13

Codes sources

Certains programmes, en particulier SPSO et APS, ont été initialement codés en C. Pour plus de souplesse tout a été réécrit en Scilab (Scilab Enterprises [2014]) – au prix d'une rapidité d'exécution très inférieure – avec quelques modifications permettant d'illustrer divers points évoqués dans ce livre. Trois remarques à propos de ces codes :

– ce sont des outils de recherche. Ils fonctionnent, mais il est possible, voire probable, que des erreurs subsistent ;

– ils ne sont pas optimisés. C'est parfois volontaire, pour permettre une transcription plus facile en d'autres langages, mais beaucoup pourrait être réécrits de manière plus concise ou plus efficace. Ou les deux ;

– ils ne sont pas couverts par un droit d'auteur, mais si vous les utilisez, même modifiés, en particulier pour un travail publié, il serait de bon goût de citer leur origine, c'est-à-dire le présent ouvrage.

13.1. Générations aléatoires et échantillonnages

Générer des nombres pseudo-aléatoires est évidemment le premier outil nécessaire aux optimiseurs stochastiques. Un certain nombre de GNA sont codés ici, y compris ceux exploitant une liste cyclique.

13.1.1. *Instructions préalables pour les codes Scilab*

```
stacksize(30000000) ; // Augmentation de la mémoire
(utile si evalMax*runMax est grand)
global RND ; // valeur courante du nombre aléatoire généré
global SEED ;  // Pour GNA à graine
RND=1 ;// Attention : pas 0 pour Multiplicative et HP
SEED=%e ; // Doit être transcendant, comme %pi, %e ou sin(1), etc.
// Pour Multiplicative, doit aussi être supérieur à 1
```

```
// Pour GNA congruentiel (code 11)
global A
global B
global M
// À commenter ou décommenter selon l'option choisie
// A=69069 ; B=0 ; M=2^32 ; // Marsaglia
// A=1664525 ; B=1013904223 ; M=2^32 ; // Knuth & Lewis
A=137 ; B=187 ; M=2^8 ; // Cycle de 256 nombres
// Pour GNA à liste
global RANK
global L
L=100 ; // Longueur de liste par défaut  À redéfinir selon les besoins
// Pour les techniques No Man's Land
global knownPos
// Pour les problèmes Voyageur de commerce
global distances
distances=[] ;
```

13.1.2. *Tirage d'un nombre pseudo-aléatoire, selon options*

```
function rnd=alea(mini,maxi,randOption)
  global RND ;
  global SEED ;
  global A
  global B
  global M
  global L
  global RANK
  // randOption peut être soit un simple nombre
  // soit [rng, var, a,b]
  [dummy, nb]=size(randOption) ;
  if nb==1 then  // GNA simples
    select randOption
    case -2
      rnd=randTrue(0,32) ; // Hasard vrai (lecture du bruit matériel)
      // Le nombre de bits peut être 8, 16, 32 ou 64
    case -1
      rnd=randTrue(1,32) ; // Hasard presque vrai
      // Lecture du bruit matériel éventuellement complété par simulation,
      //pour aller plus vite
    case 0 // GNA par défaut de Scilab (Mersenne-Twister)
      rnd=rand(1,"uniform") ;
    case 1 //  Additive
      u=RND+SEED ;
      rnd=u-floor(u) ;
      RND=rnd ;
    case 11 // Générateur congruentiel
      // Attention, utilise les variables globales A, B, M
      // et RND
      rnd=A*RND + B ;
```

```
  rnd=modulo(rnd, M) ;
  RND=rnd ;
  rnd=rnd/(M-1) ; // sur {0,1]
case 12 // ANSI C
  // Attention à initialiser la variable globale RND
  a=1103515245 ; b=12345 ; m=2^16 ;
  rnd=a*RND + b ;
  rnd=modulo(rnd, m) ;
  if rnd==0 then rnd=1 ; end
  RND=rnd ;
  rnd=rnd/(2^16-1) ;
case 2 // Multiplicative
  // Attention. Bien initialiser les variables globales
  // RND=SEED= alpha, transcendant supérieur à 1
  // Plus alpha est grand, plus la distribution est uniforme
  u=RND*SEED ;
  rnd=u-floor(u) ;
  RND=rnd ;
case 3 // HP
  beta_=5 ;
  u=(RND +SEED)^beta_ ;
  rnd=u-floor(u) ;
  RND=rnd ;
case 1000 // Mersenne-Twister, liste de longueur L(variable globale)
  if RANK==L then
    RANK=0 ;
    rand("seed",123456789) ;
  end
  rnd=rand(1,"uniform") ;
  RANK=RANK+1 ;
case 1002 // Multplicative, liste de longueur L (variable globale)
  if RANK==L then
    RND=1 ;
    RANK=0 ;
  end
  u=RND*SEED ;
  rnd=u-floor(u) ;
  RND=rnd ;
  RANK=RANK+1 ;
case 2000 // Liste ultra-courte
  //L=17 ; // Variable globale à définir au préalable
  // Note : pour accélérer l'exécution, il vaudrait mieux
          //       mettre « liste » en variable globale et
          //       et ne la calculer qu'une seule fois au début
  select L
  case 3  // L3a
    liste=[0.66636,0.48627,0.00779] ;// OK pourTripod
  else
    printf("\n L=%i", L) ;
    error("Longueur de liste non implémentée pour le GNA 2000") ;
  end
  rnd=liste(RANK) ;
```

```
      RANK=RANK+1 ;
      if RANK>L then
        RANK=1 ;
      end
    case 2001 // Liste ultra-courte
      select L
      case 3 // L3b
        liste=[%pi/1000, sin(0.5)  ,%e/2.720] ;
      case 5 // L5b
        liste=listBuilder(5,%e*sin(1)) ;
      else
        printf("\n L=%i", L) ;
        error("Longueur de liste non implémentée pour le GNA 2001") ;
      end
      rnd=liste(RANK) ;
      RANK=RANK+1 ;
      if RANK>L then
        RANK=1 ;
      end
    else // GNA par défaut de Scilab
      rnd=rand(1,"uniform") ;
    end
  else // Variantes non uniformes
    // L'option est codée [rng, var,a,b]
    // a et b : voir aleaVar()
    if nb<4 then
      printf("\n alea : %i valeurs dans randOption.",nb) ;
      printf(" Il en faut une seule ou quatre. ") ;
  error(" ") ;
end
rnd=aleaVar(randOption(2),randOption(1),randOption(3),randOption(4)) ;
  end
  rnd=mini+(maxi-mini)*rnd ;
endfunction
```

13.1.3. *Hasard vrai*

code C (c'est le seul donné ici, tous les autres sont en Scilab)

```
f_rand = fopen("/dev/urandom", "r") ;
fread((&rInt),sizeof(rInt),1,f_rand) ;
fclose(f_rand) ;
r=fabs((double)rInt/INT_MAX) ;
```

code Scilab

```
function rnd=randTrue (u, nBits) // Lecture bruit matériel (sous Linux)
  if u==1 then
    [trueRand,err]=mopen("/dev/urandom","r") ;
```

```
    if err ~=0 then
      error("\n Impossible d'ouvrir le fichier /dev/urandom") ;
    end
  else // Attention, plus "vrai" mais TRÈS lent !
    [trueRand,err]=mopen("/dev/random","r") ;
    if err ~=0 then
      error("\n Impossible d'ouvrir le fichier /dev/random") ;
    end
  end
  select nBits
  case 8
    rnd=mget(1, 'uc', trueRand) ;
  case 16
    rnd=mget(1, 'us', trueRand) ;
  case 32
    rnd=mget(1, 'ui', trueRand) ;
  case 64
    rnd=mget(1, 'ul', trueRand) ;
  else
    rnd=mget(1, 'ui', trueRand) ; // 32 bits par défaut
  end
  rnd=rnd/(2^nBits -1) ; // Sur [0,1]
  mclose ;
endfunction
```

13.1.4. *Hasard manipulé*

```
function rnd=aleaVar(randVar,randOption,a,b)
  select randVar
  case 0  //Uniforme sur [0,1]
    rnd=alea(0,1,randOption) ;
  case 1  // Gaussienne
    rnd=aleaNormal(a,b,randOption) ;
  case 11 // Gaussienne tronquée à [0,1]
    rnd=min(1,max(0,aleaNormal(0.5,0.1,randOption))) ;
  case 2 // Cloche. Unimodale sur [0,1].
    //a=0.3 ;
    b=exp(-0.5*a^-2) ;
    u1=-1 ;
    while u1<=0
      u1=alea(b,1,randOption) ;
    end
    u2=alea(0,1,randOption) ;
    v=sqrt(-2*log(u1)) ;
    rnd=0.5*(1+a*v*cos(%pi*u2)) ;
  case 3 // Trimodale sur [0,1], calcul direct avec trois Cloches
    //a=0.3 ;
    b=exp(-0.5*a^-2) ;
    u1=alea(b,1,randOption) ;
    u2=alea(0,1,randOption) ;
```

```
  v=sqrt(-2*log(u1)) ;
  m=floor(alea(0,3,randOption)) ; // 0, 1 ou 2
  select m
  case 0
    rnd=0.5*abs(a*v*cos(%pi*u2)) ;
  case 1
    rnd=0.5*(1+0.5*a*v*cos(%pi*u2)) ;
  case 2
    rnd=1-0.5*abs(a*v*cos(%pi*u2)) ;
  end
case 4 // Trimodale avec trois gaussiennes
  m=floor(alea(0,3,randOption)) ; // 0, 1 ou 2
  rnd=-1 ;
  while rnd<0 | rnd>1
    select m
    case 0
      rnd=aleaNormal(0,0.1,randOption) ;
    case 1
      rnd=aleaNormal(0.5,0.05,randOption) ;
    case 2
      rnd=aleaNormal(1,0.1,randOption) ;
    end
  end
case 5 // Lévy
  rnd=randLevy(a,b) ;
case 6 // Gauss + uniforme
  m=floor(alea(0,2,randOption)) ;
  select m
  case 0
    rnd=-1 ;
    while rnd<0 | rnd>1
      rnd=aleaNormal(0,0.1,randOption) ;
    end
  case 1
    rnd=alea(0,1,randOption) ;
  end
case 7 // Cloche + uniforme
  m=floor(alea(0,2,randOption)) ;
  select m
  case 0
    b=exp(-0.5*a^-2) ;
    u1=-1 ;
    while u1<=0
      u1=alea(b,1,randOption) ;
    end
    u2=alea(0,1,randOption) ;
    v=sqrt(-2*log(u1)) ;
    rnd=0.5*(1+a*v*cos(%pi*u2)) ;
  case 1
    rnd=alea(0,1,randOption) ;
  end
case 8 // Log-normale
```

```
    auto=0 ;
    select auto
    case 1 // Attention, nécessite la bibliothèque Distfun
      rnd=distfun_lognrnd (a,b) ;
    case 0 // Moins précis, surtout pour les petites valeurs
      u1=alea(0,1,randOption) ;
      u2=alea(0,1,randOption) ;
      rnd=exp(-(log(u1)-a)^2/(2*b^2))/(u2*b*sqrt(2*%pi)) ;
    end
  end
endfunction
//------------------------------
function rnd=aleaNormal(mu,sigma,randOption)  // Gaussienne N(mu,sigma)
  u1=alea(0,1,randOption) ;
  u2=alea(0,1,randOption) ;
  rnd=sqrt(-2*log(u1))*cos(%pi*u2) ;
  rnd=mu+rnd*sigma ;
endfunction
//------------------------------
function r=randLevy(a,b)
  rng=0 ; // GNA implicite de Scilab
  erfc_1=erfinv(alea(0,1,rng)) ; // Identique en distribution
                                 // à erfinv(1-rand()) ;
  r=a + b/(2*(erfc_1^2)) ;
endfunction
//------------------------------
function liste=listBuilder(sizeL, seed)
  liste(1)=seed/(1000*(1+int(seed))) ;
  dx=1/(sizeL-1) ;
  for l=2 :sizeL
liste(l)= dx*(l-1)-liste(1)*l/sizeL ;
end
endfunction
```

13.1.4.1. *Approximation de la distribution de Lévy*

La difficulté est de programmer une fonction approchée pour erf^{-1}, réciproque de la fonction d'erreur de Gauss. Le logiciel Scilab l'intègre en natif (fonction `erfinv`), mais il peut être intéressant de voir comment en coder une soi-même avec des opérations de bas niveau, la transposition dans d'autres langages, tels C, étant alors facile. L'inconvénient majeur de la méthode ci-dessous est qu'il n'est pas possible en pratique de générer de très petits nombres. Même avec cent coefficients, tous les nombres générés seront supérieurs à 0,11. Ce défaut pourrait être pallié grâce à une génération spécifique approchée –par exemple de densité linéaire – entre zéro et ce minimum.

```
// Coefficients pour calcul approché de la fonction d'erreur inverse
function c=coeffeErfinv(n)
  c(1)=1 ;
  for k=2 :n // Niveau d'approximation (conseillé, au moins 100)
```

```
    // particulièrement si vous devez calculer
    // randlevy(u,c) avec u proche de 1
    c(k)=0 ;
    for m=1 :(k-1)
      c(k)=c(k)+c(m)*c(k-m)/(m*( 2*m-1 )) ;
    end
  end
endfunction
// Calcul approché de la fonction d'erreur inverse
// (pour comparer avec la fonction intégrée erfinv()
// Attention, même pour un degré élevé (taille de c),
// les approximations pour u proche de 1 sont mauvaises
function erf_1=erfinvapprox(u,c)
  sizec=size(c) ;
  n=sizec(1) ;
  erf_1=0 ;
  for k=1 :n
    erf_1=erf_1+(c(k)/(2*k-1))*((u*sqrt(%pi)/2))^(2*k-1) ;
  end
endfunction
// Calcul approché de Lévy
function levy=randlevyApprox(u,c,a,b)
  levy=a+b/(2*(erfinvapprox(u,c)^2)) ;
endfunction
```

13.1.5. *Initialisations uniformes (continues, combinatoires)*

```
//----------------------------- Initialisation uniforme d'une position
function pos=initPos(posMin,posMax,randOption,quantis,combin)
  [dummy,D]=size(posMin) ;
if combin then
//pos=grand(1,'prm',1 :D) ;
pos=permut(D,randOption) ; // Cas combinatoire
else
  for d=1 :D
if quantis(d)<=0 then
pos(1,d)=alea(posMin(d),posMax(d),randOption) ;
else
n=1+(posMax(d)-posMin(d))/quantis(d) ;
r=floor(alea(0,n,randOption)) ;
pos(1,d)=posMin(d)+r*quantis(d) ;
end
end
  end // Fin de "if combin
endfunction
```

13.1.6. *Initialisations régulières (Sobol, Halton)*

```
// Attention, ceci nécessite la bibliothèque
// Lowdisc  (low discrepancy)
function rnd=sobol(D,nb)
  rng = lowdisc_new("sobol") ;
  rng = lowdisc_configure(rng,"-dimension",D) ;
  [rng,rnd]=lowdisc_next(rng,nb) ;
  rng = lowdisc_destroy(rng) ;
endfunction
function rnd=halton(D,nb)
  rng = lowdisc_new("halton") ;
  rng = lowdisc_configure(rng,"-dimension",D) ;
  [rng,rnd]=lowdisc_next(rng,nb) ;
  rng = lowdisc_destroy(rng) ;
endfunction
// Affichage 2D d'une séquence générée
function plotSequ(rnd)
  scf() ;
  plot2d(rnd( :,1),rnd( :,2),axesflag=2, rect=[0,0,1,1]) ;
  a=get("current_axes") ;
  cpd=a.children ;
  p=cpd.children ; // Polyline
  p.line_mode="off" ;
  p.mark_mode="on" ;
  p.mark_style=0 ;
  p.mark_size=4 ;
  p.mark_foreground=2 ;
  p.mark_background=2 ;
endfunction
```

