Data Mining
TP3 - Classification Hiérarchique Ascendante

Rémi MUSSARD - Thomas ROBERT

1 Calcul de distance

```
function M = distance(X, param)
% X : R ^ N x d
% N : Nb points
% M : R ^ N x N distance entre les points

    n = size(X,1);

    if (param == 'euclid')
        %M = zeros(n,n);

        %for i = 1:n
        %    for j = i+1:n
        %        dist = norm(X(i,:) - X(j,:));
        %        M(i,j) = dist;
        %        M(j,i) = dist;
        %    end
        %end

        produits = X*X'; % produits scalaires
        normes = diag(produits); % normes des vecteurs sur la diag
        M = normes*ones(1,n) + ones(n,1)*normes' - 2* produits;
    end
end
```

2 Fonctionnement d'aggclust

aggclust crée une hiérarchie ascendante des clusters. Il initialise le niveau 1 en créant 1 cluster par point. On boucle ensuite de 2 au nombre de points et en rassemblant à chaque fois les deux clusters les plus proches.

3 Fonction calc_dendro

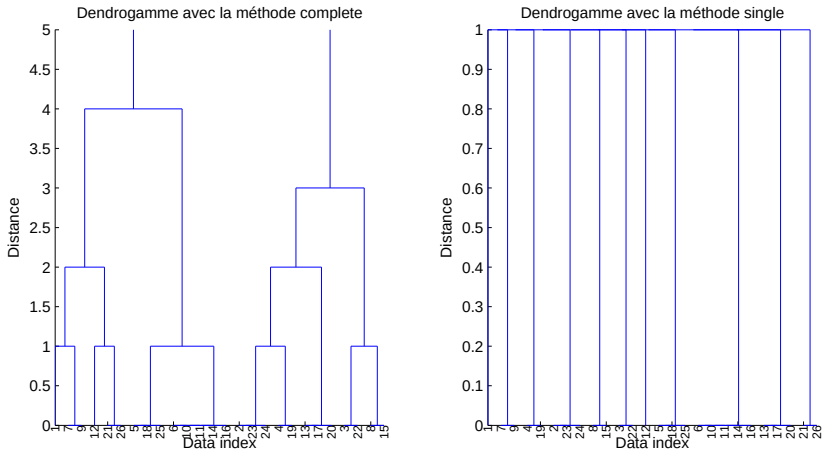
Nous avons écrit une fonction `calc_dendro` qui se charge de calculer et d'afficher les deux dendrogrammes différents (un pour chaque méthode) à partir de données.

```
function [M, level, level_single] = calc_dendro(data, show)

M = distance(data, 'euclid');
M = M + diag(inf*ones(1,size(M)));

level = aggclust(M, 'complete');
level_single = aggclust(M, 'single');

if (show)
```

```
    subplot(1,2,1);
    dendro(level);
    title('Dendrogamme avec la méthode complete');

    subplot(1,2,2);
    dendro(level_single);
    title('Dendrogamme avec la méthode single');
end
```

4 Classification ASI4

```
load('asi4.mat');
fig = figure();
calc_dendro(data, true);
set(fig, 'Position', [100 100 850 420]);
```



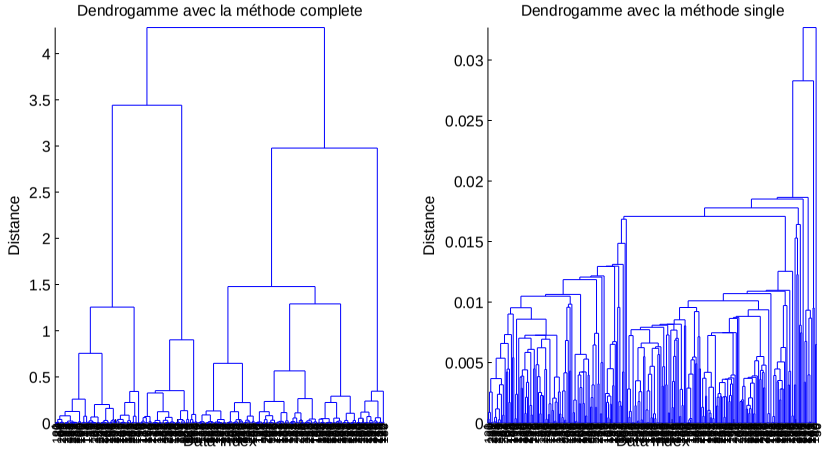
On voit que la méthode single n'arrive pas du tout à trouver de groupes alors que la méthode complete forme bien des groupes qui semblent (au vu du nombre d'étudiants par groupe) relativement bien répartis.

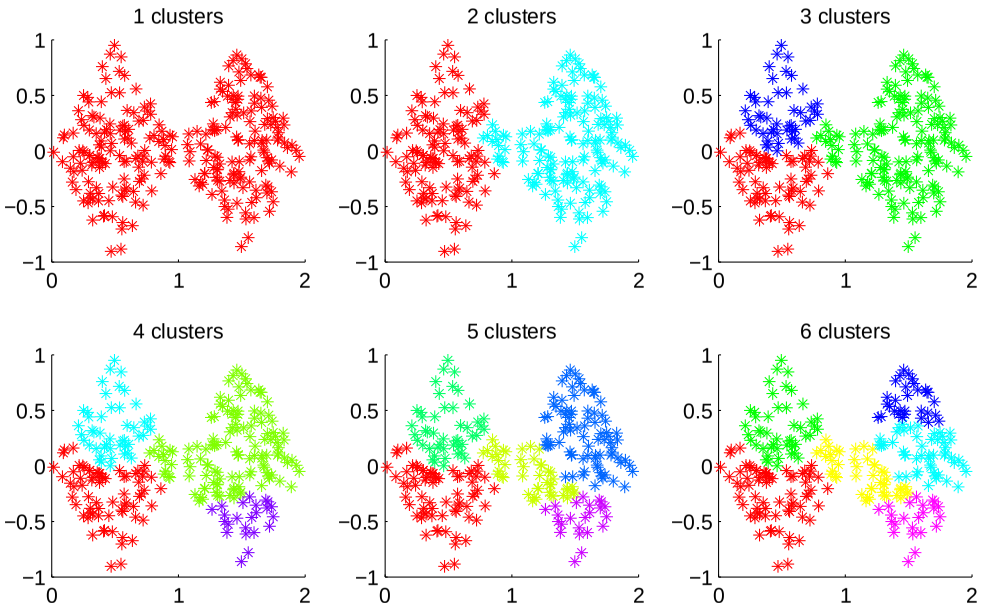
5 DS2

Afin de pouvoir visualiser graphiquement les clusters, on écrit une fonction `show_clusters(data, level, nbClust)` permettant d'afficher `nbClust` clusters.

```
function show_clusters(data, level, nbClust)

colors = hsv(nbClust);

for i = 1:nbClust

    inds = level(end - nbClust + 1).cluster{i};
    dataClust = data(inds, :);
    hold on;
    plot(dataClust(:,1), dataClust(:,2), '*', 'color', colors(i, :));

end
```

```
load ds2.dat

data = mydownsampling(ds2, 30);

fig = figure();
[M, level, ~] = calc_dendro(data, true);
set(fig, 'Position', [100 100 850 420]);

% affichage

fig = figure();
for i=1:6
    subplot(2,3,i);
    show_clusters(data, level, i);
    title([int2str(i) ' clusters']);
end
set(fig, 'Position', [100 100 750 420]);
```



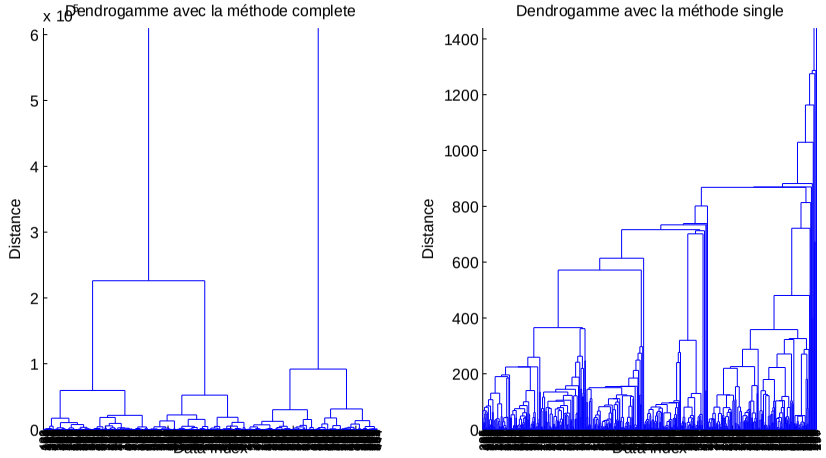


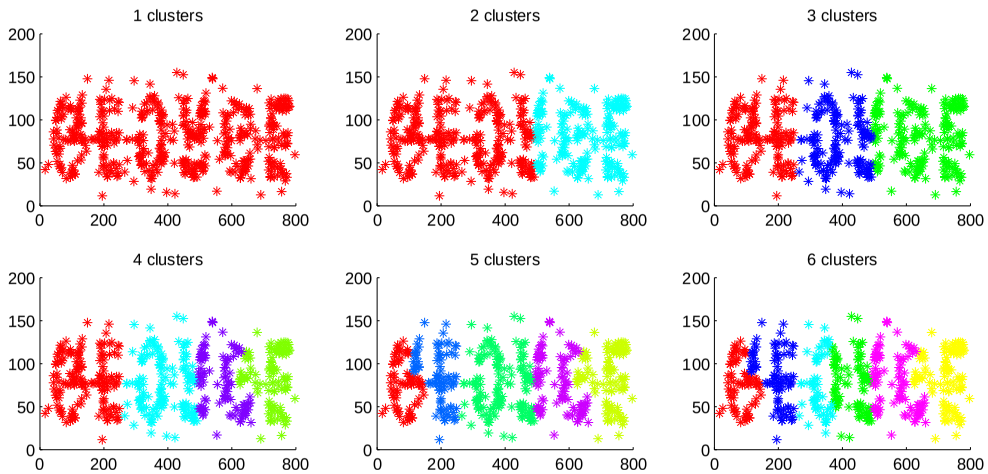
On constate que cette méthode ne donne pas toujours les deux clusters que l'on attendrait (un par losange) mais regroupe parfois les pointes (un cluster avec les pointes inférieures et un avec les pointes supérieures).

6 George

Avec les données "george", les clusters semblent plus proche de ce que l'on attends (1 cluster par lettre lorsque l'on choisi 6 clusters), et le résultat semble relativement stable vis à vis du sous-échantillonage, contraitement à ce que l'on observait avec les données "ds2".

```
load george.dat

data = mydownsampling(george, 15);

fig = figure();
[M, level, ~] = calc_dendro(data, true);
set(fig, 'Position', [100 100 850 420]);

% affichage

fig = figure();
for i=1:6
    subplot(2,3,i);
    show_clusters(data, level, i);
    title([int2str(i) ' clusters']);
end
set(fig, 'Position', [100 100 980 420]);
```





Data Mining
TP4 - Clustering K-means

Rémi MUSSARD - Thomas ROBERT

Voici les diverses fonctions codées

1 Calcul de distance

Cette fonction permet de calculer la distance entre chaque point des données de **X** par rapport à chaque centre **C** par la méthode `param`.

