Update centroid. We define a function to update the centroid after the group assignment, returning a new list of group centroids.

```
In [ ]: def update_centroid(data, grouping, centroids):
            new_centroids = [];
            for i in range(len(centroids)):
                cent = np.zeros(len(data[0]))
                count = 0
                for j in range(len(data)):
                    if grouping[j] == (i+1):
                        cent = cent+data[j]
                        count += 1
                group_average = cent/count
                new_centroids.append(group_average)
            return new_centroids
```

Clustering objective. Given the group assignment and the centroids with the data, we can compute the clustering objective as the square of the RMS value of the vector of distances.

```
In [ ]: def clustering_objective(data, grouping, centroids):
            J_obj = 0
            for i in range(len(data)):
                for j in range(len(centroids)):
                    if grouping[i] == (j+1):
                        J_obj += np.linalg.norm(data[i] - centroids[j])**2
            J_obj = J_obj/len(data)
            return J_obj
```

4.3. The k-means algorithm

We can define another function `Kmeans_alg` that uses the three functions defined in the above subsection iteratively.

```
In [ ]: def Kmeans_alg(data, centroids):
            iteration = 0
            J_obj_vector = []
            Stop = False
            while Stop == False:
                grouping = group_assignment(data, centroids)
```

```
        new_centroids = update_centroid(data, grouping, centroids)
        J_obj = clustering_objective(data, grouping,
        ↪   new_centroids)
        J_obj_vector.append(J_obj)
        iteration += 1
        if np.linalg.norm(np.array(new_centroids) -
        ↪   np.array(centroids)) < 1e-6:
            Stop = True
        else:
            centroids = new_centroids
    return new_centroids, grouping, J_obj_vector, iteration
```

Convergence. Here we use a `while` loop, which executes the statements inside the loop as long as the condition `Stop == False` is true. We terminate the algorithm when the improvement in the clustering objective becomes very small (`1e-6`).

Alternatively, we can use the `Kmeans` function in the `cluster` module of the `sklearn` package.

```
In [ ]: from sklearn.cluster import KMeans
        import numpy as np
        kmeans = KMeans(n_clusters=4, random_state=0).fit(data)
        labels = kmeans.labels_
        group_representative = kmeans.cluster_centers_
        J_clust = kmeans.inertia_
```

Here we try to apply the k-means algorithm on `data`, clustering the vectors into 4 groups. Note that the `sklearn.cluster.KMeans` function initialize the algorithms with random centroids and thus the initial values of centroids are not required as an argument but the random state to draw the random initialization is.

4.4. Examples

We apply the algorithm on a randomly generated set of $N = 300$ points, shown in Figure 4.1. These points were generated as follows.

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
```