sheet06_final

December 8, 2016

1 Weighted K-Means

In this exercise we will simulate finding good locations for production plants of a company in order to minimize its logistical costs. In particular, we would like to place production plants near customers so as to reduce shipping costs and delivery time.

We assume that the probability of someone being a customer is independent of its geographical location and that the overall cost of delivering products to customers is proportional to the squared Euclidean distance to the closest production plant. Under these assumptions, the K-Means algorithm is an appropriate method to find a good set of locations. Indeed, K-Means finds a spatial clustering of potential customers and the centroid of each cluster can be chosen to be the location of the plant.

Because there are potentially millions of customers, and that it is not scalable to model each customer as a data point in the K-Means procedure, we consider instead as many points as there are geographical locations, and assign to each geographical location a weight w_i corresponding to the number of inhabitants at that location. The resulting problem becomes a weighted version of K-Means where we seek to minimize the objective:

$$J(c_1, \ldots, c_K) = \frac{\sum_i w_i \min_k ||x_i - c_k||^2}{\sum_i w_i},$$

where c_k is the kth centroid, and w_i is the weight of each geographical coordinate x_i. In order to minimize this cost function, we iteratively perform the following EM computations:

- **Expectation step:** Compute the set of points associated to each centroid:

$$\forall\ 1 \leq k \leq K: \quad \mathcal{C}(k) \leftarrow \left\{ i \ : \ k = \arg\min_k \|x_i - c_k\|^2 \right\}$$

- **Minimization step:** Recompute the centroid as a the (weighted) mean of the associated data points:

$$\forall\ 1 \leq k \leq K: \quad c_k \leftarrow \frac{\sum_{i \in \mathcal{C}(k)} w_i \cdot x_i}{\sum_{i \in \mathcal{C}(k)} w_i}$$

until the objective $J(c_1, \ldots, c_K)$ has converged.

1.1 Getting started

In this exercise we will use data from http://sedac.ciesin.columbia.edu/, that we store in the files `data.mat` as part of the zip archive. The data contains for each geographical coordinates (latitude and longitude), the number of inhabitants and the corresponding country. Several variables and methods are provided in the file `utils.py`:

- `utils.population` A 2D array with the number of inhabitants at each latitude/longitude.
- `utils.countries` A 2D array with the country indicator at each latitude/longitude.

- `utils.nx` The number of latitudes considered.
- `utils.ny` The number of longitudes considered.
- `utils.plot(latitudes,longitudes)` Plot a list of centroids given as geographical coordinates in overlay to the population density map.

The code below plots three factories (white squares) with geographical coordinates (60,80), (60,90),(60,100) given as input.
** NOTE: DELETED THE PLOT **

1.2 Initializing Weighted K-Means (20 P)

Because K-means has a non-convex objective, choosing a good initial set of centroids is important. Centroids are drawn from from the following discrete probability distribution:

$$P(x, y) = \frac{1}{Z} \cdot \text{population}(x, y)$$

where Z is a normalization constant. Furthermore, to avoid identical centroids, we add a small Gaussian noise to the location of centroids, with standard deviation 0.01.
Tasks:

- **Implement the initialization procedure above.**
- **Run the initialization procedure for K=200 clusters.**
- **Visualize the centroids obtained with your initialization procedure using `utils.plot`.**

In [131]:
```
import numpy as np
from utils import population, countries, nx, ny
from utils import plot as uplot
%matplotlib inline
```