Elle retourne la matrice de distance **M**.

```
function M = distance(X, C, param)

    N = size(X,1);
    K = size(C,1);

    % M = zeros(N,K);
    %
    % for i = 1:N
    %   for k = 1:K
    %       if (param == 'euclid')
    %           M(i,k) = norm(X(i,:) - C(k,:));
    %       end
    %   end
    % end

    if (strcmp(param, 'euclid'))
        normX = sum(X.^2, 2);
        normC = sum(C.^2, 2);
        ps = X*C';
        M = repmat(normX, 1, K) + repmat(normC', N, 1) - 2*ps;
    end

    if (strcmp(param, 'mahal'))

        M = zeros(N,K);

        for k = 1:K
            X2 = X + repmat(mean(X),N,1) - repmat(C(k,:),N,1);
            M(:, k) = mahal(X2, X);
        end
    end
end
```

../distance.m

2 Affectation aux clusters

Cette fonction permet d'affecter chaque point au cluster dont le centre est le plus proche du point, grâce à la matrice de distance **M**.

Elle retourne les affectations sous forme d'une liste **liste** avec pour chaque point son numéro de cluster.

```
function liste = affectation(M)
    [~, liste] = min(M,[] , 2); % indices des minimums par ligne
end
```

../affectation.m

3 Coût

Cette fonction permet de calculer le critère d'inertie intra-cluster à partir de la matrice des distances **M** et des affectations **liste**.

Elle retourne les coût **Jw**.

```
function Jw = cout(M, liste)

    [N, K] = size(M);

    Jw = 0;

    % pour chaque cluster
    for k = 1:K
        % distance intra-cluster du cluster k
        Jw = Jw + sum(M(liste == k, k));
    end

    % Jw = 0;
    % for i = 1:N
    %     Jw = Jw + M(i, liste(i));
    % end
end
```

../cout.m

4 Affectation et coût

Cette fonction combine les deux précédentes en une seule plus efficace.

```
function [Jw, liste] = affectation_cout(M)
    [couts, liste] = min(M,[] , 2); % indices des minimums par ligne
    Jw = sum(couts);
end
```

../affectation_cout.m

5 Calcul des centres

Calcule les coordonnées des centres des clusters de **X** définis par la liste d'affectation **liste**.

Retourne les positions des centres **C**.

```
function C = nouveaux_centres(X, liste)
    K = max(liste);
    d = size(X, 2);

    C = zeros(K, d);

    % pour chaque cluster
```

```
    for k = 1:K
        % moyenne des points dans le cluster
        C(k, :) = mean(X(liste == k, :));
    end
end
```

../nouveaux_centres.m

6 Initialisation des centres

Calcule des coordonnées de **K** centres aléatoires dont les coordonnées sont inclues dans le nuage de points **X** (encadrées par les min et max de chaque variable).

Retourne les positions des centres initiaux **C**.

```
function C = init_centres(X, K)

    d = size(X, 2);

    % points centrés entre 0 et 1
    C = rand(K, d) - repmat(0.5, K, d);

    % etendue des données
    etendue = max(X) - min(X);
    centre = (max(X) + min(X)) / 2;

    % multiplication par l'étendue
    C = C .* repmat(etendue, K, 1);

    % centrage de C sur X
    C = C + repmat(centre, K, 1);
end
```

../init_centres.m

7 Calcul des clusters par la méthode des K-moyennes

Calcule les clusters de façon itérative jusqu'à stabilisation du critère de coût à un minimum local, sur les données **X**, à partir des centres initiaux **C0**, avec la méthode de calcul de distance **param**.

Retourne les centres finaux **C**, la liste des affectations finales **liste**, et l'évolution du critère de coût **Jw**.

```
function [C, liste, Jw] = k_moyennes(X, C0, param)
    C = C0;

    % init des couts
    JwPrec = 1;
    JwNew = 0;
    Jw = [];

    % tant que le cout varie de plus de 1e-3
    while abs(JwPrec - JwNew) > 1e-3
        JwPrec = JwNew;

        % calcul des distances
        M = distance(X, C, param);
        % affectation et couts
        [JwNew, liste] = affectation_cout(M);
        Jw = [Jw ; JwNew];
        % calcul des nouveaux centres
        C = nouveaux_centres(X, liste);
    end
end
```

../k_moyennes.m

8 Calcul des formes fortes à partir des affectations

Découpe les clusters calculés en formes fortes à partir des multiples affectations **listes**. Chaque colonne représente les affectations pour une itération.

Retourne les formes fortes sous forme de liste d'affectation **newlist**.

```
function newlist = formes_fortes_from_listes(listes)

    N = size(listes,1);

    [~, ~, listeformesfortes]=unique(listes,'rows') ;

    Nff = max(listeformesfortes);
    effectifs = zeros(Nff, 1);
    newlist = zeros(N,1);

    for c=1:Nff
        ind = find(listeformesfortes==c) ;
        effectifs(c)=length(ind);
        newlist(ind)=c;
    end

end
```

../formes_fortes_from_listes.m

9 Calcul des formes fortes

Calcule des formes fortes sur les données **X**, en appliquant **nbIte** fois la méthode k-moyennes avec **K** clusters et la fonctions de distance **param**.

Retourne les formes fortes sous forme de liste d'affectation **newlist**.

```
function newlist = formes_fortes(X, nbIte, param, K)

    N = size(X,1);

    listes = zeros(N,nbIte);

    for j = 1:nbIte
        C0 = init_centres(X, K);
        [~, liste, ~] = k_moyennes(X, C0, param);
        listes(:,j) = liste;
    end

    newlist = formes_fortes_from_listes(listes);

end
```

../formes_fortes.m

10 Test des méthodes de k-moyennes et formes fortes

Cette fonction permet de tester la méthode des k-moyennes et des formes fortes sur les données **X**, pour les nombres de clusters contenus dans le vecteur **Klist**, en réalisant **nbIte** à la fin desquelles est appliquée la méthode

des formes fortes. La méthode de calcul de distance **param** est utilisée.

Les résultats de clustering sont affichés sur la figure **figClusters**. Les variations de la fonction coût sont affichés dans la figure **figJw**.

```
function test_k_means(X, Klist, nbIte, param, figClusters, figJw)
    subI = length(Klist);
    subJ = nbIte + 1;

    % pour chaque k
    for i = 1:length(Klist)
        K = Klist(i);
        listes = zeros(size(X,1), nbIte);

        % pour chaque itération
        for j = 1:nbIte

            % on initialise les K centres
            C0 = init_centres(X, K);
            % on calcule nos clusters
            [C, liste, Jw] = k_moyennes(X, C0, param);
            listes(:,j) = liste;

            % affichage des clusters et des centres initiaux
            figure(figClusters);
            subplot(subI, subJ, (i - 1)*subJ + j);
            show_clusters(X, liste);
            title(['K = ' num2str(K) ' / Jw = ' num2str(Jw(end))]);
            hold on
            scatter(C(:,1),C(:,2), 50, 'k', 'd', 'fill');
            scatter(C0(:,1),C0(:,2), 50, 'k', 'd');

            % affichage de l'évolution de Jw
            figure(figJw);
            subplot(subI, subJ - 1, (i-1)*(subJ-1) + j);
            plot(Jw, '-*');
            title(['K = ' num2str(K)]);
        end

        % calcul des formes fortes à partir des intérations
        newlist = formes_fortes_from_listes(listes);

        % affichage formes fortes
        figure(figClusters);
        subplot(subI, subJ, i*subJ);
        show_clusters(X, newlist);
        title(['Formes fortes / Jw =' num2str(cout(X, newlist))]);
    end
end
```

../test_k_means.m

11 Lancement des tests

Ce script permet de lancer la procédure de test ci-dessus sur chaque jeu de données, avec les deux méthodes de calcul de distance.

```
% DM / TP4 / MUSSARD / ROBERT

clear all
close all

% Liste des méthodes a appliquer.
% Pour chaque méthode, liste des fichers de données ainsi que des
% paramètres de test.
methodes = struct(                                                  ...
    'methode',{'euclid'; 'mahal'},                                  ...
```

```
    'tests', {                                                          ...
        struct(                                                         ...
            'file', {'ds2.dat' ; 'ds3.dat' ; 'ds4.dat' ; 'ds5.dat' ; 'george.dat'}, ...
            'Klist',{2:7 ; 2:6 ; 4:9 ; 3:8 ; 2:6},                     ...
            'nbIte',{3 ; 3 ; 3 ; 3 ; 3});                               ...
        struct(                                                         ...
            'file', {'ds2.dat' ; 'ds3.dat' ; 'ds4.dat' ; 'ds5.dat' ; 'george.dat'}, ...
            'Klist',{2:7 ; 2:6 ; 4:9 ; 3:8 ; 2:6},                     ...
            'nbIte',{3 ; 3 ; 3 ; 3 ; 3})                                ...
        });

% pour chaque méthode (euclid, mahal, ...)
for i = 1:length(methodes)

    methode = methodes(i).methode;
    tests = methodes(i).tests;

    % pour chaque jeu de données
    for j = 1:length(tests)

        % init des données
        X = load(tests(j).file);
        Klist = tests(j).Klist;
        nbIte = tests(j).nbIte;

        % init des figures
        figClusters = figure();
        figJw = figure();

        % execution du test
        test_k_means(X, Klist, nbIte, methode, figClusters, figJw);

        % sauvegarde des figures
        set(figClusters, 'PaperUnits', 'points');
        set(figClusters, 'PaperPosition', [0 0 1000 1000]);
        saveas(figClusters, ['TP4_MUSSARD_ROBERT_' int2str(figClusters) '.png']);
        set(figJw, 'PaperUnits', 'points');
        set(figJw, 'PaperPosition', [0 0 500 600]);
        saveas(figJw, ['TP4_MUSSARD_ROBERT_' int2str(figJw) '.png']);

    end
end
```

../TP4_MUSSARD_ROBERT.m

12 Résultats

Ci-après sont présent les résultats des tests sur chaque figure, avec chaque méthode, avec plusieurs valeurs de K, plusieurs itérations suivies du calcul des formes fortes.

Des diamants indiquent les centres initiaux et finaux. Par ailleurs, un graphe permet de visualiser l'évolution de Jw.

On remarque que les clusters trouvés par cette méthode ne respectent pas les formes dessinées par les points, ce qui est logique puisque cette méthode cherche uniquement à créer des clusters de points les plus proches les uns des autres, alors que nos données comportent des groupes de points qui s'entremêlent et qui sont par ailleurs tous très proches les uns des autres. C'est pourquoi le résultat trouvé est souvent proche d'une simple partition de la "zone de dessin" en "parts" égales.

Concernant la distance de Mahalanobis, elle permet d'annuler la distorsion due à l'étalement des données. Mais pour certains jeu de données, l'étalement à une importance et une signification, et le fait de ne plus en tenir compte détériore le résultat. C'est par exemple le cas avec le jeu de données George où l'étalement des lettres n'est plus pris en compte et où les clusters sont plutôt en forme d'étoile autour du centre des données.

