warnings
warnings.filterwarnings('ignore')

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

1.1 Entropic Regularization of Optimal Transport

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

$$_n a \in_+^n \sum_i a_i = 1.$$

We consider the following discrete regularized transport

$$W_\epsilon(a,b)P \in U(a,b)CP - \epsilon E(P).$$

where the polytope of coupling is defined as

$$U(a,b)P \in (^+)^{n\times m} P_m = a, P_n^\top = b,$$

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

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

When $\epsilon = 0$ 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 $C \in (^{+})\{n \times m\}$ $defines the ground cost, i.e. C_{i,j}$ 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

$$W_\epsilon(a,b) = \epsilon P \in U(a,b) PKK_{i,j} e^{-\frac{C_{i,j}}{\epsilon}}$$

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

$$PK \sum_{i,j} P_{i,j} \log \frac{P_{i,j}}{K_{i,j}} - 1.$$

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

Given a convex set $\subset^N$, the projection according to the Kullback-Leiber divergence is defined as

$$_(\xi) = \pi \in \pi\xi.$$

1.2 Iterative Bregman Projection Algorithm

Given affine constraint sets $ (_1,_2) $, we aim at computing

$$_(K) =_1 \cap_2$$

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

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

$$P_{2\ell+1} =_1 (P_{2\ell}) P_{2\ell+2} =_2 (P_{2\ell+1}).$$

One can indeed show that $P_\ell \to_(K)$. 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).

1.3 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

$$P_\epsilon = uKv$$

where the Gibbs kernel is defined as

$$Ke^{-\frac{C}{\epsilon}}.$$

One thus needs to find two positive scaling vectors $u \in_+^n$ and $v \in_+^m$ such that the two following equality holds

$$P = u \odot (Kv) = aP^\top = v \odot (K^\top u) = b.$$

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

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

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

1.4 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 $x_i, y_i \in^2$.

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

[2]:
```
N = [300,200]
```

Dimension of the clouds.

[3]:
```
d = 2
```

Point cloud x, of n points inside a square.

[4]:
```
x = np.random.rand(2,N[0])-.5
```

Point cloud y, of m points inside an anulus.

[5]:
```
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))
```

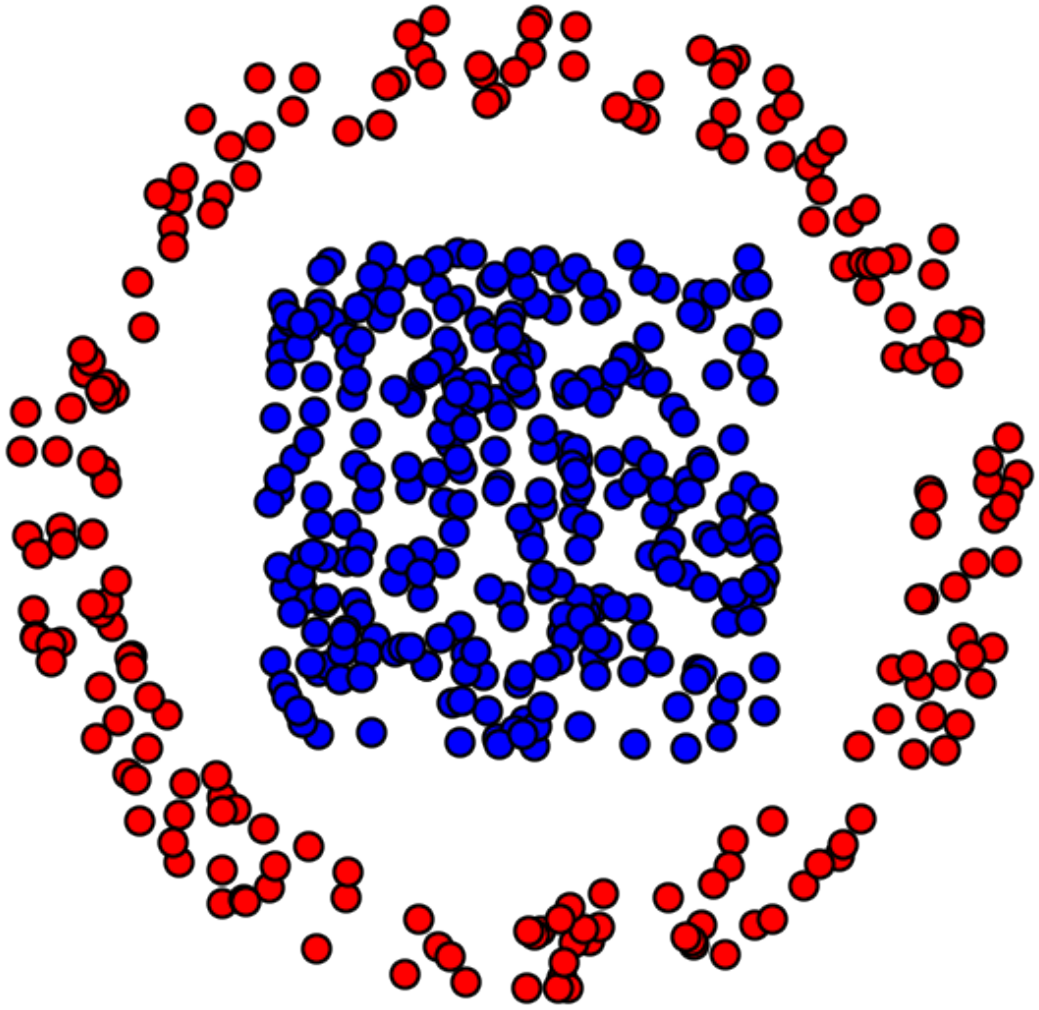
Shortcut for displaying point clouds.

[6]:
```
plotp = lambda x,col: plt.scatter(x[0,:], x[1,:], s=200, edgecolors="k", c=col,␣
 ↪linewidths=2)
```

Display of the two clouds.

[7]:
```
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.show()
```



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

[8]:
```
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.

[9]:
```
a = np.ones(N[0])/N[0]
b = np.ones(N[1])/N[1]
```

Regularization strength $\epsilon > 0$.

[10]:
```
epsilon = .01;
```

Gibbs Kernel K.

[11]:
```
K = np.exp(-C/epsilon)
```

Initialization of $v =_m$ (u does not need to be initialized).

[12]:
```
v = np.ones(N[1])
```

One sinkhorn iterations.

[13]:
```
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

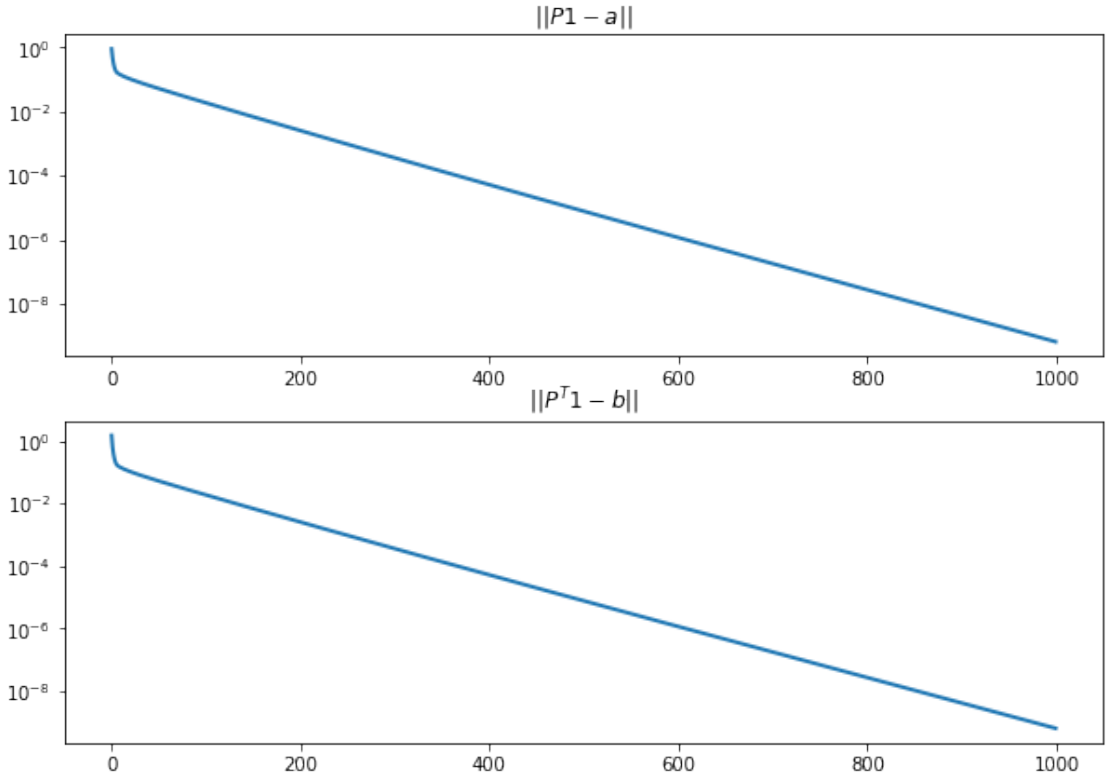
$$P - a_1 P^\top - b$$

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

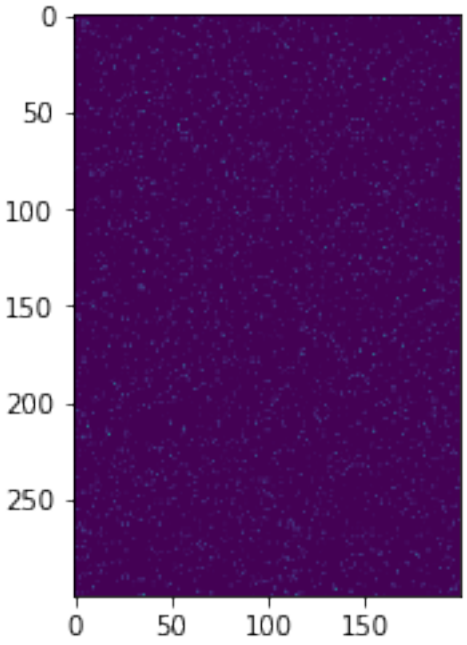
[14]:
```
def Sinkhorn(iteration, a, b, C, epsilon):
    n = np.size(a)
    m = np.size(b)
    v = np.ones(m)
    K = np.exp(-C/epsilon)
    error_a = []
    error_b = []
    for i in range(iteration):
        u = a / (np.dot(K,v))
        constrain_b = v*(np.dot(np.transpose(K), u)) - b
        error_b.append(np.linalg.norm(constrain_b, 1))
        v = b / (np.dot(np.transpose(K),u))
        constrain_a = u*np.dot(K,v) - a
        P = np.dot(np.dot(np.diag(u),K),np.diag(v))
        error_a.append(np.linalg.norm(constrain_a, 1))


    return P, error_a, error_b
```

[15]:
```
iteration = 1000
P, err1, err2 = Sinkhorn(iteration, a, b, C, epsilon=0.01)
```

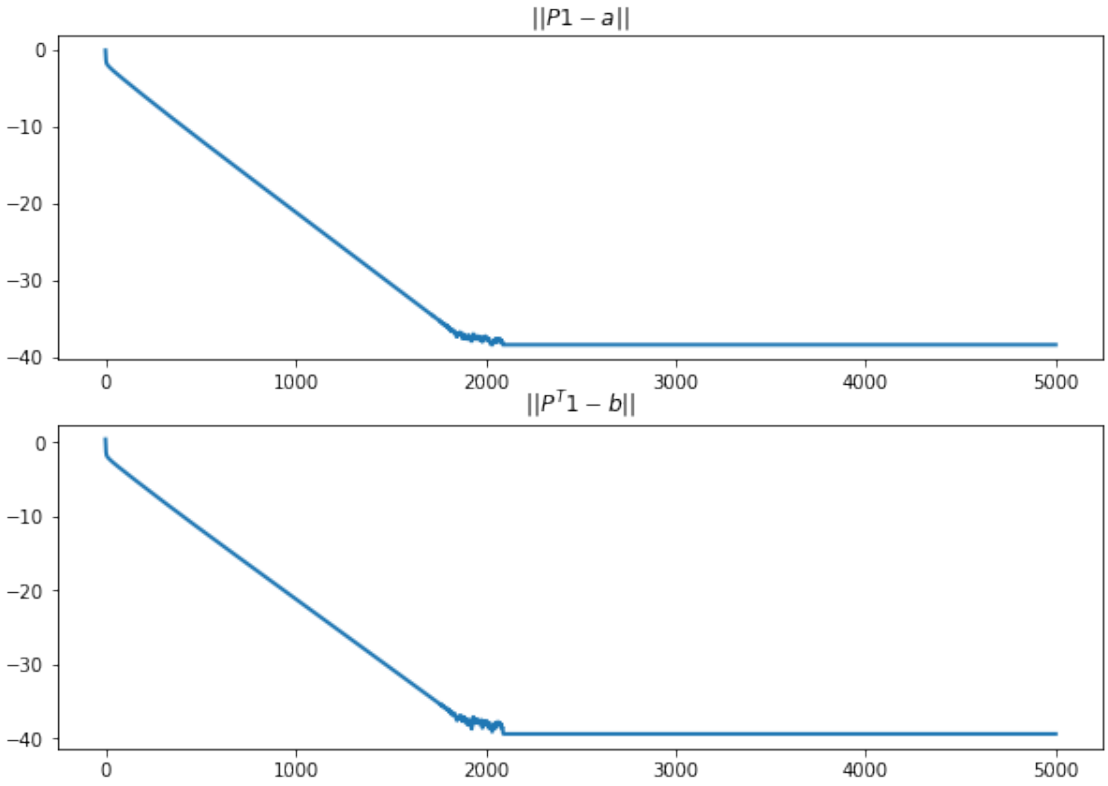
[16]:
```
plt.figure(figsize = (10,7))
plt.subplot(2,1,1)
plt.title("$||P 1 - a||$")
plt.plot(err1, linewidth = 2)
plt.yscale('log')

plt.subplot(2,1,2)
plt.title("$||P^T 1 - b||$")
plt.plot(err2, linewidth = 2)
plt.yscale('log')
plt.show()

plt.figure()
plt.imshow(P)
```



[16]:
```
<matplotlib.image.AxesImage at 0x28489f98160>
```



[17]:
```
run -i nt_solutions/optimaltransp_5_entropic/exo1
```



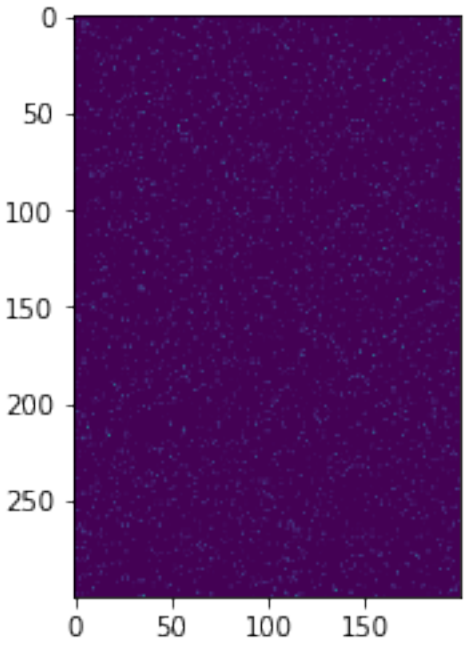
```
<Figure size 432x288 with 0 Axes>
```

Compute the final matrix P.

