Algorithme Génétique et Ant colony optimisation

Objectif :

Dans ce projet, il fallait reprendre le TP sur les algorithmes génétiques, terminer le code et l'utiliser pour trouver les optima globaux de fonctions telles que Rosenbrock, Himmelblau, Rastrigin.
La seconde étape consiste à proposer un autre algorithme basé sur les colonies de fourmis.

Démarche :

Maple :

Outre les difficultés de conception algorithmiques, ce projet a nécessité un nombre et temps de recherche très important sur le lange Maple que je ne maitrisais mal. Je me suis essentiellement appuyée sur le site MapleSoft.

Algorithmes génétiques :

Je suis parti de la version TP envoyée le 05/03/2014.
J'ai codé les 3 fonctions étudiées, Rosenbrock, Himmelblau et Rastrigin.
Dans un premier temps, j'avais repris 3 procédures différentes pour les 3 fonctions en les adaptant sur le nombre de croisements et les intervalles des tirages aléatoires.
J'ai ensuite regroupé dans la même procédure, en appelant les fonctions d'optimisation en paramètre de ma procédure Darwin:=proc(fonction_a_optimiser)
La procédure Darwin crée une liste de parents optimaux grâce à une fonction d'optimisation puis crée une nouvelle liste à partir de leurs enfants eux-mêmes optimisés par cette même fonction. Cette boucle de génération parents/enfants, s'effectue jusqu'au nombre de croisements voulus. Le résultat de l'optimisation dépend de la valeur du croisement et diffère selon les fonctions. Il me semble que l'optimisation est plus précise pour les minima de la fonction Rosenbrock.
Le programme rendu en Maple comprend la procédure Darwin et son exécution pour les 3 fonctions d'optimisation.

Ant colony optimisation :

Les algorithmes de colonies de fourmis sont une technique faisant partie des algorithmes d'optimisation. L'explication de ce concept est souvent assimilée à l'algorithme du voyageur de commerce. On synthétise l'optimisation d'un chemin partant d'un point A vers un point B grâce au schéma naturel du comportement des fourmis. Une première fourmi emprunte un chemin de manière aléatoire en laissant des phéromones qui donnent des informations à propos du chemin emprunté pour les autres fourmis à venir. Puis d'autres fourmis suivent ces chemins tout en conservant une part « d'exploration ». Les phéromones disparaissent avec le temps au profit des chemins les plus empruntés. Le but étant de tendre vers un chemin optimal. Toutes ces notions sont synthétisées par des formules, complexes au premier abord, mais compréhensibles à condition de s'attarder un petit moment dessus. Les deux formules principales sont celle du choix de la fourmi face aux chemins et celle du poids de ces chemins en fonction des phéromones.

Ex: CHOIX de la fourmi face aux chemins :

$$P^k(i,j) = \begin{cases} \dfrac{[\tau(i,j)]^{\alpha}.[\eta(i,s)]^{\beta}}{\sum[\tau(i,s)]^{\alpha}.[\eta(i,s)]^{\beta}}, & \text{(chemin connu)} \\ 0, & \text{(chemin inconnu)} \end{cases}$$

Le POIDS des chemins est calculé par une autre formule.

$$\text{Choix du chemin} = \frac{\text{Information du chemin devant lequel la fourmi se trouve (poids)}}{\text{Information de la route totale de A vers B}}$$

η = information sur la distance entre les points
τ = intensité de la phéromone (répartie sur le terrain, sorte de « marqueur post-it »)
α, β = densité en fonction du nombre de fourmis passées : l'utilisation d'un chemin augmente en fonction du nombre de passages, les fourmis suivent les renseignements directionnels donnés par les autres

Pour ma part, j'ai commencé par faire de nombreuses recherches sur internet après avoir globalement compris le principe à partir de la page wikipedia donnée. Tous les documents trouvés étaient d'un niveau très élevé, en général issus de thèses ou de travaux de recherches.
Une aide de monsieur Legrand, maıtre de Conférence, Université de Bordeaux 2, m'a permis de repositionner mon travail, non plus sous le schéma de l'algorithme du voyageur de commerce / colonies de fourmis (décrit ci-dessus), mais de concevoir le problème comme une nuage de point en 3 dimensions avec différentes profondeurs (z) permettant d'apprécier la qualité de l'optimisation.
Enfin un échange avec Hayssam Soueidan, chercheur au Centre de Bioinformatique de Bordeaux (que je remercie chaleureusement), m'a orienté vers le choix des points aléatoires par variable aléatoire non uniforme pour avoir plus de probabilité de sélectionner un point de poids minimal « au centre » de la courbe gaussienne.