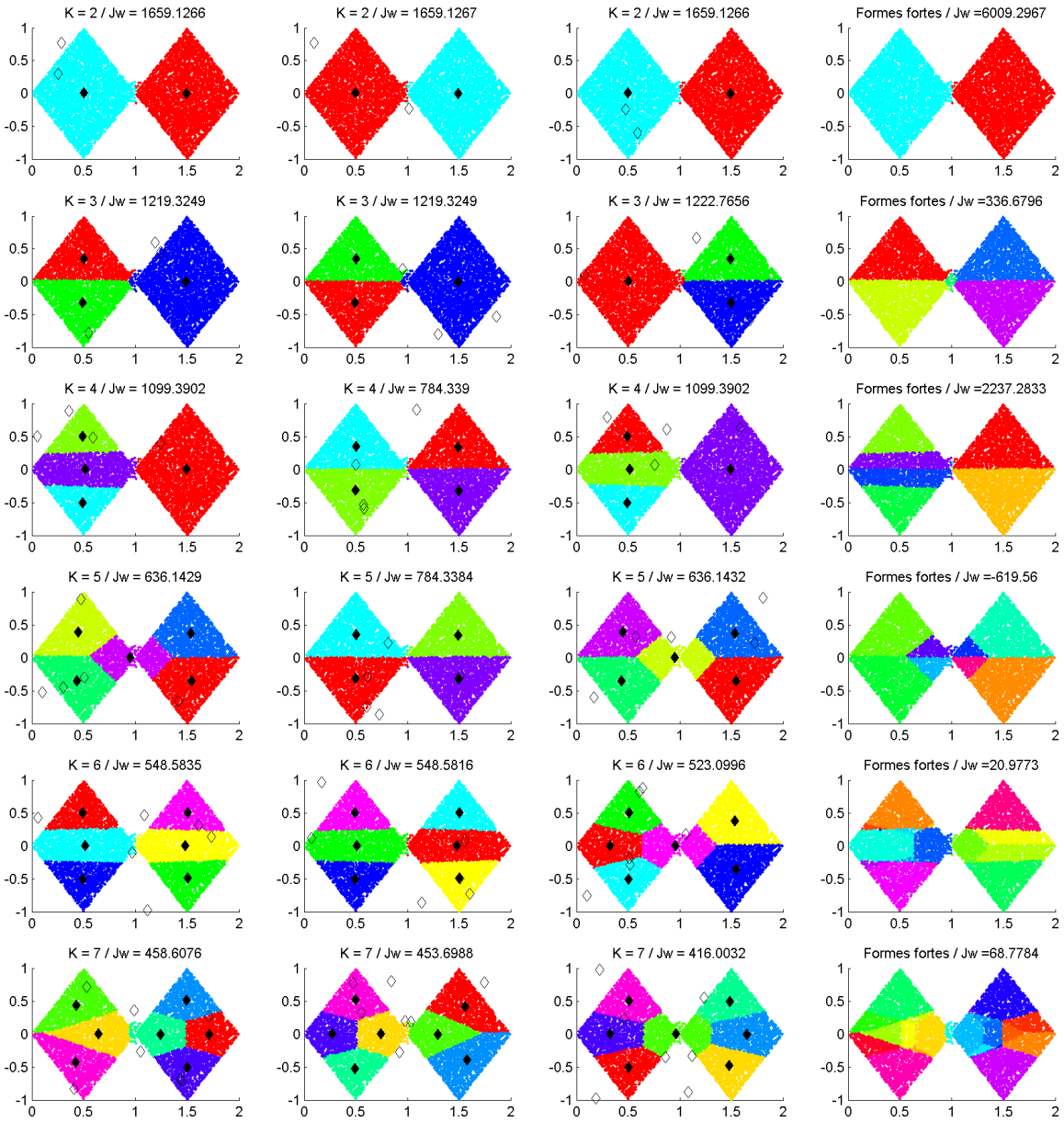


FIGURE 1 – ds2 - Distance euclidienne

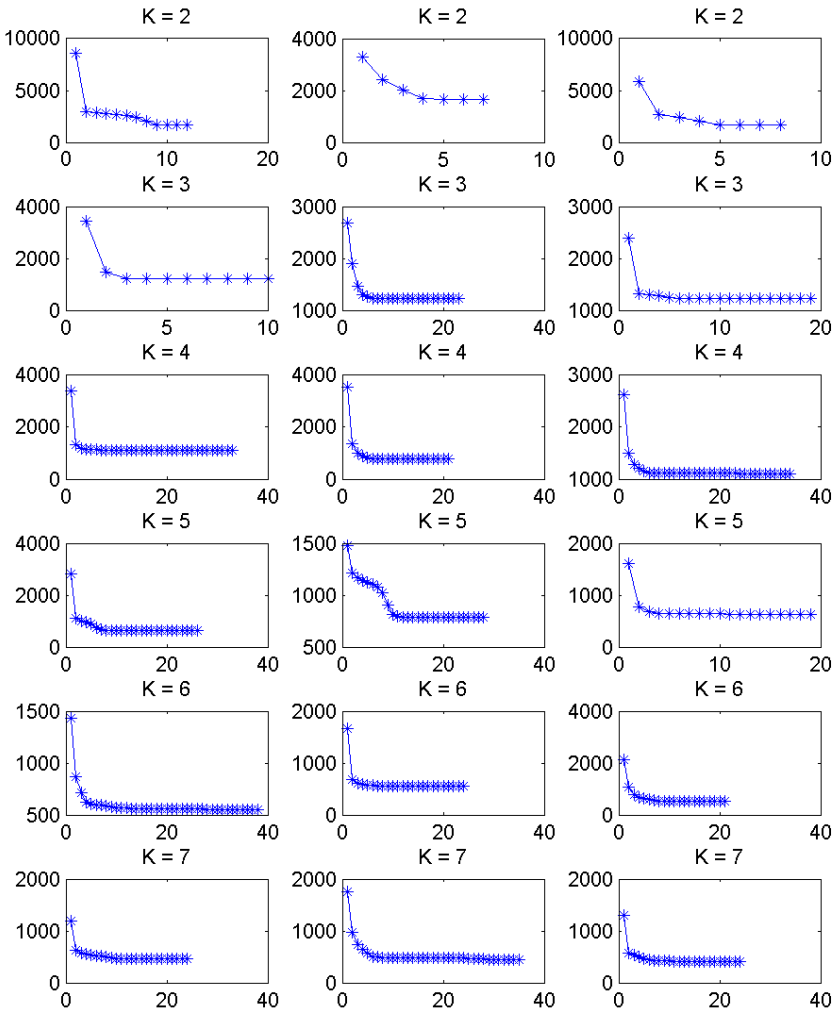


FIGURE 2 – DS2 - Distance euclidienne

FIGURE 3 – DS3 - Distance euclidienne

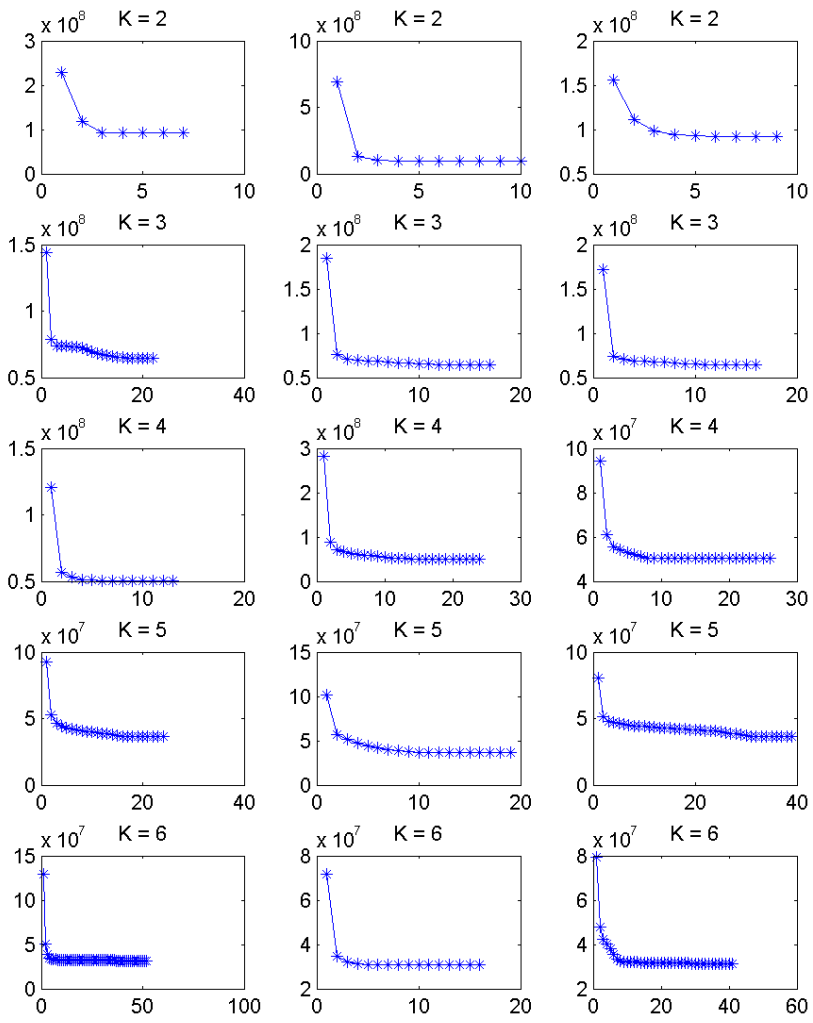


FIGURE 4 – DS3 - Distance euclidienne

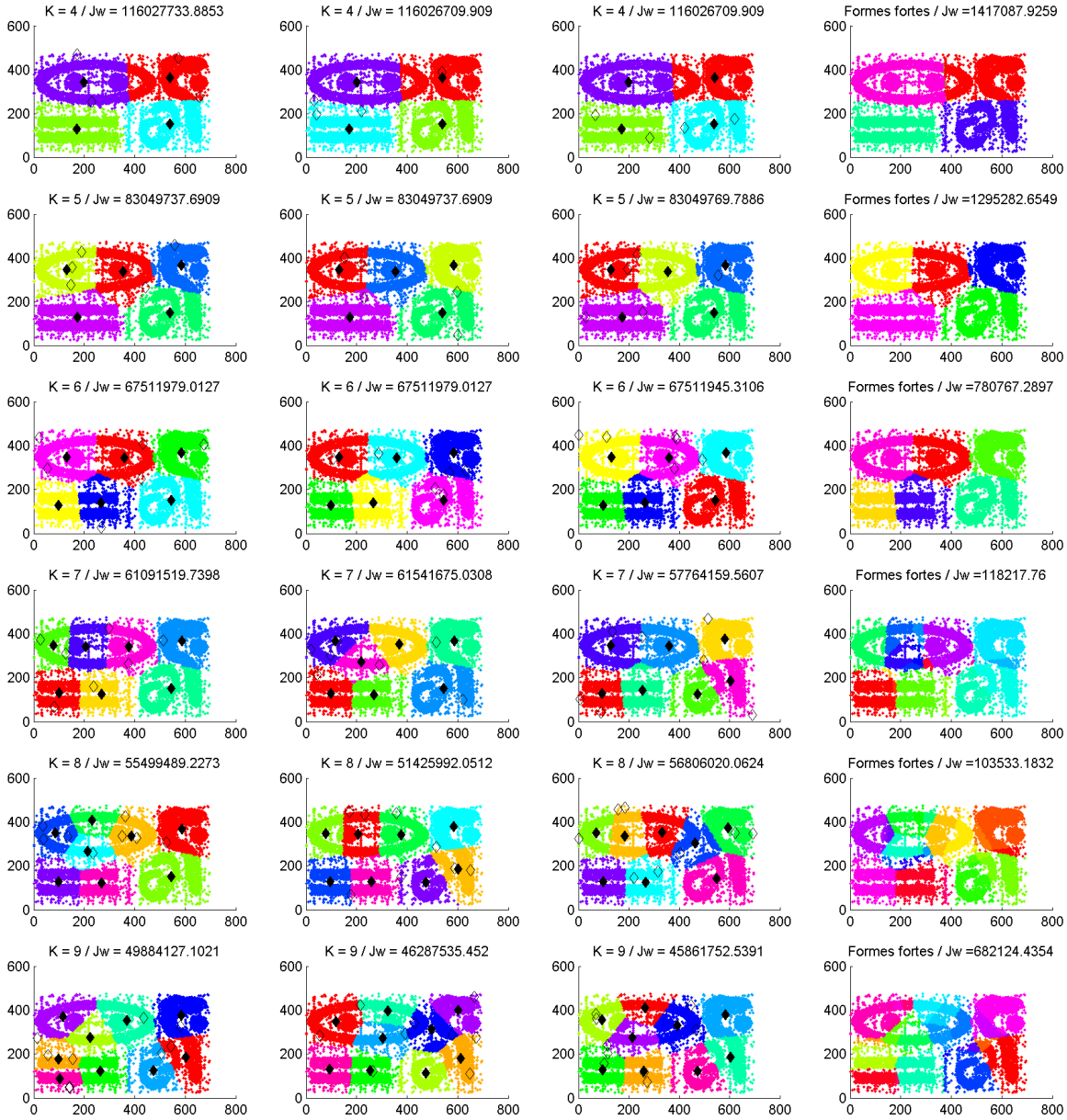


FIGURE 5 – DS4 - Distance euclidienne

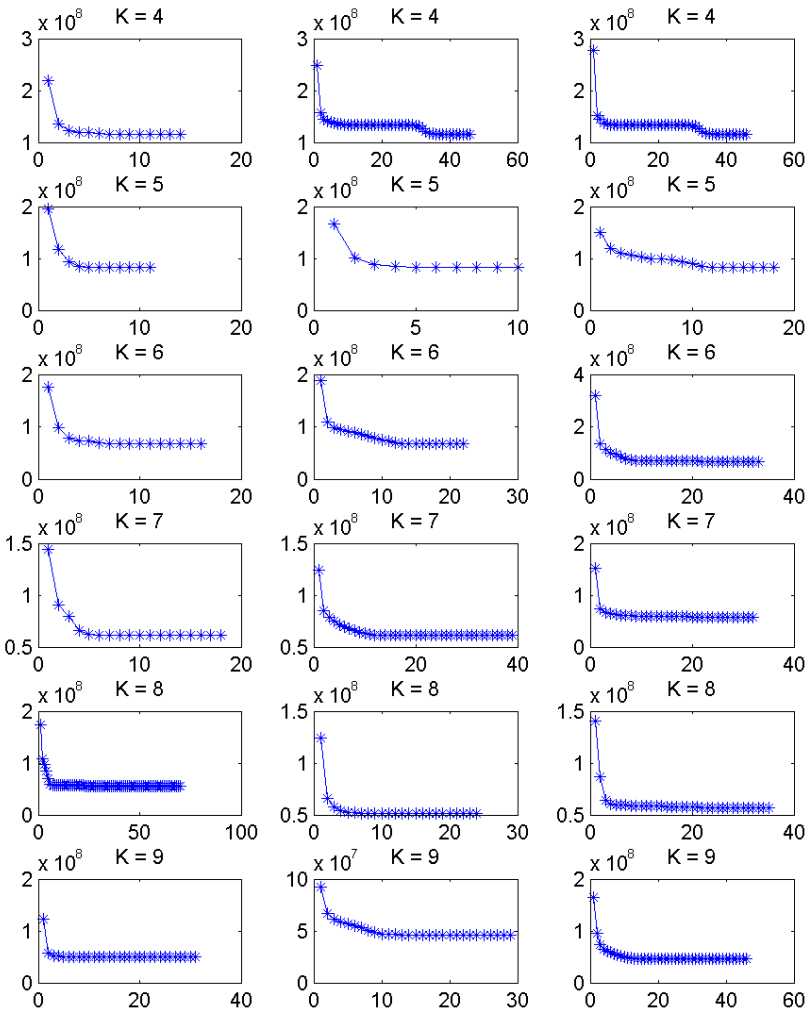


FIGURE 6 – DS4 - Distance euclidienne

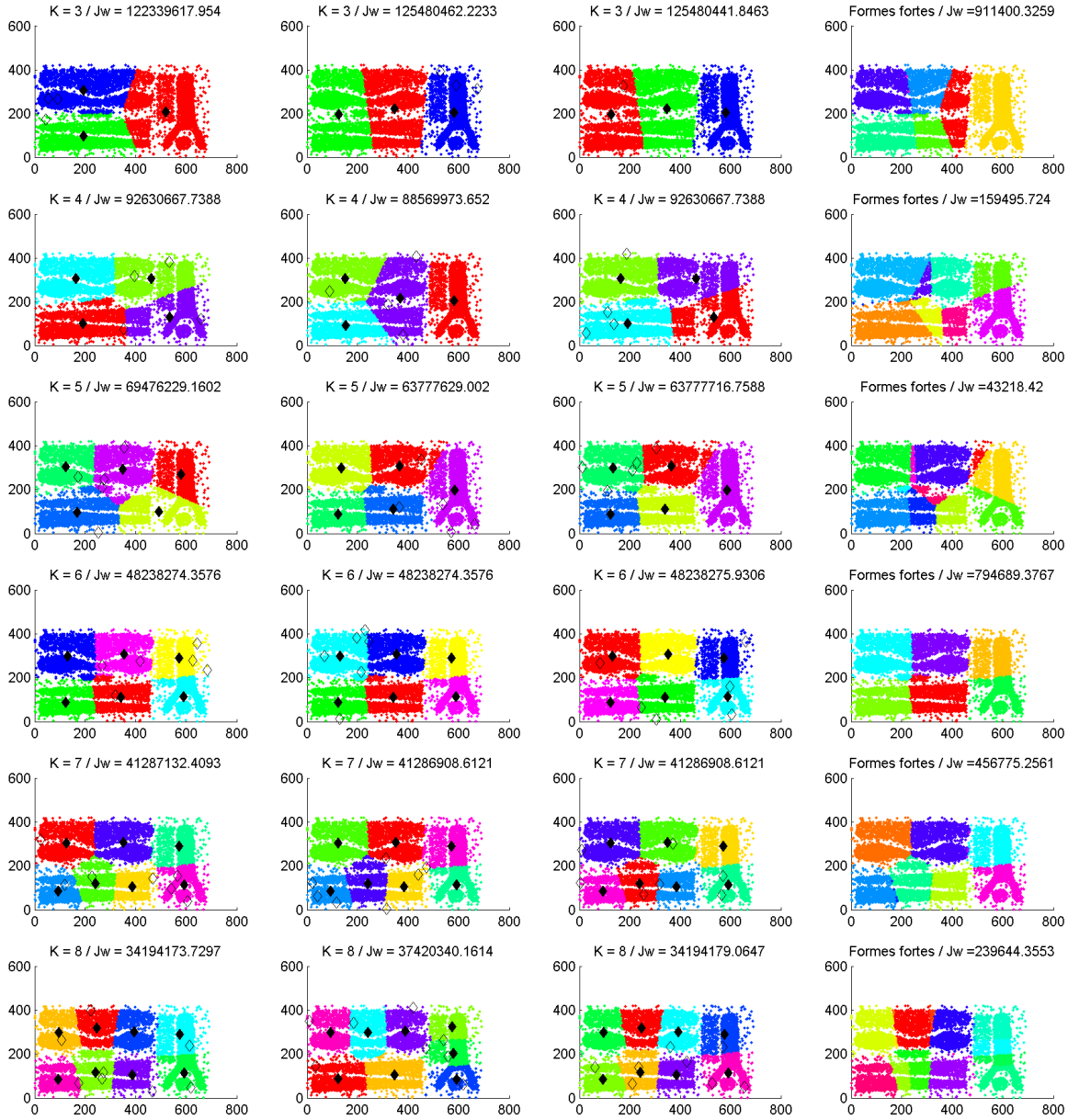


FIGURE 7 – DS5 - Distance euclidienne

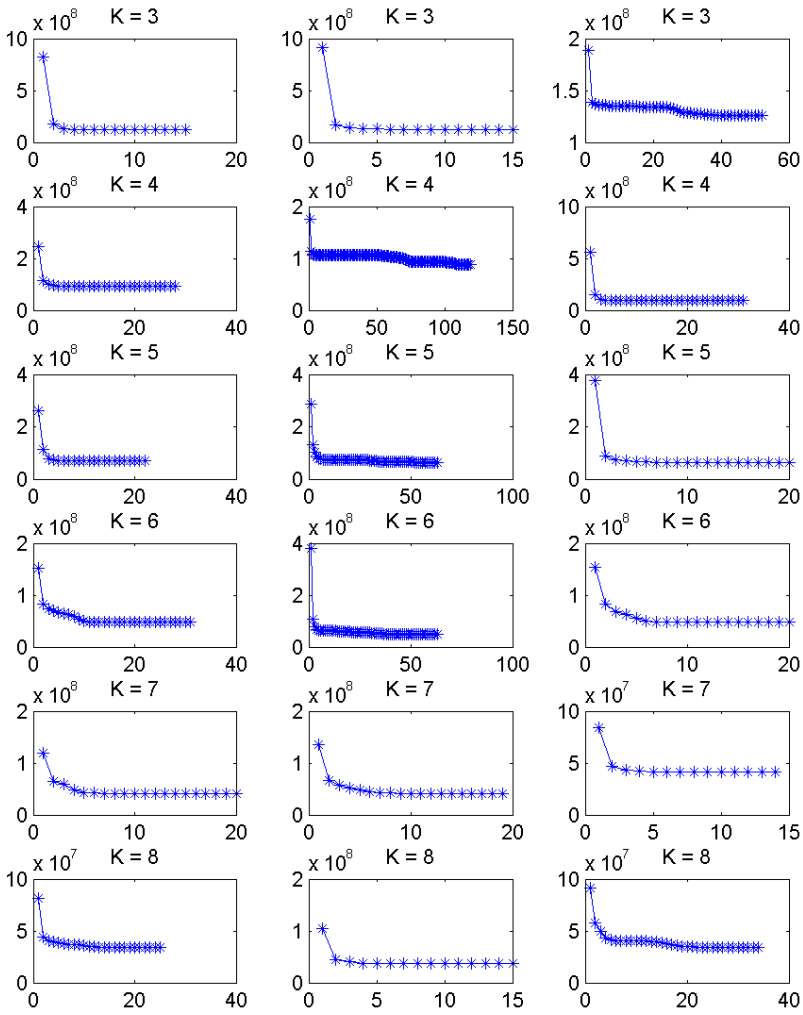


FIGURE 8 – DS5 - Distance euclidienne

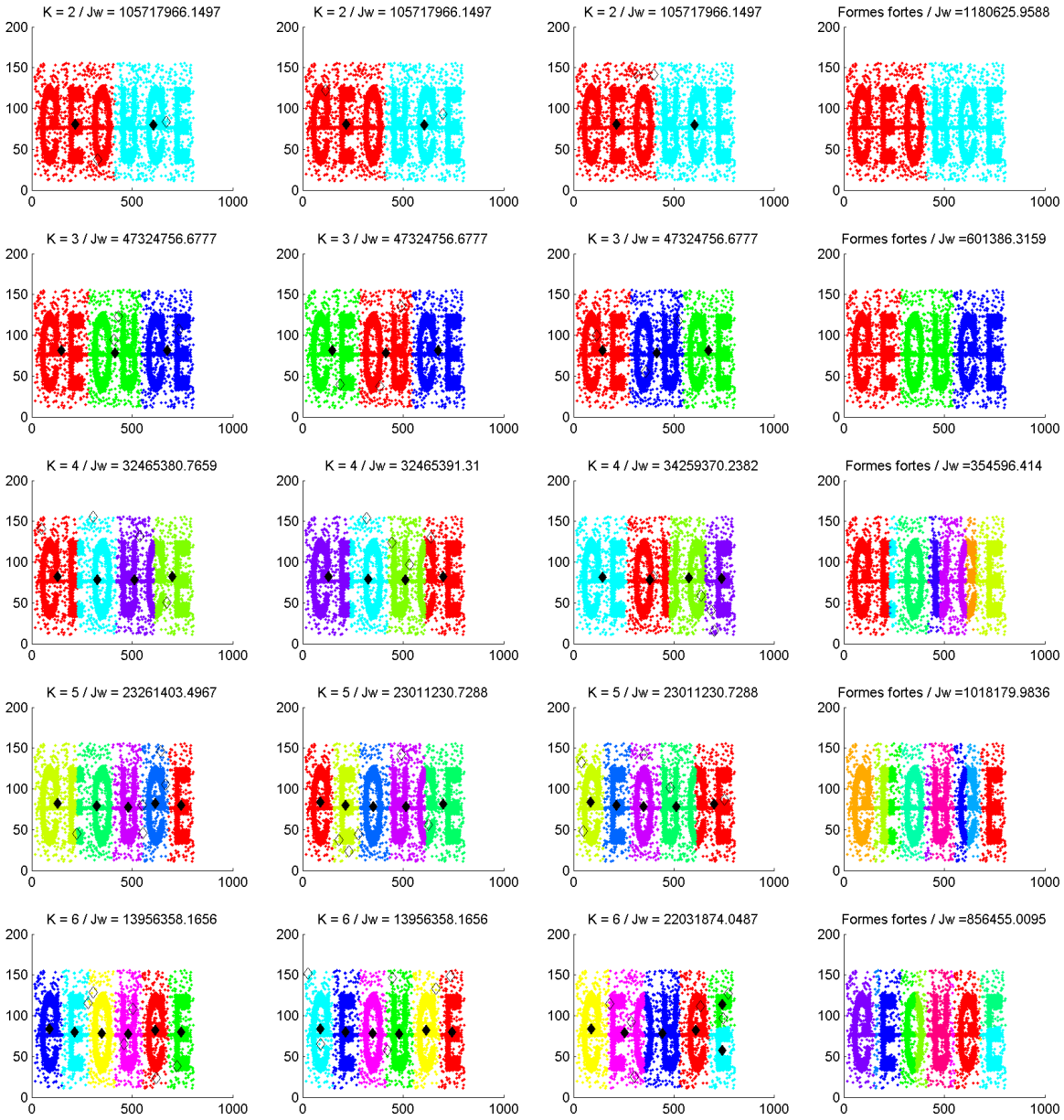


FIGURE 9 – George - Distance euclidienne

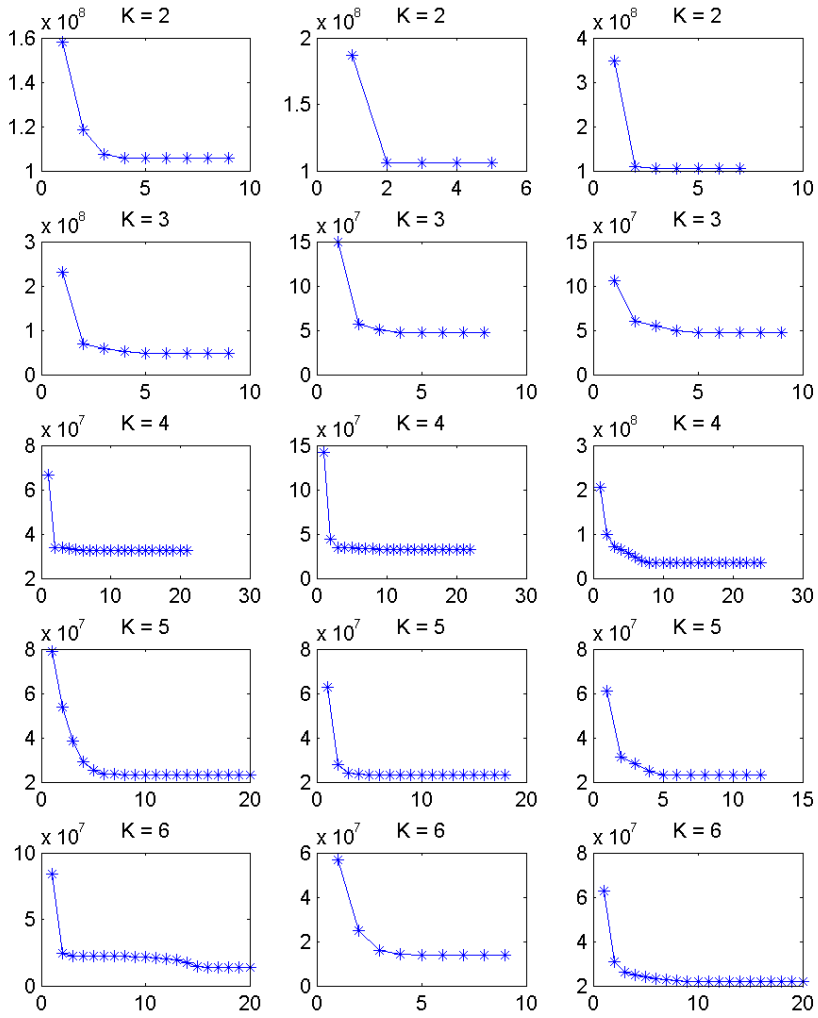


FIGURE 10 – George - Distance euclidienne