[18]:
```
P = np.dot(np.dot(np.diag(u),K),np.diag(v))
```

Display it.

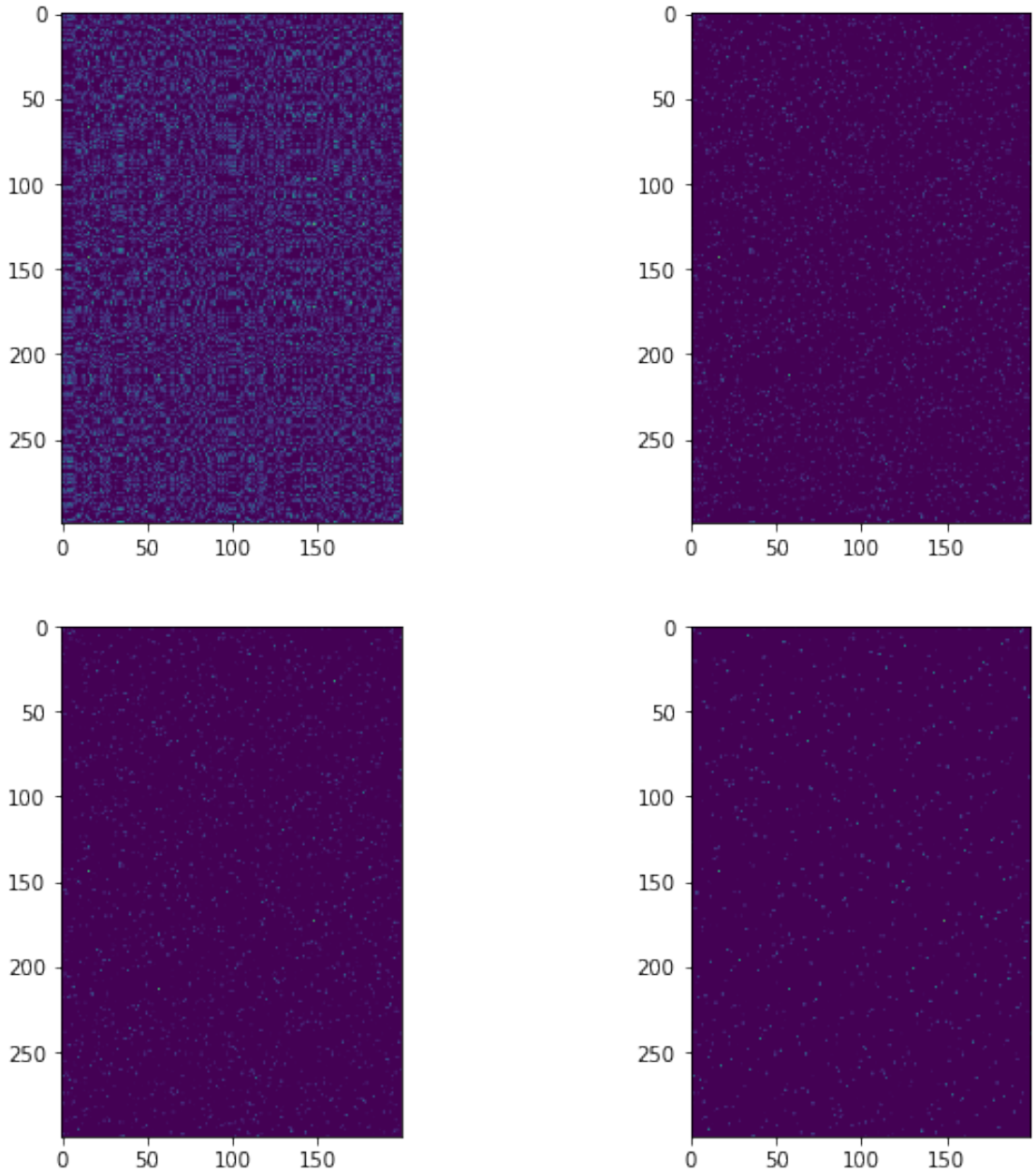
[19]:
```
plt.imshow(P);
```



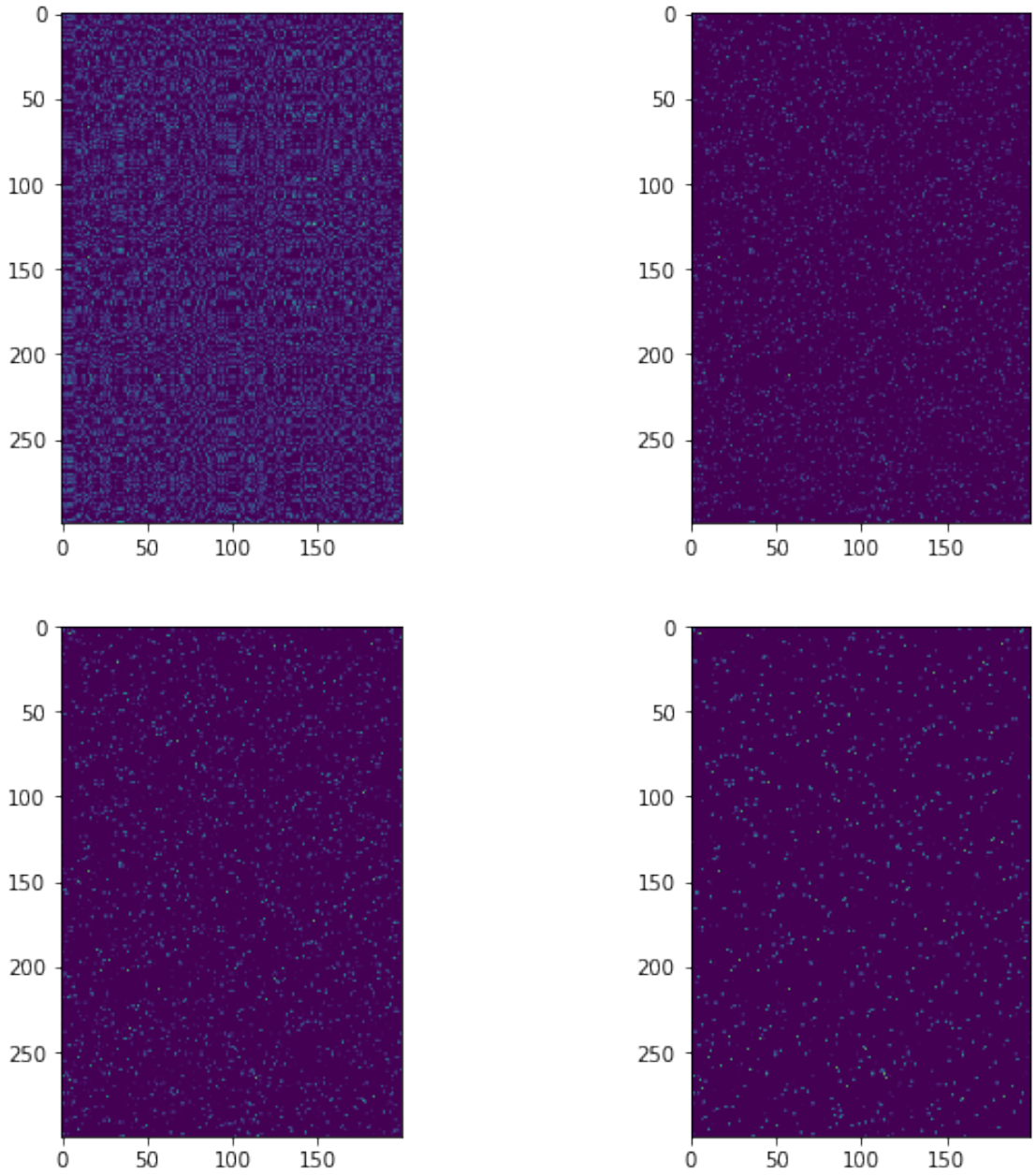
Exercise 2

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

[20]:
```
eps = [0.1, 0.01, 0.005, 0.001] #list of epsilons
plt.figure(figsize = (10,10))
for ep in eps:
    P_e, _, _ = Sinkhorn(300, a, b, C, ep)
    plt.subplot(2, 2, eps.index(ep)+1)
    plt.imshow(P_e)
```



[21]:
```
run -i nt_solutions/optimaltransp_5_entropic/exo2
```



Compute the obtained optimal P.

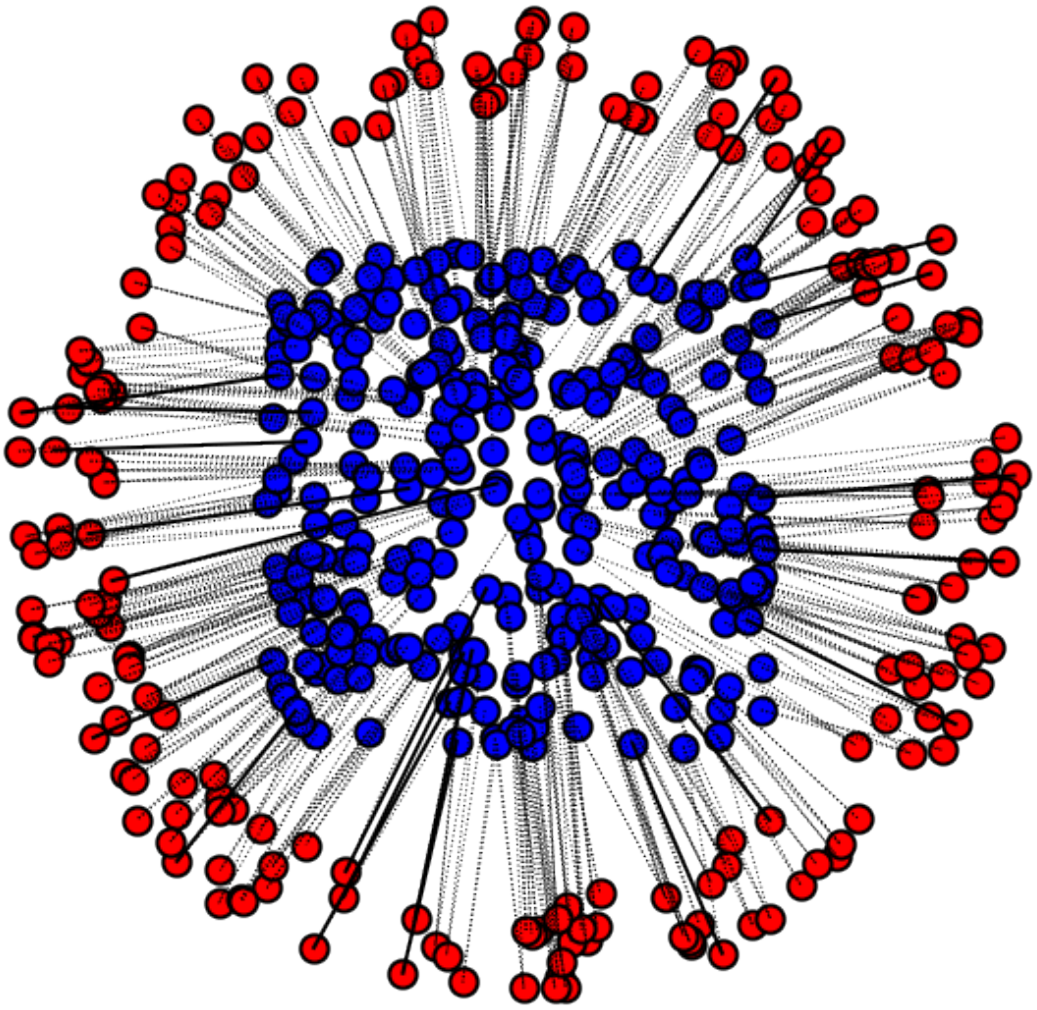
[22]:
```
P = np.dot(np.dot(np.diag(u),K),np.diag(v))
np.shape(P)
```

[22]: (300, 200)

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.

[23]:
```
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(np.min(y[0,:])-.1,np.max(y[0,:])+.1)
plt.ylim(np.min(y[1,:])-.1,np.max(y[1,:])+.1)

plt.show()
```



1.5 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.

[24]:
```
N = 200
```

We use here a 1-D square Euclidean metric.

[25]:
```
t = np.arange(0,N)/N
```

Define the histogram a, b as translated Gaussians.