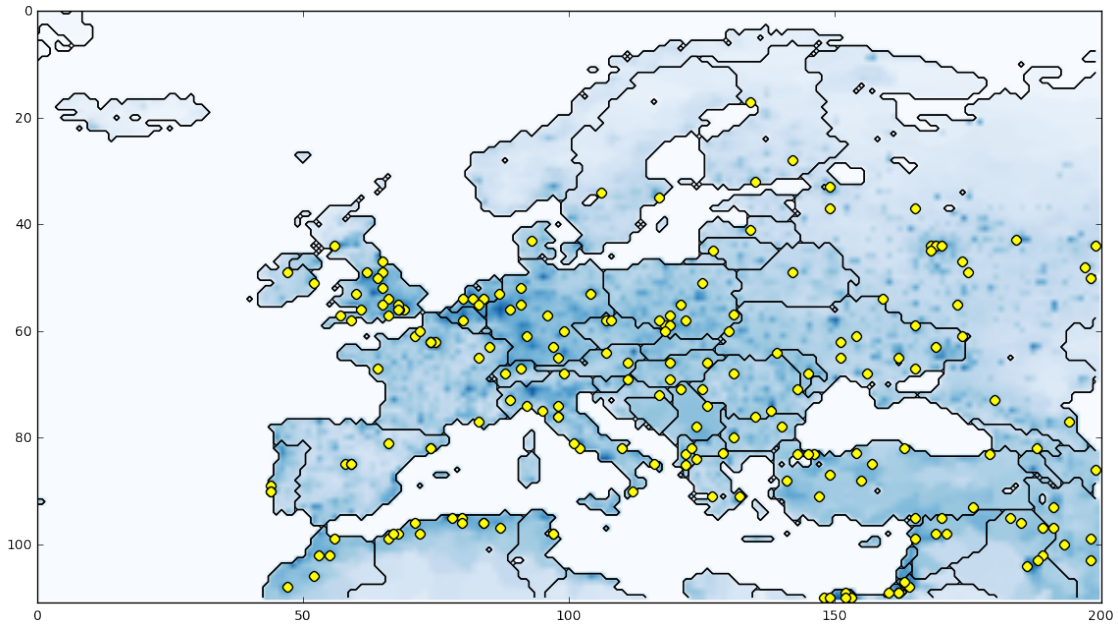
In [132]:
```
# flatten
pop_f = population.flatten()
ctr_f = countries.flatten()

# make vector of indices
ix = np.arange(len(pop_f))

# make array of lists containing coordinate pairs
xs = list(range(nx))
ys = list(range(ny))
coords = np.array(list([i,j] for i in xs for j in ys))
```

In [112]:
```
def initialise(coords, pop_f, K):
    # make vector of indices
    ix = np.arange(len(pop_f))

    # find normalising constant Z
    weight = 1/pop_f.sum() * pop_f

    # draw K indices
    res = np.random.choice(ix, size = K, replace = True, p=weight)

    # find resulting coordinate pairs [x,y]
    centr_init = coords[res]
```

```
        # draw 2D-array containing Gaussian noise
        mu, sigma = 0, 0.01
        noise = np.random.normal(mu, sigma, [K,2])

        # add noise to the selected coordinates [x+noise_x, y+noise_y]
        centr_init = centr_init + noise
        return(centr_init)
```

```
In [27]: centroid_init = initialise(coords, pop_f, K = 200)
         # re-arrange for plotting:
         init_xs = list(np.array(centroid_init)[:,0])
         init_ys = list(np.array(centroid_init)[:,1])
         # plot
         uplot(init_xs,init_ys)
```

```
C:\Users\johndoe\Anaconda3\lib\site-packages\matplotlib\collections.py:650: FutureWarning: elementwise c
  if self._edgecolors_original != str('face'):
C:\Users\johndoe\Anaconda3\lib\site-packages\matplotlib\collections.py:590: FutureWarning: elementwise c
  if self._edgecolors == str('face'):
```



1.3 Implementing Weighted K-Means (40 P)

Tasks:

- **Implement the weighted K-Means algorithm as described in the introduction.**
- **Run the algorithm with K=200 centroids until convergence (stop if the objective does not improve by more than 0.01). Convergence should occur after less than 50 iterations. If it takes longer, something must be wrong.**
- **Print the value of the objective function at each iteration.**

- **Visualize the centroids at the end of the training procedure using the methods `utils.plot`.**

In [36]:
```
# define functions
    # squared Euclidean distance
def sed(x, y):
    diff = np.array(x) - np.array(y)
    return np.dot(diff, diff)

    # find the closest centroid to some point
def close(x, centroids):
    dist = 10**10
    for i in range(len(centroids)):
        temp = centroids[i]
        res = sed(x, temp)
        if res < dist:
            dist = res
            cent = temp
            it = i
    return(dist, cent, it)
```