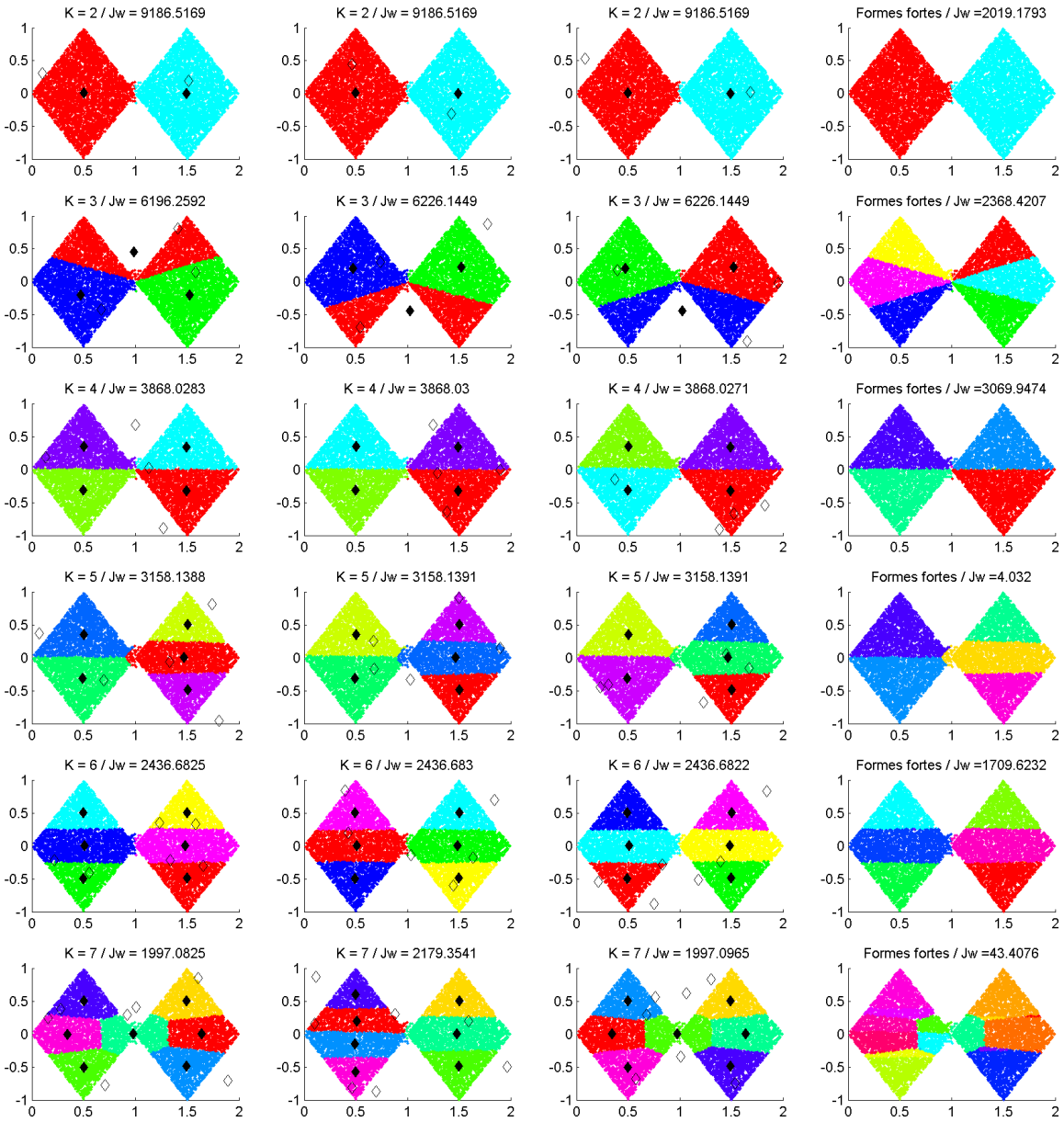


FIGURE 11 – DS2 - Distance de Mahalanobis

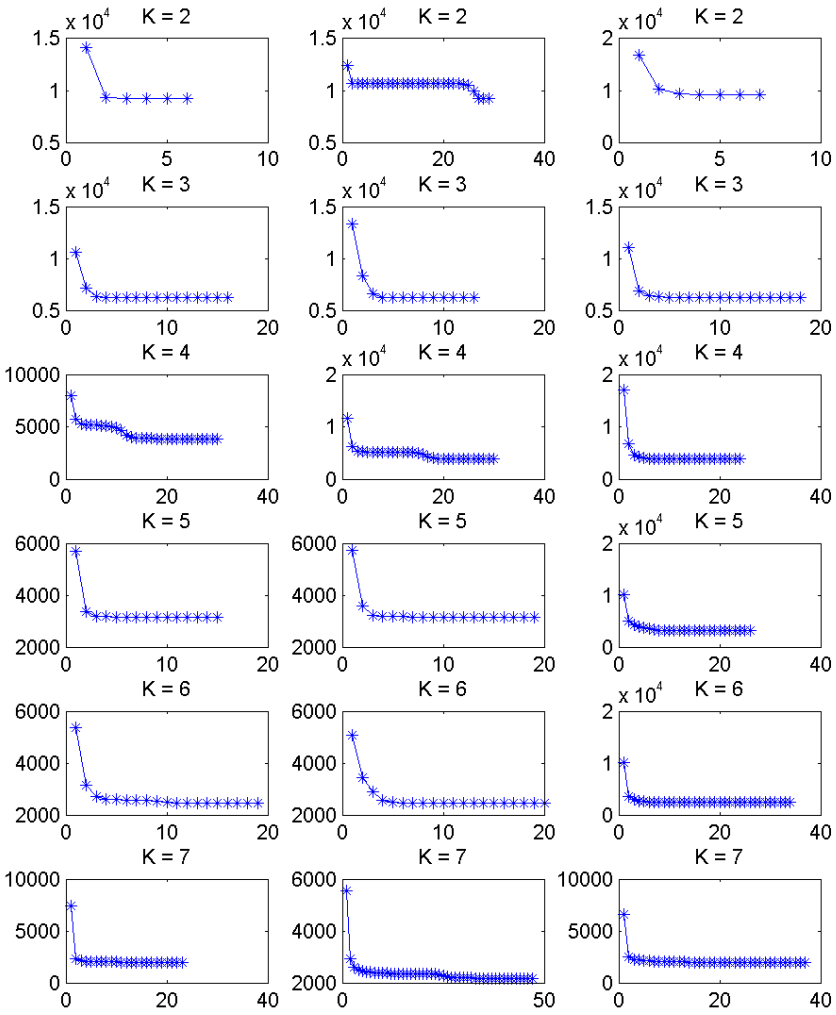


FIGURE 12 – DS2 - Distance de Mahalanobis

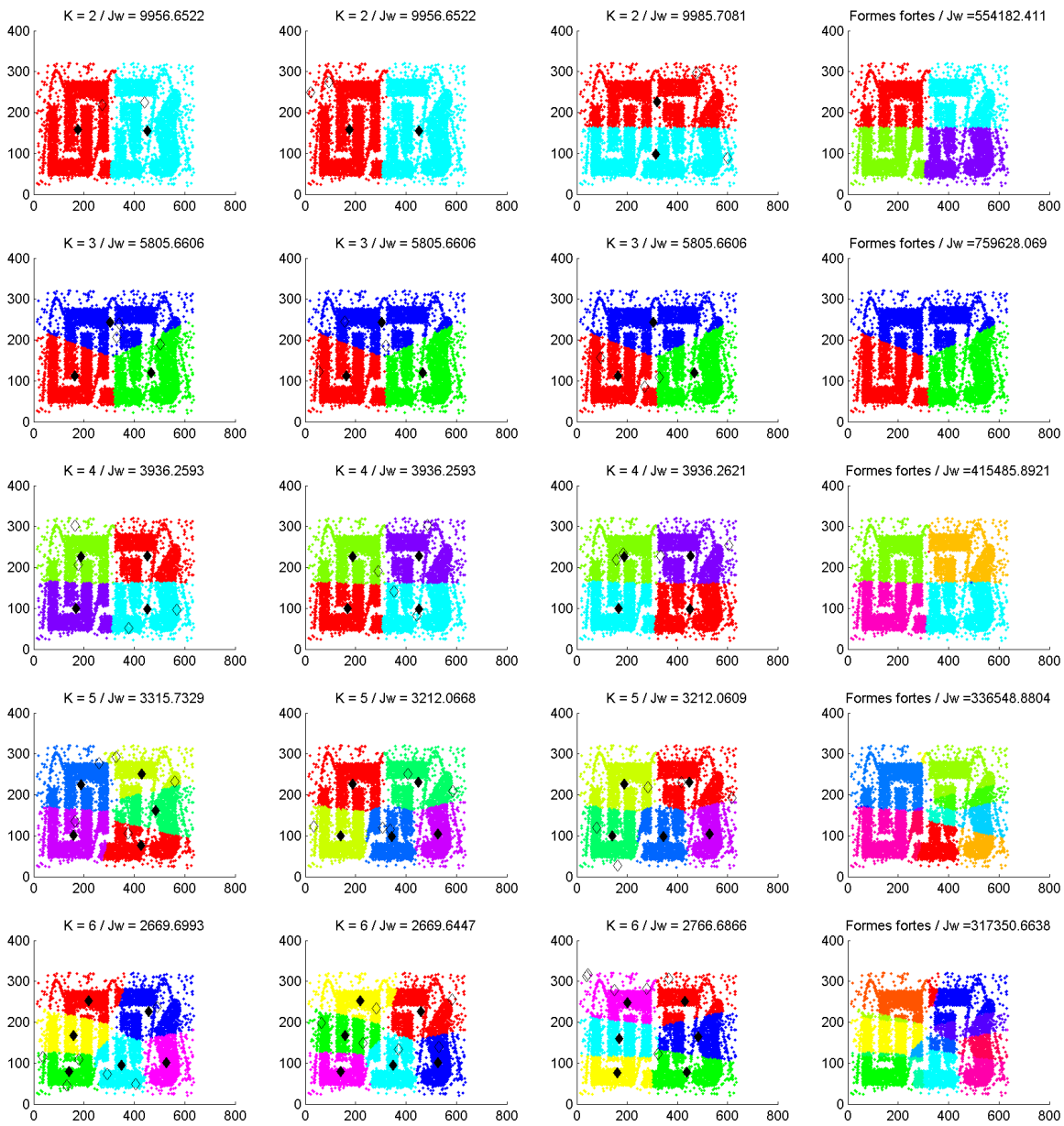


FIGURE 13 – DS3 - Distance de Mahalanobis

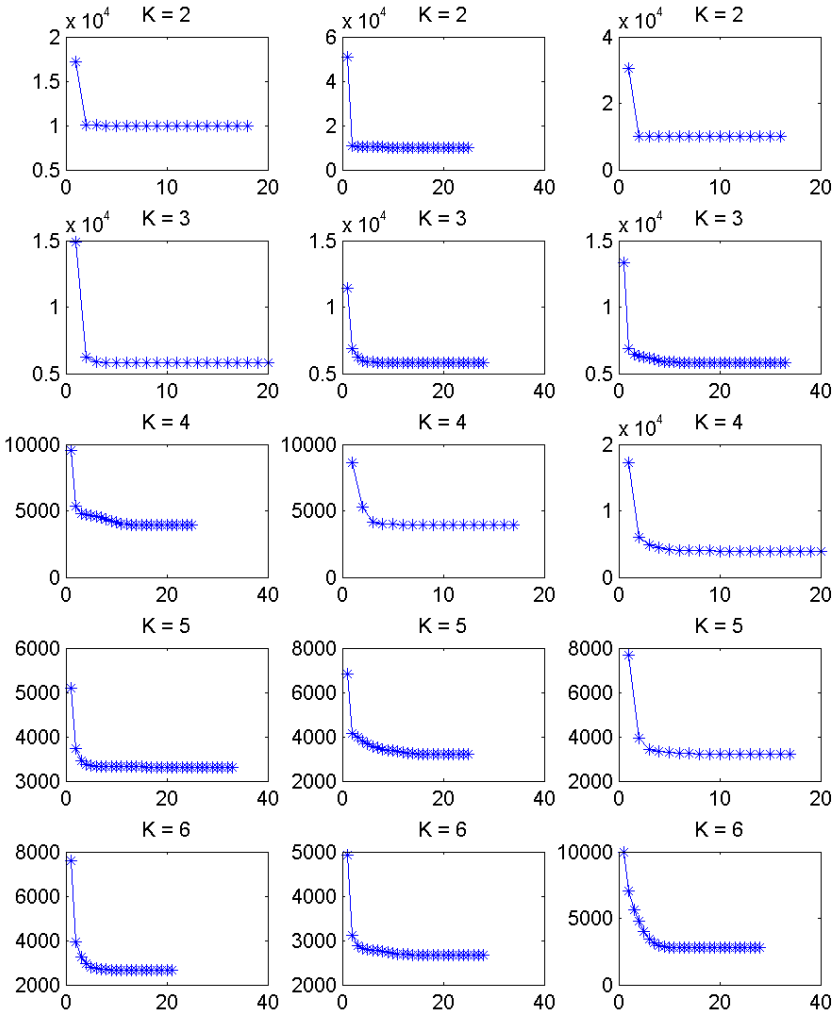


FIGURE 14 – DS3 - Distance de Mahalanobis

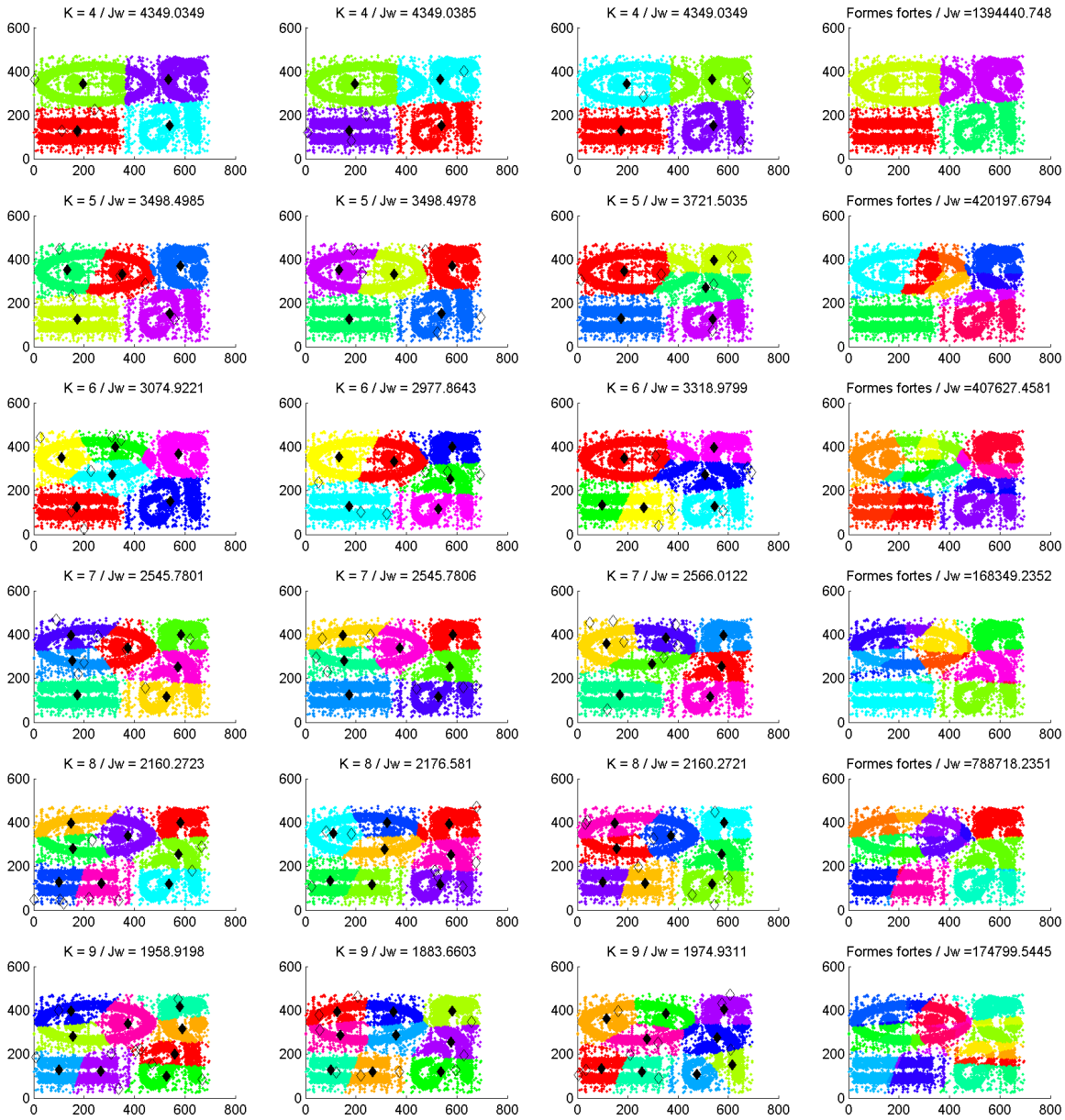


FIGURE 15 – DS4 - Distance de Mahalanobis

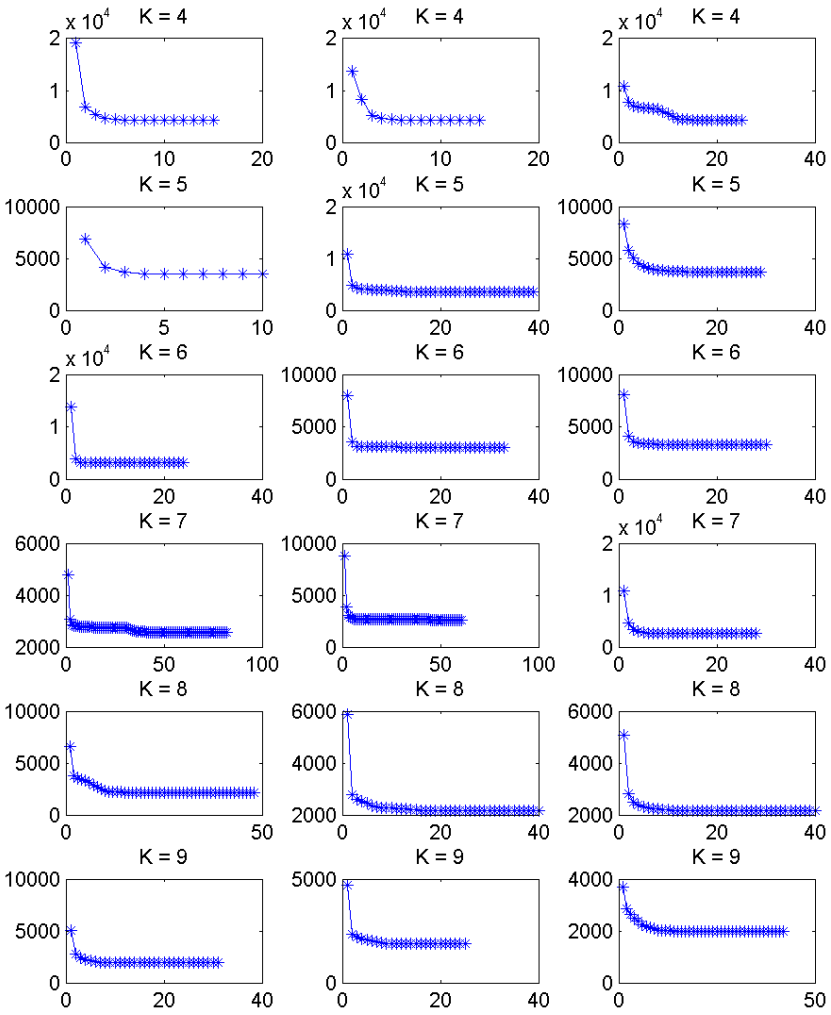


FIGURE 16 – DS4 - Distance de Mahalanobis

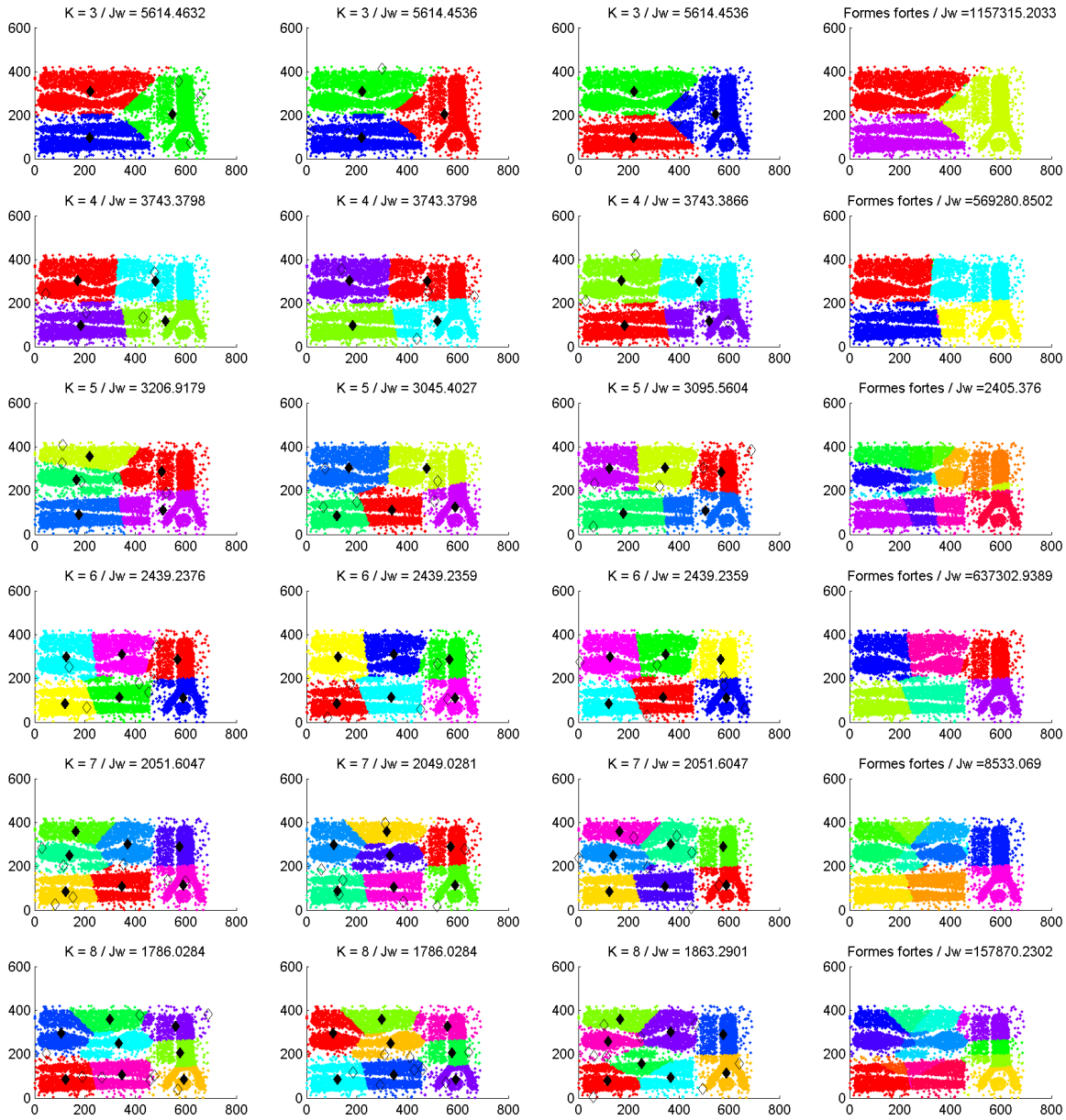


FIGURE 17 – DS5 - Distance de Mahalanobis

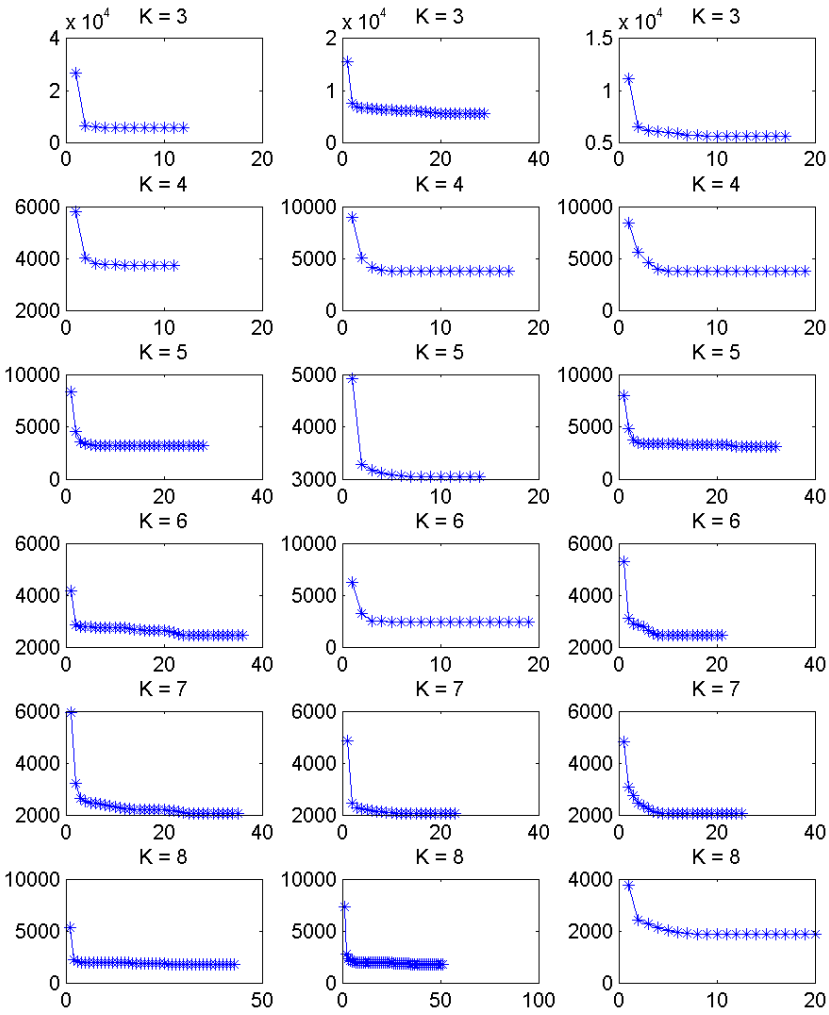


FIGURE 18 – DS5 - Distance de Mahalanobis

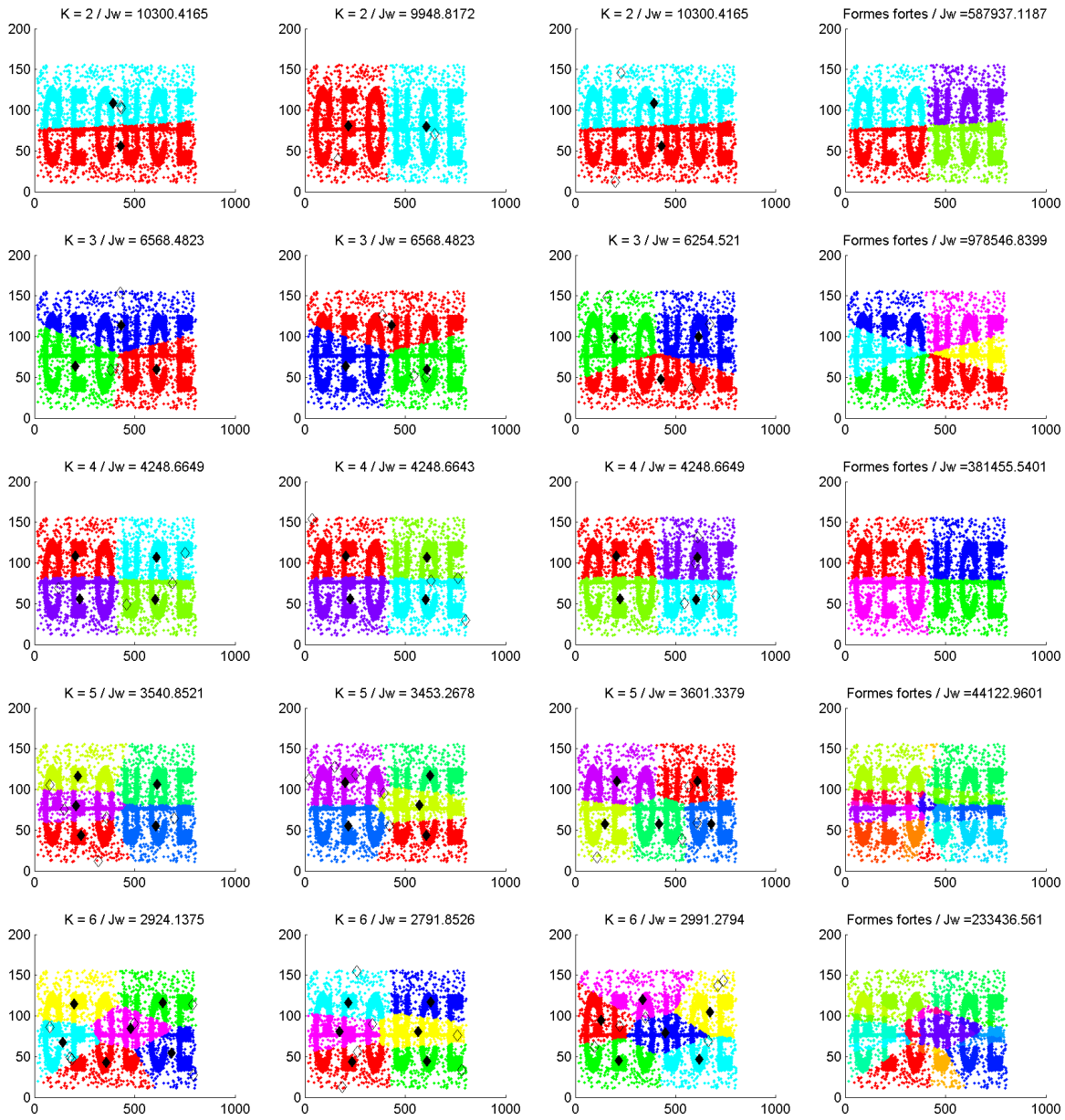