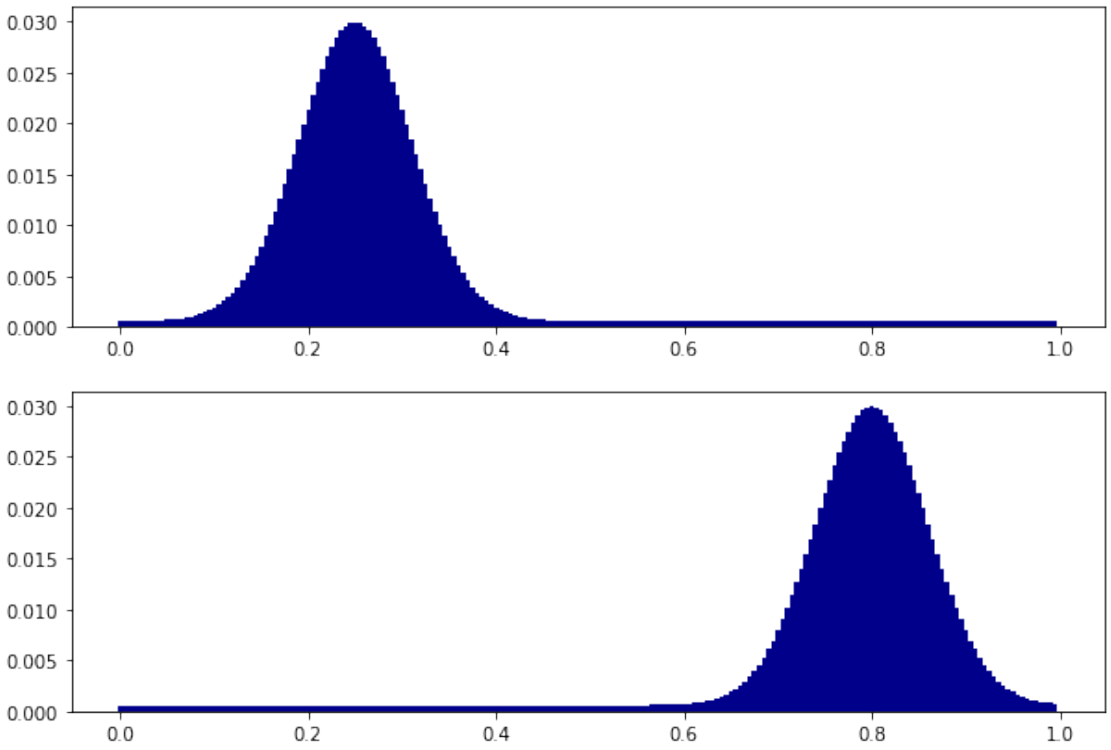
[26]:
```
Gaussian = lambda t0,sigma: np.exp(-(t-t0)**2/(2*sigma**2))
normalize = lambda p: p/np.sum(p)

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

Add some minimal mass and normalize.

[27]:
```
vmin = .02;
a = normalize( a+np.max(a)*vmin)
b = normalize( b+np.max(b)*vmin)
```

Display the histograms.

[28]:
```
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.show()
```



Regularization strength .

[29]:
```
epsilon = (.03)**2
```

The Gibbs kernel is a Gaussian convolution,

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

[30]:
```
[Y,X] = np.meshgrid(t,t)
K = np.exp(-(X-Y)**2/epsilon)
```

Initialization of $v = _N$.

[31]:
```
v = np.ones(N)
```

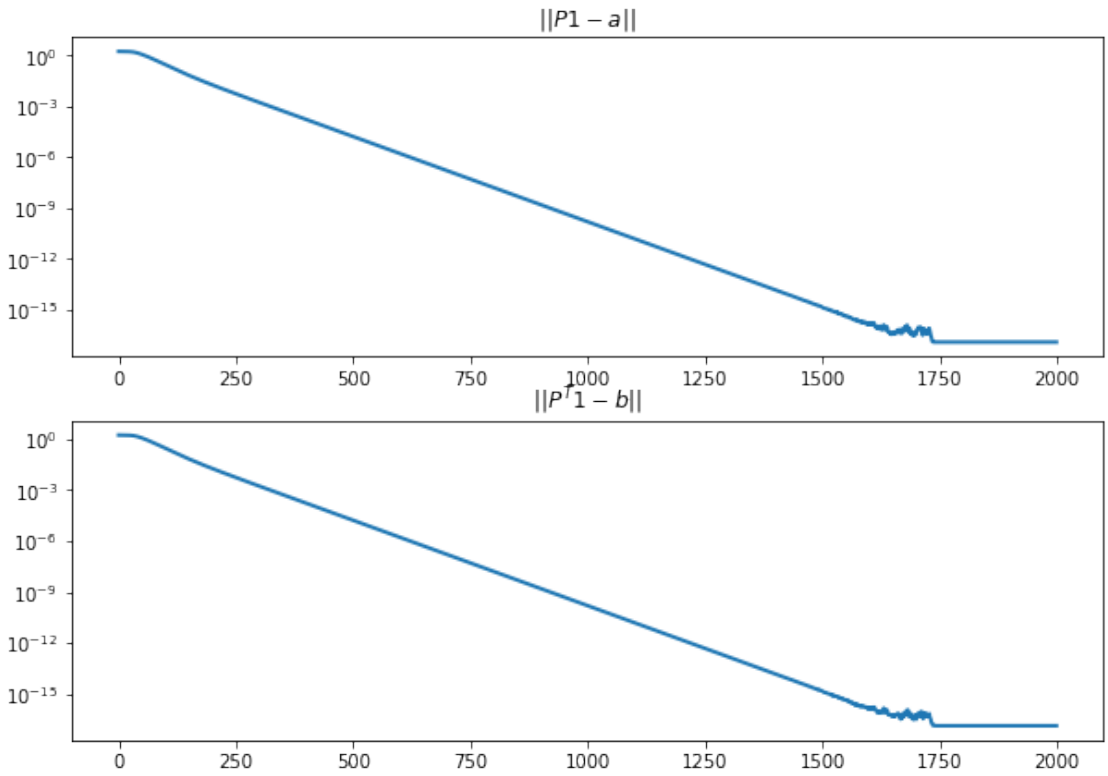
One sinkhorn iteration.

[32]:
```
u = a / (np.dot(K,v))
v = b / (np.dot(np.transpose(K),u))
```

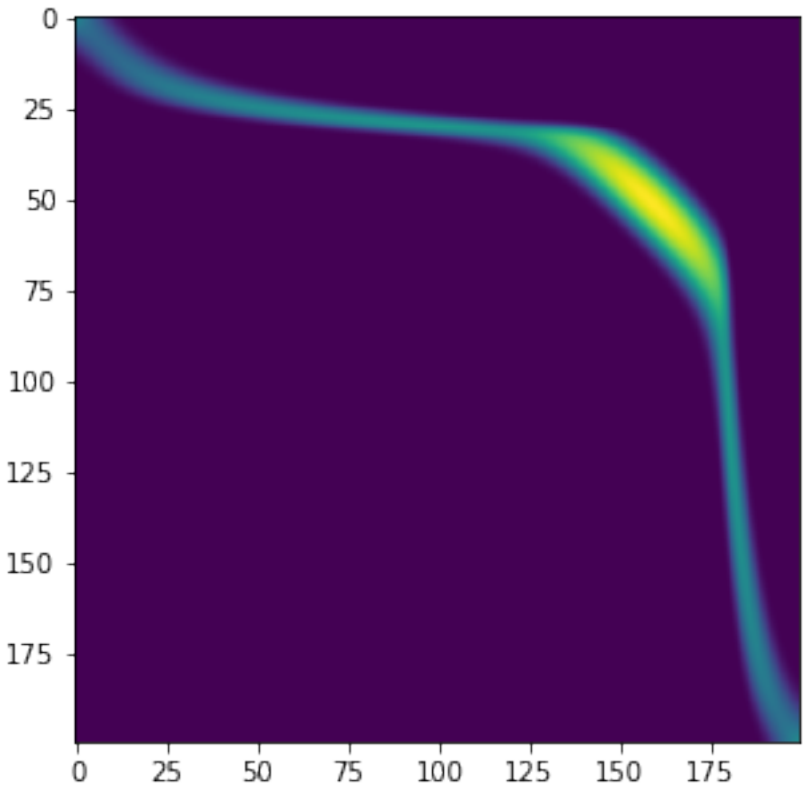
Exercise 3

Implement Sinkhorn algorithm. Display the evolution of the constraints satisfaction errors $ P - a _1, $P^\top - b_1$. You need to think how to compute it from (u, v). Display the violation of constraint error in log-plot.

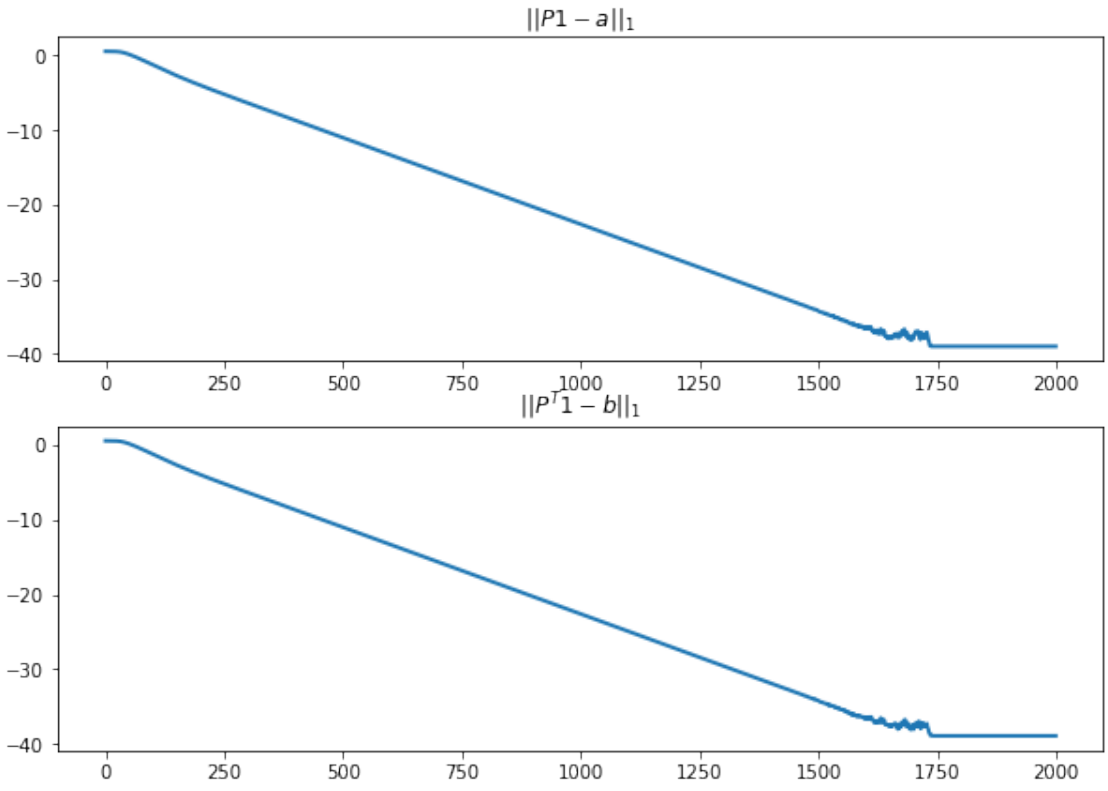
[33]:
```
iteration = 2000
C = (X-Y)**2
P, err1, err2 = Sinkhorn(iteration, a, b, C, epsilon)

plt.figure(figsize = (10,7))
plt.subplot(2,1,1)
plt.title("$||P 1 - a||$")
plt.plot(err1, linewidth = 2)
plt.yscale('log')

plt.subplot(2,1,2)
plt.title("$||P^T 1 - b||$")
plt.plot(err2, linewidth = 2)
plt.yscale('log')
plt.show()

plt.figure(figsize=(5,5))
plt.imshow(np.log(P+1e-5))
```



[33]: <matplotlib.image.AxesImage at 0x2848a07b4a8>



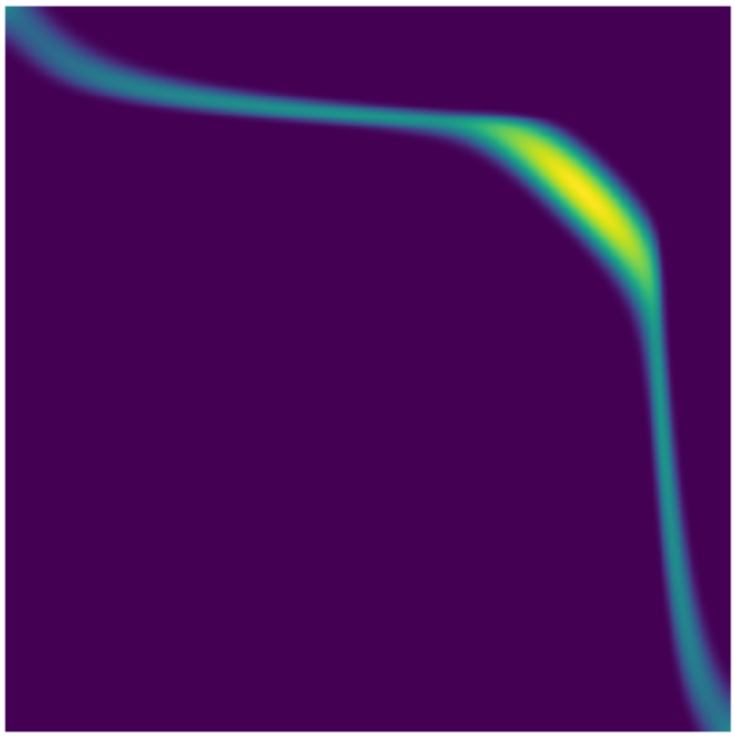
[34]:
```
run -i nt_solutions/optimaltransp_5_entropic/exo3
```




```
<Figure size 432x288 with 0 Axes>
```

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

[35]:
```
for _ in range(1000):
    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))
plt.figure(figsize=(5,5))
plt.imshow(np.log(P+1e-5))
plt.axis('off');
```



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

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

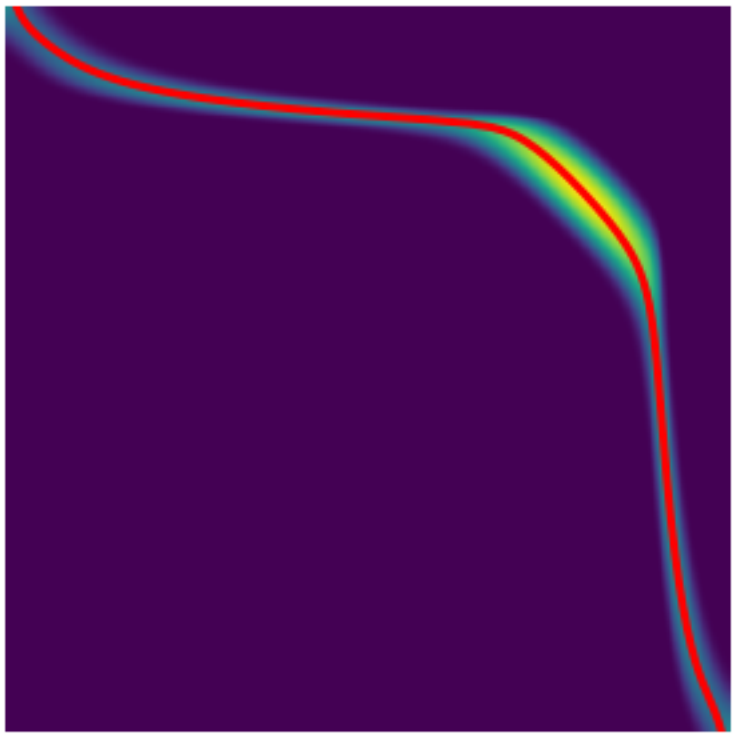
where $\odot$ and $\div$ are the enry-wise multiplication and division.

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

[36]:
```
s = np.dot(K,v*t)*u/a
```

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

[37]:
```
plt.figure(figsize=(5,5))
plt.imshow(np.log(P+1e-5))
plt.plot(s*N,t*N, 'r', linewidth=3);
plt.axis('off');
```



1.6 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

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

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

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

[39]:
```
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