13.1.7. *Techniques* No Man's Land

```
// ------------------------------------ D-rectangle  No man's land
function [SSmin, SSmax]=noMansLand(pos, posMin, posMax)
// pos : liste des positions
// Renvoi le plus grand D-rectangle vide
  [S,dummy]=size(pos) ; // Taille de la population. Peut être nulle
if S==0 then
SSmin=posMin ;
SSmax=posMax ;
else
[dummy, D]=size(posMin) ; // Dimension
   for d=1 :D // Pour chaque dimension, on classe les coordonnées
// par ordre croissant
// et on cherche le plus grand intervalle
     x(1)=posMin(d) ;
     x(2 :S+1)=gsort(pos( :,d),"g","i") ;
     x(S+2)=posMax(d) ;
     delta=0 ;
```

```
    for s=1 :S+1
      dx=x(s+1)-x(s) ;
      if  dx> delta then
        delta=dx ;
        SSmin(1,d)=x(s) ; SSmax(1,d)=x(s+1) ;
      end
    end
  end
end
endfunction
//-----------------------------------------------------------
function pos=noMansLand2(knownPos, posMin,posMax)
// Ajoute une position "au plus loin" des autres
// On cherche à minimiser un potentiel
// par exemple avec l'optimiseur Minimalist
// (mais pourrait être remplacé par un autre)
// Attention knownPos est une variable globale pour Minimalist
[dummy,D]=size(posMin) ;
[S,dummy]=size(knownPos) ;
minimalist(11,40,500*D,10,%inf,0,0) ;
pos=knownPos ;
pos(S+1, :)=posBest ;
endfunction
//-----------------------------------------------------------
function  pos=noMansLandSeq(posInit, posMin, posMax, randOption,nb)
// Génère nb positions en plus de celles de posInit
// Chaque nouveau point est généré dans le plus grand no man's land
// laissé par les précédents
// Attention, dans cette version, ni la quantisation ni l'aspect
//  combinatoire éventuel ne sont pris en compte
// Pour générer à partir de rien (initialisation totale), l'appel est
// pos=noMansLandSeq([], posMin, posMax, randOption,nb)
[popSize, D]=size(posInit) ;
[dummy,D]=size(posMin) ;
pos=posInit ;
for n=1 :nb
[SSmin, SSmax]=noMansLand(pos, posMin, posMax) ;
//
pos(popSize+1,1 :D)=initPos(SSmin,SSmax,randOption,zeros(1 :D),%F) ;
pos(popSize+1,1 :D)=initPos(SSmin,SSmax,[0,2,0.3, 0],zeros(1 :D),%F) ;
popSize=popSize+1 ;
end
// Affichage 2D
plotSequ(pos) ;
endfunction
```

13.1.8. *Échantillonnages*

```
//--------------------------------- Tirage aléatoire dans une D-sphère
function pos=randSphere(centre,radius,nonUnif,randOption)
  // Renvoi un vecteur-ligne
```

```
  [line, D]=size(centre) ; // Sauf erreur, line=1
  //direction=rand(1,D,"normal")) ; // Direction aléatoire
  for d=1 :D direction(1,d)=aleaNormal(0,1,randOption) ; end
  if nonUnif==0 then  // Rayon aléatoire pour distribution uniforme
    //rad=radius*(rand(1,"uniform")^(1/D)) ;
    rad=radius*(alea(0,1,randOption)^(1/D)) ;
  else // Rayon aléatoire pour distribution NON uniforme
       // (plus dense vers le centre)
    // rad=radius*rand() ;
    rad=alea(0,radius,randOption) ;
  end
  normDir=norm(direction) ;
  // pos=centre + rad*direction/normDir ;
  for d=1 :D
    pos(1,d)=centre(1,d) + rad*direction(1,d)/normDir ;
  end
endfunction
```

La procédure `randAround` recherche au hasard un point situé à moins d'une certaine distance r d'un « centre », tout en étant dans l'espace de recherche. Par défaut, elle utilise la 2-distance (distance euclidienne) et, donc, tire un point au hasard dans une hypersphère. Néanmoins, comme il a été expliqué dans le chapitre 4, dès que la dimension D augmente, cette recherche peut être très longue, car la probabilité que ce point soit en dehors de l'espace de recherche est une fonction rapidement croissante de D et il faut alors itérer de nombreuses fois pour en trouver un d'acceptable. En fonction du paramètre `countMax`, l'algorithme peut tenter alors d'utiliser une distribution non uniforme (plus dense vers le centre), puis la 1-distance (qui donne à coup sûr une position acceptable), ou alors directement celle-ci.

```
//--------------------------- Sélection d'un point "autour" d'une position,
//                            avec différentes options
function pos=randAround(centre,position,posMin,...
posMax,countMax,randOption)
  [dummy, D]=size(centre) ;
  //countMax
  // -1 => uniquement selon la 1-distance
  // valeur positive (suggestion : 2*D) =>
  // a) au plus countMax tentatives avec la
  //       2-distance et distribution uniforme
  // b) si toujours pas de position acceptable,
  //    alors au plus countMax tentatives avec la
  //        2-distanc et distribution non uniforme
  //    c) si toujours pas de position acceptable,
  //       utilisation de la 1-distance
  // %inf => uniquement selon la 2-distance, à vos risques et périls
  //              car si D est grand cela peut être TRÈS long !
  // Note : si D=1, on pourrait penser obtenir le même résultat
  //        avec la 1-distance et la 2-distance.
  //     En fait, ce n'est pas le cas, à cause du confinement :
```

```
// - a posteriori avec la 2-distance
// - mais a priori avec la 1-distance
// Premières tentatives
radius=distance(centre,position) ;
inside=0 ;
count=0 ;
while inside==0 & count<=countMax
  pos=randSphere(centre,radius,0,randOption) ;
  count=count+1 ;
  inside=1 ;
  for d=1 :D
    if pos(d)<posMin(d) | pos(d)>posMax(d) then
      inside=0 ;
      break ;
    end
  end
end // Fin premières tentatives
// Secondes tentatives (n'est activé que si inside=0)
// On "triche" un peu, en utilisant une distribution NON uniforme
// (plus dense vers le centre)
count=0 ;
while inside==0 & count<=countMax
  pos=randSphere(centre,radius, 1,randOption) ;
  count=count+1 ;
  inside=1 ;
  for d=1 :D
    if pos(d)<posMin(d) | pos(d)>posMax(d) then
      inside=0 ;
      break ;
    end
  end
end // Fin secondes tentatives
// Recherche en 1-distance (activée si « inside » est toujours nul)
//  Avec cette distance, on est certain de tomber directement
// sur un point dans l'espace de recherche
if inside==0 then
 for d=1 :D
   z=position(d)-centre(d) ;
   if z>0 then
     mini=max(posMin(d),centre(d)-z) ;
     maxi=position(d) ;
   else
     mini=position(d) ;
     maxi=min(posMax(d),centre(d)-z) ;
   end
   //pos(1,d)= mini+rand(1,"uniform")*(maxi-mini) ;
   pos(1,d)= alea(mini,maxi,randOption) ;
 end
end
endfunction
```

13.1.9. *Déplacements et confinements*

```
//----------------------------- Quantification d'un vecteur de nombres
function q=quant(x0, quantis)
  // Le résultat est un vecteur-ligne
  // x doit être un vecteur-ligne
  // Test et redressement éventuel
  [a,b]=size(x0) ;
  if a>1 then x=x0', else x=x0 ; end
  [dummy,D]=size(x) ;
  for d=1 :D
    qd=quantis(d) ;
    if qd>0 then
      q(1,d)=qd*floor(0.5+x(d)/qd) ;
    else
      q(1,d)=x(d) ;
    end
  end
endfunction
//--------------------------------------------- Confinement d'une position
function x=confinPos(poss,posMin,posMax)
  // poss est un vecteur-ligne
  // La sortie x est un vecteur-ligne
  // Test et redressement éventuel
  [a,b]=size(poss) ;
  if a>1 then x=poss' ; else x=poss ; end
  [dummy,D]=size(poss) ;
  for d=1 :D
    if poss(d) <posMin(d) then
      x(1,d)=posMin(d) ;
    end
    if poss(d) >posMax(d) then
      x(1,d)=posMax(d) ;
    end
  end
endfunction
//-------------- Confinement d'une position avec modification de vitesse
// Utile uniquement pour SPSO et ses variantes
function [x,v]=confinPosVel(pos,vel,posMin,posMax,randOption)
  [dummy,D]=size(pos) ;
  x=pos ;
  v=vel ;
  for d=1 :D
    if pos(d) <posMin(d) then
      x(d)=posMin(d) ;
      //v(d)=0 ;  // SPSO 2007 originel
      v(d)=-alea(0,0.5,randOption)*vel(d) ; // Version modifiée
    end
    if pos(d) >posMax(d) then
      x(d)=posMax(d) ;
      // v(d)=0 ;  // SPSO 2007 originel
```

```
      v(d)=-alea(0,0.5,randOption)*vel(d) ; // Version modifiée
    end
  end
endfunction
```

13.2. Utilitaires

```
//----------- Distance euclidienne (norme 2) entre deux positions
function dist=distance(position1, position2)
// Les positions sont des vecteurs-lignes
differ=position1-position2 ;
dist=sqrt(differ*differ') ;
endfunction
//---------------------------------------------- Distance norme 1
function dist=distance1(position1, position2)
differ=position1-position2 ;
dist=max(abs(differ)) ;
endfunction
//---------------------------------------------- Volume d'un cube
function vol=volumCube(side, D)
vol=side^D ;
endfunction
//---------------------------------------------- Volume d'une sphère
function vol=volumSphere(radius, D)
d2=D/2 ;
vol=(radius^D)*(%pi^d2)/gamma(d2+1) ;
endfunction
```

13.3. Opérations combinatoires

```
//----------------------- Permutation aléatoire des entiers de 1 à N
function perm=permut(N,randOption)
  //perm=grand(1,'prm',1 :N) ;  Instruction standard en SciLab, mais
  // ATTENTION.  Sous certains systèmes d'exploitation (Linux Ubuntu 14.10)
  // l'initialisation "setsd" ne fonctionne pas correctement,ce qui empêche
  // de réeffectuer exactement la même série d'essais.
  // Dans ce cas, il vaut mieux utiliser le code donné ici
  temp=1 :N ; tempSize=N ;
  for n=1 :N
    // Rang aléatoire dans temp
    // r=1+floor(tempSize*rand(1,"uniform")) ;
    r=1+floor(tempSize*alea(0,1,randOption)) ;
    // Affecte la valeur de temp(r)
    perm(1,n)=temp(r) ;
    // Compacte temp
    if r<tempSize then
      for d=r :tempSize-1 temp(d)=temp(d+1) ; end
    end
    tempSize=tempSize-1 ;
```

```
  end
endfunction
//-----------------------------------------------------------
function pos=randAroundCombin(centre,position,randOption,distanceType)
  //distanceType=1 ; // 0 => Kendall-Tau
  // 1 => Cayley
  [dummy, D]=size(centre) ;
  // Distance
  radius=permutDist(centre,position,distanceType) ;
  // On choisit au hasard un nombre de transpositions
  if radius>1 then
    nbTrans=1+floor(alea(0,radius,randOption)) ;
  else
    nbTrans=1 ; radius=1 ;
  end
  // On effectue ces transpositions sur le "centre", au hasard
  // Avec Kendall-Tau, la méthode n'est pas garantie,
  //alors on boucle tant que la distance est supérieure au rayon
  dist=%inf ;
  while dist>radius
    pos=centre ; // Position initiale
    for n=1 :nbTrans
      p1=0 ; p2=0 ;
      while p1==p2  // Sélection de deux éléments différents
        p1=1+floor(alea(0,D,randOption)) ;
        p2=1+floor(alea(0,D,randOption)) ;
      end
      // Transposition
      temp=pos(p1) ; pos(p1)=pos(p2) ; pos(p2)=temp ;
    end
    if distanceType==0 then
      // Teste si la distance est OK
      dist=permutDist(centre,pos,distanceType) ;
      //if dist>radius then printf("\n trop loin") ; end
    else
      dist=radius-1 ; // Valeur arbitraire, juste pour terminer
                      // la boucle while
    end
  end
endfunction
//-----------------------------------------------------------
function pos=randLineCombin(centre,position,randOption)
  // Définit une permutation au hasard « entre » deux autres
  transpo=permutDecompCayley(centre,position,[]) ;
  [dummy, nTrans]=size(transpo) ;
  // On choisit au hasard un nombre de transpositions
  if nTrans>1 then
    n=floor(alea(1,nTrans,randOption)) ;
  else
    n=1 ;
  end
  // On effectue  ces transpositions sur le "centre",
```

```
  pos=transpoApply(centre,transpo( :,n)) ;
endfunction
//-------------------------------------------------------
function dist=permutDist(perm1,perm2,distanceType)
  // Distance entre deux permutations
  select distanceType
  case 0  // Distance de Kendall-Tau
    dist=permutDistKT(perm1,perm2) ;
  else  // Distance de Cayley
    dist=permutDistCayley(perm1,perm2) ;
  end
endfunction
//-------------------------------------------------------
function dist=permutDistKT(perm1,perm2)
  // Distance de Kendall-Tau
  // C'est le nombre de paires qui ne sont pas dans le même ordre
  // dans les deux permutations
  transpo=permutDecompKT(perm1,perm2) ;
  [dummy,dist]=size(transpo) ;
endfunction
//-------------------------------------------------------
function transpo=permutDecompKT(perm1,perm2)
  // Décomposition minimale d'une permutation en séquence de transposition
  // pour la distance de Kendall-Tau
  [dummy,D]=size(perm1) ;
  transpo=[] ;
  dist=0 ;
  for d1=1 :D-1  // Pour toutes les paires de perm1 ...
    for d2=d1+1 :D
      m1=perm1(d1) ;
      m2=perm1(d2) ;
      // ... on cherche dans perm2 le rang de son premier élément
      //      et le rang de son second élément
      for d3=1 :D
        if perm2(d3)==m1 then rank1=d3 ;
        else
          if perm2(d3)==m2 then rank2=d3 ; end
        end
      end
      if rank1>rank2 then
        dist=dist+1 ;
        transpo(1,dist)=d1 ;
        transpo(2,dist)=d2 ;
      end
    end
  end
endfunction
//-------------------------------------------------------
function dist=permutDistCayley(permutInit,permutEnd)
  // Distance de Cayley
  // Nombre minimum de transpositions pour passer
  // d'une permutation à l'autre
```

```
  // Plus longue à calculer, mais plus fine que Kendall-Tau
  transpo=permutDecompCayley(permutInit,permutEnd,[]) ;
  [dummy,dist]=size(transpo) ;
endfunction
//-----------------------------------------------------------
function transpo=permutDecompCayley(permutInit,permutEnd,transpo0)
  // Construit une séquence minimum de transpositions
  // pour aller d'une permutation de 1 :D à une autre de 1 :D
  // Attention, on appelle cette fonction récursive avec transpo=[]
  [dummy,D]=size(permutEnd) ;
  [dummy,p]=size(transpo0) ;
  permut=permutInit ;
  transpo=transpo0 ;
  // Si les permutations sont identiques, on ne fait rien
  ident=1 ;
  for k=1 :D
    if permutInit(k) ~=permutEnd(k) then
      ident=0 ;
      break ;
    end
  end
  // Sinon, k est le premier point non fixe,
  // On cherche la transposition qui replace
  // permutInit(k) en bonne position
  if ident==0 then
    p=p+1 ;
    d=k+1 ;
    while permutEnd(d)~=permutInit(k)
      d=d+1 ;
    end
    // Mémorise la transposition
    transpo(1,p)=k ;
    transpo(2,p)=d ;
    // Applique la transposition
    temp=permut(d) ;
    permut(d)=permut(k) ;
    permut(k)=temp ;
    //printf("\npermut ") ; for d=1 :D printf(" %i",permut(d)) ; end
    // Appel récursif
    transpo=permutDecompCayley(permut,permutEnd,transpo) ;
  end
endfunction
//-----------------------------------------------------------
function permut=transpoApply(permut,transpo)
  // Applique une séquence de transpositions à une permutation
  // Attention : pas de contrôle des tailles ni des valeurs,
  // on suppose qu'elles sont OK
  [dummy2,nbTranspo]=size(transpo) ;
  if nbTranspo>0 then
    for n=1 :nbTranspo
      r1=transpo(1,n) ;
      r2=transpo(2,n) ;
```

```
      temp=permut(r1) ;
      permut(r1)=permut(r2) ;
      permut(r2)=temp ;
    end
  end
endfunction
//-------------------------------------------------------------
function pr=probaTsp(D,noRepeat,t)
  // Pour un problème "Voyageur de commerce" symmétrique
  // avec une seule solution
  // D = nombre de villes
  // Probabilité de trouver cette solution en tirant au hasard :
  // noRepeat 0 => avec remise, on peut tirer plusieurs
  // fois la même permutation
  // noRepeat 1 => sans remise
  // t = nombre de tirages
  select noRepeat
  case 0
    pr=1-(1-2/factorial(D-1))^t ; // 2*D/factorial(D). 2 pour la symmétrie
    //et D pour les permutations cycliques
  case 1
    // if t>=factorial(D)-2*D then pr=1 ;
    //else
    if t==1 then
      pr=2/factorial(D-1) ;
    else
      z=2*D/(factorial(D)-t+1) ;
      pr=probaTsp(D,1,t-1)*(1-z)+z ;
    end
    //end
  end
endfunction
//-------------------------------------------------------------
function probaTspPlot(D,tMax)
  //Courbes de comparaisons des probabilités de succès avec et sans remise
  for t=1 :tMax
    pr0(t)=probaTsp(D,0,t) ;
    pr1(t)=probaTsp(D,1,t) ;
  end
  scf() ;
  plot2d([1 :tMax],[pr0,pr1],style=[2,3]) ;
  xtitle(" ","Nombre de tirages","Probabilité de succès") ;
  legends(["avec remise" ;"sans remise"],[2,3],1) ;
endfunction
```

