TD noté, jeudi 10 décembre 2009

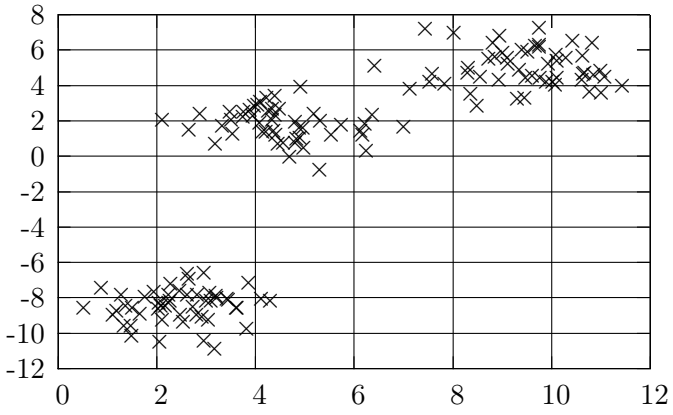
Le programme construit au fur et à mesure des questions devra être imprimé à la fin du TD et rendu au chargé de TD. Il ne faut pas oublier de mentionner son nom en commentaires au début du programme et l'ajouter sur chaque page. Les réponses autres que des parties de programme seront insérées sous forme de commentaires. Les définitions de fonctions proposées ne sont que des suggestions.

Glossaire :

barycentre — Un barycentre est le centre d'un ensemble de points $(x_i, y_i)_{1 \leqslant i \leqslant n}$. Dans le plan, il a deux coordonnées (X, Y) égales à $X = \frac{1}{n} \sum_i x_i$ et $Y = \frac{1}{n} \sum_i y_i$. Ses coordonnées le placent au milieu des points dans il est le barycentre : il est situé dans l'enveloppe convexe formée par l'ensemble des points.

centree — En dimension deux, même si c'est une expression employée dans la suite de l'énoncé, une loi normale de centre (x, y) n'est pas une expression correcte. On devrait dire une loi normale de moyenne (x, y). De même, cette loi n'a pas une variance $\sigma \in \mathbb{R}$, on devrait dire une variance $\begin{pmatrix} \sigma & 0 \\ 0 & \sigma \end{pmatrix}$.

Les nuées dynamiques servent à construire automatiquement des classes dans un ensemble d'observations. C'est une façon de regrouper entre elles des observations qui sont proches les unes des autres. Prenons par exemple le nuage de points suivant qui inclut trois sous-nuages.



Le nuage de points contient trois sous-ensembles de points. Chacun est un ensemble de points simulés selon une loi normale de variance 1 et de moyenne identique à l'intérieur d'un sous-ensemble.

1) La fonction `gauss` du module `random` permet de générer un nombre selon une loi normale. Le premier objectif est de créer une fonction qui retourne un ensemble de points simulés selon une loi normale de variance v et de centre (x, y). (2 points)

```
def sous_nuage (nb, x, y, v) :  # retourne une liste de 2-uples
```

2) On cherche à créer un nuage regroupant n sous-nuages de même variance 1 avec la fonction précédente. Chaque sous-nuage est centré autour d'une moyenne choisie aléatoirement selon une loi de votre choix. La fonction dépend de deux paramètres : le nombre de points dans chaque classe et le nombre de classes. (2 points)

```
def n_sous_nuages (n, nb) :     # retourne une liste de 2-uples
```

3) Dessiner le nuage avec le module `matplotlib` pour vérifier que le résultat correspond à vos attentes. On pourra s'appuyer sur l'extrait qui suit. (1 point)

```
import matplotlib.pyplot as plt
x   = [ ... ]
y   = [ ... ]
fig = plt.figure()
ax  = fig.add_subplot(111)
ax.plot (x,y, 'o')
plt.savefig ("im1.png") # ou plt.show () pour afficher le graphe
```

4) L'algorithme des nuées dynamiques commence par affecter chaque point à une classe choisie au hasard. Pour cette tâche, on pourra utiliser la fonction `randint` du module `random`. On veut créer une fonction qui retourne une classe aléatoire pour chaque point du nuage. Elle doit prendre comme entrée le nombre de classes souhaité. (2 points)

```
def random_class (points, n) :  # retourne une liste d'entiers
```

5) L'algorithme des nuées dynamiques répète ensuite alternativement deux étapes :

Etape 1 On calcule le barycentre de chaque classe.
Etape 2 On associe à chaque point la classe dont le barycentre est le plus proche (au sens de la distance euclidienne).