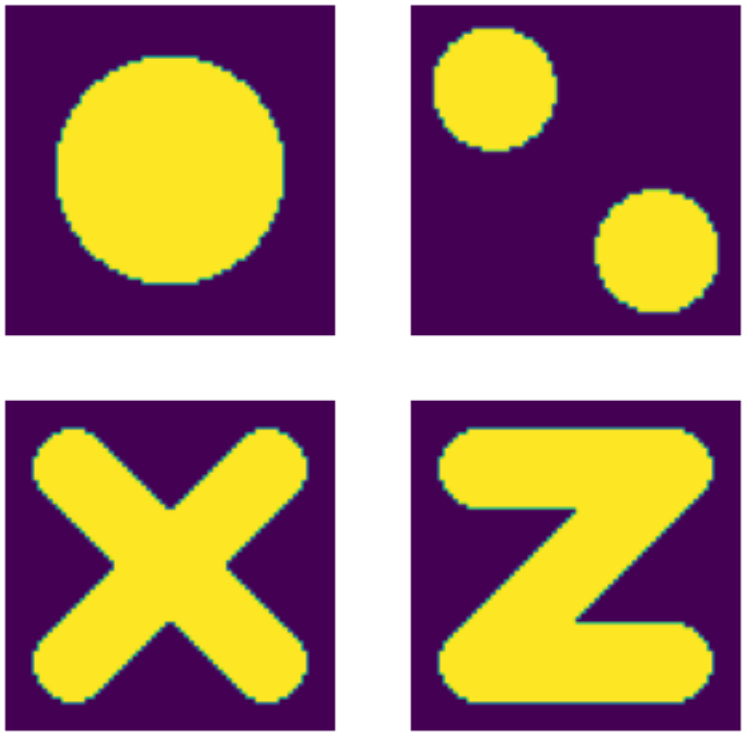
Load input histograms $(a_k)_{k=1}^R$, store them in a tensor A.

[40]:
```
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)
```

Display the input histograms.

[41]:
```
plt.figure(figsize=(5,5))
for i in range(R):
    plt.subplot(2,2,i+1)
    plt.imshow(A[:,:,i])
    plt.axis('off');
```



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

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

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 $\epsilon > 0$.

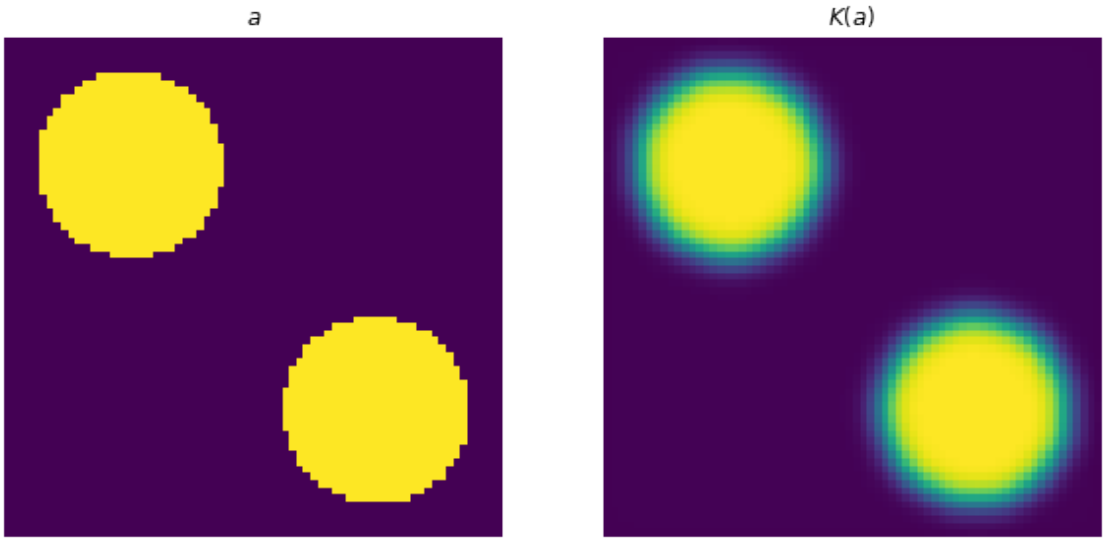
[42]:
```
epsilon = (.04)**2
```

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.

[43]:
```
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.

[44]:
```
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.imshow(A[:,:,1])
plt.title("$a$")
plt.axis('off');
plt.subplot(1,2,2)
plt.imshow(K(A[:,:,1]))
plt.title("$K(a)$")
plt.axis('off');
```



Weights $_k$ for isobarycenter.

[45]:
```
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

$$P_k = u_k K v_k$$

for some unknown positive weights (u_k, v_k).

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

[46]:
```
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 $P^k = a_k$. This is achieved by updating

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

[47]:
```
for k in range(R):
    u[:,:,k] = A[:,:,k]/K(v[:,:,k])
```

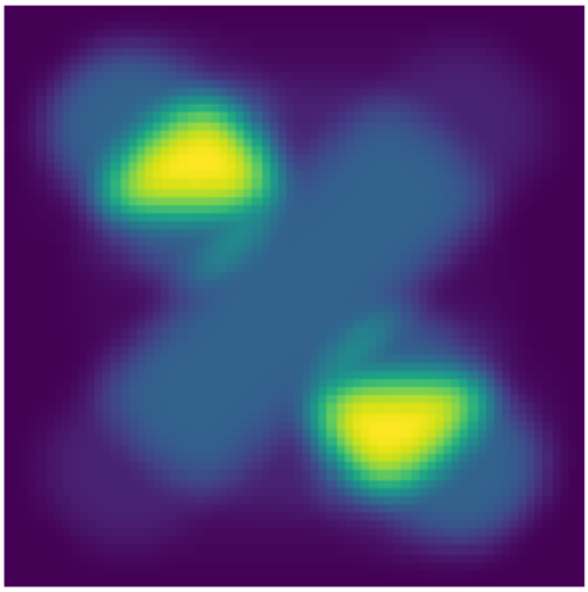
The second step of the Bregman projection method corresponds to the projection on the equal marginals constraints $\forall k, P_k^\top = b$ for a common barycenter target b. This is achieved by first computing the target barycenter b using a geometric means

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

[48]:
```
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.

[49]:
```
plt.imshow(b);
plt.axis('off');
```



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):

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

[50]:
```
for k in range(R):
    v[:,:,k] = b/K(u[:,:,k])
```

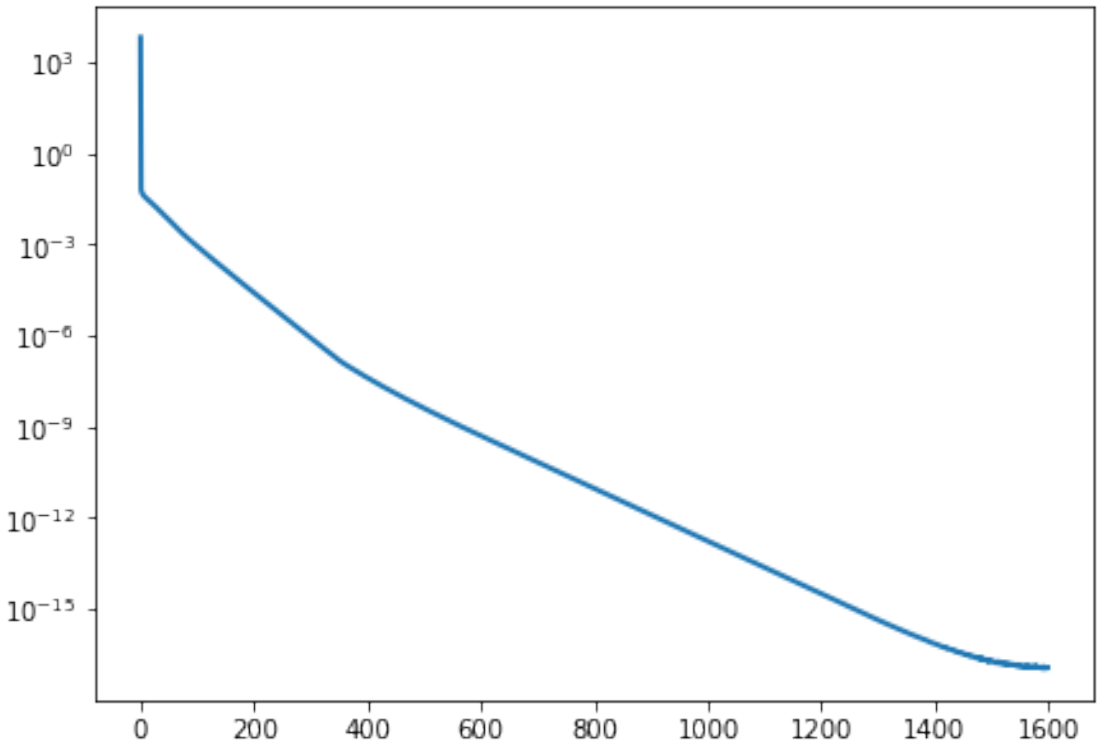
Exercise 4

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

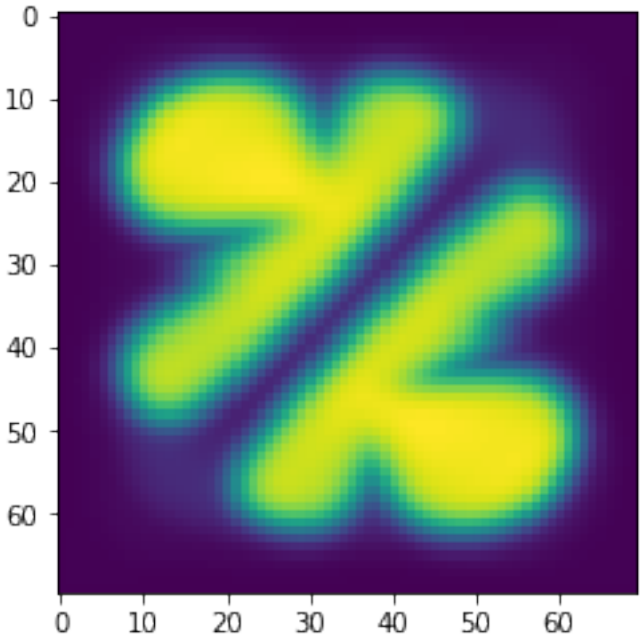
[51]:
```
def barycenter(iteration, A, K1, lambd):
    K = lambda x: np.dot(np.dot(K1,x),K1)

    N = A.shape[0]
    R = A.shape[2]

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

    err = np.zeros(iteration)
    for i in range(iteration):

        for k in range(R):
            err[i] = err[i] + np.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])
    return b, err
```

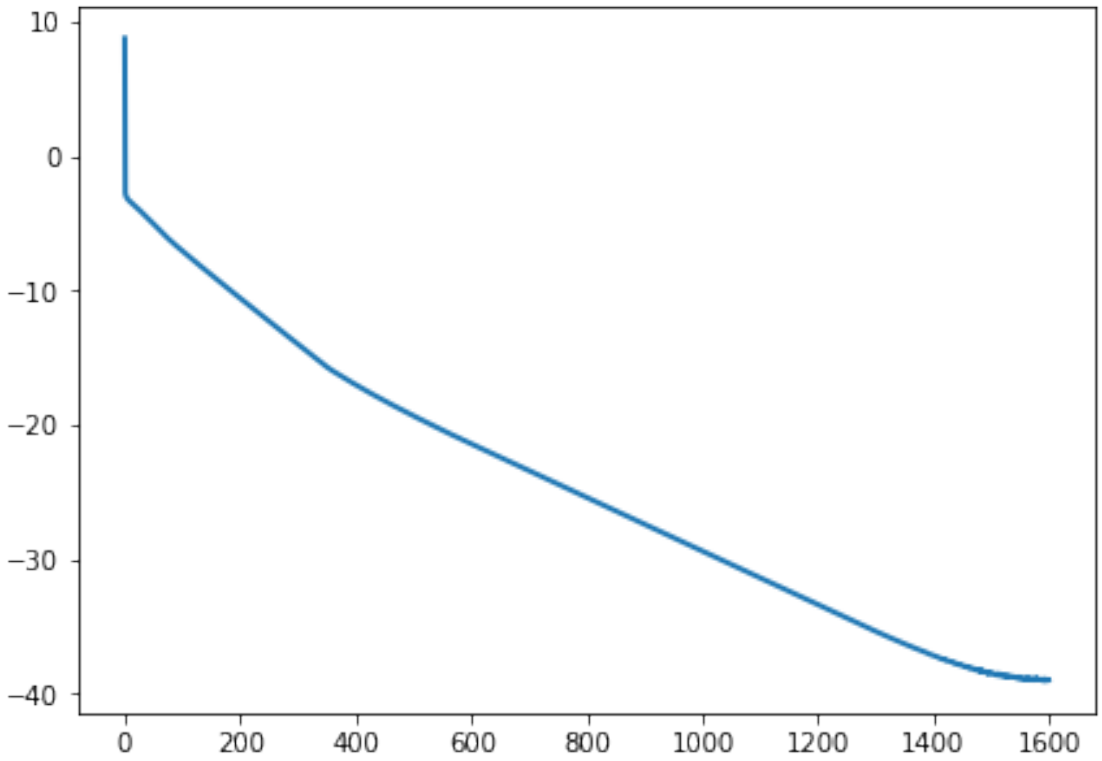
[52]:
```
iteration = 1600
bary, err = barycenter(iteration, A, K1, lambd)

plt.figure(figsize=(7,5))
plt.plot(err,linewidth = 2)
plt.yscale('log')
plt.show()

plt.figure()
plt.imshow(bary)
```



[52]: <matplotlib.image.AxesImage at 0x2848a4154a8>



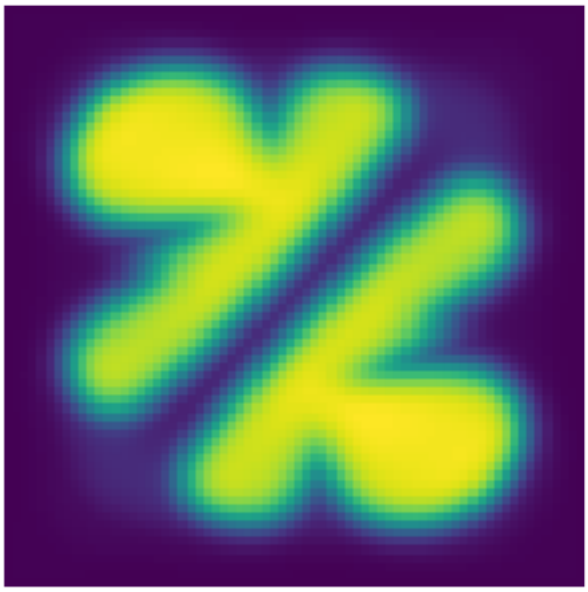
[53]:
```
run -i nt_solutions/optimaltransp_5_entropic/exo4
```




```
<Figure size 432x288 with 0 Axes>
```

Display the barycenter.