La procédure *Fourmi* à qui l'on passe en paramètre la fonction d'optimisation (*Rosenbrock, Himmelblau* ou *Rastrigin*), le nombre d'itération et la taille de la population (*N*), crée puis classe de manière croissante, une liste *init[ ]* de *N* points générés aléatoirement pour (*x,y*) et générés par la fonction d'optimisation pour *z*.
Une boucle exécute le nombre d'itération choisie, extrait une liste des points (*x,y*) et une liste des poids des points (*z*), sélectionne aléatoirement deux des points (*p1, p2*) de la liste, calcule le poids d'un point (*p3*) situé aléatoirement entre les deux points choisis (*p1+λ*p2*) avec la fonction d'optimisation choisie (*p3:=[op(p3),function_a_opt(p3[1],p3[2])];*). Ce point *p3* est ajouté à la liste *init[ ]*, liste ensuite triée de manière croissante selon le poids, puis conservée pour ses *N* premiers points de poids les plus faibles.
A la sortie de la boucle, on affiche le premier point de la liste init[ ], celui de poids le plus faible.

La procédure aleatoire_non_uniforme permet d'échantillonner des distributions discrètes non uniformes et est utilisée pour sélectionner à chaque itération deux points (p1,p2) de la liste init[ ], en favorisant les points de poids faibles et où le poids correspond à la valuation de la fonction à minimiser. Pour effectuer ce tirage, on transforme tout d'abord la liste des poids en une liste des poids normalisés (somme de tous les poids – poids initial). Un nombre aléatoire uniformément distribué entre 1 et la nouvelle somme des poids normalisés est ensuite tiré. Une boucle sélectionne finalement le premier élément de la liste tel que la somme des poids des éléments précédents soit supérieure au nombre tiré aléatoirement.

Bilan :

Au final ce projet a été riche d'enseignements : l'algorithme qui me semblait incompréhensible (mais passionnant) à programmer dans un langage qui me semblait incompréhensible (peut-être est-il passionnant… ?) s'est finalement révélé compréhensible pour moi, petit à petit, par mes recherches en ligne et mes rencontres physiques.
J'ai pu également constater l'intérêt de cet algorithme lors de mes rencontres avec des chercheurs ou ingénieurs dans le cadre du Projet Professionnel de l'Etudiant (PPE) : ces chercheurs (tel qu'André Garenne ou Hayssam Soueidan) travaillent sur des algorithmes se basant sur des modèles naturels appliqués à l'informatique (algorithmes de colonies d'abeilles, algorithmes simulant des lésions cérébrales, algorithmes de pousses de maïs). Cela m'a conforté dans mes orientations universitaires centrées sur la cognitique et l'informatique.

Ressources consultées :

Aide et syntaxe Maple :
http://www.maplesoft.com/support/help/

Document donné par M. Legrand :
http://math.unice.fr/~dreyfuss/rapport_11-12_2.pdf

Documents consultés :
http://iridia.ulb.ac.be/IridiaTrSeries/link/IridiaTr2013-002.pdf
http://link.springer.com/article/10.1134%2FS1064230710010053#page-1
http://archives.math.utk.edu/ICTCM/VOL13/C039/paper.pdf (Bruno Guerrieri / Florida A&M University)