On propose de commencer par écrire une fonction qui retourne pour un point donné le barycentre le plus proche. (2 points)

```
def proche_barycentre (point, barycentres) :   # retourne un entier
```

6) La fonction suivante retourne le barycentre le plus proche pour chaque point. (2 points)

```
def association_barycentre (points, barycentres) :  # retourne une liste d'entiers
```

7) On découpe la première étape de la même façon :

1. Première fonction : calcule le barycentre d'une classe.
2. Seconde fonction : calcule le barycentre de toutes les classes.

Il faut implémenter ces deux fonctions. (3 points sans utiliser `numpy`, 4 points avec `numpy` et une fonction).

```
def barycentre_classe (points, classes, numero_class) :   # retourne un 2-uple
def tous_barycentres  (points, classes) :       # retourne une liste de 2-uples
```

8) L'algorithme commence par la création des classes (fonction `n_sous_nuages`) et l'attribution d'une classe au hasard (fonction `random_class`). Il faut ensuite répéter les fonctions `tous_barycentres` et `association_barycentre`. L'enchaînement de ces opérations est effectué par la fonction `nuees_dynamiques`. (2 points)

```
def nuees_dynamiques (points, nombre_classes) : # retourne une liste d'entiers
```

9) Dessiner le résultat permettra de vérifier que tout s'est bien passé, toujours avec un code similaire à celui-ci. (2 points)

```
import matplotlib.pyplot as plt
x1  = [ ... ]
y1  = [ ... ]
x2  = [ ... ]
y2  = [ ... ]
fig = plt.figure()
ax  = fig.add_subplot(111)
ax.plot (x1,y1, 'o')
ax.plot (x2,y2, 'x')     # ligne ajoutée, 'x', 'bo', ...
plt.savefig ("im2.png")  # 'rx', 'go', 'gs', 'bs', ...
```

10) Question facultative : comment savoir quand s'arrêter ? (0 point)

Correction

Le corrigé final apparaît après les commentaires qui suivent. Ils sont inspirés des réponses des élèves.

1) 2) Le centre de chaque sous-nuages a été généré selon diverses lois aléatoires, des lois normales, uniformes réelles ou discrètes.

```
def n_sous_nuages (n, nb):
    m = []
    for i in range (0,n):
        x = 5*random.random()
        y = 5*random.random()
        d = sous_nuage(nb,x,y,1)
        m += d
    return m
```

```
def n_sous_nuages (n, nb):
    m = []
    for i in range (0,n):
        x = random.randint(0,20)
        y = random.randint(0,20)
        d = sous_nuage(nb,x,y,1)
        m += d
    return m
```

L'exemple de droite utilise la même loi pour générer aléatoirement à la fois le centre de chaque nuage et les points qu'ils incluent. Il sera alors difficile de distinguer visuellement plusieurs sous-nuages avec le graphe dessiné à la question suivante.

```
def n_sous_nuages (n, nb):
    m = []
    for i in range (0,n):
        x = random.gauss(0,1)
        y = random.gauss(0,1)
        d = sous_nuage(nb,x,y,1)
        m += d
    return m
```

Quels que soient les points simulés, les réponses aux questions suivantes n'en dépendaient pas. L'algorithme des centres mobiles s'appliquent à n'importe quel ensemble de points bien que le résultat ne soit pas toujours pertinent.

Certains élèves ont ajouté [d] au lieu de d seul. Au lieu d'obtenir comme résultat une liste de 2 coordonnées (une matrice de deux colonnes), le résultat est alors une liste de matrices de deux colonnes : c'est un tableau à trois dimensions. Ce n'est pas faux mais cela complique inutilement l'écriture des fonctions qui suivent en ajoutant une boucle à chaque fois qu'on parcourt l'ensemble des points.

```
def n_sous_nuages (n, nb):
    m = []
    for i in range (0,n):
        x = random.gauss(0,1)
        y = random.gauss(0,1)
        d = sous_nuage(nb,x,y,1)
        m +=  [ d ]
           # le résultat n'est
           # plus une liste
    return m
```

3) Utiliser l'exemple de l'énoncé n'a pas posé de problème excepté un cas particulier :

```
import matplotlib.pyplot as plt
x = [ p [0] for p in n_sous_nuages (3,50) ]
y = [ p [1] for p in n_sous_nuages (3,50) ]
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot (x,y, 'o' )
plt.savefig ("im1.png")
```

Dans cet exemple, la fonction `n_sous_nuages` est appelée une fois pour extraire les abscisses, une seconde fois pour extraire les ordonnées. Etant donné que cette fonction retourne un résultat aléatoire, il est très peu probable qu'elle retourne deux fois le même résultat. Par conséquence, les abscisses et les ordonnées ne proviennent pas du même nuage : le graphique résultant ne montrera pas trois nuages séparés.