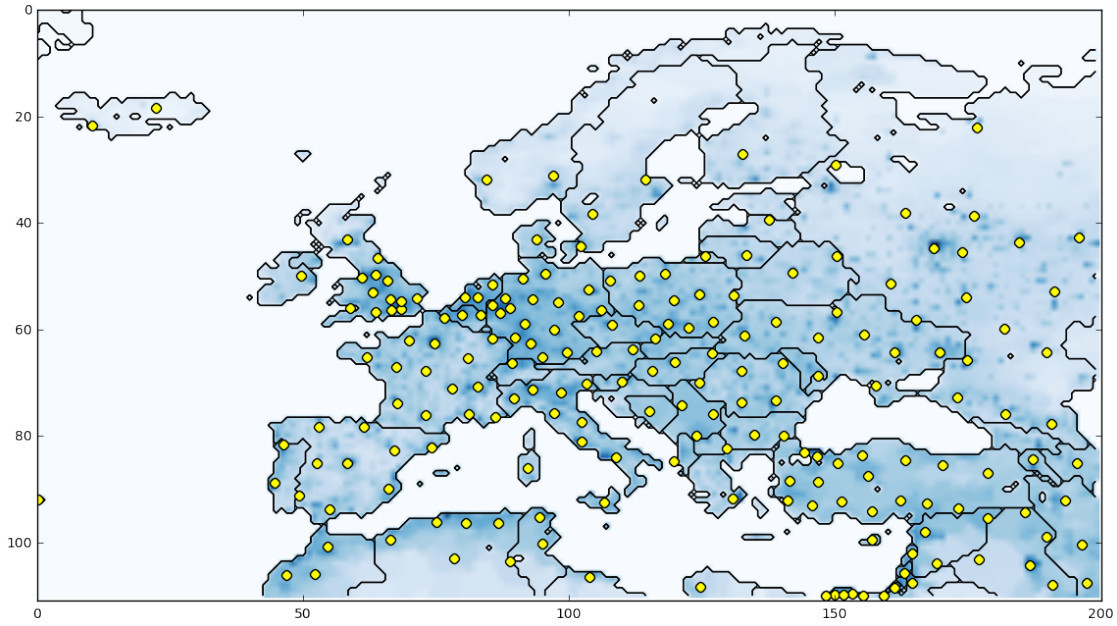
In [133]:
```
# E-step
def estep(coords, pop_f, centroids, J_old):
    Dist = []
    Cent = []
    It = []
    for x in coords:
        dist, cent, it = close(x,centroids)
        Dist.append(dist)
        Cent.append(cent)
        It.append(it)
    # calculate J (flattened weights are in 'pop_f')
    J = (pop_f * Dist).sum() / pop_f.sum()
    print("J =", J)
    J_old = J
    return(Dist, Cent, It, J_old)

# M-step
def mstep(coords, pop_f, It, K):
    coord_upd = np.zeros((K,2))
    for i in range(K):
        indices = [j for j, x in enumerate(It) if x == i]
        if len(indices) == 0:
            print("Found cluster with no members.")
            # in this case, we could also re-initialize this centroid,
            # but here we decided to just continue
            # since the issue is usually resolved in the next iteration.
            continue
        myx = np.array(coords[indices])[:,0]
        myy = np.array(coords[indices])[:,1]
        cent_x = (pop_f[indices] * myx).sum() / (pop_f[indices]).sum()
        cent_y = ((pop_f[indices] * myy).sum()) / (pop_f[indices].sum())
        coord_upd[i] = [cent_x, cent_y]
        # what happens if a cluster has NO members?
    return(coord_upd)
```

In [13]:
```
# EM until threshold is reached
    # initialize
go = True
J_old = 10**4
centroids = np.copy(centroid_init) # copied results from initialization in number 1
K = len(centroids)
    # run
while go:
    Dist, Cent, It, J = estep(coords, pop_f, centroids, J_old)
    centroids = mstep(coords, pop_f, It, K)
    go = J_old-J > 0.01
    J_old = J
print("Converged.")
```

```
J =  16.5465293085
J =  10.4804750271
J =  8.67876761118
J =  8.2086116817
J =  7.86369035238
J =  7.62200622104
J =  7.50976953302
J =  7.42967351176
J =  7.35553533291
J =  7.28408536107
J =  7.23896470713
J =  7.21251668317
J =  7.19886273376
J =  7.1866613348
J =  7.17389933467
J =  7.16751465318
Converged.
```