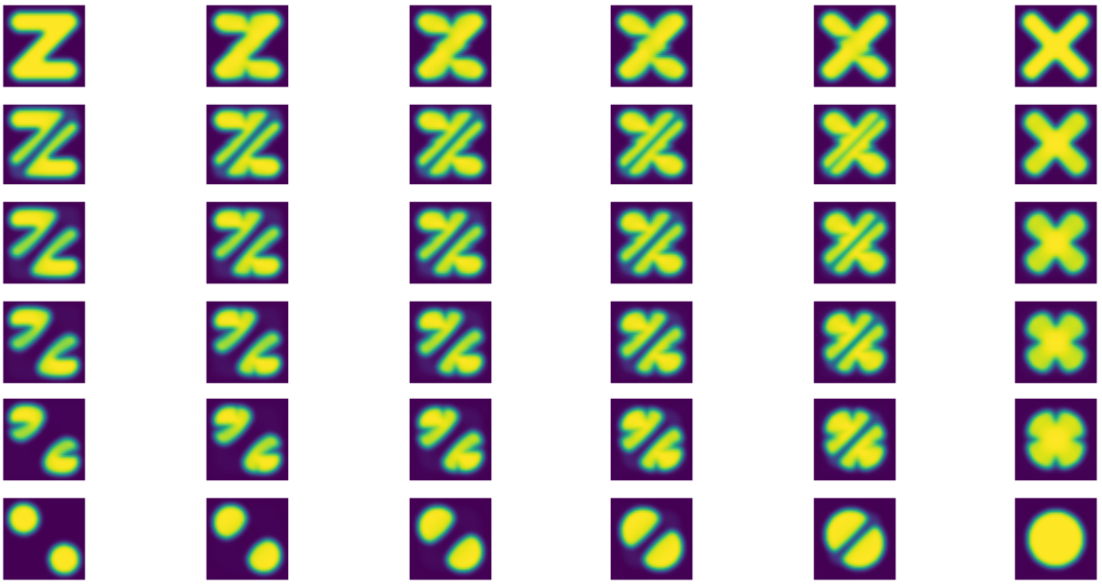
[54]:
```
plt.imshow(b)
plt.axis('off');
```



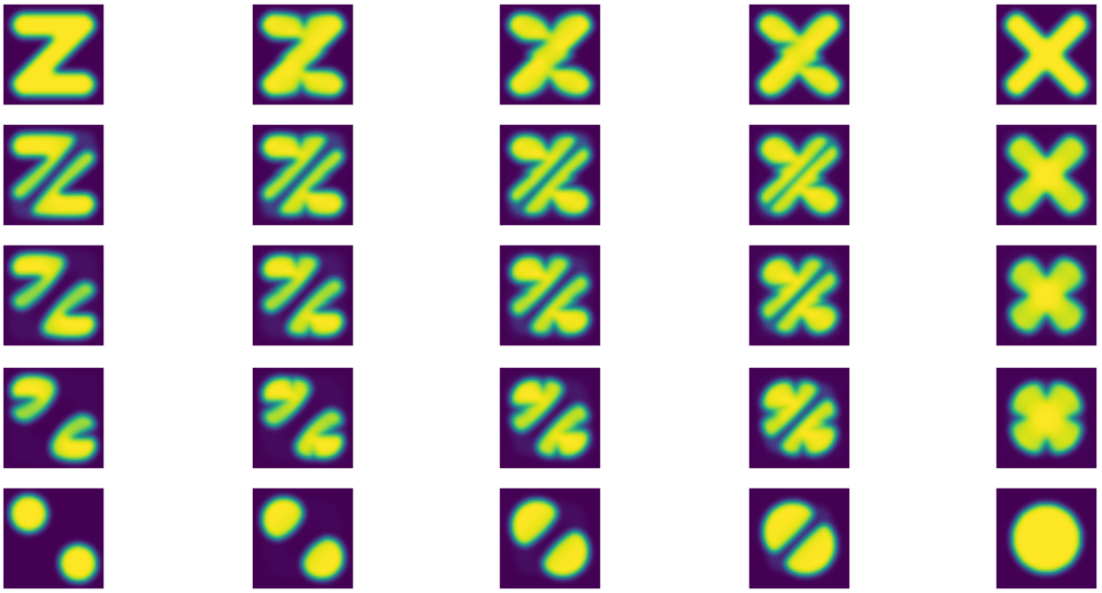
Exercise 5

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

[55]:
```
m = 6
[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")

iteration = 1000
plt.figure(figsize=(10,5))
for j in range(m**2):
    lambd_j = np.hstack((S[j]*T[j], (1-S[j])*T[j], S[j]*(1-T[j]),␣
 ↪(1-S[j])*(1-T[j])))
    bary_j, _ = barycenter(iteration, A, K1, lambd_j)
    plt.subplot(m,m,j+1)
    plt.imshow(bary_j)
    plt.axis('off')
```



[56]:
```
run -i nt_solutions/optimaltransp_5_entropic/exo5
```



1.7 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.

Numerical Experiment 3

January 8, 2021

1 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].

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.1 Dual OT and c-transforms

The primal Kantorovitch OT problem reads

$$W_c(,) = \pi \int_{\times} c(x,y)\mathrm{d}\pi(x,y)\pi_1 =, \pi_2 =.$$

It dual is

$$W_c(,) = f,g \int_f \mathrm{d} + \int_g \mathrm{d}f(x) + g(y) \leq c(x,y).$$

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

$$f^c(y)ic(x_i,y) - f_i.$$

The function to maximize is then

$$W_c(,) = f \in^n(f) \sum_i f_i a_i + \int f^c(y)\mathrm{d}(y).$$

1.2 Semi-discret via Gradient Ascent

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

$$= \bigcup_i L_i(f) L_i(f) y \forall j, c(x_i, y) - f_i \leq c(x_j, y) - f_j.$$

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".

[2]:
```
p = 400  # 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 $= \sum_{i=1}^n a_{ix_i}$.

[3]:
```
n = 60
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)))
a = np.ones(n)/n
```

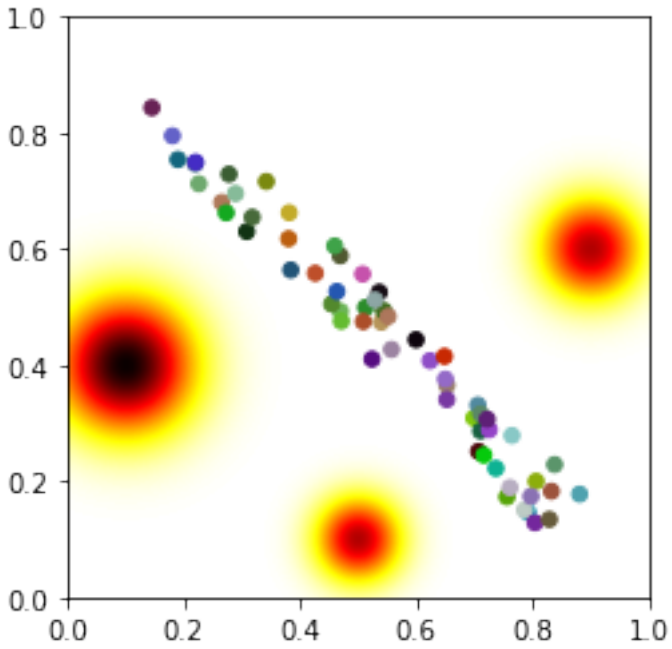
Second measure , potentially a continuous one (i.e. with a density), mixture of Gaussians. Here we discretize $\beta = \sum_{j=1}^m b_{jy_j}$ on a very fine grid.

[4]:
```
def Gauss(mx, my, s): return np.exp((-(U-mx)**2-(V-my)**2)/(2*s**2))


Mx = [.6, .4, .1]  # means
My = [.9, .1, .5]
S = [.06, .08, .05]  # variance
W = [.3, .4, .3]  # 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])
b = b/np.sum(b.flatten())
```

Display the two measures.

[5]:
```
Col = np.random.rand(n, 3)
plt.imshow(-b[::-1, :], extent=[0, 1, 0, 1], cmap='hot')
plt.scatter(X[1, :], X[0, :], s=30, c=.8*Col)
```

[5]:
```
<matplotlib.collections.PathCollection at 0x7f10f8ae3c70>
```



Initial potentials.

[6]:
```
f = np.zeros(n)
```

compute Laguerre cells and c-transform

[7]:
```
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, $fa + f^c$.

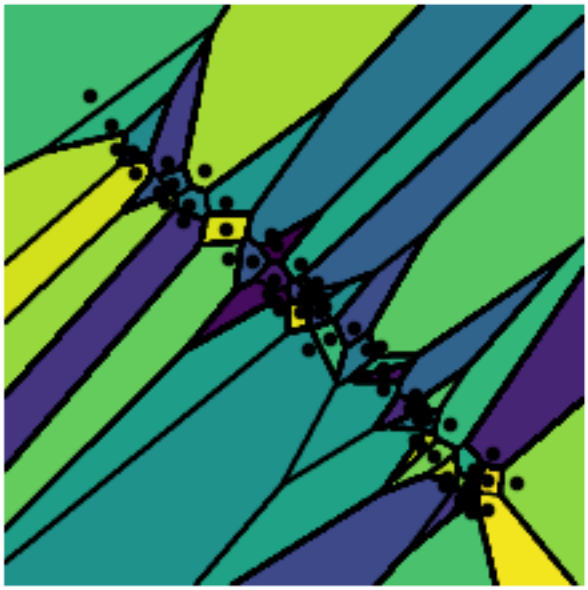
[8]:
```
OT = np.sum(f*a) + np.sum(fC*b.flatten())
print(OT)
```

```
0.09200566152064497
```

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

[9]:
```
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')
```

[9]: (0.0, 1.0, 0.0, 1.0)



Where has a density with respect to Lebesgue measure, then is smooth, and its gradient reads

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

sum area captured

Exercise 1

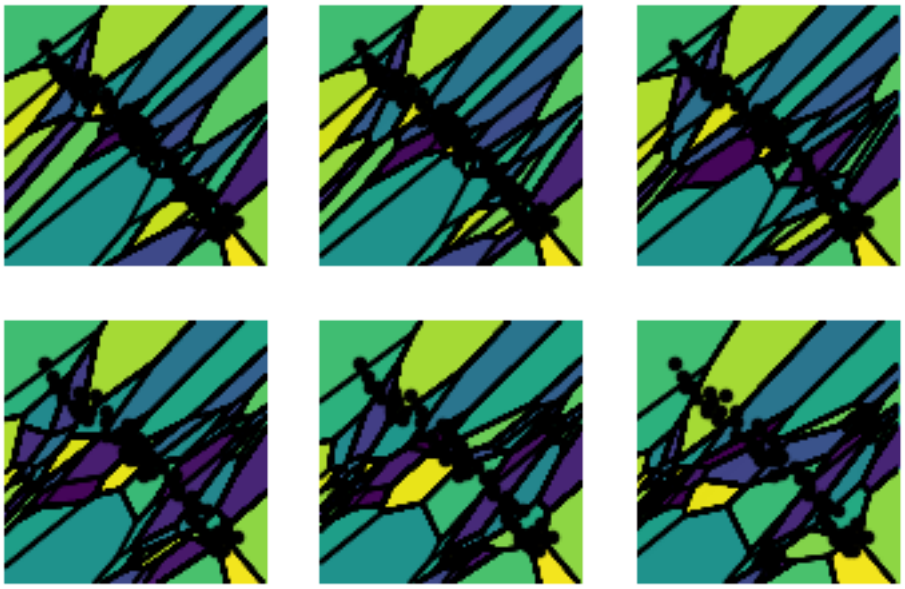
Implement a gradient ascent

$$f \leftarrow f + \tau\nabla(f).$$

Experiment on the impact of τ, display the evolution of the OT value and of the Laguerre cells.