4) La fonction `randint(a,b)` retourne un nombre entier aléatoire compris entre `a` et `b` **inclus**. Il fallait donc bien choisir `a` et `b`. Le meilleur choix était $a = 0$ et $b = n - 1$. Un autre choix assez fréquent était $a = 1$ et $b = n$ comme dans l'exemple suivant :

```
def random_class(l,n):
    l = []
    for i in range(0,len(l)):
        l += [ random.randint (1,n) ]
    return l
```

Les deux réponses sont correctes. Toutefois, la solution ci-dessus implique de faire un peu plus attention par la suite car elle complique la correspondance entre les barycentres et le numéro de la classe qu'il représente. En effet, qu'en est-il de la classe 0 dans ce cas. Dans cet exemple, la fonction `random_class` n'associe aucun point à la classe 0. On peut alors se demander à quoi correspond le premier élément du tableau `barycentre` utilisé dans les fonctions des questions suivantes. Quoi qu'il en soit, la fonction `proche_barycentre` retourne l'indice du barycentre le plus proche, pas le numéro de la classe à laquelle il correspond. Selon les programmes, avec un peu de chance, les numéros des classes ont commencé à 0 après le premier appel à la fonction `proche_barycentre`. Le calcul du barycentre de la première classe amène une division par zéro à moins que ce cas ait été pris en compte.

Dans l'exemple suivant, on tire une classe aléatoire parmi $n + 1$ numéros. Il y a donc une classe de plus qu'attendu mais là encore, cette erreur peut être compensée par une autre plus loin.

```
def random_class(l,n):
    l = []
    for i in range(0,len(l)):
        l += [ random.randint (0,n) ]
    return l
```

Un élève a fait une allusion à la probabilité qu'une classe soit vide : un numéro entre 0 et $n-1$ n'est jamais attribué. On peut déjà se pencher sur la probabilité que la classe 0 soit vide. Chaque point a un probabilité $\frac{1}{n}$ d'être associé à la classe 0. La probabilité cherchée est donc : $\left(\frac{n-1}{n}\right)^N$ où N est le nombre de points. On peut ainsi majorer la probabilité qu'une classe soit vide par : $n\left(\frac{n-1}{n}\right)^N$.

5) La fonction `proche_barycentre` a été plutôt bien traitée malgré deux erreurs fréquentes. La première concerne la fonction puissance lors du calcul de la distance euclidienne :

```
d= (  (p[0]-f[0])**2+(p[1]-f[1])**2 ) ** (1/2)
```

Dans l'exemple précédente, `1/2` est une division entière et son résultat est nul. Comme $\forall x$, $x^0 = 1$ (pour *Python* du moins), toutes les distances calculées sont donc égales à 1. Il faut noter que la racine carrée n'était pas indispensable puisqu'on cherchait le barycentre le plus proche : seule la plus petite valeur comptait et non la valeur elle-même.

L'autre erreur fréquente est celle-ci :

```
def proche_barycentre (point,barycentres):
    d=distance_euclidienne(point,barycentres[0])
    for i in range (0,len(barycentres)):
        if distance_euclidienne(point,barycentres[i])<=d:
            d=distance_euclidienne(point,barycentres[i])
    return d
```

On retourne non pas l'indice du barycentre le plus proche mais la distance de ce barycentre au point considéré.

6) Cette question a été bien traitée. Les erreurs introduites dans la fonction précédentes se sont propagées sans provoquer d'erreur d'exécution.