In [15]:
```
# once we're done, re-arrange and plot
centr_xs = list(np.array(centroids)[:,0])
centr_ys = list(np.array(centroids)[:,1])
# plot
uplot(centr_xs,centr_ys)
```

```
C:\Users\johndoe\Anaconda3\lib\site-packages\matplotlib\collections.py:650: FutureWarning: elementwise c
  if self._edgecolors_original != str('face'):
C:\Users\johndoe\Anaconda3\lib\site-packages\matplotlib\collections.py:590: FutureWarning: elementwise c
  if self._edgecolors == str('face'):
```



1.4 Focus on German-Speaking Countries (20 P)

Market analysis has shown that people in German-speaking countries, more precisely, Germany (country 111), Austria(country 104), and Switzerland (country 109), are 25 times more likely to be customers than in other countries.

Tasks:

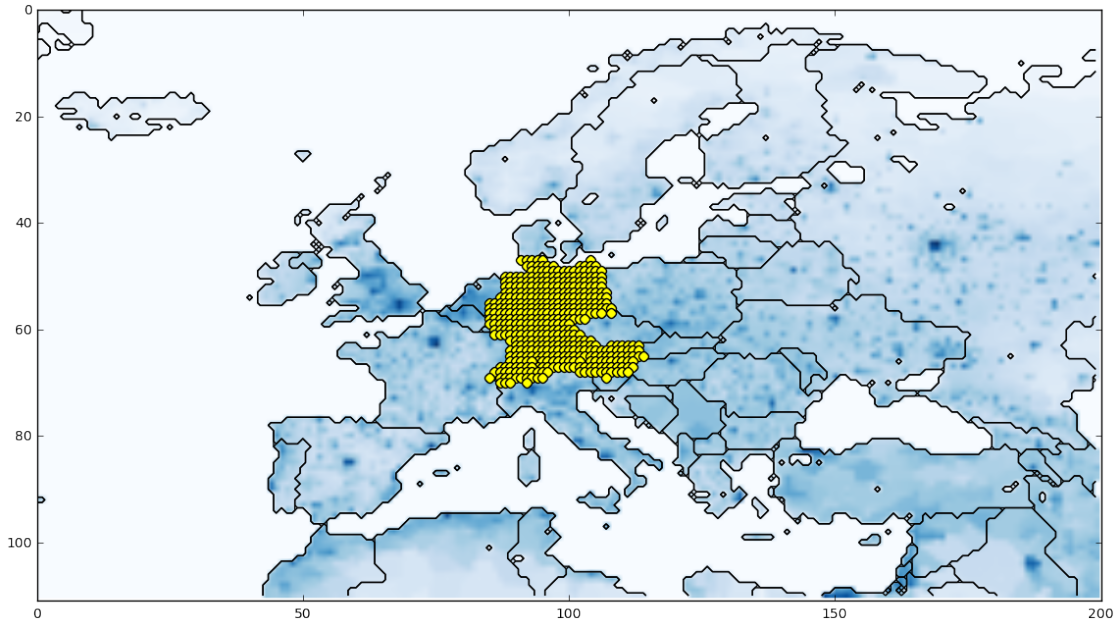
- **Describe the necessary changes to the problem setup to take into account this new constraint.**
- **Run k-means on the modified problem.**
- **Visualize the newly obtained centroids using the methods `utils.plot`.**

Multiply the weights for all the coordinates in these countries by 25, and re-estimate (I use the final result from the last exercise to initialize).

```
In [28]: # make a flag for coordinates corresponding to German-speaking countries
         ger = [111, 104, 109]
         flag = []
         for c in ger:
             flag.append([i for i, x in enumerate(ctr_f) if x==c])
         flag = [item for sublist in flag for item in sublist]
```

```
In [29]: # let's see if we got this right:
         ger_xs = list(np.array(coords[flag])[:,0])
         ger_ys = list(np.array(coords[flag])[:,1])
         # plot
         uplot(ger_xs,ger_ys)
```

```
C:\Users\johndoe\Anaconda3\lib\site-packages\matplotlib\collections.py:650: FutureWarning: elementwise c
  if self._edgecolors_original != str('face'):
C:\Users\johndoe\Anaconda3\lib\site-packages\matplotlib\collections.py:590: FutureWarning: elementwise c
  if self._edgecolors == str('face'):
```