13.4. Algorithme Aléatoire

```
function randomSearch(pb,evalMax,runMax,randOption,tracePos)
  // Exemple : randomSearch(3,1000,100,0,0)
  global RND ;
```

```
global SEED ;
global A
global B
global M
global L
global RANK
RANK=0 ; // Pour GNA à liste
rand("seed",123456789) ; // Initialisation de la graine,
//                          pour résultats reproductibles
// Définition du problème
[D,posMin0,posMax0,quantis,normalise]=pbDef(pb) ;
combin=pb>1000 ; // Problème combinatoire
global distances // Pour les problèmes « Voyageur de commerce »
// En cas de normalisation, l'espace de recherche est [0,1]^D
if normalise==0 then
  posMin=posMin0 ;
  posMax=posMax0 ;
else
  posMin=zeros(1,D) ;
  posMax=ones(1,D) ;
end
// Boucle sur les essais
for run=1 :runMax
  evalNb=0 ;
  costMin=%inf ;
  // Boucle sur les évaluations
  while evalNb<evalMax
    // Échantillonnage aléatoire
    pos= initPos(posMin,posMax,randOption,quantis,combin) ;
    cost=pbEval(pb,pos(1, :),posMin0,posMax0,normalise) ;
    evalNb=evalNb+1 ;
    if tracePos==1 then
      if normalise==0 then
        for d=1 :D  positions(evalNb,d)=pos(d) ; end
      else
        for d=1 :D
          positions(evalNb,d)=posMin0(d)+pos(d)*(posMax0(d)-posMin0(d)) ;
        end
      end
    end
    if cost<costMin then
      costMin=cost ; // Meilleure valeur
      bestPos=pos ;  // Meilleure position
    end
    costBest(run,evalNb)=costMin ;
  end // Fin boucle sur les évaluations
  // Affichage du résultat
  printf("\n Essai %i => %e",run,costBest(run,evalMax)) ;
  // Sauvegarde du résultat
  if normalise==0 then
    resultBest(run,1 :D)=bestPos ;
  else
```

```
      for d=1 :D
        resultBest(run,d)=posMin0(d) +bestPos(d)*(posMax0(d)-posMin0(d)) ;
      end
    end
    resultBest(run,D+1)=costMin ;
  end  // Fin boucle sur les essais
  printf("\nALÉATOIRE") ;
  printf("\nProblème %i,dimension %i",pb,D) ;
  if normalise==1 then printf("\nEspace de recherche normalisé") ; end
  printf("\n randOption %i",randOption) ;
  if randOption >=1000 then
    printf("\n  Liste cyclique de longueur %i",L) ;
  end
  printf("\n %i évaluations,%i essais",evalMax,runMax) ;
  printf("\n Minimum %e",min(costBest( :,evalMax))) ;
  printf("\n Moyenne %e",mean(costBest( :,evalMax))) ;
  printf("\n Écart-type %e",stdev(costBest( :,evalMax))) ;
  printf("\n Médiane %e",median(costBest( :,evalMax))) ;
  if tracePos==1 then
    [resultBest,costBest,positions]=resume(resultBest,costBest,positions) ;
  else
    [resultBest,costBest]=resume(resultBest,costBest) ;
  end
endfunction
```

13.5. Algorithme Minimaliste

```
function minimalist(pb,popSize0,evalMax,runMax,...
countMax,line,randOption)
  // Exemples
  // minimalist(310,40,5000,100,%inf,0,0) Alpine 10D,
  //   population de 40
  // minimalist(310,-1,5000,100,%inf,0,0)  Alpine 10D,
  //   population automatique
  // minimalist(3,40,1000,100,%inf,0,1000)  Alpine 10D, avec liste
  // Longueur de liste L, variable globale à initialiser
  // minimalist(3,40,10000,100,%inf,0,[0,6,0,0]) // GNA = Gauss+uniforme
  // minimalist(3,40,10000,100,%inf,0,[0,7,0.3,0]) // GNA = Cloche+uniforme
  // minimalist(11,40,1000,1,%inf,0,0) // Test initialisation.
  //             La variable globale knownPos doit être renseignée
  //  minimalist(3,9,100,100,-1,0,2000) // Test liste extra-courte de longueur L
  //                               (variable globale à initialiser).
  global RND ;
  global SEED ;
  global A
  global B
  global M
  global L
  global RANK
  global knownPos
  rand("seed",123456789) ; // Initialisation de la graine,
                           // pour résultats reproductibles
```

```
//randOption=0 ; Type de GNA. Voir alea()
//countMax=-1 ; // Voir randAround(). Sans effet si problème combinatoire
// Pour les problèmes combinatoires :
distanceType=1 ; // 0 => Kendall-Tau
//* 1 => Cayley  (préférable)
// line = 0 ; //  0 => autour d'une position
// 1 => entre deux positions
// Définition du problème
[D,posMin0,posMax0,quantis,normalise]=pbDef(pb) ;
combin=pb>1000 ; // Traitement particulier si problème combinatoire
toQuantis=sum(quantis)>0 ; // Pour discrétisation sur certaines dimensions
// En cas de normalisation, l'espace de recherche est [0,1]^D
if normalise==0 then
  posMin=posMin0 ;
  posMax=posMax0 ;
else
  posMin=zeros(1,D) ;
  posMax=ones(1,D) ;
end
// Taille de la population (si >0, doit être >=2)
if popSize0>0 then popSize=popSize0 ;
else
  select popSize0
  case -1  // Comme SPSO 2007
    popSize = floor(10 + 2 * sqrt(D)) ;
  case -2 // Comme APS standard
    popSize=sqrt(40*40 + (D+2)^2) ; z=40 + 2 * sqrt(D) ;
    popSize=floor(max(popSize,z)) ;
  end
end
// Boucle sur les essais
for run=1 :runMax
  RANK=1+modulo(run-1,L) ; // Pour les GNA à liste.
                           // Attention à avoir défini L au préalable
  // Initialisation aléatoire
  for pop=1 :popSize
    pos(pop,1 :D)=initPos(posMin,posMax,randOption,quantis,combin) ;
  end
  evalNb=0 ;
  cost=%inf*ones(1,popSize) ;
  best=1 ;
  for pop=1 :popSize // Évaluations initiales
    cost(pop)=pbEval(pb,pos(pop,1 :D),posMin0,posMax0,normalise) ;
    evalNb=evalNb+1 ;
    // Sauvegarde le meilleur résultat
    if cost(pop)<cost(best) then best=pop ; end ;
    costBest(run,evalNb)=cost(best) ;
  end
  // Boucle sur l'effort (nombre d'évaluations)
  while evalNb<=evalMax
    // Boucle sur la population
    for pop=1 :popSize // Pour chaque individu/position ...
      if pop==best then continue ; end
      //... définit une position...
```

```
      if combin then // Problème combinatoire
        if line==0 then
          posAttempt=randAroundCombin(pos(best, :), ...
                     pos(pop, :),randOption,distanceType);
        else
          posAttempt=randLineCombin(pos(best, :),pos(pop, :),...
                     randOption,distanceType);
        end
      else // Problème continu ou discret ou mixte
        if line==0 then
          posAttempt=randAround(pos(best, :),pos(pop, :),posMin,posMax,...
                     countMax,randOption);
        else
          posAttempt=randLine(pos(best, :),pos(pop, :),posMin,posMax,...
                     countMax,randOption);
        end
        // Discrétisation éventuelle
        posAttempt=quant(posAttempt,quantis);
        // Confinement, peut être nécessaire après la discrétisation
        if toQuantis then
          posAttempt=confinPos(posAttempt,posMin,posMax);
        end
      end
      // ... et l'évalue
      costAttempt=pbEval(pb,posAttempt,posMin0,posMax0,normalise);
      evalNb=evalNb+1;
      if evalNb>evalMax then break; end
      nEval(run,evalNb)=evalNb; // Données pour graphiques d'évolution
      costBest(run,evalNb)=cost(best);
      //  ... si amélioration, "déplace" l'individu sur la nouvelle position
      if costAttempt<cost(pop) then
        pos(pop, :)=posAttempt;
        cost(pop)=costAttempt;
      end
      //  ... si meilleure que la meilleure, la remplace
      if cost(pop)<cost(best) then best=pop; end
    end // Fin boucle sur les individus
    if evalNb>evalMax then break; end
  end // Fin boucle sur les évaluations
  //Affichage du résultat
  printf("\n Essai %i => %e",run,cost(best));
  // Sauvegarde du résultat
  resultBest(run, :)=pos(best, :); // Une ligne de D coordonnées par essai.
end  // Fin boucle sur les essais
printf("\nMINIMALISTE");
printf("\nProblème %i, dimension %i",pb,D);
if normalise==1 then printf("\nEspace de recherche normalisé"); end
printf("\n %i essais de %i évaluations",runMax,evalMax);
printf("\n Population  %i",popSize);
printf("\n countMax  %i",countMax);
printf("\n line %i",line);
[dummy,nb]=size(randOption);
printf("\n randOption  ");
for n=1 :nb printf(" %i",randOption(1,n)); end
```

```
  if randOption(1,1) >=1000 then
    printf("\n  Liste cyclique de longueur %i",L) ;
  end
  printf("\n Résultat =>") ;
  printf("\n Minimum %e",min(costBest( :,evalMax))) ;
  printf("\n Moyenne %e",mean(costBest( :,evalMax))) ;
  printf("\n Écart-type %e",stdev(costBest( :,evalMax))) ;
  printf("\n Médiane %e",median(costBest( :,evalMax))) ;
  [resultBest,costBest,posBest]=resume(resultBest,costBest,pos(best, :)) ;
endfunction
```

13.6. Algorithme SPSO

Il s'agit d'une version réécrite à partir du code C de SPSO 2007, disponible sur le *Particle Swarm Central* (PSC [2014]), avec quelques compléments, en particulier le traitement « naïf » des problèmes combinatoires dont l'espace de recherche est l'ensemble des permutations de $\{1, 2, \ldots, D\}$. Le code permet l'utilisation de la distance de Kendall-Tau ou celle de Cayley, mais cette dernière est plus pertinente pour remplacer les entités et les opérateurs classiques de SPSO, de la manière suivante :

Position $\rightarrow$ permutation de $\{1, 2, \ldots, D\}$.

Déplacement (souvent appelé « vitesse ») $\rightarrow$ séquence de transpositions $s = \left((r_{1,1}, r_{1,2}), (r_{2,1}, r_{1,2}), \ldots, \left(r_{L(s),1}, r_{L(s),2}\right)\right)$, où $L(s)$ est la longueur de la séquence et les $r_{i,j}$ des entiers donnant les rangs des éléments à transposer.

Application d'un déplacement s à une position $P \rightarrow s(P)$.

Différence de deux positions $P_1 - P_2 \rightarrow$ séquence minimale s de transpositions, telle que $s(P_1) = P_2$. Le résultat est donc un déplacement.

Somme de deux déplacements s_1 et $s_2 \rightarrow s = s_2 \circ s_1$ (les transpositions de s_1 puis celles de s_2). L'ordre à son importance, car l'opération n'est pas commutative.