7) Dans la fonction suivante, si la plupart ont pensé à ne prendre que les points de la classe `numero_class`, ils ont parfois oublié de diviser par le bon nombre d'éléments. Ici, c'est la variable `n` qui n'est définie nulle part. Si le programme ne provoque pas d'erreurs, c'est donc une variable globale déclarée avant.

```
def barycentre_classe (points, classes, numero_class):
    x=0
    y=0
    for i in range (0,len(classes)):
        if classes[i]==numero_class:  # ligne importante
            l=point[i]
            x=x+l[0]
            y=y+l[1]
    c=[x/n,y/n]                       # ligne importante
    return c
```

La variable `n` a parfois été remplacée par `len(classes)` qui est aussi faux puisque cela correspond au nombre total de points et non celui de la classe `numero_class`.

Il arrive que la fonction provoque une division par zéro lorsqu'une classe est vide. C'est un cas à prendre en compte. L'algorithme peut alors évoluer dans deux directions. La première consiste à supprimer la classe. Le second choix évite la disparition d'une classe en affectant un ou plusieurs points désignés aléatoirement à la classe disparue. L'énoncé ne demandait à ce qu'il en soit tenu compte même si cela serait souhaitable.

La fonction `tous_barycentre` a été victime de deux erreurs. La première est la suivante où on construit autant de barycentres qu'il y a de points alors qu'on souhaite autant de barycentres qu'il y a de classes :

```
def tous_barycentres (points,classes):
    c=[]
    for i in classes :   # or on a len(classes) == len(points)
        c+=[barycentre_classe (points,classes,i)]
    return c
```

La seconde erreur intervient lors du numéro de classes et fait écho à la fonction `randon_class` et ses erreurs. Dans l'exemple suivant, la classe dont le numéro est le plus grand a pour numéro `max(classes)`. Or dans la boucle `for i in range(0,mx)` :, elle est oubliée car la fonction `range` va jusqu'à `mx - 1` inclus. Il aurait fallu écrire `mx = max(classes) + 1`. Dans le cas contraire, on perd une classe à chaque fois qu'on appelle la fonction `tous_barycentres`.

```
def tous_barycentres (points,classes):
    c=[]
    mx=max(classes)        # il faut ajouter +1
    for i in range(0,mx) :
        c+=[barycentre_classe (points,classes,i)]
    return c
```

Il faut noter également que la classe 0 reçoit un barycentre après la fonction `tous_barycentres` même si la fonction `random_class` ne lui en donnait pas lorsqu'elle utilise l'instruction `random.randint(1,n)`.

Peu d'élèves ont utilisé le module `numpy`. Son usage avait pour but d'éviter une boucle sur les points : elle ne disparaît pas mais est prise en charge par les fonctions de calcul matriciel proposées par le module `numpy`. Cette boucle a persisté dans la grande majorité des solutions envisagées. La difficulté réside dans la construction d'une ou deux matrices qui mènent au calcul des barycentre par quelques manipulations matricielles. La correction qui suit présente deux implémentations dont les seules boucles portent sur le nombre de classes.

8) La dernière fonction de l'algorithme de classification a connu trois gros défauts. Le premier est l'oubli de la boucle qui permet de répéter les opérations plus d'une fois. Le second défaut apparaît lorsque le résultat de la fonction `association_barycentre` n'est pas utilisé. Dans l'exemple suivant, le calcul des barycentres a toujours lieu avec la liste `l` qui n'est jamais modifiée : le résultat `a` est toujours celui de l'algorithme après la première itération qui est répétée ici 10 fois exactement de la même manière.

```
def nuees_dynamiques (points,nombre_classes):
    l = random_class (points,nombre_classes)
    for j in range (0,10):
        c = tous_barycentres (points, l)
        a = association_barycentre (points,c)
                        # il faut ajouter ici l = a pour corriger la fonction
    return a
```

La dernière erreur suit la même logique : l'instruction `l = a` est bien présente mais son effet est annulé par le fait de générer aléatoirement un numéro de classe à chaque itération.

```
def nuees_dynamiques (points,nombre_classes):
    for j in range (0,10):
        l = random_class (points,nombre_classes)
        c = tous_barycentres (points, l)
        a = association_barycentre (points,c)
        l = a
    return a
```

Enfin, une erreur grossière est parfois survenue : l'exemple suivant change les données du problème à chaque itération. Le résultat a peu de chance de signifier quoi que ce soit.

```
def nuees_dynamiques (n,nb):
    for j in range (0,10):
        points = n_sous_nuage (n,nb)
        l = random_class (points,nombre_classes)
        c = tous_barycentres (points, l)
        a = association_barycentre (points,c)
        l = a
    return a
```