FIGURE 19 – George - Distance de Mahalanobis

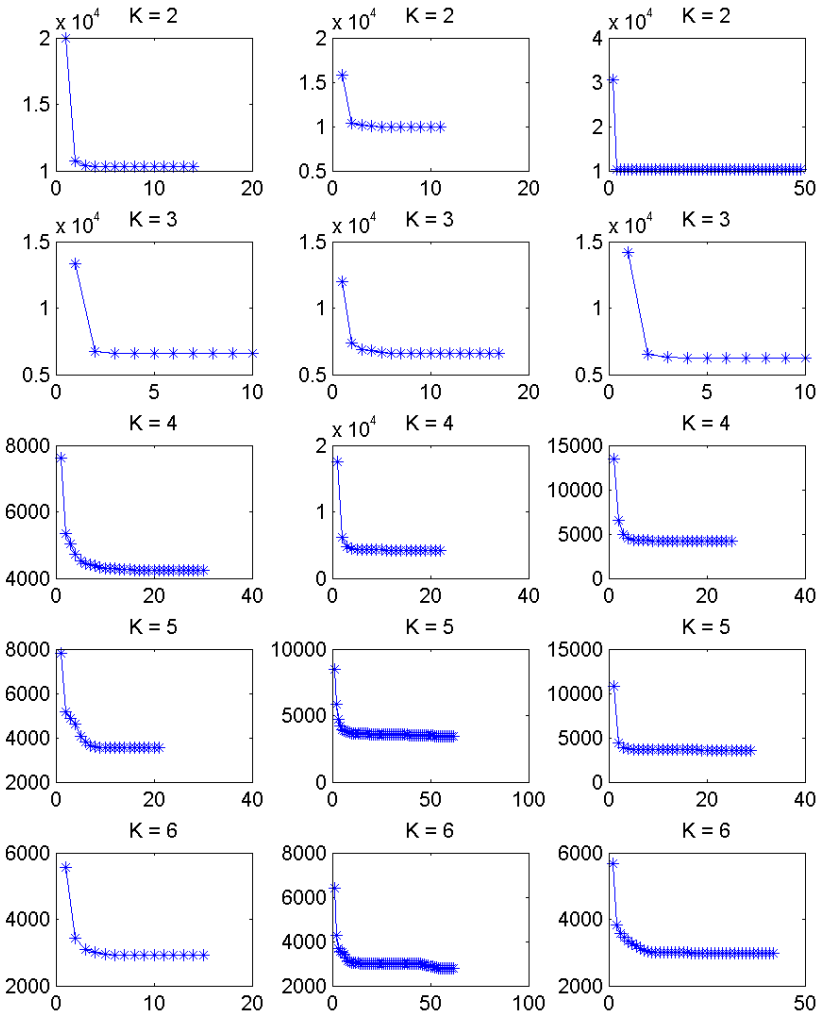


FIGURE 20 – George - Distance de Mahalanobis

Data Mining
TP5 - Clustering par mélange de gaussiennes

Rémi MUSSARD - Thomas ROBERT

1 Code

Afin de réaliser un clustering par la méthode du mélange de gaussiennes, on code l'algorithme EM. On retrouve dans ce code les deux étapes délimitées par des commentaires. Le code en commentaire est le code avant optimisation de plusieurs étapes de calcul.

```
%%
close all;
clc;
clear all;

load gauss3.mat;
%X = data;
X = load('george.dat');
X = mydownsampling(X, 7);

[N, d] = size(X);
K = 6;

figure();
hold on;

% Initialisation
mu = init_centres(X, K);
S = {};
for j = 1:K
    S{j} =  eye(d)*var(X(:,1));

end
pi = ones(K,1)*1/K;

mu_prec = mu + 1;