In [32]:
```
# looks good, so let's update the weight vector by multiplying by 25 all elements
# that correspond to German-speaking coordinates (new weight vector is called 'w')
w = np.copy(pop_f)
w[flag] = w[flag] * 25

# we could re-initialize using the updated weights and the results from number 1,
# but I decided to just take the solution from number 2.
centroid_init_ger = np.copy(centroids)
# instead of:
#centroid_init_ger = initialise(coords, w, K = 200)
```

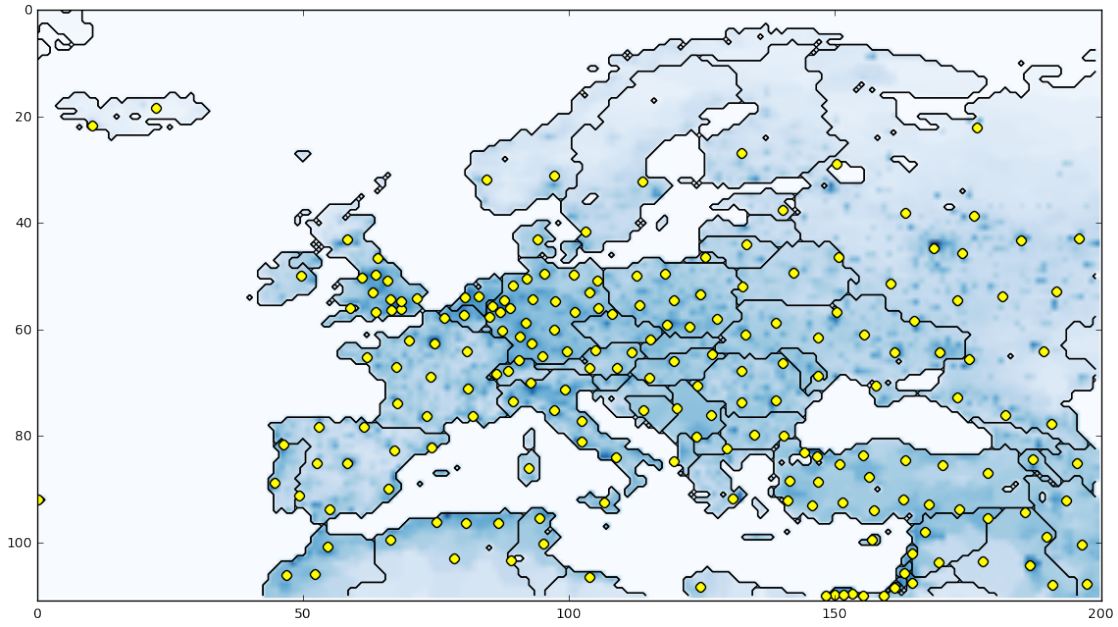
In [33]:
```
# EM until threshold is reached
    # initialize
go = True
J_old = 10**4
centroids_ger = np.copy(centroid_init_ger)
K = len(centroids_ger)
    # run
while go:
    Dist, Cent, It, J = estep(coords, w, centroids_ger, J_old)
    centroids_ger = mstep(coords, w, It, K)
    go = J_old-J > 0.01
    J_old = J
print("Converged.")
```

```
J =  4.08230082164
J =  3.81718276006
J =  3.71699091335
J =  3.65073750584
J =  3.59505827358
J =  3.57144974134
```

```
J =  3.55585594353
J =  3.55044387587
Converged.
```

```
In [34]: # once we're done, re-arrange and plot
         centr_ger_xs = list(np.array(centroids_ger)[:,0])
         centr_ger_ys = list(np.array(centroids_ger)[:,1])
         # plot
         uplot(centr_ger_xs,centr_ger_ys)
```

```
C:\Users\johndoe\Anaconda3\lib\site-packages\matplotlib\collections.py:650: FutureWarning: elementwise c
  if self._edgecolors_original != str('face'):
C:\Users\johndoe\Anaconda3\lib\site-packages\matplotlib\collections.py:590: FutureWarning: elementwise c
  if self._edgecolors == str('face'):
```



1.5 Shipping Restrictions (20 P)

We now suppose that deliveries across national borders are taxed heavily, and should be avoided as much as possible.