[10]:
```
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
```

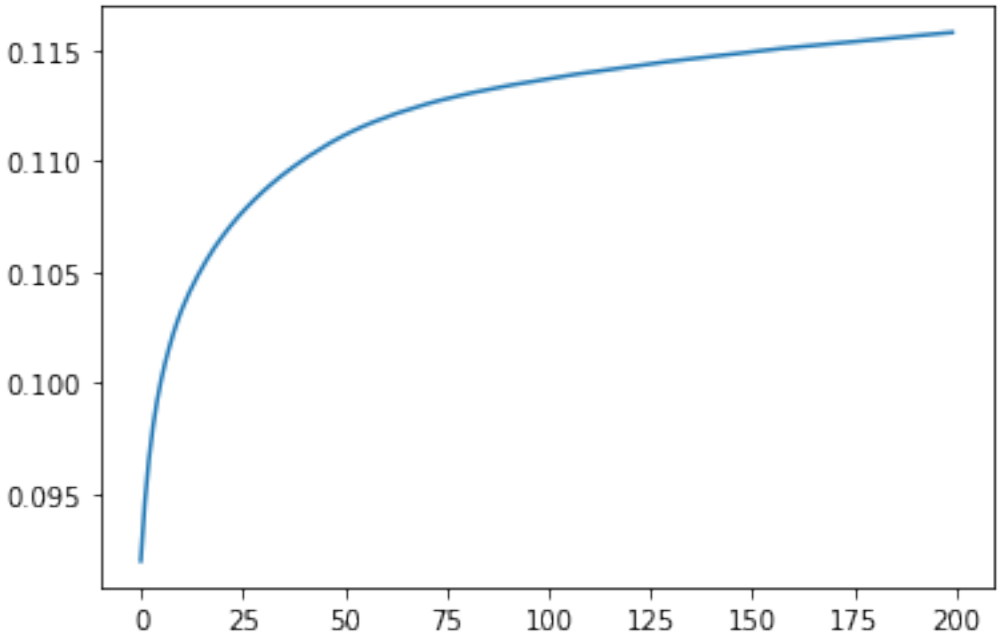
```
    if (kdisp < len(ndisp)) and (ndisp[kdisp] == it):
        plt.subplot(2, 3, kdisp+1)
        plt.imshow(I[::-1, :], extent=[0, 1, 0, 1])
        plt.scatter(X[1, :], X[0, :], s=20, c='k')
        plt.contour(t, t, I, np.linspace(-.5, n-.5, n), colors='k')
        plt.axis('off')
        kdisp = 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.

[11]:
```
plt.plot(E, '-')
```

[11]:
```
[<matplotlib.lines.Line2D at 0x7f10f7c4a940>]
```



1.3 Stochastic Optimization

The function to minimize can be written as an expectation over a random variable $Y \sim$

$$(f) = (E(f,Y))E(f,y) = fa + f^c(y).$$

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 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.

[12]:
```
f = np.zeros(n)
```

Draw the sample.

[13]:
```
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.

[14]:
```
R = np.sum(y**2) + np.sum(X**2, axis=0) - 2*y.transpose().dot(X) - f[:]
i = np.argmin(R)
```

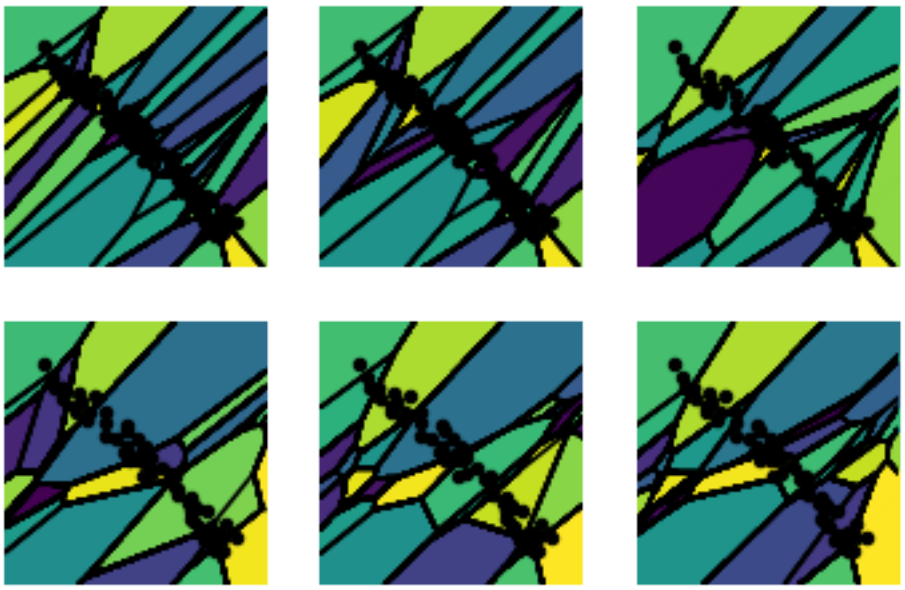
Randomized gradient.

[15]:
```
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.

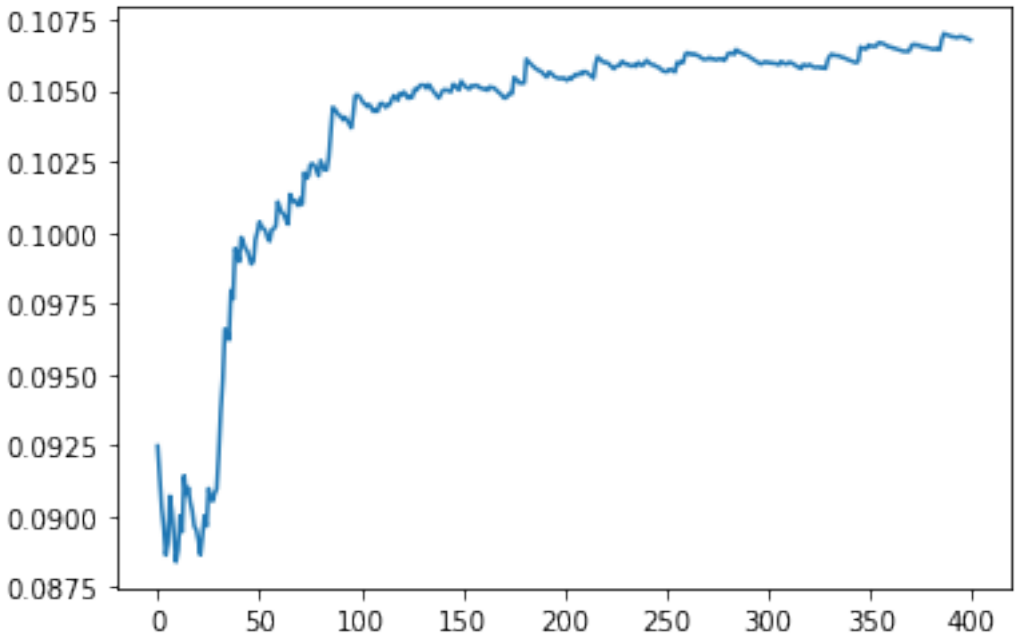
[16]:
```
niter = 400
q = 6
ndisp = np.unique(np.round(1 + (niter/2-1)*np.linspace(0, 1, q)**2))
kdisp = 0
E = np.zeros(niter)
for it in range(niter):
    # sample
    k = np.int(np.random.rand(1) < W[1])  # select one of the two Gaussian
    y = np.array((S[k] * np.random.randn(1) + Mx[k],
                  S[k] * np.random.randn(1) + My[k]))
    # 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 = 10  # warmup phase.
    tau = .1/(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[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
```



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

[17]:
```
plt.plot(E)
```

[17]:
```
[<matplotlib.lines.Line2D at 0x7f10f7d07640>]
```



1.4 Optimal Quantization and Lloyd Algorithm

We consider the following optimal quantization problem [Gruber02]

$$(a_i)_i, (x_i)_i W_c \sum_i a_{i x_i},.$$

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

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

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

The gradient reads

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

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

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

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

Intialize the centroids positions.

[18]:
```
X1 = X.copy()
```

Compute the Voronoi cells $V_i(x)$.

[19]:
```
D = D = distmat(Y, X1)
fC = np.min(D, axis=1)
I = np.reshape(np.argmin(D, axis=1), [p, p])
```

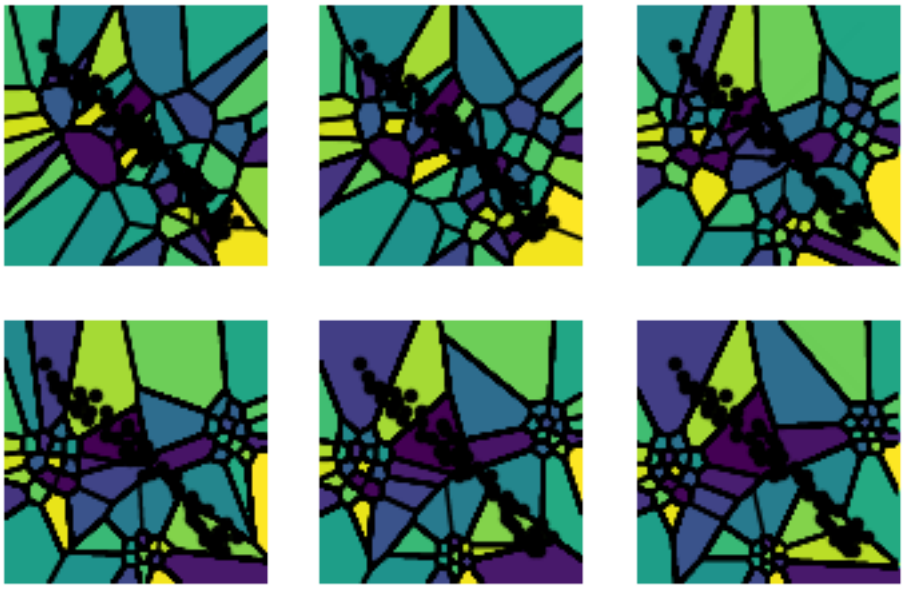
Update the centroids to the barycenters.

[20]:
```
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.

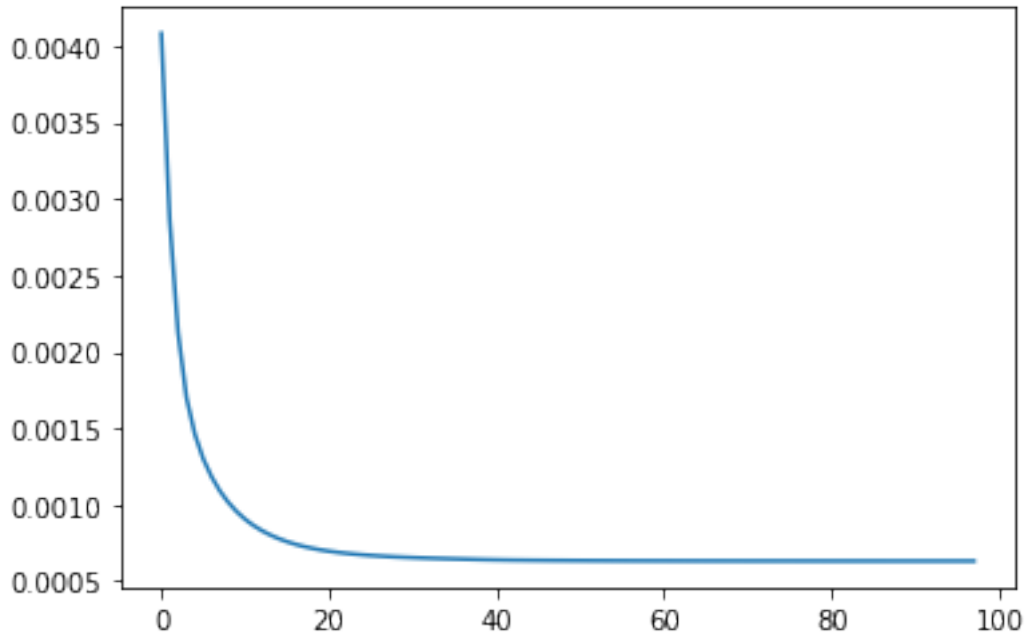
[21]:
```
niter = 100
q = 6
ndisp = np.unique(np.round(1 + (niter/4-1)*np.linspace(0, 1, q)**2))
kdisp = 0
E = np.zeros(niter)
X1 = X.copy()
for it in range(niter):
    # compute Voronoi cells
    D = D = distmat(Y, X1)
    fC = np.min(D, axis=1)
    I = np.reshape(np.argmin(D, axis=1), [p, p])
    E[it] = 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.

[22]:
```
plt.plot(E[1:-1])
```

[22]:
```
[<matplotlib.lines.Line2D at 0x7f10f60aa670>]
```



1.5 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.

Numerical Experiment 4

January 8, 2021

1 Advanced Topics on Sinkhorn Algorithm

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

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.1 Log-domain Sinkhorn

For simplicity, we consider uniform distributions on point clouds, so that the associated histograms are $ (a,b) ∈■𝑛 ×■𝑛$$beingconstant1/𝑛 and $1/𝑚$.

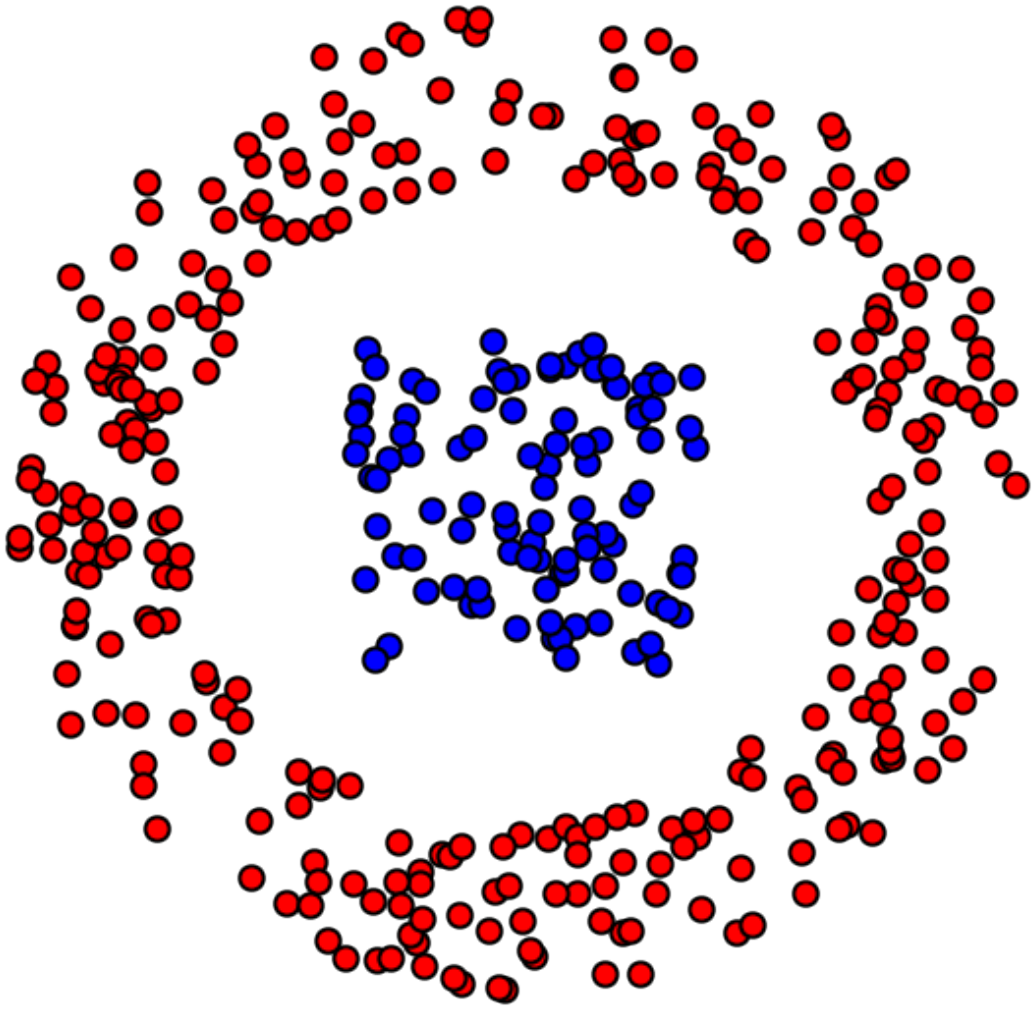
[2]:
```
n = 100
m = 300
a = np.ones((n,1))/n
b = np.ones((1,m))/m
```

Point clouds x and y.

[3]:
```
x = np.random.rand(2,n)-.5
theta = 2*np.pi*np.random.rand(1,m)
r = 1. + .5*np.random.rand(1,m)
y = np.vstack((np.cos(theta)*r,np.sin(theta)*r))
```

Display of the two clouds.

[4]:
```
plotp = lambda x,col: plt.scatter(x[0,:], x[1,:], s=150, edgecolors="k", c=col,␣
 ↪linewidths=2)
plt.figure(figsize=(10,10))
plotp(x, 'b')
plotp(y, 'r')
plt.axis("off");
```



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

[5]:
```
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) ∈■𝑛 ×■𝑛$.