while abs(sum(mu-mu_prec)) > 1e-6
    mu_prec = mu;

    % Etape E

    P = zeros(N,K);
    for j = 1:K
        P(:,j) = pi(j) * mvnpdf(X, mu(j,:), S{j});
    end
    P = P./repmat(sum(P,2), 1, K);

    % Etape M

    mu = (X'*P./repmat(sum(P),d,1))';
    %mu = zeros(K, d);
    %for j = 1:K
    %    mu(j,:) = sum(repmat(P(:,j), 1, d).*X)/sum(P(:,j));
    %end

    pi = sum(P)/N;
    %pi = zeros(K, 1);
    %for j = 1:K
```

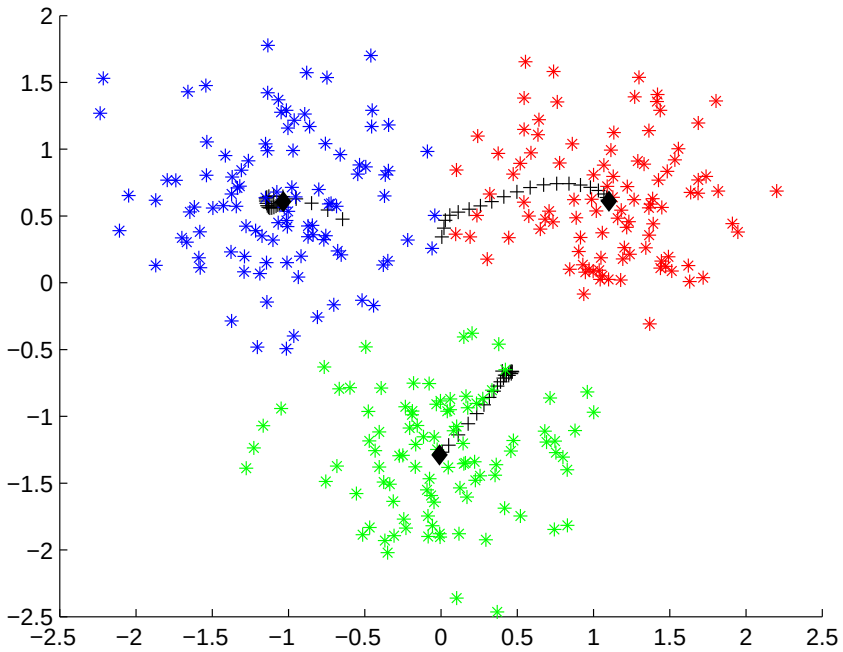
```
49      %     pi(j) = sum(P(:,j))/N;
50      %end
51
52      for j = 1:K
53          S{j} =  (X - repmat(mu(j,:),N,1))'*diag(P(:,j))*(X - repmat(mu(j,:),N,1)) / sum(P(:,j));
54
55          %S{j} = zeros(d);
56          %for i = 1:N
57          %   S{j} = S{j} + P(i,j)*(X(i,:)-mu(j,:))'*(X(i,:)-mu(j,:));