9) La dernière question fut plus ou moins bien implémentée, souvent très bien pour le cas particulier de deux classes. Le cas général où le nombre de classes est variable n'a pas été souvent traité. La correction complète suit.

```
# coding: latin-1
import random
import numpy

def dessin (nuage, image = None) :
    """dessine un nuage de points
    @param      nuage       le nuage de points
    @param      image       si None, on affiche le nuage au travers d'une fenêtre,
                            sinon, image correspond à un nom d'image
                            sur disque dur qui contiendra le graphique final"""
    import matplotlib.pyplot as plt
    x = [ p[0] for p in nuage ]
    y = [ p[1] for p in nuage ]
    plt.clf ()
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot (x,y, 'o')
    if image == None :  plt.show ()
    else :              plt.savefig (image)

def dessin_classes (nuage, classes, image = None) :
    """dessine un nuage, donne des couleurs différentes
    selon que le point appartient à telle ou telle classes
    @param      nuage           nuage[i], c'est le point i
    @param      classes         classes [i] est la classe associée au point i
    @param      image           voir la fonction précédente
    """
```

```
    import matplotlib.pyplot as plt
    x = {}
    y = {}
    for i in range (0, len (nuage)) :
        cl = classes [i]
        if cl not in x :
            x [cl] = []
            y [cl] = []
        x [cl].append ( nuage [i][0] )
        y [cl].append ( nuage [i][1] )
    plt.clf ()
    fig = plt.figure()
    ax = fig.add_subplot(111)
    for cl in x :
        ax.plot (x [cl], y [cl], "+")
    if image == None :  plt.show ()
    else :              plt.savefig (image)

def sous_nuage (nb, x, y) :
    """retourne un ensemble de points tirés aléatoirement selon
    une loi normale centrée autour du point x,y
    @param      nb              nombre de points
    @param      x               abscisse du centre
    @param      y               ordonnée du centre
    @return                     une liste de points ou matrice de deux colonnes
                                - la première correspond aux abscisses,
                                - la seconde aux ordonnées
    """
    res = []
    for i in xrange (0, nb) :
        xx = random.gauss (0,1)
        yy = random.gauss (0,1)
        res.append ( [x+xx, y+yy] )
    return res

def n_sous_nuages (nb_class, nb_point) :
    """crée un nuage de points aléatoires
    @param      nb_class        nombre de sous nuages
    @param      nb_point        nombre de points dans chaque sous nuage
    @return                     une liste de points ou matrice de deux colonnes
                                - la première correspond aux abscisses,
                                - la seconde aux ordonnées"""
    res = []
    for c in xrange (0, nb_class) :
        x = random.gauss (0,1) * 5
        y = random.gauss (0,1) * 5
        res += sous_nuage (nb_point, x,y)
    return res

def random_class ( nuage, n) :
    """choisis aléatoirement un entier pour chaque point du nuage
    @param      nuage           un nuage de points (matrice de deux colonnes)
    @param      n               nombre de classes
    @return                     une liste d'entiers
    """
    res = [ ]
    for p in nuage :
        c = random.randint (0, n-1)
        res.append (c)
```

```
    return res

def proche_barycentre (point, barycentres) :
    """détermine le barycentre le plus d'un point
    @param      point           liste de 2 réels : [x,y]
    @param      barycentres     liste de n points = matrice de deux colonnes,
                                chaque ligne correspond à un barycentre
    @return                     un entier qui correspond à l'index
                                du barycentre le plus proche"""
    dmax = 1e6
    for i in range (0, len (barycentres)) :
        b  = barycentres [i]
        dx = point [0] - b [0]
        dy = point [1] - b [1]
        d  = (dx**2 + dy**2) ** 0.5
        if d < dmax :
            dmax = d
            m    = i
    return m

def association_barycentre (points, barycentres) :
    """détermine pour chaque point le barycentre le plus proche
    @param      points          nuage (matrice de deux colonnes)
    @param      barycentres     c'est aussi une matrice de deux colonnes mais
                                avec moins de lignes
    @return                     liste d'entiers, chaque entier
                                correspond à la classe du point points[i],