Multiplication d'un déplacement s par un coefficient $\lambda \rightarrow$ si $\lambda \leq 1$ alors $k = \max(1, \lfloor \lambda L(s) \rfloor)$ sinon $k = \lfloor (\lambda - 1) L(s) \rfloor$.

```
// ----------------------------------------SPSO 2007
// modifié par Maurice Clerc, 2014
// Variante à partir du code source en C  disponible sur le
// Particle Swarm Central http://particleswarm.info
// Modifications :
//- méthode de confinement, en cas de sortie de l'espace de recherche,
//  non seulement la particule est ramenée sur une frontière
//  mais sa vitesse est inversée de manière partiellement aléatoire.
// - traitement de problèmes combinatoires type "Voyageur de commerce".
//     Attention, il s'agit d'un traitement rudimentaire
```

```
// juste pour montrer que, puisqu'un tel problème
//    est à corrélation positive, on peut faire mieux
//    que l'aléatoire en le traitant selon les mêmes
//    principes qu'un problème continu, en adaptant les opérateurs
// de la combinaison linéaire utilisée dans SPSO.
//    On ne cherche pas ici la performance,
//    mais l'illustration expérimentale d'un concept mathématique
//
function spso(pb,popSize,evalMax,runMax,tracePos,randOption)
  // Si popSize<0, on utilise son estimation automatique
  // en fonction de la dimension du problème
  // tracePos :  1 => sauvegarde toutes les positions, 0 sinon
  // Exemples :
  // spso(3,-1,1000,100,0,0)   // Alpine 2D, population calculée
  global RND;
  global SEED;
  global A
  global B
  global M
  global L
  global RANK
  // randOption// Type de GNA. Voir alea()
  nPos=0;
  rand("seed",123456789); // Initialisation de la graine,
  //                          pour résultats reproductibles
  grand("setgen","mt"); // Mersenne-Twister
  grand("setsd",123456789); // ATTENTION, cela ne fonctionne pas avec
  //                  certains systèmes d'exploitation (Ubuntu 14.10)
  // Il vaut alors mieux utiliser permut(N,randOption).
  // Cela est d'ailleurs nécessaire si l'on veut appliquer un GNA
  // non défini dans Scilab (il faut alors le définir dans alea())
  //
  // Définition du problème
  [D,posMin0,posMax0,quantis,normalise]=pbDef(pb);
  combin=pb>1000; // Pour problèmes combinatoires. Distance de Cayley
  toQuantis=sum(quantis)>0;
  if normalise==0 then
    posMin=posMin0;
    posMax=posMax0;
  else
    posMin=zeros(1,D);
    posMax=ones(1,D);
  end
  // Taille de l'essaim
  if popSize<0 then
    //S=sqrt(40*40 + (D+2)^2); z=40 + 2 * sqrt(D); // Formule d'APS
    // S=floor(max(S,z));
    S = floor(10 + 2 * sqrt(D)); // Formule d'origine de SPSO 2007
  else
    S=popSize;
  end
  disp(S,"Swarm size");
  // Paramètre p de la topologie variable
  K=3;  // C'est le nombre moyen approximatif d'informatrices
```

```
//        pour chaque particule
p=1-(1-1/S)^K;
// Note : pour simuler la topologie globale, faire p=1.
// Inertie et Coefficients de confiance
w = 1/(2 * log (2)); // 0.721.Inertie, tendance à suivre la même direction
c = 0.5 + log (2); // 1.193. Tendance conservatrice et tendance panurgienne
// Note : certaines variantes utilisent deux valeurs différentes
// Meilleur résultat sur l'ensemble des essais
bestBest=%inf;
// Index initial pour les rangs des particules
index=linspace(1,S,S);
//------------------------------------------ Boucle sur les essais
for run=1 :runMax
  RANK=1+modulo(run-1,L); // Pour les GNA à liste.
  //                        Attention à avoir défini L au préalable
  evalNb=0;
  cost=%inf*ones(1,S);
  best=1;
  clear pos;
  clear vel;
  //---------------------------------Initialisation des positions
  for s=1 :S
    pos(s,1 :D)=initPos(posMin,posMax,randOption,quantis,combin);
  end
  // -------------------------------Initialisation des vitesses
  if combin then  // Une "vitesse" est une liste de transpositions
    // On peut aussi mettre simplement transpo=[],
    // c'est-à-dire initialiser avec une "vitesse" nulle
    for s=1 :S
      transpo(s,1,1 :D)=permut(D,randOption); // Premier élément
                                              // des transpositions
      transpo(s,2,1 :D)=permut(D,randOption); // Second élément
      transpo(s,1,D+1)=D; // Longueur de la séquence
    end
  else   // L'idée est qu'aucune particule ne quitte
               // l'espace de recherche
               // dès la première itération
    for s=1 :S
      for d=1 :D
        vel(s,d)=(alea( posMin(d),posMax(d),randOption ) - pos(s,d))/2;
      end
    end
  end
  // Prise en compte du caractère discret éventuel de certaines variables
  if  ~combin then
    if toQuantis then
      for s=1 :S
        pos(s,1 :D)=quant(pos(s, :),quantis);
        // Confinement. Peut parfois (rarement) être nécessaire,
        // à cause du traitement précédent
        [x,v]=confinPosVel(pos(s, :),vel(s, :),posMin,posMax,randOption);
        //for d=1 :D pos(s,d)=x(d); vel(s,d)=v(d); end
        pos(s,1 :D)=x;
        vel(s,1 :D)=v;
```

```
    end
  end
end
// Optionnellement, sauve les positions, pour analyses à posteriori
// (signature, trajectoires, etc.)
if tracePos==1 then
  for s=1 :S
    nPos=nPos+1;
    if normalise==0 then
      for d=1 :D  positions(nPos,d)=pos(s,d); end
    else
      for d=1 :D
        positions(nPos,d)=posMin0(d)+pos(s,d)*(posMax0(d)-posMin0(d));
      end
    end
  end
end
// Évaluations
for s=1 :S
  cost(s)=pbEval(pb,pos(s, :),posMin0,posMax0,normalise);
  evalNb=evalNb+1;
  // Meilleure position mémorisée
  // On sauvegarde aussi sa valeur, dans l'élement D+1
  pBest(s,1 :D)=pos(s,1 :D);
  pBest(s,D+1)=cost(s);
  // Sauvegarde du résultat, pour traitements ultérieurs
  // (Eff-Rés, en particulier).
  nEval(run,evalNb)=evalNb;
  costBest(run,evalNb)=cost(best);
  // Rang "best" de la meilleure particule
  if cost(s)<cost(best) then best=s; end;
end // Fin d'initialisation
errorPrev=cost(best); // Meilleur résultat courant
initLinks=1; // Drapeau pour modifier ou non la topologie
//-------------------------------- Boucle sur les évaluations
while evalNb<=evalMax
  // Modifie la topologie. Qui informe qui, au hasard
  if initLinks==1
    for s = 1 :S
      for m=1 :S
        if  alea (0,1,randOption)<p  then
          LINKS(m,s) = 1; // Méthode probabiliste
        else LINKS(m,s) = 0;
        end
      end
      LINKS(s,s)=1; // Chaque particule s'informe elle-même
    end
  end
  // Permutation aléatoire des rangs des particules
  // Note : pas franchement nécessaire,
  // mais l'algorithme est ainsi un peu plus robuste
  // index=grand(1,'prm',index);
  index=permut(S,randOption);
  // ---------- Boucle sur les particules
```

```
for s0=1 :S // Pour chaque individu/position ...
  s=index(s0);
  if s==best then // La meilleure position ne bouge pas ...
    continue;
    // ... mais on pourrait aussi effectuer un "petit" mouvement aléatoire
    // < choix d'une autre position, pour définir le "rayon" du mouvement>
    //  < randAround ou randAroundCombin>
    // continue;
  end
  // Trouve la première informatrice. Son indice est s1
  s1=1;
  while LINKS(s1,s)==0
    s1=s1+1;
  end
  // Trouve la meilleure informatrice. Son indice est g.
  g=s1;
  for m=s1 :S
    if LINKS(m,s)==1 & cost(m)<=cost(g) then g=m;end
  end
  if ~combin then
    // Calcule et applique le déplacement
    for d=1 :D
      vel(s,d)=w*vel(s,d) +alea(0,c,randOption)*(pBest(s,d)-pos(s,d))...
               +alea(0,c,randOption)*(pBest(g,d)-pos(s,d));
      pos(s,d)=pos(s,d)+vel(s,d);
    end
    // Tient compte de la discrétisation
    if toQuantis then
      pos(s, :)=quant(pos(s, :),quantis);
    end
    // Confinement
    [x,v]=confinPosVel(pos(s, :),vel(s, :),posMin,posMax,randOption);
    //for d=1 :D pos(s,d)=x(d); vel(s,d)=v(d); end
    pos(s,1 :D)=x;
    vel(s,1 :D)=v;
  else // Cas combinatoire. Forme une séquence de transpositions
    // On utilise la distance de Cayley
    // (nb minimum de transpositions pour aller
    //   d'une permutation à une autre)
    // La prise en compte du coefficient c est rudimentaire,
    // on peut sûrement l'améliorer
    // On applique successivement les transpostions.
    // Attention, l'ordre est important (pas de commutativité)
    posPrev=pos(s, :);
    //------ Tendance panurgienne (aller vers la meilleure)
    transpoC=  permutDecompCayley(pos(s, :),pBest(g,1 :D),[]);
    [dummy2,tSize]=size(transpoC);
    if tSize>0 then
      ct=tSize*alea(0,c,randOption);
      ct=round(ct);
      if ct<tSize then  // On va de de pos(s) vers pBest(g)
        transpoC2=transpoC( :,1 :ct);
        pos(s, :)=transpoApply(pos(s, :),transpoC2);
      else  // On va "au-delà" de pBest(g)
```

```
      ct=modulo(ct,tSize);
      if ct>0 then
        transpoC2=transpoC( :,1 :ct);
        pos(s, :)=transpoApply(pBest(g,1 :D),transpoC2);
      else
        pos(s, :)=pBest(g,1 :D);
      end
    end
  end
  // -------Tendance conservatrive
  // (retourner vers la meilleure position de la particule)
  transpoC=  permutDecompCayley(pos(s, :),pBest(s,1 :D),[]);
  [dummy2,tSize]=size(transpoC);
  if tSize>0 then
    ct=tSize*alea(0,c,randOption); ct=round(ct);
    if ct<tSize then // On va de pos(s) vers pBest(s)
      transpoC1=transpoC( :,1 :ct);
      pos(s, :)=transpoApply(pos(s, :),transpoC1);
    else // On va "au-delà" de pBest(s)
      ct=modulo(ct,tSize);
      if ct>0 then
        transpoC1=transpoC( :,1 :ct);
        pos(s, :)=transpoApply(pBest(s,1 :D),transpoC1);
      else
        pos(s, :)=pBest(s,1 :D);
      end
    end
  end
  // Inertie (conserver la même "vitesse", mais en ralentissant)
  transpoSize=transpo(s,1,D+1);
  wt=floor(w*transpoSize);
  wt=max(wt,1); // Au moins une transposition
  for i=1 :2
    for j=1 :wt
      transpoW(i,j)=transpo(s,i,j);
    end
  end
  pos(s, :)=transpoApply(pos(s, :),transpoW);
  //-------- Recalcule la "vitesse"
  transpoTemp=permutDecompCayley(posPrev,pos(s, :),[]);
  [dummy2,tSize]=size(transpoTemp);
  for i=1 :2
    for j=1 :tSize
      transpo(s,i,j)=transpoTemp(i,j);
    end
    transpo(s,i,D+1)=tSize;// Pour mémoriser la longueur réelle
                          // de la séquence
  end
end  // Fin  if ~combin
// Optionnellement, sauve la position, pour analyses a posteriori
if tracePos==1 then
  nPos=nPos+1;
  if normalise==0 then
    for d=1 :D  positions(nPos,d)=pos(s,d); end
```

```
        else
          for d=1 :D
            positions(nPos,d)=posMin0(d)+pos(s,d)*(posMax0(d)-posMin0(d));
          end
        end
      end
      // Évalue la nouvelle position
      cost(s)=pbEval(pb,pos(s, :),posMin0,posMax0,normalise);
      evalNb=evalNb+1;
      // Test de fin
      // (en cas de fin, on aura fait une évaluation de trop pour rien,
      //  mais tant pis)
      if evalNb>evalMax then break; end
      // On continue
      nEval(run,evalNb)=evalNb; // Données pour traitements ultérieurs
      costBest(run,evalNb)=cost(best);
      // Met à jour la meilleure position mémorisée par la particule
      if  cost(s)<=pBest(s,D+1) then
        for d=1 :D  pBest(s,d)=pos(s,d); end
        pBest(s,D+1)=cost(s);
        //  ... si meilleure que la meilleure, la remplace
        if cost(s)<pBest(best,D+1) then best=s; end
      end
    end // Fin boucle sur les particules
    // Si pas d'amélioration on changera la topologie
    // pour la prochaine itération
    if pBest(best,D+1)<errorPrev then initLinks=1;
    else initLinks=0;
    end
    // if evalNb>evalMax then break; end
    errorPrev=pBest(best,D+1);
  end // Fin boucle sur les évaluations
  //
  // Sauvegarde du résultat
  if normalise==0 then
    resultBest(run, :)=pBest(best, :); // Une ligne de D coordonnées
    //                                    par essai + le résultat
  else
    for d=1 :D
      resultBest(run,d)=posMin0(d) +pBest(best,d)*(posMax0(d)-posMin0(d));
    end
    resultBest(run,D+1)=pBest(best,D+1);
  end
  if pBest(best,D+1)<bestBest then
    bestRun=run;
    bestBest=pBest(best,D+1);
  end
  // Affichage
  printf("\n Essai %i => %e",run,cost(best));
end  // Fin boucle sur les essais
printf("\nSPSO   ");
[dummy,nb]=size(randOption);
printf("\n randOption  ");
for n=1 :nb printf(" %i",randOption(1,n)); end
```

```
  if randOption(1,1) >=1000 then
    printf("\n  Liste cyclique de longueur %i",L);
  end
  printf("\nProblème %i, dimension %i",pb,D);
  printf("\n %i essais de %i évaluations",runMax,evalMax);
  printf("\n Taille de l'essaim %i",S);
  printf("\n Minimum : %e",min(costBest( :,evalMax)));
  printf("\n Moyenne  %e",mean(costBest( :,evalMax)));
  printf("\n Écart-type : %e",stdev(costBest( :,evalMax)));
  printf("\n Médiane : %e",median(costBest( :,evalMax)));
  if tracePos==1 then
    [resultBest,costBest,positions,bestRun]=...
     resume(resultBest,costBest,positions,bestRun);
  else
    [resultBest,costBest,bestRun]=resume(resultBest,costBest,bestRun);
  end
endfunction
```