Tasks:

- **Describe the necessary changes to the problem setup to take into account this new constraint.**
- **Run k-means on the modified problem.**
- **Visualize the newly obtained centroids using the methods `utils.plot`.**

Solution (sketch)

- Adapt the E-step: When looking for the closest cluster centroid, only consider centroids that are in the same country as the coordinate.

- Possibly adapt initialisation in such a way that we get at least one cluster centroid per country (cf. `initialiseNational` – works, also see map below.)
- Determine "nationality" of cluster centroid by nationality of nearest coordinate (cf. `findCountry`).
- Unfortunately, implementing the actual updating process was too complicated for us! :(

In [58]:
```
# we have 11225 coordinates w/o country and w/ zero weight
print((pop_f == 0).sum())
print((ctr_f == -9999).sum())
# the indeces are the same!
print(np.array_equal(np.where(pop_f == 0), np.where(ctr_f == -9999)))
```

```
11225
11225
True
```

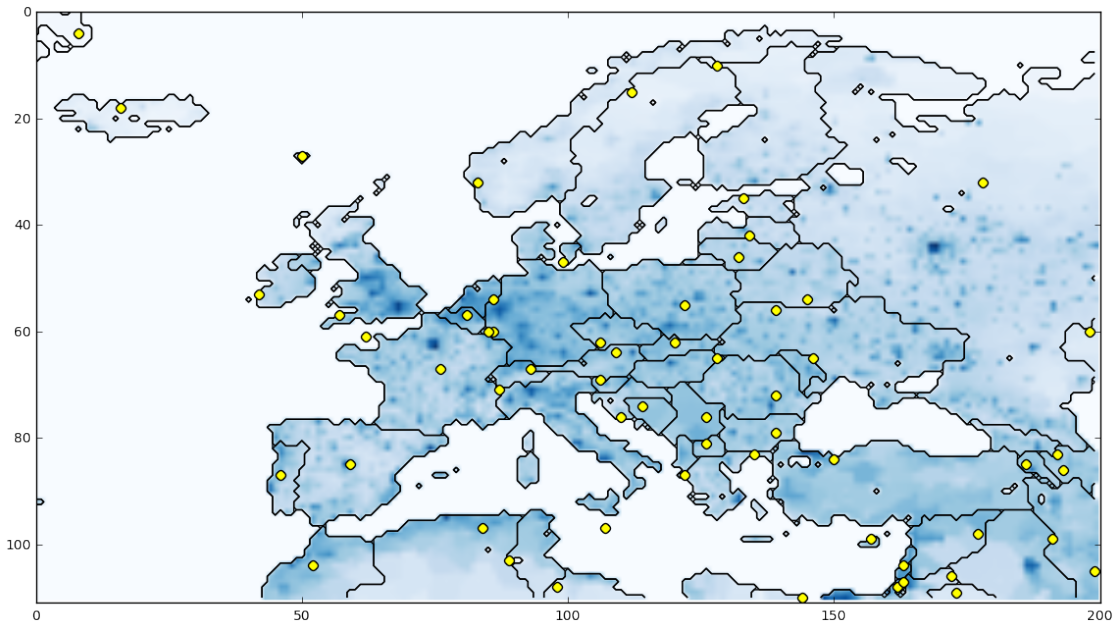
In [144]:
```
def initialiseNational(coords, ctr_f, pop_f, K):
    # adapt initialization: first sample one centroid per country (60 countries)
    # make list of unique countries
    first_draws = list()
    ctr_uniq = np.unique(ctr_f)[1:]
    for ctr in ctr_uniq:
        # for some country, get all points that belong to this country
        pool = [i for i, x in enumerate(ctr_f) if x == ctr]
        # now draw one index
        draw = np.random.choice(pool, size=1).item(0)
        # get the coordinate pair that belongs to this index
        #first_draws.append(coords[draw])
        first_draws.append(draw)
    res1 = np.array(first_draws)

    # find normalising constant Z
    weight = 1/pop_f.sum() * pop_f
    # make vector of indices
    ix = np.arange(len(pop_f))

    # draw K-nCtr indices (here: 200-60)
    res2 = np.random.choice(ix, size = K-len(res1), replace = True, p=weight)

    # append first and second results
    res = np.append(res1,res2)

    # find resulting coordinate pairs [x,y]
    centr_init = coords[res]

    # draw 2D-array containing Gaussian noise
    mu, sigma = 0, 0.01
    noise = np.random.normal(mu, sigma, [K-len(res1),2])

    # add noise to the additional coordinates only (not the ctr-specific ones!)
    centr_init[len(res1):] = centr_init[len(res1):] + noise
    return(centr_init)
```