58          %end
59          %S{j} = S{j} / sum(P(:,j));
60      end
61
62      plot(mu(:,1), mu(:,2), '+','Color',[0.5,0.5,0.5]);
63
64  end
65
66  % extraction de la liste des clusters
67  [~, liste] = max(P');
68  % affichage des clusters
69  show_clusters(X, liste);
70  % affichage des centres
71  scatter(mu(:,1), mu(:,2), 50, 'd', 'k', 'fill');
```

../TP5_MUSSARD_ROBERT.m

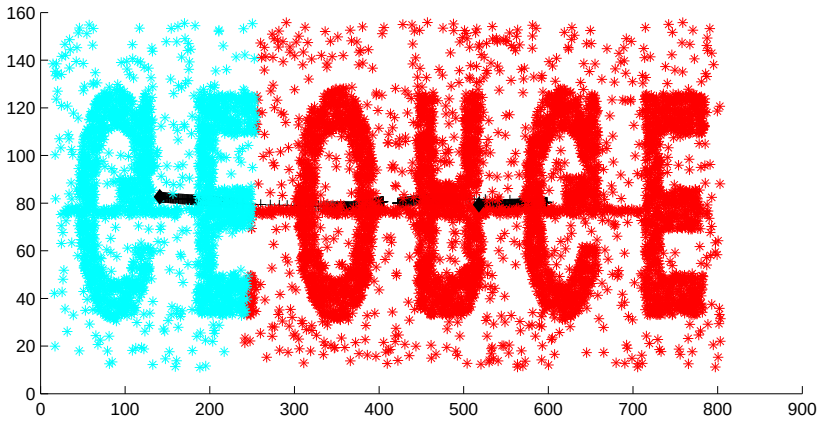
2 Résultats et commentaires

On voit que sur un jeu de données appropriés (gauss3.mat), la méthode fonctionne bien et permet de déterminer les centres des points répartis selon une loi normale.

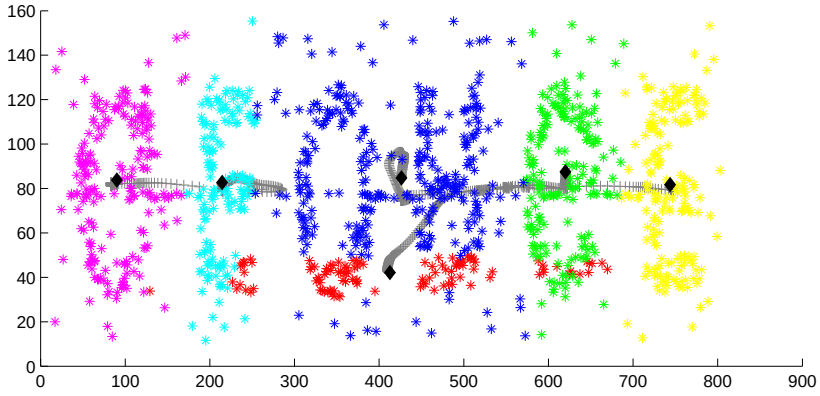
En revanche, sur un jeu de données où les points ne respectent pas une loi normale, comme par exemple george.dat où les données représentent des lettres, la méthode a beaucoup de mal à converger et ne donne pas un résultat satisfaisant.



Données gauss avec 3 clusters



Données george avec 2 clusters



Données george avec 6 clusters

3 Conclusion

Après avoir testé diverses méthodes de clustering, on a pu constater que les diverses méthodes fonctionnent globalement bien. Cependant, les résultats fournis par les diverses méthodes regroupent dans un clusters des points les plus proches, ce qui ne correspond pas toujours vraiment aux données fournies, qui contiennent plutôt des formes et non des groupes distincts de points.

La méthode CHAméléon présentée dans le cours semble pouvoir résoudre plus ou moins ce problème d'après les exemples fournis.