13.7. Algorithme APS

```
// Adaptive Population-based Simplex (APS)
// modifié par Maurice Clerc, 2014
// Version Scilab à partir de la description et des codes disponibles
//  sur le site http://aps-optim.info
//
// Principales modifications
// - option d'initialisation par no man's lands successifs
//    - variantes de recherche locale
//    - problèmes combinatoires (rudimentaires)
// Le traitement d'un problème combinatoire est ici complètement calqué
// sur celui d'un problème continu. Il ne cherche pas à être particulièrement
// performant mais juste à mettre en évidence que c'est possible, avec des
// résultats meilleurs que l'aléatoire pur, du fait que même pour un problème
// combinatoire la corrélation distance-qualité est positive
//
global Smin;
global Smax;
function aps(pb, popSize,evalMax, runMax, tracePos, localSearch,countMax,...
             init,randOption)
  // suggestion : aps(pb, -5,evalMax, runMax, 0, -2,-1,0,0)
  global Smin;
  global Smax;
  global RND;
  global SEED;
  global A
  global B
  global M
  global L
  global RANK
  //
  // popSize    -1 => dépendant de la dimension (formule spécifique)
  //            -2 => dépendant de la dimension (même formule que SPSO)
  // -3 => population adaptative, nouvel individu au hasard
```

```
//     -4 => population adaptative, nouvel individu
//                  dans un no-man's land D-rectangle
// -5 => population adaptative, nouvel individu par mutation du meilleur
popAdapt=popSize== -3 | popSize== -4 | popSize== -5;
// tracePos :  1 => save all positions, 0 sinon
// randOption   Type de GNA. Voir alea()
// localSearch=0; // 0 => Recherche locale dans un petit D-rectangle
// 10 => dans le D-rectangle englobant la population
// (ne semble pas très bon)
// -1 => fait appel à la fonction randAround().
//       Nécessite de définir countMax
//* -2 => gaussienne dimension/dimension
// 1 => petite D-sphère
// 11 => grande D-sphère
// countMax=-1;    // Voir randAround().
// Pour les options suivantes, il vaut mieux mettre normalise=1
//   (ou que l'espace de recherche soit déjà un D-cube)
//  dans la définition du problème (voir pdDef())
// Attention, c'est aussi valable si localSearch=0 et countMax>0
// 1 => dans une petite D-sphère
//* 11 =>  dans une D-sphère englobante  (APS standard)
// init=0; // 0 => au hasard dans l'espace de recherche
// 1 => répartition aléatoire progressive
//       selon le plus grand  D-rectangle no man's land
expand=1; // 1, 2 ou 3. Voir la phase d'expansion probabiliste
nPos=0;
rand("seed",123456789); // Initialisation de la graine,
                        // pour résultats reproductibles
grand('setsd',123456789); // ATTENTION, cela ne fonctionne pas avec
//                    certains systèmes d'exploitation (Linux Ubuntu 14.10)
//                    Il vaut alors mieux utiliser permut(N,0)
zero=%eps; // Epsilon machine. C'est le "zéro".
//            Tout calcul impliquant des valeurs inférieures est douteux.
alpha=1; // Pour future adaptation dans la phase d'expansion.
         // Voir la version originelle d'APS
// Définition du problème
[D,posMin0,posMax0,quantis,normalise]=pbDef(pb);
factD=factorial(D); // Pour ne le calculer qu'une seule fois
combin=pb>1000; // Problème combinatoire.
//                 Une "position" est une permutation des entiers de 1 à D
// Note : calculs avec la distance de Cayley
toConfin=(popSize==-5 |  localSearch==-2 | localSearch==1...
         | localSearch==11) & ~combin;
toQuantis=sum(quantis)>0;
// En cas de normalisation, l'espace de recherche est [0,1]^D
if normalise==0 then
  posMin=posMin0;
  posMax=posMax0;
else
  posMin=zeros(1,D);
  posMax=ones(1,D);
end
// Taille de la population
Smin=2*(D+1);
```

```
Smax=%inf;
select popSize
case -1
  S0=sqrt(40*40 + (D+2)^2); z=40 + 2 * sqrt(D);
  S0=floor(max(S0,z));  // Formule d'APS
case -2
  S0 = max(D+1,floor(10 + 2 * sqrt(D))); // Inspiré de SPSO 2007
case -3 // Adaptative
  S0= max(D+1,floor(10 + 2 * sqrt(D)));  // Valeur initiale
case -4 // Adaptative
  S0= max(D+1,floor(10 + 2 * sqrt(D)));  // Valeur initiale
case -5 // Adaptative
  S0= max(D+1,floor(10 + 2 * sqrt(D)));  // Valeur initiale
else
  S0=popSize;
end
printf("\n Taille de la population : %i",S0);
// Meilleur résultat sur l'ensemble des essais
bestBest=%inf;
//------------------------------------------ Boucle sur les essais
for run=1 :runMax
  RANK=1+modulo(run-1,L); // Pour les GNA à liste.
  //                        Attention à avoir défini L au préalable
  S=S0;
  evalNb=0;
  cost=%inf*ones(1,S);
  best=1;
  clear pos;
  clear vel;
  sVolI=1;
  sVol(sVolI)=0;
  //--------------------------------Initialisation des positions
  select init
  case 0
    for s=1 :S
      pos(s,1 :D)=initPos(posMin,posMax,randOption,quantis,combin);
    end
  case 1
    pos(1,1 :D)=initPos(posMin,posMax,randOption,quantis,combin);
    for s=2 :S
      [SSmin, SSmax]=noMansLand(pos, posMin, posMax);
      pos(s,1 :D)=initPos(SSmin,SSmax,randOption,quantis,combin);
    end
  end
  // Optionnellement, sauve la position, pour analyses a posteriori
  // (signature, trajectoires, etc.)
  if tracePos==1 then
    for s=1 :S
      nPos=nPos+1;
      if normalise==0 then
        for d=1 :D  positions(nPos,d)=pos(s,d); end
      else
        for d=1 :D
          positions(nPos,d)=posMin0(d)+pos(s,d)*(posMax0(d)-posMin0(d));
```

```
      end
    end
  end
end
// Évaluation
for s=1 :S
  cost(s)=pbEval(pb,pos(s, :),posMin0,posMax0,normalise);
  evalNb=evalNb+1;
  // Sauvegarde du résultat, pour traitements ultérieurs
  // (Eff-Rés, en particulier).
  nEval(run,evalNb)=evalNb;
  costBest(run, evalNb)=cost(best);
  // Rang "best" de la meilleure particule
  if cost(s)<cost(best) then
    best=s;
    if cost(best)<bestBest then
      bestBest=cost(best);
    end
  end;
end
// printf("\n Résultat après initialisation : %e",cost(best));
// Cout total
costTot=sum(cost);
//-------------------------------- Boucle sur les évaluations
while evalNb<=evalMax
  costBestPrev=cost(best);
  improvNb=0;
  // ---------- Boucle sur les individus
  for s0=1 :S // Pour chaque individu/position ...
    if evalNb>=evalMax then break; end
    // Permutation aléatoire des rangs des individus
    clear index;
    //   index=grand(1,'prm',1 :S);
    index=permut(S,randOption);
    s=index(s0);
    // On s'assure que l'individu courant est dans les D+1 premiers.
    OK=0;
    for j=1 :D+1
      if  index(j)==s then
        OK=1;
        break;
      end
    end
    if OK==0 then index(D+1)=s; end
    // Définition du simplexe. Ce sont les D+1 individus
    // Les trois premiers serviront aux opérations Expansion et Contraction
    // Le premier servira à la recherche locale
    // Le simplexe complet ne servira qu'à calculer
    // un seuil de probabilité adaptatif
    // On classe les trois premiers, par ordre croissant de cout
    // (pour une minimisation)
    i1=1; i2=2; i3=3;
    if cost(index(i2))>cost(index(i3)) then
      j=i3; i3=i2; i2=j;
```

```
end
if cost(index(i1))>cost(index(i3)) then
  j=i3 ; i3=i1 ; i1=j ;
end
if cost(index(i1))>cost(index(i2)) then
  j=i2 ; i2=i1 ; i1=j ;
end
rBest=index(i1) ;
rWorst2=index(i2) ;
rWorst=index(i3) ;
// Sauvegarde du meilleur
if cost(rBest)<=cost(best) then
  best=rBest ;
end
if combin then
  pv=0.5 ; // Pour l'instant, en combinatoire, pas d'adaptation
else
  // Calcul de l'écart-type
  index2 = [rBest,rWorst,rWorst2] ;
  sMin = posMin ;
  sMax = posMax ;
  for d=1 :1 :D
    for j=1 :1 :3
      ri = index2(j) ;
      sMin(d) = min(sMin(d),pos(ri,d)) ;
      sMax(d) = max(sMax(d),pos(ri,d)) ;
    end
  end
  stDev = (posMax - posMin) - (sMax - sMin) ;
  // Adaptation du seuil de probabilité
  // L'idée est la suivante :
  // plus le volume du simplexe augmente, moins l'algorithme
  // est efficace et plus il faut utiliser d'aléatoire
  // pour augmenter la diversité. Et inversement.
  // On concatène les D+1 positions dans une matrice
  // et on ajoute une ligne de 1
  //printf("\nindex") ; for d=1 :D+1 ; printf(" %i ",index(d)) ; end
  clear M ;
  for j=1 :D+1
    for d=1 :D
      M(d,j)=pos(index(j),d) ;
    end
    M(D+1,j)=1 ;
  end
  // Calcul du volume du simplexe
  sVol(2-sVolI)=abs(det(M))/factD ;  // C'est un théorème
  // Calcul du seuil de probabilité
  if sVol(sVolI)<=zero then pv=0 ;
  else
    pv = (sVol(2-sVolI)-sVol(sVolI))/sVol(sVolI) ;
  end
  pv = 1/(1+exp(-pv)) ; // Logistic curve, so that pv is in [0,1]
  sVolI=2-sVolI ;
end
```

```
// ------------- Expansion probabiliste (quasi-réflexion)  => pos1
clear pos1;
if ~combin then
  for d=1 :D
    if alea(0,1,randOption)<pv then
      select expand
      case 1
        pos1(1,d)=pos(rBest,d)+alpha*(pos(rWorst2,d)-pos(rWorst,d));
      case 2
        z=(pos(rWorst2,d)+pos(rWorst,d))/2;
        pos1(1,d)=pos(rBest,d) +alpha*(z - pos(rBest,d));
      case 3
        z=(pos(rWorst2,d)+pos(rBest,d))/2;
        pos1(1,d)=z + alpha*(z-pos(rWorst,d));
      end
    else
      pos1(1,d)=pos(s,d);
    end
  end
  // Tient compte de la discrétisation
  // Note : comme pos1 est un vecteur-colonne, il faut le transposer
  if toQuantis then
    pos1=quant(pos1,quantis); // Le résultat est un vecteur-ligne
  end
  // Confinement
  if toConfin then
    pos1=confinPos(pos1,posMin,posMax); // la sortie est un vecteur-ligne
  end
else // Cas combinatoire
  // On définit le « milieu » de [pos(rWorst2),pos(rWorst)]
  transpo=permutDecompCayley(pos(rWorst2, :), pos(rWorst, :),[]);
  [dummy,dist]=size(transpo); // Note : Si dist=1, le « milieu »
  //                              sera en fait pos(rWorst2)
  nTrans=floor(dist/2);
  midPos=transpoApply(pos(rWorst2, :),transpo( :,1 :nTrans));
  // "Segment"  pos(rBest, :), midPos
  transpo=permutDecompCayley(pos(rBest, :), midPos,[]);
  [dummy,dist]=size(transpo);
  // On va un peu "au-delà" de midPos
  nTranspoRbestMid=max(1,round(pv*dist));
  n=floor(alea(0,nTranspoRbestMid,randOption));
  n=max(1,n);
  pos1=transpoApply(midPos,transpo( :,1 :n));
end
// Évalue la nouvelle position
cost1=pbEval(pb,pos1,posMin0,posMax0,normalise);
evalNb=evalNb+1;
// Optionnellement, sauve la position, pour analyses a posteriori
if tracePos==1 then
  nPos=nPos+1;
  if normalise==0 then
    for d=1 :D  positions(nPos,d)=pos1(d); end
  else
    for d=1 :D
```

```
      positions(nPos,d)=posMin0(d)+pos1(d)*(posMax0(d)-posMin0(d));
    end
  end
end
if cost1<cost(s) then
  improvNb=improvNb+1;
  improv=1;
  costTot=costTot-(cost(s)-cost1); // Si amélioration,
  //                                diminue le cout total
  for d=1 :D  pos(s,d)=pos1(d); end;
  cost(s)=cost1;
  if cost(s)<=cost(best) then
    best=s;
  end
else improv=0;
end
// Sauve la meilleure position courante, pour analyses ultérieures
costBest(run,evalNb)=cost(best);
// -----------  Si pas d'amélioration,
// contraction probabiliste=> pos2
if evalNb>=evalMax then break; end
if improv==0 then
  clear pos2;
  if ~combin then
    for d=1 :D
      if alea(0,1,randOption)<pv then
        pos2(1,d)=(pos(rBest,d)+pos(rWorst2,d)+pos(rWorst,d))/3;
      else
        pos2(1,d)=pos(s,d);
      end
    end
    // Tient compte de la discrétisation
    if toQuantis then
      pos2=quant(pos2,quantis);
      // Normalement, en contraction, il n'y a pas lieu de tester
      // le confinement, mais cela peut quand même être nécessaire
      // après discrétisation
      if toConfin then
        pos2=confinPos(pos2,posMin,posMax);
      end
    end
  else // Combinatoire
    n=floor(alea(0,nTranspoRbestMid,randOption));
    n=max(1,n);
    pos2=transpoApply(pos(rBest, :),transpo( :,1 :n));
  end
  // Évalue la nouvelle position
  cost2=pbEval(pb,pos2,posMin0,posMax0,normalise);
  evalNb=evalNb+1;
  // Optionnellement, sauve la position, pour analyses a posteriori
  if tracePos==1 then
    nPos=nPos+1;
    if normalise==0 then
      for d=1 :D  positions(nPos,d)=pos2(d); end
```

```
    else
      for d=1 :D
        positions(nPos,d)=posMin0(d)+pos2(d)*(posMax0(d)-posMin0(d));
      end
    end
  end
  if cost2<cost(s) then
    improvNb=improvNb+1;
    improv=1;
    costTot=costTot-(cost(s)-cost2); // Si amélioration,
    //                                  diminue le cout total
    for d=1 :D  pos(s,d)=pos2(d); end
    cost(s)=cost2;
    if cost(s)<=cost(best) then
      best=s;
    end
  else improv=0;
  end
  // Sauve la meilleure position courante, pour analyses ultérieures
  costBest(run, evalNb)=cost(best);
end  // Fin de "if improv==0" (Contraction)
// ---------------------- Si pas d'amélioration,
//                        recherche locale =>pos3
if evalNb>=evalMax then break; end
if improv==0 & cost(s)>=costTot/S then  // Si pas d'amélioration et
//                                        en-dessous de la moyenne
  clear pos3;
  if ~combin then
    if  localSearch==-2 then
      pos3=mutPop(pos(rBest, :),pv,stDev,randOption); // * "mutation"
    else
      pos3=around(pos, rBest, posMin,posMax,localSearch,randOption);
    end
    // Tient compte de la discrétisation
    if toQuantis then
      pos3=quant(pos3, quantis);
    end
    if toConfin then
      pos3=confinPos(pos3,posMin,posMax);
    end
  else // Combinatoire
    // Note : le choix de pos(rWorst2, :) comme position
    // pour calculer le « rayon » est expérimental
    //Un autre choix pourrait donner de meilleurs résultats
    pos3=randAroundCombin(pos(rBest, :),  pos(rWorst2, :), randOption,1);
            // Avec la distance de Cayley
  end
  // Évalue la nouvelle position
  cost3=pbEval(pb,pos3,posMin0,posMax0,normalise);
  evalNb=evalNb+1;
  // Optionnellement, sauve la position, pour analyses à posteriori
  if tracePos==1 then
    nPos=nPos+1;
    if normalise==0 then
```

```
        for d=1 :D  positions(nPos,d)=pos3(d); end
      else
        for d=1 :D
          positions(nPos,d)=posMin0(d)+pos3(d)*(posMax0(d)-posMin0(d));
        end
      end
    end
    if cost3<cost(s) then  improvNb=improvNb+1; end
    // Amélioration ou pas, on déplace l'individu
    for d=1 :D  pos(s,d)=pos3(d); end
    cost(s)=cost3;
    costTot=costTot-(cost(s)-cost3);
    if cost(s)<=cost(best) then
      best=s;
    end
    // Sauve la meilleure position courante, pour analyses ultérieures
    costBest(run, evalNb)=cost(best);
  end  // Fin de "if improv==0" pour recherche locale
end // Fin boucle sur les individus
if evalNb>=evalMax then break; end
if popAdapt then // Adaptation de la population
  if improvNb>0.75*S  & cost(best)<costBestPrev then
   // Si assez d'améliorations
    if S>Smin then   // ...  et si la population n'est pas trop petite ...
      // ... on cherche le rang du moins bon individu
      rw=1;
      for s=2 :S
        if cost(s)>cost(rw) then rw=s;end
      end
      // ... et on le supprime
      if rw<S then
        for s=rw :S-1
          pos(s, :)=pos(s+1, :);
          cost(s)=cost(s+1);
        end
      end
      S=S-1;
    printf("\n    Pop-1 => %i ",S); end
  end
  if improvNb<0.5*S & cost(best)>costBestPrev
    // Si pas suffisamment d'amélioration ...
    if S<Smax then // ... et population pas trop grande,
                   // alors ajouter un individu
      S=S+1;  printf("   Pop+1 => %i ",S);
      select popSize
      case -3 // Au hasard dans tout l'espace de recherche
       pos(S,1 :D)=initPos(posMin,posMax,randOption,quantis,combin);
      case -4 // Au hasard dans le plus grand D-rectangle no man's land
        [SSmin, SSmax]=noMansLand(pos,posMin, posMax);
        pos(S,1 :D)=initPos(SSmin,SSmax,randOption,quantis,combin);
      case -5 //* Mutation du meilleur
        mut=mutPop(pos(rBest, :),pv,stDev,randOption);
        mut=quant(mut, quantis);
        pos(S,1 :D)=confinPos(mut,posMin,posMax);
```

```
          end
          if toQuantis then
            pos(S, :)=quant(pos(S, :),quantis);
          end
          cost(S)=pbEval(pb,pos(S, :),posMin0,posMax0,normalise);
          evalNb=evalNb+1;
        end
      end
    end
  end // Fin boucle while sur les évaluations
  // Sauvegarde du résultat. Une ligne de D coordonnées par essai
  // + le résultat
  if normalise==0 then
    resultBest(run, 1 :D)=pos(best,1 :D);
  else
    for d=1 :D
      resultBest(run, d)=posMin0(d) +pos(best,d)*(posMax0(d)-posMin0(d));
    end
  end
  resultBest(run,D+1)=cost(best);
  // Affichage
  printf("\nEssai %i, => %e",run,cost(best));
  // Mémorisation du rang du meilleur essai
  if cost(best)<bestBest then
    bestRun=run;
    printf("\nMeilleur essai courant : %i",run);
    bestBest=cost(best);
  end
end  // Fin boucle sur les essais
printf("\nAPS");
printf("\nProblème %i, dimension %i",pb,D);
printf("\n %i essais de %i évaluations",runMax,evalMax);
printf("\n popSize %i",popSize);
printf("\n localSearch %i",localSearch);
printf("\n Smin %i, Smax %i",Smin,Smax);
printf("\n countMax %i",countMax);
[dummy,nb]=size(randOption);
printf("\n randOption  ");
for n=1 :nb printf(" %i", randOption(1,n)); end
if randOption(1,1) >=1000 then
  printf("\n  Liste cyclique de longueur %i",L);
end
printf("\nRésultats =>");
printf("\n Minimum : %e",min(costBest( :,evalMax)));
printf("\n Moyenne  %e",mean(costBest( :,evalMax)));
printf("\n Écart-type : %e",stdev(costBest( :,evalMax)));
printf("\n Médiane : %e",median(costBest( :,evalMax)));
if tracePos==1 then
  [resultBest, bestBest,  costBest,positions,bestRun]=...
   resume(resultBest, bestBest,costBest,positions,bestRun);
else
  [resultBest, bestBest,costBest,bestRun]=...
   resume(resultBest,bestBest,costBest,bestRun);
end
```

```
endfunction
//--------------------------------------------- Mutation d'une position
function  pos=mutPop(pos0,pv,stDev,randOption);
  [dummy,D]=size(pos0);
  for d=1 :D
    if (alea(0,1,randOption)<pv)
      pos(1,d)=pos0(d);
    else
      pos(1,d)=pos0(d)+ alea(0,stDev(d),3); // Distribution gaussienne
    end
  end
endfunction
//--------------------------------------------- Recherche locale
// Dans la version originale d'APS, elle est faite dans une D-sphère
// Ici, on peut  la faire dans un D-rectangle (voir l'option localSearch)
function   x=around(pos,rBest,posMin,posMax,localSearch,randOption)
  [S,D]=size(pos);
  alpha=1/3;
  select localSearch
  case 0 // Dans un petit D-rectangle
    for d=1 :D  // Pour chaque dimension ...
      //... trouve les coordonnées des individus les plus proches
      //  (ou bien les frontières)
      mini=posMin(d);  // Borne inférieure
      for s=1 :S
        if s==rBest then continue; end
        if pos(s,d)>pos(rBest) then continue; end
        if pos(s,d)>mini then mini=pos(s,d); end
      end
      maxi=posMax(d); // Borne supérieure
      for s=1 :S
        if s==rBest then continue; end
        if pos(s,d)<pos(rBest) then continue; end
        if pos(s,d)<maxi then maxi=pos(s,d); end
      end
      mini= pos(rBest,d)-(pos(rBest,d)-mini)*alpha;
      maxi=pos(rBest,d)+(maxi-pos(rBest,d))*alpha;
      x(d)=alea(mini,maxi,randOption);
    end
    x=x'; // On veut un vecteur-ligne
  case -1 // D-rectangle avec le plus proche en sommet
    radius=%inf;
    nearest=1;
    for s=1 :S
      if s==rBest then continue; end
      dist=distance(pos(rBest, :), pos(nearest, :));
      if dist<radius then nearest=s; radius=dist; end
    end
    x=randAround(pos(rBest, :) ,pos(nearest, :), posMin, posMax,-1,randOption);
    // case -2 //* Gaussienne dimension par dimension.
    //               Peut être vue comme une mutation
    //               Pour mémoire. Cas traité directement
    //               dans le programme principal aps()
  case 10 // Dans le D-rectangle englobant l'ensemble des positions
```

```
    // Note : apparemment peu efficace
    for d=1 :D
      mini(d)=min(pos( :,d));
      maxi(d)=max(pos( :,d));
      x(d)=alea(mini,maxi,randOption);
    end
    x=x';
  case 1  // Dans une  petite D-sphère
    radius=%inf;
    for s=1 :S
      if s==rBest then continue; end
      dist=distance(pos(rBest, :),pos(s, :));
      if dist<radius then radius=dist; end
    end
    nonUnif=0; // Uniforme
    x=randSphere(pos(rBest, :),radius,nonUnif,randOption);
  case 11 // Dans une D-sphère englobante (APS standard)
    radius=0;
    for s=1 :S
      if s==rBest then continue; end
      dist=distance(pos(rBest,1 :D), pos(s,1 :D));
      if dist>radius then radius=dist; end
    end
    // if alea(0,1,randOption)<0.5 then nonUnif=0; else nonUnif=1; end
    nonUnif=0;
    x=randSphere(pos(rBest,1 :D),radius,nonUnif,randOption);
  end // Fin de "select localSearch"
endfunction
```