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

$$f_i \leftarrow \min_\epsilon^b (C_{i,\cdot} - g)$$

$$g_j \leftarrow \min_\epsilon^a (C_{\cdot,j} - f),$$

where the regularized minimum operator reads

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

[6]:
```
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 $c \in$, one has

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

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

[7]:
```
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);
```

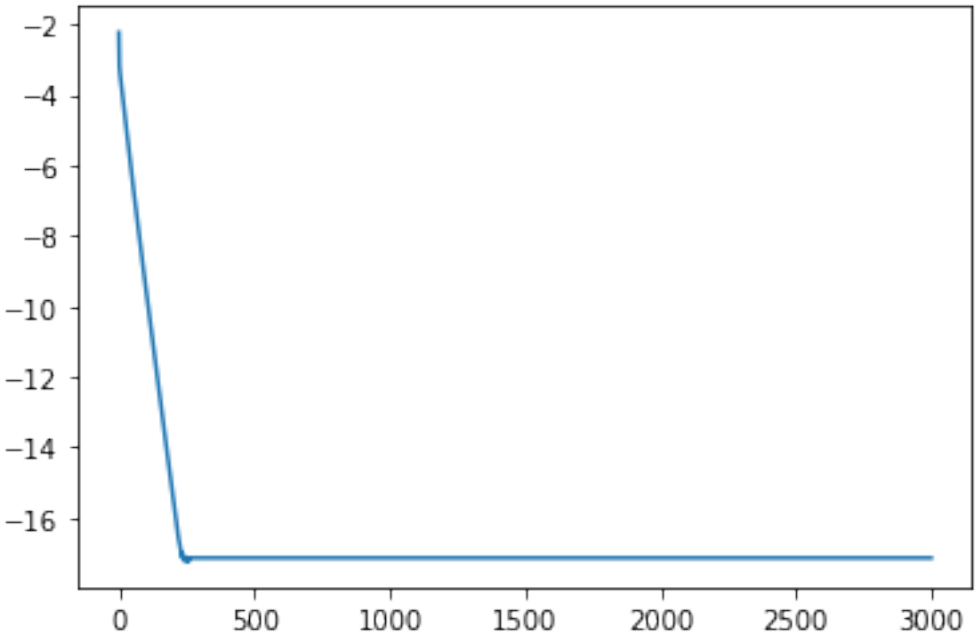
Value of ϵ.

[8]:
```
epsilon = 0.1
```

Exercise 1

Implement Sinkhorn in log domain.

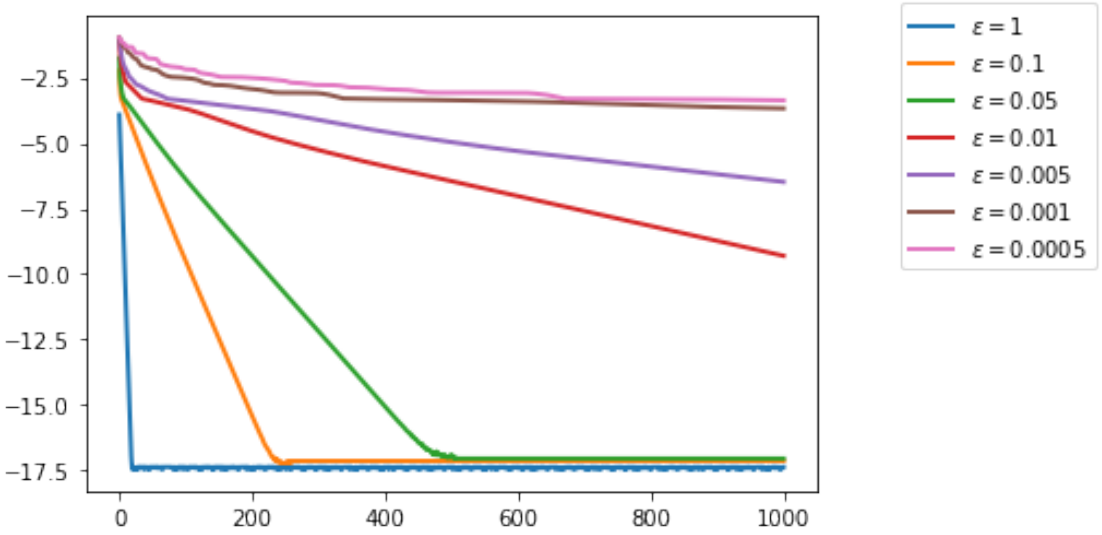
[9]:
```
def Sinkhorn(C,epsilon,f,niter = 500):
    Err = np.zeros(niter)
    for it in range(niter):
        g = mina(C-f[:,None],epsilon)
        f = minb(C-g[None,:],epsilon)
        # generate the coupling
        P = a * np.exp((f[:,None]+g[None,:]-C)/epsilon) * b
        # check conservation of 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.

[10]:
```
plt.figure()
for epsilon in (1,0.1, 0.05, 0.01, 0.005,0.001, 0.0005):
    (P,Err) =  Sinkhorn(C,epsilon,np.zeros(n),1000)
    plt.plot(np.log10(Err), label='$\epsilon=$' + str(epsilon),linewidth=2)
plt.legend(bbox_to_anchor=(1.1, 1.05));
plt.show()
```



1.2 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

$$(z) W_\epsilon \frac{1}{n}\sum_i z_i, \frac{1}{m}\sum_i y_i.$$

Here we have denoted the Sinkhorn score as

$$W_\epsilon(,)PC - \epsilon \mathrm{KL}(P|ab^\top)$$

where $= \frac{1}{n}\sum_i x_i$ and $= \frac{1}{m}\sum_i y_i$ are the measures (beware that C depends on the points positions).

[11]:
```
z = x # initialization
```

The gradient of this energy reads

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

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

$$(\bar{\nabla}(z))_i = z_i - \bar{y}_i \bar{y}_i \frac{\sum_j P_{i,j} y_j}{a_i}.$$

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.

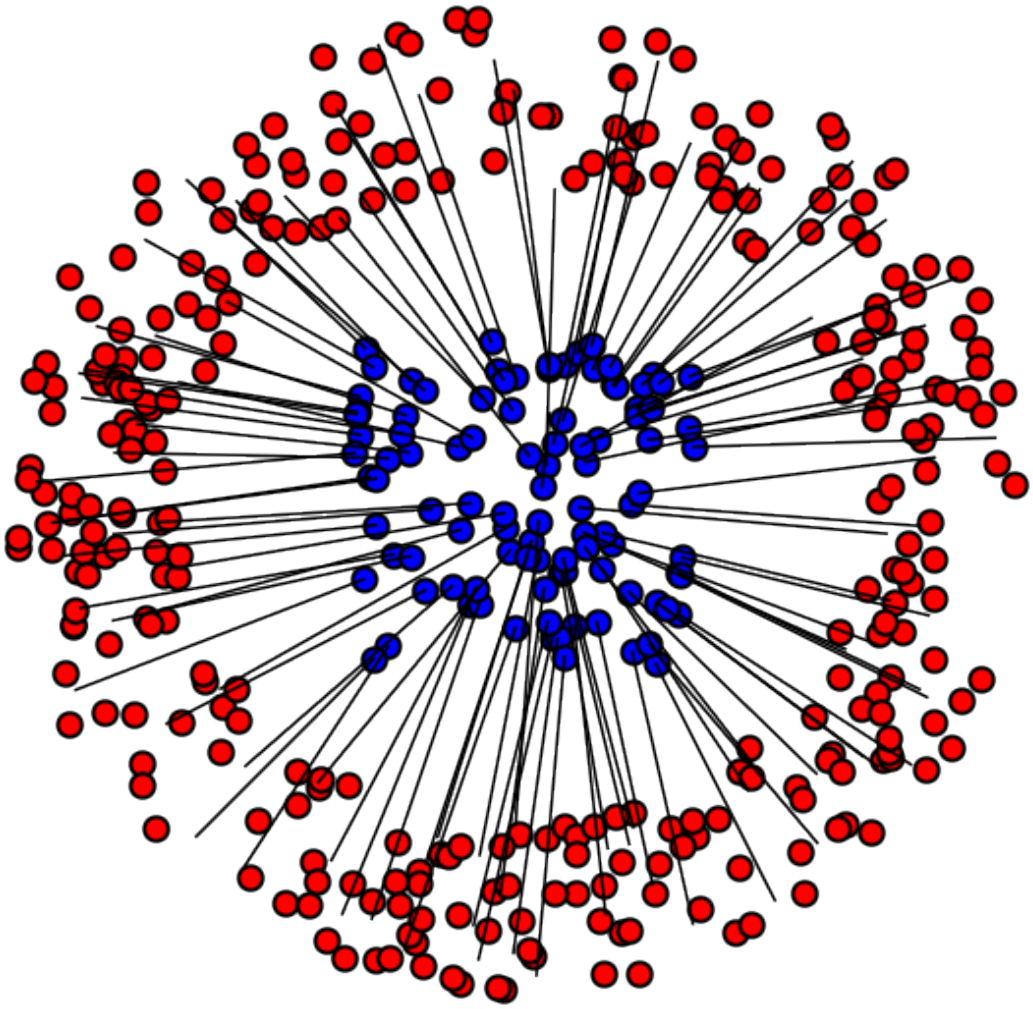
[12]:
```
epsilon = .005
niter = 300
(P,Err) = Sinkhorn(distmat(z,y), epsilon,np.zeros(n),niter);
```

Compute the gradient

[13]:
```
G = z - ( y.dot(P.transpose()) ) / a.transpose()
```

Display the gradient field.