Ci-dessous, listing des procédures *Darwin* et *Fourmi.*

Darwin :

```
STUDENT > Rosenbrock:=proc(x,y)
              local f;
              f:=((1-x)^2)+100*((y-(x^2))^2);
          end:
STUDENT > Himmelblau:=proc(x,y)
              local f;
              f:=((x^2)+y-11)^2+(x+(y^2)-7)^2;
          end:
STUDENT > Rastrigin:=proc(x,y)
              local f;
              f:=evalf(20+((x^2)-10*cos(2*Pi*x))+((y^2)-10*cos(2*Pi*y)));
          end:

STUDENT > Darwin:=proc(fonction_a_optimiser)

          local N, Nb_enf, GenR, Croisement, new_list, i, x, y, z, parents, indiv1, indiv2, indiv3, indiv4, k, j, pere, mere, p1,
          p2, puissance_p1, puissance_p2, m1, m2, puissance_m1, puissance_m2, a, b, c, d, X_enf, Y_enf, Z_enf;

          # N, nombre d'individus parents initiaux
          N:=250;
          Nb_enf:=100;
          #Avant 10 croisements, les resultats sont pas tres precis, on arrive aux points (1,1) a partir de 10
          Croisement:=20;
          GenR:=rand(-6000..6000)/1000;

          #création de listes pas encores remplies
          parents:=[NULL,NULL,NULL];
          new_list:=[NULL,NULL,NULL];

          #Remplissage de ces listes avec notion d'aleatoirite introduite par Rosenbrk
          while nops(parents) < N do
           for i from 1 to N do
              x:=evalf(GenR(),3);
              y:=evalf(GenR(),3);
              z:=fonction_a_optimiser(x,y);
           parents:=[op(parents),[x,y,z]];
           od;
          od;
          parents:=sort(parents,(indiv1,indiv2)-> evalb(indiv1[3]<indiv2[3]));

          ###BOUCLE PRINCIPALE: CROISEMENTS###
          #Boucle A de croisements
          for k from 1 to Croisement do
          #Boucle B de générations d'enfants
            for j from 1 to Nb_enf do

         #Selection de deux peres
              p1:=rand(1..25);
              puissance_p1:=p1();
              p2:=rand(1..25);
              puissance_p2:=p2();

         #On prends le meilleur
              if parents[puissance_p1][3]<parents[puissance_p2][3] then
                  pere:=puissance_p1;
           else
                  pere:=puissance_p2;
              fi;

         #On le met dans la liste des peres balezes
         new_list:=[op(new_list),[parents[pere][1],parents[pere][2],parents[pere][3]]];

         #Selection de deux meres
              m1:=rand(1..25);
              puissance_m1:=m1();
              m2:=rand(1..25);
              puissance_m2:=m2();

         #On prends la meilleure
         if puissance_m1<puissance_m2 then
                  mere:=puissance_m1;
              else
                  mere:=puissance_m2;
              fi;

         #On la met dans la liste des meres balezes
         new_list:=[op(new_list),[parents[mere][1],parents[mere][2],parents[mere][3]]];
```

```
#Génération d'un enfant par deux parents balezes qui sont ensuites retires de la liste initiale
    a:=parents[pere][1];
    b:=parents[mere][1];
    X_enf:=evalf((a+b)/2,8);

    c:=parents[pere][2];
    d:=parents[mere][2];
    Y_enf:=evalf((c+d)/2,8);

#Nouvelle liste d'enfants (avec un enfant aleat) qui vont devenir la nouvelle liste de parents
    subsop(puissance_p1=NULL,puissance_p2=NULL,parents);
    subsop(puissance_m1=NULL,puissance_m2=NULL,parents);
    Z_enf:=evalf(fonction_a_optimiser(X_enf,Y_enf),8);
    new_list:=[op(new_list),[X_enf,Y_enf,Z_enf]];
od;
#Fin de la Boucle B de générations d'enfants

new_list:=sort(new_list,(indiv3,indiv4)->evalb(indiv3[3]<=indiv4[3]));
parents:=[NULL,NULL,NULL];
parents:=new_list;
new_list:=[NULL,NULL,NULL];
parents;
od;
#Fin de la Boucle A de croisements
end:
```

```
STUDENT > Darwin(Rastrigin);
STUDENT >
[%1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1,
  %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1,
  %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1]
  %1 := [.0030625000, -.0076562500, .0134875]
STUDENT > Darwin(Rosenbrock);
[%1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1,
  %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1,
  %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1]
  %1 := [1.3186719, 1.7496407, .11309750]
STUDENT > Darwin(Himmelblau);
[%1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1,
  %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1,
  %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1, %1]
  %1 := [3.0025000, 1.9975000, .00021255558]
STUDENT >
```