13.8. Algorithme μPSO

```
//--------------------------------- microPSO
// par Maurice Clerc, 2014
// Version spécialement adaptée pour micro-robots
// À chaque itération, ceux-ci ont la même vitesse en valeur absolue
// qui, optionnellement, décroit au cours du temps
function micropso(pb,S,evalMax,runMax,tracePos,randOption)
  global SEED;
  global A
  global B
  global M
  global L
  global RANK
  // micropso(3, 10,100, 100, 0,0) // Alpine 2D, GNA Mersenne-Twister
  // global L; L=5;  micropso(3,10,100,L,0,2001) // Alpine 2D, GNA L5b
  // S=taille de la population
  // randOption// Type de GNA. Voir alea()
  nPos=0;
  rand("seed",123456789); // Initialisation de la graine,
                          // pour résultats reproductibles
  // Définition du problème.
  [D,posMin0,posMax0,quantis,normalise]=pbDef(pb);
  toQuantis=sum(quantis)>0;
```

```
if normalise==0 then
  posMin=posMin0;
  posMax=posMax0;
else
  posMin=zeros(1,D);
  posMax=ones(1,D);
end
// Paramètre K de la topologie variable
// Chaque robot/particule s'informe auprès des K plus proches
K=D+1;
if K>S-1 then
  printf("\n S=%i < K+1=%i",S,K+1);
  error("Revoir la taille de l''essaim et/ou le nombre d''informatrices");
end
// Paramètre de diminution ou non du coefficient d'inertie
wVar=0; //* 0 =>  toujours égal à w0
//          1 => décroissance au fil des itérations
w0=0.1; //* 0.1;   Si wVar=1, c'est la valeur initiale
// Coefficients de confiance
cMin=0.5;
cMax=1.5;
//---------------------------------- Boucle sur les essais
bestRun=1;
for run=1 :runMax
  evalNb=0;
  cost=%inf*ones(1,S);
  best=1;
  RANK=1+modulo(run-1,L); // Pour les GNA à liste.
  //                        Attention à avoir défini L au préalable
  w=w0; //  Si wVar=1, sera diminué pour simuler
  //        la perte d'énergie des robots/particules
  clear pos;
  clear vel;
  //--------------------------------Initialisation des positions
  for s=1 :S
    pos(s,1 :D)=initPos(posMin,posMax,randOption,quantis,%F);
  end
  // -------------------------------Initialisation des vitesses
  clear v
  diago=1;
  for d=1 :D
    diago=diago*(posMax(d)-posMin(d))
  end
  V=sqrt(diago)/S; // Le déplacement maximum est une fraction
                   // de la diagonale
  for s=1 :S
    for d=1 :D
      // v(d)=alea(0,1,randOption); // A. Direction aléatoire non uniforme
      v(d)=aleaNormal(0,1,randOption); // B. Direction aléatoire uniforme
    end
    vel(s, :)=v'*V/norm(v); // Longueur V
  end
  // Prise en compte du caractère discret éventuel de certaines variables
  if toQuantis then
```

```
  for s=1 :S
    pos(s,1 :D)=quant(pos(s, :), quantis);
  end
end
// Optionnellement, sauve les positions, pour analyses a posteriori
// (signature, trajectoires, etc.)
if tracePos==1 then
  for s=1 :S
    nPos=nPos+1;
    if normalise==0 then
      for d=1 :D  positions(nPos,d)=pos(s,d); end
    else
      for d=1 :D  positions(nPos,d)=posMin0(d)...
       +pos(s,d)*(posMax0(d)-posMin0(d)); end
    end
  end
end
// Évaluations
for s=1 :S
  cost(s)=pbEval(pb,pos(s, :),posMin0,posMax0,normalise);
  evalNb=evalNb+1;
  // Meilleure position mémorisée par la particule
  // On sauvegarde aussi sa valeur, dans l'élement D+1
  pBest(s,1 :D)=pos(s,1 :D);
  pBest(s,D+1)=cost(s);
  // Meilleure dans l'essaim
  if cost(s)<cost(best) then best=s; end;
  // Sauvegarde du résultat, pour traitements ultérieurs
  // (Eff-Rés, en particulier).
  costBest(run, evalNb)=cost(best);
end
// printf("\nMeilleur résultat aprês initialisation = %f",cost(best));
//--------------------------------- Boucle sur les évaluations
while evalNb<=evalMax
  // ---------- Boucle sur les particules
  for s=1 :S // Pour chaque individu/position
    // Trouve les K plus proches
    n=0;
    for s2=1 :S
      if s2==s then
        //pos(s, :)=pos(s, :)+w*vel(s, :);
        continue;
      end
      n=n+1;
      dist(n)=distance(pos(s, :),pos(s2, :));
      rankS(n)=s2;
    end
    [dist,r]=gsort(dist,"g","i"); // Classement par ordre croissant
                                  // de distance
    // Le vecteur r contient les rangs d'origine dans dist
    // Choisit la meilleure position mémorisée par ces K particules
    pBestBest=pBest(rankS(r(1)), :);
    kBest=1;
    for k=2 :K
```

```
    f=pBest(rankS(r(k)),D+1);
    if f<pBestBest(D+1) then
      pBestBest=pBest(rankS(r(k)), :);
      kBest=k;
    end
  end
  //  Calcule et applique le déplacement
  for d=1 :D
    rd=alea(cMin,cMax,randOption);
    xTemp(d)=pos(s,d)+vel(s,d)+rd*(pBestBest(d)-pos(s,d)-vel(s,d));
    v(d)=xTemp(d)-pos(s,d);
    //vel(s,d)=w*v(d);  // A
    vel(s,d)=w*vel(s,d)+ v(d); // B
  end
  //radius=distance(pos(s, :)+vel(s, :), pBestBest(1 :D))/2;
  //xTemp=randSphere(pBestBest, radius,0,randOption)
  pos(s, :)=pos(s, :)+vel(s, :);
  // Tient compte de la discrétisation
  if toQuantis then
    pos(s, :)=quant(pos(s, :), quantis);
  end
  // Confinement
  x=confinPos(pos(s, :),posMin,posMax);
  pos(s,1 :D)=x;
  // Optionnellement, sauve la position, pour analyses a posteriori
  if tracePos==1 then
    nPos=nPos+1;
    if normalise==0 then
      for d=1 :D  positions(nPos,d)=pos(s,d); end
    else
      for d=1 :D
        positions(nPos,d)=posMin0(d)+pos(s,d)*(posMax0(d)-posMin0(d));
      end
    end
  end
  // Évalue la nouvelle position
  cost(s)=pbEval(pb,pos(s, :),posMin0,posMax0,normalise);
  evalNb=evalNb+1
  // Test de fin
  // (en cas de fin, on aura fait une évaluation de trop pour rien,
  //  mais tant pis)
  if evalNb>evalMax then break; end
  // On continue
  // nEval(run,evalNb)=evalNb; // Données pour traitements ultérieurs
  // Met à jour la meilleure position mémorisée par la particule
  if  cost(s)<=pBest(s,D+1) then
    pBest(s,1 :D)=pos(s, :);
    pBest(s,D+1)=cost(s);
  end
  //  ... si meilleure que la meilleure, la remplace
  if cost(s)<pBest(best,D+1) then best=s; end;
  costBest(run, evalNb)=pBest(best,D+1); // Sauvegarde le meilleur
 //                                       résultat courant
end // Fin boucle sur les particules
```

```
      if wVar==1 then
        w=w0*sqrt(1-(evalNb-S)/evalMax); // Pour diminuer l'énergie
        //                                  des particules
      end
    end // Fin boucle sur les évaluations
    // Sauvegarde du résultat
    if normalise==0 then
      resultBest(run, :)=pBest(best, :); // Une ligne de D coordonnées
      //                                   par essai + le résultat
    else
      for d=1 :D
        resultBest(run, d)=posMin0(d) +pBest(best,d)*(posMax0(d)-posMin0(d));
      end
      resultBest(run,D+1)=pBest(best,D+1);
    end
    if resultBest(run,D+1)<resultBest(bestRun,D+1) then
      bestRun=run;
    end
    // Affichage
    printf("\n Essai %i => %e",run,resultBest(run,D+1));
    printf("  RANK %i",RANK);
  end  // Fin boucle sur les essais
  printf("\nmicroPSO   ");
  [dummy,nb]=size(randOption);
  printf("\n randOption  ");
  for n=1 :nb printf(" %i", randOption(1,n)); end
  if randOption(1,1) >=1000 then
    printf("\n  Liste cyclique de longueur %i",L);
  end
  printf("\nProblème %i, dimension %i",pb,D);
  printf("\n Taille de l'essaim %i",S);
  if wVar==0 then printf("\n Coefficient d'inertie constant = %f",w0);
  else printf("\n Coefficient d'inertie décroissant à partir de %f",w0);
  end
  printf("\nCoefficients de confiance %f et %f",cMin,cMax);
  printf("\n %i essais de %i évaluations",runMax,evalMax);
  printf("\n =>");
  printf("\n Minimum : %e",min(costBest( :,evalMax)));
  printf("\n Moyenne  %e",mean(costBest( :,evalMax)));
  printf("\n Écart-type : %e",stdev(costBest( :,evalMax)));
  printf("\n Médiane : %e",median(costBest( :,evalMax)));
  if tracePos==1 then
    [resultBest,costBest, positions,bestRun]=...
    resume(resultBest,costBest,positions,bestRun);
  else
    [resultBest,costBest, bestRun]=...
    resume(resultBest,costBest,bestRun);
  end
endfunction
```

13.9. Problèmes

Un problème est défini par :

– sa dimension D ;

– son espace de recherche, qui est un D-rectangle (sauf pour les problèmes combinatoires) ;

– éventuellement, pour certaines des dimensions, la liste des valeurs discrètes possibles ;

– une méthode pouvant assigner une valeur numérique à chaque point de l'espace de recherche.

Par ailleurs, dans certains cas, en particulier quand on veut estimer un taux de réussite, on donne une valeur minimale à atteindre et un seuil de tolérance. Enfin, parfois, il peut être utile de normaliser l'espace de recherche, en particulier si celui-ci n'est pas un D-cube et que l'algorithme fait usage de D-sphères.

13.9.1. *Définitions des problèmes*

```
//-------------------- Choix du problème
function [D,posMin,posMax, quantis,normalise]=pbDef(pb0 )
  normalise=0 ;
  global distances // Pour les problèmes "Voyageur de commerce"
  global knownPos  // Pour initialisation par no man's land
  select pb0
  case 2  // Rosenbrock pour différentes dimensions
    D=2 ; pb=2 ;
  case 203
    D=3 ; pb=2 ;
  case 204
    D=4, pb=2 ;
  case 210
    D=10, pb=2 ;
  case 3 // Alpine, pour différentes dimensions
    D=2 ; pb=3 ;
  case 310
    D=10 ; pb=3 ;
  case -310
    D=2 ;
  case 10 // Sphère
    D=30 ; pb=10 ;
  case 102
    D=2, pb=10 ;
  else
    pb=pb0 ;
  end
  select pb
    // Trompeuse 3
  case -4
```

```
  D=2;
  posMin=zeros(1,D);
  posMax=10*ones(1,D);
  quantis=zeros(1,D);
  // Trompeuse 2
case -3
  D=1;
  posMin=zeros(1,D);
  posMax=10*ones(1,D);
  quantis=zeros(1,D);
  // Trompeuse 1
case -2
  D=1;
  posMin=zeros(1,D);
  posMax=ones(1,D);
  quantis=zeros(1,D);
  // Trompeuse 11
case -21
  D=1;
  posMin=zeros(1,D);
  posMax=ones(1,D);
  quantis=zeros(1,D);
  // Plateau
case -1
  D=2;
  posMin=[-1,-1];
  posMax=-posMin;
  quantis=zeros(1,D);
  // Racine Carrée
case 0
  D=1;
  posMin(1)=0;
  posMax(1)=1;
  quantis=zeros(1,D);
  // Rosenbrock
case 2
  // D=4;
  posMin=-100*ones(1,D);
  posMax=-posMin;
  quantis=zeros(1,D);
  // Alpine
case 3
  posMin=zeros(1,D);
  posMax=4*D*ones(1,D);
  quantis=zeros(1,D);
case 4 // Enceinte sous pression
  // Solution (1.125,0.625,58.290155,43.692656) => 7197.72893
  D=4;
  posMin=[1.125,0.625,0,0];
  posMax=[12.5,12.5,240,240];
  quantis=[0.0625,0.0625,0,0];
  normalise=1;
case 5 // Tripod
  D=2;
```

```
  posMin=[-100,-100];
  posMax=-posMin;
  quantis=zeros(1,D);
case 10  // Paraboloïde (Sphère)
  //D=30;
  posMin=zeros(1,D);
  posMax=100*ones(1,D);
  quantis=zeros(1,D);
case 11 // Minimisation d'un potentiel pour initialisation
        // par No man's land
  // Attention : répéter ici les éléments du problème
  // réellement à résoudre
  D=2;
  posMin=zeros(1,D);
  posMax=ones(1,D);
  quantis=zeros(1,D);
case 12 // Train d'engrenages
  D=4;
  posMin=12*ones(1 :D);
  posMax=60*ones(1 :D);
  quantis=ones(1 :D);
  // Pour les codes > 1000, le problème est combinatoire
case 1001  // Voyageur de commerce, 6 villes
  // [ 1,6,2,3,4,5] => 20
  // Note : pour ce petit exemple, la matrice des distances
  // est directement codée ici
  D=6;
  posMin=ones(1,D);
  posMax=D*ones(1,D);
  quantis=ones(1,D);
  I=1000; // Grande valeur arbitraire
  distances=[0,9, I, I,3,5;
  9, 0,5, I, I,4;
  I, 5, 0,2, I,8;
  I, I, 2, 0,1,7;
  3, I, I, 1, 0, 5;
  5, 4, 8, 7, 5,0];
case 1002 // Voyageur de commerce, 14 villes
  //  [1,2,14,3,4,5,6,12,7,13,8,11,9,10]=> 3323
  [ftsp,err]=mopen("burma14.tsp","r");
  if err~=0 then error("Impossible d'ouvrir le fichier burma14.tsp"); end
  // Lit la dimension
  D=mfscanf(ftsp,"%i");
  posMin=ones(1,D);
  posMax=D*ones(1,D);
  quantis=ones(1,D);
  printf("\n Burma 14");
  // Lit la matrice contenant des données. Pour chaque ligne
  // numéro du nœud, latitude, longitude
  nodes=mfscanf(D,ftsp,"%i %lg %lg");
  printf("\n Construction de la matrice des distances. Un peu de patience, SVP");
  // Chaque donnée est sous la forme <degré>.<minute>
  for d=1 :D
    lat=nodes(d,2); long=nodes(d,3);
```

```
      degr(1)=floor(lat); degr(2)=floor(long);
      minute(1)=lat-degr(1); minute(2)=long-degr(2);
      latitude(d) = %pi*(degr(1)+5*minute(1)/3)/180;
      longitude(d) = %pi*(degr(2)+5*minute(2)/3)/180;
    end
    distances=[];
    R=6378.388; // Rayon moyen de la terre dans cette région
    for n1=1 :D-1
      for n2=n1+1 :D
        q(1)=cos( longitude(n1) -longitude(n2) );
        q(2)=cos( latitude(n1) -latitude(n2) );
        q(3)=cos( latitude(n1) +latitude(n2) );
        z=R*acos (0.5*((1+q(1))*q(2)-(1-q(1))*q(3)))+1;
        distances(n1,n2)=floor(z);
      end
    end
    // Symétrisation
    for n1=2 :D
      for n2=1 :n1-1
        distances(n1,n2)=distances(n2,n1);
      end
    end
    // Annulation de la diagonale
    for n1=1 :D  distances(n1,n1)=0; end
    printf("\n La matrice est construite");
    mclose(ftsp);
    //---------
  else
    "Attention, code problème faux"
    abort;
  end
  // Si espace de recherche normalisé, on normalise aussi la discrétisation
  if normalise>0 then
    for d=1 :D
      quantis(1,d)=quantis(d)/(posMax(d)-posMin(d));
    end
    printf("\n Espace de recherche normalisé");
  end
endfunction
//-------------------------------- Cout d'une position, selon le problème
function f=pbEval(pb0,position0,posMin0,posMax0,normalise);
  // "position" est censé être un vecteur-ligne [1,D]
  //   mais on teste par précaution
  [n1,n2]=size(position0);
  D=max(n1,n2)
  I= 1000; //%inf; pour Voyageur de commerce
  if normalise==0 then
    position=position0;
  else
    for d=1 :D
      position(1,d)=posMin0(d)+position0(d)*(posMax0(d)-posMin0(d));
    end
  end
  select pb0
```

```
case 2  // Rosenbrock pour différentes dimensions
  D=2 ; pb=2 ;
case 203
  D=3 ; pb=2 ;
case 204
  D=4, pb=2 ;
case 210
  D=10, pb=2 ;
case 3 // Alpine, pour différentes dimensions
  D=2 ; pb=3 ;
case 310
  D=10 ; pb=3 ;
case 10 // Sphère
  D=30 ; pb=10 ;
case 102
  D=2, pb=10 ;
else
  pb=pb0 ;
end
select pb
  // Trompeuse 3
case -4
  c1=10 ; c2=1 ;
  f=0 ;
  for d=1 :D
    xs=position(1,d)-d ;
    f=f+abs(xs*sin(c1*xs))+abs(xs)/c1 ;
  end
  f=min(f,c2) ;
  // Trompeuse 2
case -3
  c1=10 ;
  c2=1 ;
  u=position(1) ;
  f=1+sin(c1*u) +u/c1 ;
  f=min(f,c2) ;
  // Trompeuse 11
case -21
  c1=0.2 ;
  c2=0.5 ;
  u=position(1) ;
  if u<=c1 then f=u/c1 ;
  else
    if u<=2*c1 then
      f=1+(c2-1)*(u-c1)/c1 ;
    else
      f=c2 ;
    end
  end
  // Trompeuse 1
case -2
  c1=0.1 ;
  c2=0.5 ;
  u1=position(1) ;
```

```
  if u1<2*c1 | u1>5*c1 then f=c2;
  else
    u2=u1-2*c1;
    if u2<=c1 then f=c2*(1-u2/c1);
    else
      u3=u1-3*c1;
      if u3<c1 then f=u3/c1;
      else
        u4=u1-4*c1;
        f=1-c2*u4/c1;
      end
    end
  end
  // Plateau (impossible)
case -1
  f=1;
  // Racine carrée
case 0
  f=sqrt(position(1));
  // Rosenbrock
case 2
  f=rosenbrock(position);
  // Alpine
case 3
  f=alpine(position,1);
  // Enceinte sous pression
case 4
  target=7197.72893;
  x1=position(1);
  x2=position(2);
  x3=position(3);
  x4=position(4);
  f=0.6224*x1*x3*x4 + 1.7781*x2*x3*x3 + 3.1611*x1*x1*x4 + 19.84*x1*x1*x3;
  // Contraintes
  ff1=0.0193*x3-x1;
  ff2=0.00954*x3-x2;
  ff3=750*1728-%pi*x3*x3*(x4+(4/3)*x3);
  // Pénalités
  if ff1>0 then  f = f + 1e6; end
  if ff2>0 then  f = f + 1e6; end
  if ff3>0 then  f = f + 1e6; end
  f=abs(f-target);
case 5 // Tripod
  x1 = position(1);
  x2 = position(2);
  s11 = (1.0 - sign (x1)) / 2;
  s12 = (1.0 + sign (x1)) / 2;
  s21 = (1.0 - sign (x2)) / 2;
  s22 = (1.0 + sign (x2)) / 2;
  //f = s21 * (fabs (x1) - x2); // Solution en (0,0)
  f = s21 * (abs (x1) +abs(x2+50)); // Solution en (0,-50)
  f = f + s22 * (s11 * (1 + abs (x1 + 50) + ...
  abs (x2 - 50)) + s12 * (2 + abs (x1 - 50) + abs (x2 - 50)));
case 10 // Sphere
```

```
    f=0;
    for d=1 :D
      f=f+(position(d) -d)^2;
    end
  case 11 // Potentiel (pour initialiser une position dans un no man's land
    // Attention, knownPos est une variable globale
    f=potential(position,knownPos, posMin, posMax);
  case 12 // Train d'engrenages
    target=2.7e-12;
    //epsilon=1.e-13;
    a=6.931; b=2;
    f=(1/a -position(1)*position(2)/(position(3)*position(4)))^b;
    f=abs(f-target); // Erreur
    // Pour les codes > 1000, le problème est combinatoire
  case 1001  // Voyageur de commerce, D=6
    // La matrice des distances a été définie dans pbDef()
    // et est passée en tant que variable globale
    target=0; // En fait le minimum est 20
    // La position est une permutation
    f=distances(position(D), position(1));
    for d=1 :D-1
      f=f+distances(position(d),position(d+1));
    end
    f=f-target;  // Erreur
  case 1002 // Voyageur de commerce
    target=0; // En fait le minimum est 3323
    f=distances(position(D), position(1));
    for d=1 :D-1
      f=f+distances(position(d),position(d+1));
    end
    f=f-target;  // Erreur
  end
endfunction
//-------------------------------- Rosenbrock
function f=rosenbrock(position)
  [dummy,D]=size(position);
  f = 0;
  // t0 = position(1) + 1; // Solution on (0,...0) when offset=0
  target=0;
  t0=position(1); // Solution on (1,...,1)
  for  d=2 :D
    // t1 = position(d) + 1;
    t1 = position(d);
    tt = 1 - t0;
    f=f+ tt * tt;
    tt = t1 - t0 * t0;
    f = f+100 * tt * tt;
    t0 = t1;
  end
endfunction
//-------------------------------- Alpine
// shiftXY : décalage, pour que le minimum ne soit pas en (0,...0)
// Si shift=1, le minimum est x*=(1,2, ..; D), avec f(x*)=0
function f=alpine(x, shiftXY)
```

```
  [dummy,D]=size(x);
  f=0;
  for d=1 :D
    xs=x(d)-d*shiftXY;
    f=f+abs(xs*sin(xs))+0.1*abs(xs);
  end
endfunction
//------------------------------- Potentiel
function f=potential(pos,knownPos,posMin,posMax)
  // Calcule le « potentiel » d'une position en fonction de positions connues
  // et de la « boite » espace de recherche
  infinity=10^20;
  [dummy,D]=size(posMin); // Dimension
  [S,dummy]=size(knownPos); // Nombre d'autres positions (peut être nul)
  // Potentiel par rapport aux coins de la boite
  // p1=distCorner(pos,posMin,posMax);
  // Potentiel par rapport aux faces de la boite
  // p1=distFace(pos,posMin,posMax)
  // Attraction par le centre
  // ATTENTION, on suppose que l'espace de recherche est un D-cube
  // (le plus simple est de positionner normalise=1
  //  dans la définition du problème)
  distMax=0.5*(posMax(1)-posMin(1));
  centre=0.5*(posMin + posMax);
  //distCentre=distance1(pos,centre);
  distCentre=distance(pos,centre);
  if distCentre>=distMax then
    p1=infinity;
  else
    p1=1/(distMax-distCentre) -1/distMax;
    p1=D*p1;
  end
  // Potentiel par rapport aux autres positions
  p2=0;
  if S>0 then
    for s=1 :S
      //dist=distance1(pos,knownPos(s, :));
      dist=distance(pos,knownPos(s, :));
      if dist<%eps the d2=infinity;
      else p2=p2+1/dist;
      end
    end
  end
  f=p1+p2;
endfunction
function p1=distCorner(pos,posMin,posMax)
  // Calcul le potentiel en fonction des distances aux coins
  // Attention, peut être TRÈS long quand la dimension D augmente :
  //  y a 2^D coins!
  infinity=10^20;
  [dummy,D]=size(posMin);
  nCorner=2^D;
  p1=0;
  for n=1 :nCorner
```

```
    bin=dec2bin(n-1); // Conversion en chaine de caractères binaires
    lBin=length(bin); // Au plus D
    // Extraction des bits et transformation en min ou max
    posB=posMin;
    for l=1 :lBin
      if part(bin,l)=="1" then posB(lBin-l+1)=posMax(l); end
    end
    dist=distance(pos,posB);
    if dist<%eps then p1=infinity;
    else p1=p1+1/dist;
    end
  end
endfunction
function p1=distFace(pos,posMin,posMax)
  // Calcule le potentiel en fonction des distances aux faces
  // Il y a 2D faces
  infinity=10^20;
  [dummy,D]=size(posMin);
  weight=1;
  for d=1 :D weight=weight*(posMax(d)-posMin(d)); end
  p1=0;
  for d=1 :D
    dist=pos(d)-posMin(d);
    if dist<%eps then p1=infinity;
    else
      w=weight/(posMax(d)-posMin(d));
      p1=p1+w/dist;
    end
    dist=posMax(d)-pos(d);
    if dist<%eps then p1=infinity;
    else
      w=weight/(posMax(d)-posMin(d));
      p1=p1+w/dist;
    end
  end
endfunction
```