In [145]:
```
# let's check if drawing one obs per ctr is working (looks good!):
test = initialiseNational(coords, ctr_f, pop_f, K = 200)
# once we're done, re-arrange and plot
```

```
        test_xs = list(np.array(test)[:len(ctr_uniq),0])
        test_ys = list(np.array(test)[:len(ctr_uniq),1])
        print(len(test_xs)) # correct (should be 60, 1 for each ctr!)
        # plot
        uplot(test_xs,test_ys)
```

```
60
```

```
C:\Users\johndoe\Anaconda3\lib\site-packages\matplotlib\collections.py:650: FutureWarning: elementwise c
  if self._edgecolors_original != str('face'):
C:\Users\johndoe\Anaconda3\lib\site-packages\matplotlib\collections.py:590: FutureWarning: elementwise c
  if self._edgecolors == str('face'):
```



In [177]:
```
# find the nationality of a centroid
def findCountry(centroids, coords, ctr_f):
    centroids_ctr = list()
    for centroid in centroids:
        # round the given centroid:
        ctr_r = [round(i) for i in centroid]

        # find the nationality of the corresponding coordinate
        coords_x = list(np.array(coords)[:,0])
        coords_y = list(np.array(coords)[:,1])

        sol_x = [i for i, x in enumerate(coords_x) if x == ctr_r[0]]
        sol_y = [i for i, x in enumerate(coords_y) if x == ctr_r[1]]

        sol = list(set(sol_x) & set(sol_y))[0]

        centroids_ctr.append(ctr_f[sol])
    return(np.array(centroids_ctr))
```

```
    # adapted close
    def closeNational(x, centroids, index):
        dist = 10**10
        for i in index:
            temp = centroids[i]
            res = sed(x, temp)
            if res < dist:
                dist = res
                cent = temp
                it = i
        return(dist, cent, it)

    # adapted E-step
    def estepNational(coords, pop_f, centroids, J_old):
        Dist = []
        Cent = []
        It = []
        centroids_ctr = findCountry(centroids, coords, ctr_f)
        for i in range(len(coords)):
            x = coords[i]
            ctr_x = ctr_f[i]
            # 'index' contains the indices of centroids_ctr where ctr is the same as coord.
            index = np.where(centroids_ctr == ctr_x)[0]
            dist, cent, it = closeNational(x,centroids, index)
            Dist.append(dist)
            Cent.append(cent)
            It.append(it)
        # calculate J (flattened weights are in 'pop_f')
        J = (pop_f * Dist).sum() / pop_f.sum()
        print("J =", J)
        J_old = J
        return(Dist, Cent, It, J_old)
```

In [178]:
```
# EM until threshold is reached
    # initialize
go = True
J_old = 10**4
centroids_nat = initialiseNational(coords, ctr_f, pop_f, K = 200)

K = len(centroids_nat)
    # run
while go:
    Dist, Cent, It, J = estepNational(coords, pop_f, centroids_nat, J_old)
    centroids_nat = mstep(coords, pop_f, It, K)
    go = J_old-J > 0.01
    J_old = J
print("Converged.")
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-178-6b1975739ee3> in <module>()
```

```
      8     # run
      9 while go:
---> 10     Dist, Cent, It, J = estepNational(coords, pop_f, centroids_nat, J_old)
     11     centroids_nat = mstep(coords, pop_f, It, K)
     12     go = J_old-J > 0.01


    <ipython-input-177-836311af996b> in estepNational(coords, pop_f, centroids, J_old)
     37     Cent = []
     38     It = []
---> 39     centroids_ctr = findCountry(centroids, coords, ctr_f)
     40     for i in range(len(coords)):
     41         x = coords[i]


    <ipython-input-177-836311af996b> in findCountry(centroids, coords, ctr_f)
     14
     15         sol = list(set(sol_x) & set(sol_y))[0]
---> 16         if len(sol) == 0:
     17             print("Hey")
     18


    TypeError: object of type 'int' has no len()
```