Fourmi :

```
STUDENT > #Creation d'une variable aleatoire non uniforme pour avoir plus de proba de piocher un point au centre de la courbe
          gaussienne
          aleatoire_non_uniforme_v2:=proc(points,poids)
          local cible,total_en_cours,indice_en_cours,element_aleat,tot_poids,tireur_aleat,poids_v2,i,poids_norm;
          #somme de poids pour les points
          tot_poids:=sum('poids[i]','i'=1..nops(poids));
          #poids normalise par Gauss
          poids_norm:=[seq(tot_poids-poids[i],i=1..nops(poids))];
          #nouvelle somme des poids normalises
          tot_poids:=sum('poids_norm[i]','i'=1..nops(poids_norm));

          #tirage d un poids aleatoire
          tireur_aleat:=rand(1..floor(tot_poids));
          cible:=tireur_aleat();
          total_en_cours:=poids_norm[1];
          indice_en_cours:=2;
          #tant que poids total des points est inferieur au tirage aleatoire afin de prendre une valeur proche du centre de la
          gaussienne
          while total_en_cours < cible do
             total_en_cours:=total_en_cours+poids_norm[indice_en_cours];
             if(indice_en_cours > nops(poids_norm)) then
                break;
             fi;
             indice_en_cours:=indice_en_cours+1;
          od;
          #on retourne le point correspondant à l indice obtenu
          element_aleat:=points[indice_en_cours-1];
          element_aleat;
          end:
```

```
STUDENT > #Fonctions a optimiser
          Himmelblau:=proc(x,y)
              local f;
              f:=((x^2)+y-11)^2+(x+(y^2)-7)^2;
          end:
          Rastrigin:=proc(x,y)
              local f;
              f:=evalf(20+((x^2)-10*cos(2*Pi*x))+((y^2)-10*cos(2*Pi*y)));
          end:
          Rosenbrock:=proc(x,y)
              local f;
              f:=((1-x)^2)+100*((y-(x^2))^2);
          end:
```

```
STUDENT > #creation de 3 valeurs x,y,z randoms pour une fonction d optimisation
          GenR:=rand(-2000..2000)/1000;
          point_random:=proc(function_a_opt)
             local point;
             point:=[GenR(),GenR(),0];
             point[3]:=function_a_opt(point[1],point[2]);
             evalf(point,8);
          end:

          #BOUCLE PRINCIPALE
          fourmi:= proc(function_a_opt,n_iteration,N)
             local i,points,p1,p2,p3,lambda,compteur, poids, init;
             #generation de ces 3 valeurs randoms N fois
             init:=[seq(point_random(function_a_opt),i=1..N)];
             #classement par ordre croissant des 3 valeurs selon la valeur z
             init:=sort(init,(indiv1,indiv2)-> evalb(indiv1[3]<indiv2[3]));


             compteur:=0;
             while (compteur<n_iteration) do
                #extraction des deux premieres valeurs x,y correspondants aux points
                points:=[seq([op(1..2,init[i])],i=1..nops(init))];
                #extraction de la troisieme valeur z correspondant au poids
                poids:=[seq(init[i][3],i=1..nops(init))];
                #choix de deux points aleatoires
                p1:=aleatoire_non_uniforme_v2(points,poids);
                p2:=aleatoire_non_uniforme_v2(points,poids);
                #calcul du poids situe aleatoirement entre p1 et p2
                lambda:=evalf(rand(0..10000)()/10000);
                p3:=p1+lambda*p2;
                p3:=[op(p3),function_a_opt(p3[1],p3[2])];
                #creation d une nouvelle liste avec le nouveau poids du point p3 en z
                init:=[op(init),p3];
                init:=sort(init,(indiv1,indiv2)-> evalb(indiv1[3]<indiv2[3]));
                init:=[op(1..N,init)];
                #Pour voir l evolution des etapes de calcul, decommenter ci dessous
                #print(op(1..4,init));
                compteur:=compteur+1;
             od:
             init[1];
          end:
```

```
STUDENT > fourmi(Himmelblau,600,15);
```

[3.034942000, 2.275259539, 1.704657931]

```
STUDENT > fourmi(Rosenbrock,600,15);
```

[.9882827801, .9769478366, .0001432949190]

```
STUDENT > fourmi(Rastrigin,600,15);
```

[.00736398654, .05776894023, .665639877]