13.9.2. *Paysage d'un problème*

```
//---------------------------- // Paysage d'un problème 1D
function plotPb1D (pb,nx)
  [D,posMin, posMax, quantis,normalise]=pbDef(pb );
  X=linspace(posMin(1),posMax(1),nx);
  for n=1 :nx
    Y(n)=pbEval(pb,X(n),posMin,posMax,normalise);
  end
  scf();
  plot2d(X,Y,style=2);
  xtitle(" ","Position","Valeur");
endfunction
//--------------------------------------- Paysage d'un problème 2D
function plotPb2D (pb,nx,ny)
```

```
  clear X ;
  clear Y ;
  clear Z ;
  [D,posMin,posMax,quantis,normalise]=pbDef(pb) ;
  X=linspace(posMin(1,1),posMax(1,1),nx) ;
  Y=linspace(posMin(1,2),posMax(1,2),ny) ;
  for m=1 :nx
    for n=1 :ny
      pos=[X(m),Y(n)] ;
      Z(m,n)=pbEval(pb,pos,posMin,posMax,normalise) ;
    end
  end
  scf() ; // Nouvelle fenêtre graphique
  plot3d1(X,Y,Z) ;
  ax=gca() ;
  ax.cube_scaling="on" ;
  ax.z_label.font_angle=270 ;
  xtitle("","x","y","Valeur") ;
  fig=gcf() ;
  fig.color_map=pinkcolormap(32) ;
  pl=ax.children ;
  pl.thickness=0 ;
endfunction
```

13.10. Traitements des résultats

Chaque optimiseur, exécuté un certain nombre de fois, génère des résultats qui sont stockés dans le tableau `costBest`. On peut traiter ces résultats pour en déduire des indicateurs de performance, tenant éventuellement compte d'une fonction de qualité, à définir par l'utilisateur.

C'est aussi à partir de ces résultats bruts que l'on peut construire un Eff-Rés, en particulier pour calculer des indicateurs de performance globaux.

13.10.1. *Qualité (y.c. courbes)*

```
// ---------------------------------------- Qualité d'un résultat
function q=qualityResult(u,alpha,epsilon)
  // Attention, ici u est le résultat normalisé r/rMax
  // et, donc, le epsilon doit être en fait celui donné par l'utilisateur,
  // divisé par rMax
  if u<epsilon then
    q=1-0.5*(u/epsilon)^alpha ;
  else
    v=(u-epsilon)/(1-epsilon) ;
    b=(1-epsilon)*alpha/(2*epsilon) ;
    q=0.5*(1-v)^b ;
  end
```

```
endfunction
//------------------------------------ Qualité versus résultat normalisé
function plotQualityResult (alpha,epsilon)
  nuMax=100 ;
  u=linspace(0,1,nuMax) ;
  for nu=1 :nuMax
    q(nu)=qualityResult(u(nu),alpha,epsilon) ;
  end
  scf() ;
  plot2d(u,q, style=2) ;
  xtitle(" ","Résultat normalisé","Qualité") ;
endfunction
// ------------------------  Qualité versus résultat normalisé, 3 courbes
function plot3QualityResult(epsilon)
  nuMax=100 ;
  u=linspace(0,1,nuMax) ;
  for nu=1 :nuMax
    q1(nu)=qualityResult(u(nu),2,epsilon) ;
     // (modifier éventuellement) les valeurs de alpha)
    q2(nu)=qualityResult(u(nu),8,epsilon) ;
    q3(nu)=qualityResult(u(nu),100,epsilon) ;
  end
  scf() ;
  plot2d(u,[q1,q2,q3], style=[2,3,5]) ;
  xtitle(" ","Résultat normalisé","Qualité") ;
  legends(["2" ;"8 " ;"100"],[2,3,5],1) ; // Modifier la légende
endfunction
```

13.10.2. *Critères divers (y.c. courbes)*

```
//--- Évolution des résultats en fonction du nombre d'évaluations,
//    pour un essai donné
function plotResult(costBest,run)
  [runMax,phiMax]=size(costBest) ;
  absc=linspace(1,phiMax,phiMax) ;
  scf() ;
  plot2d(absc,costBest(run, :),style=2) ;
  xtitle(" ","Nombre d'évaluations","Résultat") ;
endfunction
//----  Évolution du résultat moyen en fonction de l'effort
function plotMeanResultEffort(costBest)
  // Attention, on suppose que tous les essais ont la même population
  // et le même nombre d'évaluations.
  [runMax,phiMax]=size(costBest) ;
  absc=linspace(1,phiMax,phiMax) ;
  for phi=1 :phiMax
    meanResultEffort(phi)=mean(costBest( :,phi)) ;
  end
  scf() ;
  plot2d(absc,meanResultEffort,style=2) ;
  xtitle(" ","Effort","Résultat moyen") ;
  [absc,meanResultEffort]=resume(absc,meanResultEffort) ;
```

```
endfunction
//--  Évolution du résultat médian en fonction de l'effort
function plotMedianResultEffort(costBest)
  // Attention, on suppose que tous les essais ont la même population
  // et le même nombre d'évaluations.
  [runMax,phiMax]=size(costBest);
  absc=linspace(1,phiMax,phiMax);
  for phi=1 :phiMax
    medianResultEffort(phi)=median(costBest( :,phi));
  end
  scf();
  plot2d(absc,medianResultEffort,style=2);
  xtitle(" ","Effort","Résultat médian");
  [absc,medianResultEffort]=resume(absc,medianResultEffort)
endfunction
//---- Évolution du résultat moyen en fonction du nombre d'essais
function plotMeanResultRuns(costBest)
  [runMax, phiMax]=size(costBest);
  clear meanResultRun;
  clear absc;
  for run=1 :runMax
    meanResultRun(run)=mean(costBest(1 :run,phiMax));
                // Le meilleur résultat de l'essai est pour l'effort maximum
    absc(run)=run;
  end
  scf();
  plot2d(absc,meanResultRun,style=2);
  xtitle(" ","Nombre d'essais","Résultat moyen");
  [absc,meanResultRun]=resume(absc,meanResultRun);
endfunction
//--------- Évolution du résultat médian en fonction du nombre d'essais
function plotMedianResultRuns(costBest)
  [runMax, phiMax]=size(costBest);
  clear medianResultRun;
  clear absc;
  for run=1 :runMax
    absc(run)=run;
    medianResultRun(run)=median(costBest(1 :run,phiMax));
  end
  scf();
  plot2d(absc,medianResultRun,style=2);
  xtitle(" ","Nombre d'essais","Résultat médian");
  [absc,medianResultRun]=resume(absc,medianResultRun);
endfunction
//------ Taux de réussite versus Nombre d'essais, pour un effort donné
function plotSuccessRate(costBest, phi, epsilon)
  // Attention, phi est un rang dans la liste des efforts
  // Mais si, comme c'est souvent le cas, l'effort est un
  // nombre d'évaluations à partir de 1, c'est aussi
  // la valeur même de cet effort
  // Exemple : plotSuccessRate(costBest,50,20.1)
  [runMax,phiMax]=size(costBest);
  absc=linspace(1,runMax,runMax);
  if costBest(1,phi)<epsilon then sr(1)=1; else sr(1)=0; end
```

```
  for n=2 :runMax
    if costBest(n,phi)<epsilon then
      sr(n)=sr(n-1)+1;
    else
      sr(n)=sr(n-1);
    end
  end
  for n=2 :runMax
    sr(n)=sr(n)/n;
  end
  rMin=floor(0.9*runMax);
  printf("\n Taux de réussite moyen sur les essais %i à %i : %f",...
   rMin,runMax,mean(sr(rMin :runMax)));
  scf();
  plot2d(absc,sr,style=2);
  xtitle(" ","Nombre d'essais","Taux de réussite");
  [absc,sr]=resume(absc,sr);
endfunction
// Taux de réussite généralisé versus Nombre d'essais, pour un effort donné
function plotSuccessRateAlpha(costBest,phi,epsilon, alpha)
  [runMax,phiMax]=size(costBest);
  absc=linspace(1,runMax,runMax);
  rMax=max(costBest);
  eps=epsilon/rMax;
  u=costBest(1,phi)/rMax;
  sr(1)=qualityResult(u,alpha,eps);
  for n=2 :runMax
    u=costBest(n,phi)/rMax;
    sr(n)=sr(n-1)+qualityResult(u,alpha,eps);
  end
  for n=2 :runMax
    sr(n)=sr(n)/n;
  end
  scf();
  plot2d(absc,sr,style=2);
  xtitle(" ","Nombre d'essais","Taux de réussite généralisé");
  [absc,srAlpha]=resume(absc,sr);
endfunction
//------------------- Taux de réussite généralisé moyen versus effort
function plotSuccessRateEffAlpha(costBest,epsilon,alpha)
  [runMax,phiMax]=size(costBest);
  rMax=max(costBest);
  eps=epsilon/rMax;
  for phi=1 :phiMax
    for run=1 :runMax
      u=costBest(run,phi)/rMax;
      sr(run)=qualityResult(u,alpha,eps);
    end
    srMean(phi)=mean(sr);
    absc(phi)=phi;
  end
  scf();
  plot2d(absc,srMean,style=2);
  xtitle(" ","Effort","Taux de réussite généralisé moyen");
```

```
  [absc,srMean]=resume(absc,srMean);
endfunction
//-------------------- Qualité globale, à partir des données brutes
function Q=globalQuality(costBest,alpha,epsilon)
  // Attention, on suppose ici que les efforts sont 1,2,...,ny
  [nRun, ny]=size(costBest);
  rMax=max(costBest);
  eps=epsilon/rMax;
  Q=0;
  for run=1 :nRun
    for phi=1 :ny
      // qualité(résultat) * effort
      Q=Q+qualityResult(costBest(run,phi)/rMax,alpha,eps)*phi;
    end
  end
  sumPhi=0.5*ny*(ny+1);
  Q=Q/(nRun*sumPhi);
  //disp(Q, "Qualité moyenne globale");
  Q=resume(Q);
endfunction
//------------------- Qualité globale versus epsilon
function plotGlobalQuality(costBest, alpha,epsilonMax)
  epsMax=100;
  epsilon=linspace(min(costBest),epsilonMax,epsMax);
  for e=2 :epsMax
    q(e-1)=globalQuality(costBest,alpha,epsilon(e));
    disp([q(e-1),e]);
  end
  scf();
  plot2d(epsilon(2 :epsMax),q,style=2);
  xtitle(" ","epsilon","Qualité globale");
  [epsilon,q]=resume(epsilon,q);
endfunction
//----- Efficacité normalisée globale, à partir des données brutes
function globalEffic(costBest)
  // Attention, on suppose que les efforts vont de 1 en 1
  [runMax, ny]=size(costBest);
  rMax=max(costBest);
  for phi=1 :ny
    meanr=mean(costBest( :,phi));
    effic(phi)=1-meanr/rMax;
  end
  EFF=mean(effic)
  disp(EFF,"Efficacité moyenne globale normalisée");
  EFF=resume(EFF);
endfunction
```