[14]:
```
plt.figure(figsize=(10,10))
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.

[15]:
```
tau = 0.1
```

Update the point cloud.

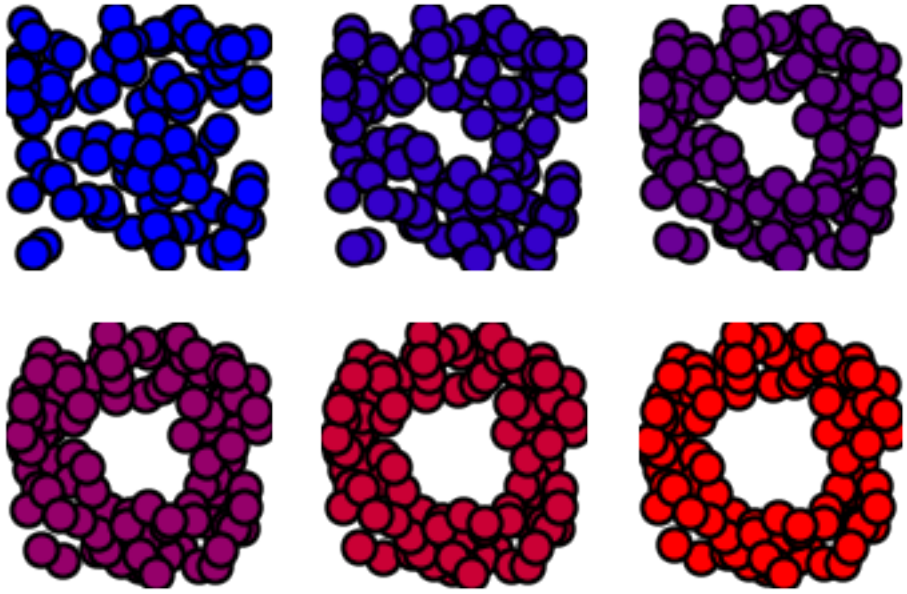
[16]:
```
z = z - tau * G
```

Exercise 3

Implement the gradient flow.

[17]:
```
z = x; # initialization
tau = .02; # step size for the descent
giter = 30; # 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
    # Sinkhorn
    (P,Err) = Sinkhorn(distmat(z,y), epsilon,f,niter)
    # gradient
    G = z - ( y.dot(P.transpose()) ) / a.transpose()
    z = z - tau * G;
```



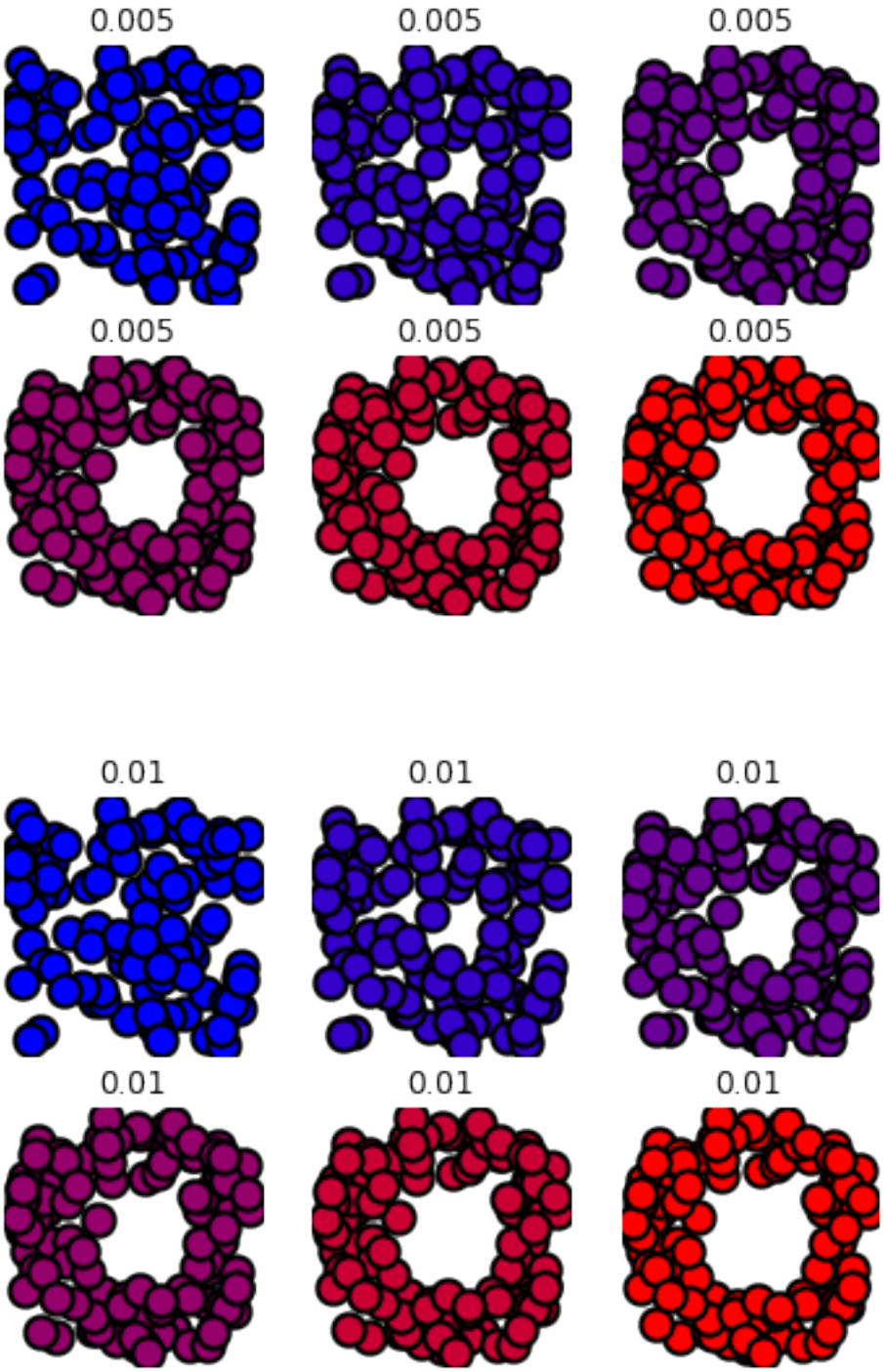
Exercise 4

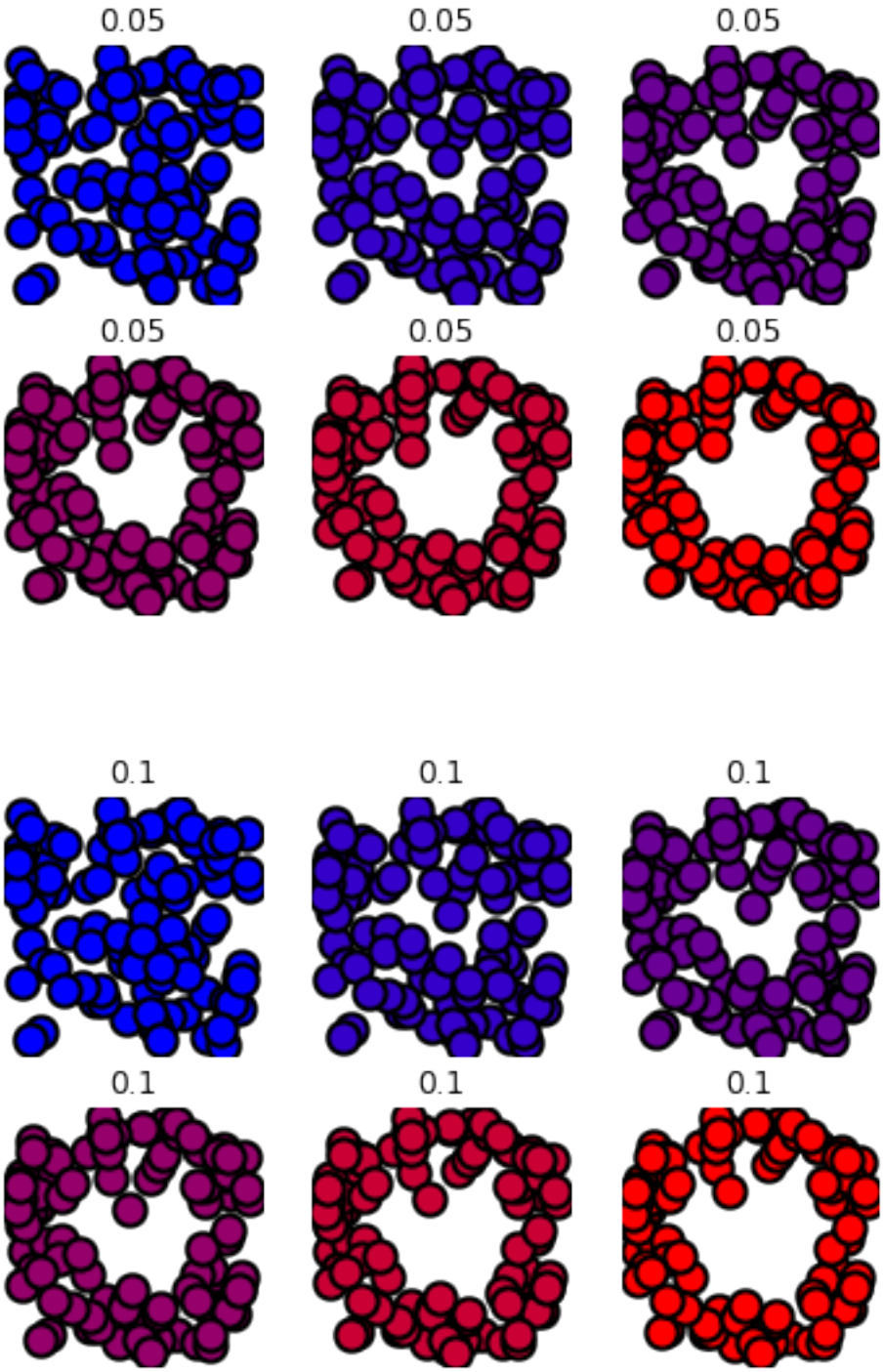
Show the evolution of the fit as ϵ increases. What do you observe. Replace the Sinkhorn score $W_\epsilon(,)$ by the Sinkhorn divergence $W_\epsilon(,) - W_\epsilon(,)/2 - W_\epsilon(,)/2$.

[18]:
```
## Insert your code here.

def gradient_flow(x,a,y,giter,tau,epsilon):
    n = x.shape[1]
    z = x; # initialization
```

```
    ndisp = np.round( np.linspace(0,giter-1,6) )
    kdisp = 0
    f = np.zeros(n)
    plt.figure()

    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.title(epsilon)
            plt.axis("off")
            kdisp = kdisp+1

        # Sinkhorn
        (P,Err) = Sinkhorn(distmat(z,y), epsilon,f,niter=500)
        # gradient
        G = z - ( y.dot(P.transpose()) ) / a.transpose()
        z = z - tau * G;
```

[19]:
```
tau = .02; # step size for the descent
giter = 30; # iter for the gradient descent
for epsilon in (0.005,0.01,0.05,0.1):
    gradient_flow(x,a,y,giter,tau,epsilon)
```


0.005
0.005
0.005
0.005
0.005
0.005
0.01
0.01
0.01
0.01
0.01
0.01



1.3 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 of encalled a generative model in the machine learning litterature, and correspon

The matching is achieved by solving

$$\min_{()} (G_{(z))=W\epsilon} \frac{1}{n} \Sigma_i {}_{g(z_i)}, \frac{1}{m} \Sigma_i y_i,$$

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

The gradient reads

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

where $\partial g_{(z_i)^*}$ is the adjoint of the Jacobian of g.

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

Denoting $ v_i = \nabla (G_(z))_i$ $ the gradient of the Sinkhorn loss (which is computed as in the previous section), the gradient w.r.t. h is $\sum_i v_i z_i^\top \nabla_h() = \sum_i v_i$.

Generate the data.

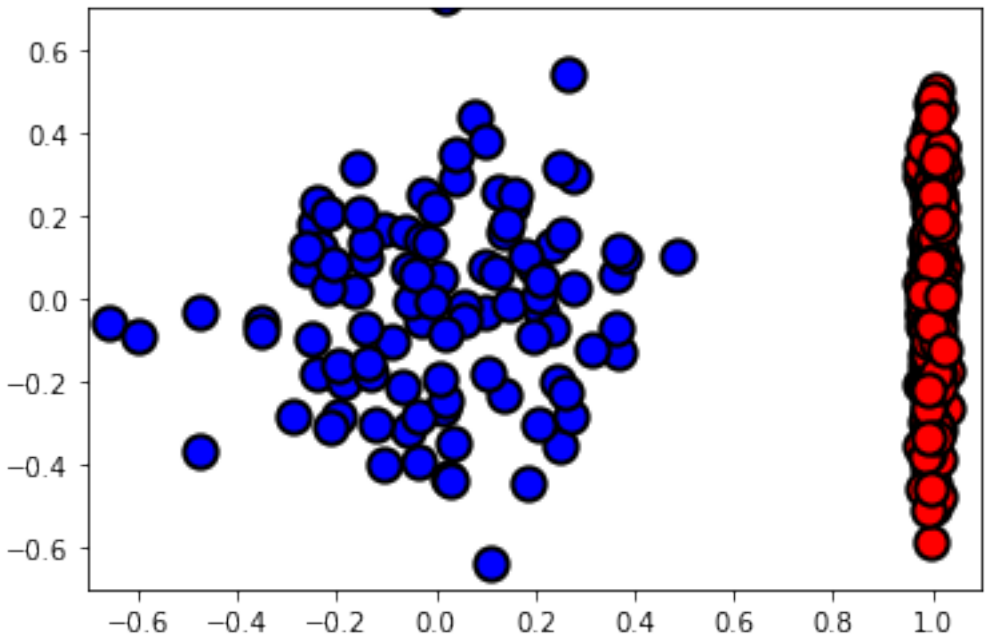
[20]:
```
z = np.random.randn(2,n)*.2
y = np.random.randn(2,m)*.2
y[0,:] = y[0,:]*.05 + 1
```

Initialize the parameters.

[21]:
```
A = np.eye(2)
h = np.zeros(2)
```

Display the clouds.

[22]:
```
plotp(A.dot(z)+h[:,None], 'b')
plotp(y, 'r')
plt.xlim(-.7,1.1)
plt.ylim(-.7,.7);
```



Run Sinkhorn.

[23]:
```
x = A.dot(z)+h[:,None]
f = np.zeros(n)
(P,Err) = Sinkhorn(distmat(x,y), epsilon,f,niter)
```

Compute gradient with respect to positions.

[24]:
```
v = a.transpose() * x - y.dot(P.transpose())
```

gradient with respect to parameters

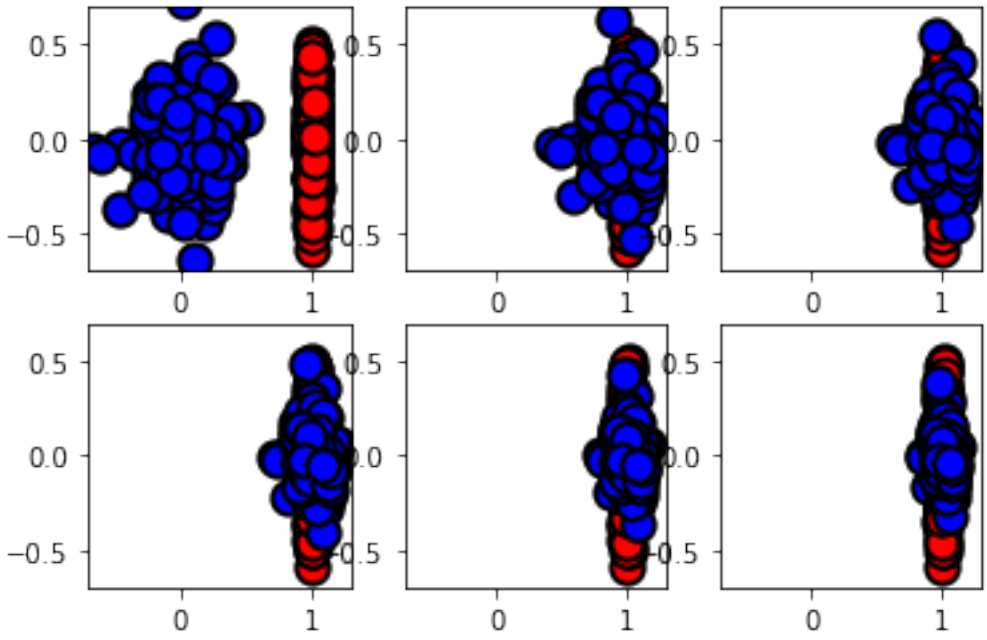
[25]:
```
nabla_A = v.dot(z.transpose())
nabla_h = np.sum(v,1)
```

Exercise 5

Implement the gradient descent.

[26]:
```
A = np.eye(2)
h = np.zeros(2)
# step size for the descent
tau_A = .5
tau_h = .2
# #iter for the gradient descent
giter = 60
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.

[27]:
```
###  create new points y
theta = np.pi*np.random.rand(1,m)/2-np.pi/4
r = 1. + .1*np.random.rand(1,m)
y = np.vstack((np.cos(theta)*r,np.sin(theta)*r))
```

```
A = np.random.rand(2, 2)
h = np.zeros(2)
# step size for the descent
tau_A = .2
tau_h = .1
# #iter for the gradient descent
giter = 60
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(-0.5,1.2)
        plt.ylim(-1,2)
    (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
```