                                c'est-à-dire l'index du barycentre le plus proche
                                ici:
                                point:      points [i]
                                classe:     res[i]
                                barycentre: barycentres[ res[i] ]
    """
    res = []
    for p in nuage :
        m = proche_barycentre (p, barycentres)
        res.append (m)
    return res

def barycentre_classe (points, classes, numero_class) :
    """calcule le barycentre d'une classe
    @param      points          ensemble de points (matrice de deux colonnes)
    @param      classes         liste d'entiers de même longueur,
                                chaque élément classes[i] est la classe de point[i]
    @param      numero_class    classe pour laquelle on doit calculer le barycentre
    @return                     résultat barycentre x,y

    dans cette fonction, on doit calculer le barycentre d'une classe
    c'est-à-dire le barycentre des points points[i]
    pour lesquelles classes[i] == numero_class
    """
    mx,my = 0.0,0.0
    nb    = 0
    for i in range (0, len (points)) :
        p = points [i]
        c = classes [i]
        if c != numero_class : continue
        nb += 1
        mx += p [0]
```

```
        my += p [1]
    return mx/nb, my/nb

def tous_barycentres (points, classes) :
    """calcule les barycentres pour toutes les classes
    @param      points      points, nuage, matrice de deux colonnes
    @param      classes     liste d'entiers
    @return                 liste de barycentre = matrice de deux colonnes
    """
    mx          = max (classes)+1
    barycentre  = []
    for m in range (0,mx) :
        b = barycentre_classe (points, classes, m)
        barycentre.append (b)
    return barycentre

def numpy_tous_barycentres (points, classes) :
    """écriture de barycentre_classe et tous_barycentres
    en une seule fonction avec numpy
    """
    nbcl = max (classes)+1
    mat  = numpy.matrix (points)
    vec  = numpy.array ( classes )
    clas = numpy.zeros ( (len (points), nbcl) )
    for i in range (0, nbcl) :
        clas [ vec == i, i ] = 1.0
    nb   = clas.sum (axis = 0)
    for i in range (0, nbcl) :
        clas [ vec == i, i ] = 1.0 / nb [i]
    ba = mat.transpose () * clas
    ba = ba.transpose ()
    ba = ba.tolist ()
    barycentre = [ b for b in ba ]
    return barycentre

def numpy_tous_barycentres2 (points, classes) :
    """écriture de barycentre_classe et tous_barycentres
    en une seule fonction avec numpy
    """
    nbcl        = max (classes)+1
    mat         = numpy.matrix (points)
    matt        = mat.transpose ()
    matcl       = numpy.matrix (classes).transpose ()
    barycentre  = []
    for c in xrange (0, nbcl) :
        w  = numpy.matrix (matcl)
        w [matcl==c] = 1
        w [matcl!=c] = 0
        wt = w.transpose ()
        r  = matt * w
        n  = wt * w
        r /= n [0,0]
        barycentre += [ [ r [0,0], r [1,0] ] ]

    return barycentre

def nuees_dynamiques (points, nbcl) :
    """algorithme des nuées dynamiques
    @param      points          ensemble points = matrice de deux colonnes
```

```
    @param      nbcl            nombre de classes demandées
    @return                     un tableau incluant la liste d'entiers
    """
    classes = random_class (points, nbcl)

    # on a le choix entre la version sans numpy
    for i in range (0,10) :
        print "iteration",i, max (classes)+1
        barycentres = tous_barycentres (points, classes)        # ou l'un
        classes     = association_barycentre (points, barycentres)
    cl1 = classes

    # ou la première version avec numpy
    for i in range (0,10) :
        print "iteration",i, max (classes)+1
        barycentres = numpy_tous_barycentres (points, classes)  # ou l'autre
        classes     = association_barycentre (points, barycentres)
    cl2 = classes

    # ou la seconde version avec numpy
    for i in range (0,10) :
        print "iteration",i, max (classes)+1
        barycentres = numpy_tous_barycentres2 (points, classes)  # ou l'autre
        classes     = association_barycentre (points, barycentres)
    cl3 = classes

    # on doit trouver cl1 == cl2 == cl3
    if cl1 != cl2 or cl1 != cl3 :
        print "erreur de calculs dans l'une des trois fonctions"
    return classes

# début du programme : on construit un nuage de points
nuage = n_sous_nuages (3, 50)
# on appelle l'algorithme
classes = nuees_dynamiques (nuage, 3)
# on dessine le résultat
dessin_classes (nuage, classes)
```