13.10.3. *Construction d'un Eff-Rés*

```
//-------------------------------- Construction de l'Eff-Rés (X,Y,Z)
function [X,Y,Z]=effRes(costBest,nx,rMax)
// X = résultats, Y  = efforts, Z = densité de probabilité
// Attention :
```

```
// - les efforts vont de 1 en 1 (ce sont des nombres d'évaluations)
// - l'effort minimum est 1
// - le résultat minimum est supposé être 0
// La matrice costBest contient une ligne par essai
    // et une colonne par nombre d'évaluations
  // nx = nombre de divisions de l'intervalle des résultats
  // Exemple :
  // [X,Y,Z]=effRes(costBest,100,1) ;  // N'oubliez-pas le point-virgule !
  clear X ;
  clear Y ;
  clear Z ;
  [runMax,ny]=size(costBest) ;
  //rMax=max(costBest) ;
  X=linspace(0,rMax,nx) ; // Définition des classes de résultats
  Y=linspace(1,ny,ny) ;  // On ne traite ici que ce cas particulier
  // Dans le cas général, on aurait des classes d'effort
  // contenant plus d'un élément.
  // Dans le cas continu, la valeur minimale serait zéro.
  // Attention, dans ce cas, le calcul du cout médian d'un effort
  // peut donner 0.
  Z=zeros(nx,ny) ;
  // Affectation des résultats dans les "cellules" de Z
  for run=1 :runMax
    for phi=1 :ny
      r=costBest(run, hi) ;
      if r<=rMax then
        rRank=1+floor((nx-1)*r/rMax) ;
        Z(rRank,phi)=Z(rRank,phi)+1 ;
      end
    end
  end
  // Normalisation
  dX=X(2)-X(1) ;
  dY=Y(2)-Y(1) ; // Juste pour le cas où les classes d'effort
  // contiennent plus d'un élément
  sumZ=sum(Z)*dX*dY ;
if sumZ>0 then
   Z=Z/sumZ ;
else
error("Toutes les densités sont nulles. Revoir nx et rMax") ;
end
  [X,Y,Z]=resume(X,Y,Z) ;
endfunction
//------------------------------- Test de validité de l'Eff-Rés
// et, si nécessaire, recherche dichotomique du
// nombre optimum d'intervalles de résultats
function effResDicho(costBest,rMax,X0,Y0,Z0)
  [nx,ny]=size(Z0) ;
  nxInf=1 ;
  nxSup=nx ;
  nxMid=floor(0.5*(nx+1)) ;
  while (nxSup-nxInf)>1
```

```
    disp(nxMid,"Nouvel nx") ;
    [X,Y,Z]=effRes(costBest, nxMid, rMax) ;
    OK=1 ;
    for r=1 :nxMid
      if sum(Z(r, :))<=0 then OK=0 ; break ; end
    end
    if OK==0 then
      nxSup=nxMid ;
      nxMid=floor(0.5*(nxInf+nxMid)) ;
    else
      nxInf=nxMid ;
      nxMid=floor(0.5*(nxMid+nxSup)) ;
    end
  end
  if OK==0 then
    [X,Y,Z]=effRes(costBest, nxMid-1, rMax) ;
  end
  [X,Y,Z]=resume(X,Y,Z) ;
endfunction
```

13.11. Traitements d'un Eff-Rés

L'objet Eff-Rés peut utilement être représenté graphiquement. Il permet aussi d'estimer des valeurs d'indicateurs de performance. Certains de ces indicateurs ont pu également être estimés à partir des résultats bruts. En elle-même, une différence entre les deux estimations est généralement un signe que le nombre d'essais n'a pas été suffisant pour construire une densité de probabilité relativement correcte. À la rigueur, on peut la reconstruire par interpolation, mais cela ne devrait être fait qu'en dernier recours, si vraiment il n'est pas possible de produire des résultats plus complets.

13.11.1. *Représentation graphique*

```
// ---------------------------------  Affichage 3D d'un Eff-Rés
function plotEffRes(X,Y,Z)
  scf() ; // Nouvelle fenêtre graphique
  plot3d1(X,Y,Z) ;
  ax=gca() ;
  ax.cube_scaling="on" ;
  ax.z_label.font_angle=270 ;
  xtitle("","Résultat","Effort","Densité de probabilité") ;
  fig=gcf() ;
  fig.color_map=pinkcolormap(32) ;
  pl=ax.children ;
  pl.thickness=0 ;
endfunction
```

13.11.2. *Interpolation*

```
//----------------------------------- Interpolation d'un Eff-Rés
function Z1=EffResInterpol(X,Y,Z)
//  On remplit les cases vides (densité zéro) en faisant la moyenne
// des valeurs non nulles les plus proches
// Attention, cela peut être TRÈS long
// À utiliser avec prudence, en particulier pour les fonctions multimodales
// Il est conseillé de lancer plotEffRes(X,Y,Z1), pour voir si l'Eff-Rés
// obtenu semble "plausible".
  [nx,ny]=size(Z) ;
  for r=1 :nx
    for phi=1 :ny
      if Z(r,phi)>0 then  Z1(r,phi)=Z(r,phi) ;
      else
        neighb=0 ;
        clear ZZ ;
        // On recrute les cases non vides les plus proches
        if r>1 then
          r1=r-1 ;
          while r1>0 & Z(r1,phi)<=0  then r1=r1-1 ; end
          if r1>0 then
            neighb=neighb+1 ;
            ZZ(neighb)=Z(r1,phi) ;
          end
        end
        if r<nx then
          r1=r+1 ;
          while r1<=nx & Z(r1,phi)<=0  then r1=r1+1 ; end
          if r1<=nx then
            neighb=neighb+1 ;
            ZZ(neighb)=Z(r1,phi) ;
          end
        end
        if phi>1 then
          phi1=phi-1 ;
          while phi1>0 &  Z(r,phi1)<=0 then phi1=phi1-1 ; end
          if phi1>0 then
            neighb=neighb+1 ;
            ZZ(neighb)=Z(r,phi1) ;
          end
        end
        if phi<ny then
          phi1=phi+1 ;
          while phi1<=ny & Z(r,phi1)<=0   then phi1=phi1+1 ; end
          if phi1<=ny then
            neighb=neighb+1 ;
            ZZ(neighb)=Z(r,phi1) ;
          end
        end
        if neighb==0 then Z1(r,phi)=0 ;
        else Z1(r,phi)=mean(ZZ) ; end
```

```
        //if neighb==0 then disp(phi,"phi",r,"r",Z1(r,phi),"Z") ; end
      end // end of "if Z(r,phi)>0
    end // end of "for phi
  end // end of "for r
  // Renormalisation
  dX=X(2)-X(1) ;
  dY=Y(2)-Y(1) ;
  Z1=Z1/(dX*dY*sum(Z1)) ;
  //plotEffRes(X,Y,Z1) ;
  [X,Y,Z1]=resume(X,Y,Z1) ;
endfunction
```

13.11.3. *Critères de performance (y.c. courbes)*

```
//  Distribution de probabilités des résultats pour un effort donné phi
function plotProbaDensityResult(phi,X,Y,Z)
// Attention :
// phi est en fait le rang dans la liste des efforts
// (d'ailleurs effectivement égal à l'effort, si la liste va de 1 en 1)
  [nx,ny]=size(Z) ;
  dX=X(2)-X(1) ;
  sumr=sum(Z( :,phi))*dX ;
  pr=Z( :,phi)/sumr ;
  scf() ;
  plot2d(X,pr,style=2) ;
  xtitle(" ","Résultat","Densité de probabilité") ;
  pr=resume(pr) ;
endfunction
// ---- Probabilités des résultats pour un effort donné phi
function plotProbaResult(phi,X,Z)
  //       (fonction de répartition)
  // Note : pour chaque r, on considère la probabilité
  // d'avoir un résultat inférieur ou égal
  // Attention, phi est supposé être un nombre d'évaluations
  [nx,ny]=size(Z) ;
  sumr=sum(Z( :,phi)) ;
  pr=Z( :,phi)/sumr ;
  for r=2 :nx
    pr(r)=pr(r-1)+pr(r) ;
  end
  scf() ;
  plot2d(X,pr,style=2) ;
  xtitle(" ","Résultat","Probabilité") ;
  pr=resume(pr) ;
endfunction
//---- Distribution de probabilité des efforts pour un résultat donné r
function plotProbaDensityEffort(r,Y,Z)
// Attention : r est le _rang_ du résultat dans la liste X des résultats
  [nx,ny]=size(Z) ;
  dY=Y(2)-Y(1) ;
  for phi=1 :ny
    pr(phi)=Z(r,Y(phi)) ;
```

```
  end
  sumpr=sum(pr)*dY;
  pr=pr/sumpr;
  scf();
  plot2d(Y,pr,style=2);
  xtitle(" ","Effort","Densité de probabilité");
  pr=resume(pr);
endfunction
//----------------- Probabilité des efforts pour un résultat égal à r
function plotProbaEffort(r,Y,Z)
// Attention : r est le _rang_ du résultat dans la liste X des résultats
  [nx,ny]=size(Z);
  for phi=1 :ny
    pr(phi)=Z(r,Y(phi));
  end
  for phi=2 :ny
    pr(phi)=pr(phi-1)+pr(phi);
  end
  pr=pr/pr(ny);
  scf();
  plot2d(Y,pr,style=2);
  xtitle(" ","Effort","Probabilité");
  pr=resume(pr);
endfunction
// - Probabilités des efforts pour un résultat donné  inférieur ou égal à r
function plotProbaEffort2(r,Y,Z)
  //       (fonction de répartition)
  // Attention : r est le _rang_ du résultat dans la liste X des résultats
  [nx,ny]=size(Z);
  dX=X(2)-X(1);
  for phi=1 :ny
    pr(phi)=sum(Z(1 :r,Y(phi)))*dX;
  end
  for r=2 :nx
    pr(r)=pr(r-1)+pr(r);
  end
  scf();
  plot2d(Y,pr,style=2);
  xtitle(" ","Effort","Probabilité");
  pr=resume(pr);
endfunction
// ------------------ Cout moyen versus résultat r
function plotMeanCostResult(X,Y,Z)
  // Attention, on suppose que aucun sum(Z(r,1 :ny)) n'est nul
  // voir effResDicho()
  [nx,ny]=size(Z);
  for r=1 :nx
    meanc(r)=Y*Z(r, :)'/sum(Z(r, :));
  end
  scf();
  plot2d(X,meanc,style=2);
  xtitle(" ","Résultat","Cout moyen");
  meanc=resume(meanc);
  //-------------------- Qualité globale, à partir de l'Eff-Rés
```

```
  function globalQualityEffRes(X,Y,Z,alpha,epsilon)
    [nx,ny]=size(Z);
    dX=X(2)-X(1);
    dY=Y(2)-Y(1);
    rMax=X(nx);
    eps=epsilon/rMax;
    Q=0;
    sumPhi=0;
    for phi=1 :ny  // Pour chaque effort ...
      m=0;
      // ... résultat moyen
      for r=1 :nx-1
        // Quand on a créé l'Eff-Rés,
        // le nombre de résultats dans cette classe était Z(r,phi)*dX*dY
        result=0.5*(X(r)+X(r+1));
        q=qualityResult(result/rMax,alpha,eps);
        m=m+q*Z(r,phi);
      end
      Q=Q+m*Y(phi);
      sumPhi=sumPhi+Y(phi)*sum(Z(1 :nx-1,phi));
    end
    Q=Q/sumPhi;
    disp(Q, "Qualité moyennne globale, à partir de l'Eff-Rés");
    Q=resume(Q);
  endfunction
  //------------- Efficacité normalisée globale, à partir de l'Eff-Rés
 function globalEfficEffRes(X,Y,Z)
    [nx,ny]=size(Z);
    // Attention, on suppose que les efforts vont de 1 en 1
    // donc que Y(phi)=phi et que l'Eff-Rés est « complet »
    // (pour chaque effort, il y a au moins un résultat)
    dY=Y(2)-Y(1);
    ind=0;
    for  phi=1 :ny
      effic(phi)=1-X*Z( :,phi)/sum(Z( :,phi));
    end
    EFF=sum(effic)*dY/(Y(ny)-Y(1));
    disp(EFF,"Efficacité moyenne globale normalisée");
    EFF=resume(EFF);
  endfunction
// ------------------------------- Cout moyen versus résultat  <=r
function plotMeanCostResultInf(X,Y,Z)
  // Attention : le temps de calcul est assez long
  [nx,ny]=size(Z);
  for r=1 :nx
    meancinf(r)=0;
    sumcr=0;
    for rinf=1 :r
      for phi=1 :ny
        meancinf(r)=meancinf(r)+Y(phi)*Z(rinf,phi);
      end
      sumcr=sumcr+sum(Z(rinf,1 :ny));
    end
    if  sumcr>0 then
```

```
      meancinf(r)=meancinf(r)/sumcr;
    else
      meancinf(r)=[]; //Y(ny);
       // Attention, dans ce cas, c'est que l'échantillonnage
       // n'est pas assez fin.
       // On peut aussi, par défaut, mettre l'effort maximum,
       // mais ce peut être en fait n'importe quelle valeur encore plus grande.
    end
  end
  scf();
  plot2d(X,meancinf,style=2);
  xtitle(" ","Résultat","Cout moyen pour <= Résultat");
  meancinf=resume(meancinf);
endfunction
//---------------------------- Efficacité normalisé versus effort
function  efficEff(X,Y,Z)
  [nx,ny]=size(Z);
  for  phi=1 :ny
    effic(phi)=1-X*Z( :,phi)/sum(Z( :,phi));
  end
  scf();
  plot2d(Y,effic,style=2);
  xtitle(" ","Effort","Efficacité normalisée");
  effic=resume(effic);
endfunction
//---------------------------- Efficacité marginale versus effort
function  efficMarg(X,Y,Z)
  [nx,ny]=size(Z);
  dY=Y(2)-Y(1);
  for  phi=1 :ny
    effic(phi)=1-X*Z( :,phi)/sum(Z( :,phi));
  end
  for phi=2 :ny
    efficMarg(phi-1)=(effic(phi)-effic(phi-1))/dY;
  end
  scf();
  plot2d(Y(2 :ny),efficMarg,style=2);
  xtitle(" ","Effort","Gain en efficacité");
  efficMarg=resume(efficMarg);
endfunction
//----- Taux de réussite généralisé versus effort, dérivé de l'Eff-Rés
function plotSuccessRateEffRes(alpha,epsilon,X,Y,Z)
  // Attention, phi est un rang dans Y
  [nx,ny]=size(Z);
  rMax=X(nx);
  eps=epsilon/rMax;
  for phi=1 :ny
    for r=1 :nx
      qp(r)=qualityResult(X(r)/rMax,alpha,eps)*Z(r,phi);
    end
    sr(phi)=sum(qp)/sum(Z( :,phi));
  end
  scf();
  plot2d(Y,sr, style=2);
```

```
  xtitle(" ","Effort","Taux de réussite") ;
  sr=resume(sr) ;
endfunction
```

13.12. Histogrammes, diagrammes polaires

```
//------------------------------------------ Test GNA, histogramme
function RNGhisto(randOption,n,nbClass,rMax)
// randOption peut être de la forme [rng,var,a,b]
//  (voir alea() et aleaVar()) ou bien simplement le code rng.
// Exemples
// RNGhisto(rng,50000,100,%inf) Test d'uniformité
// RNGhisto([0,4,0,0],10000,100,%fin)  Trimodale
// RNGhisto([0,5,0,1],10000,100,5)  Lévy
// RNGhisto([0,8,2.94,1.01],10000,100,200)  Log-normale
mini=0 ;, maxi=200 ;
for i=1 :n
r(i)=alea(0,1,randOption) ;
end
r=gsort(r,"g","i") ; // Tri par ordre croissant
// Histogramme
scf() ;
i=1 ;
while r(i)<rMax & i<n
i=i+1 ;
end
histplot(nbClass,r(1 :i)) ;
a=get("current_axes") ;
cpd=a.children ;
p=cpd.children ; // Polyline
p.foreground=4 ; // Couleur
  r=resume(r) ;
endfunction
//--------------------------------- Test GNA, diagramme polaire
function RNGpol(randOption,n)
for i=1 :n
random(i)=alea(0,1,randOption) ;
end
// Diagramme polaire
// pour mettre en évidence les éventuelles régularités
scf() ;
absc=1 :n ;
theta=2*%pi*(1/n) *absc ;
polarplot(theta,random,style=2) ;
fig=get("current_figure") ;
ax=fig.children ;
compo=ax.children ;
polyl=compo.children ;
polyl.line_mode = "off" ;
//polyl.line_style=7 ; // Ligne en pointillé si line_mode="on"
polyl.mark_style=9 ; // Marques circulaires ...
```

```
polyl.mark_background=2 ; // ... en bleu
polyl.mark_foreground=2 ;
// Suppression des étiquettes autour du cercle
mat=ax.children ;
for i=1 :2 :24
mat(i).text="" ;
end
[absc,random]=resume(absc,random) ;
endfunction
```

13.13. Figures diverses

```
// ----------------------- Figure 2D d'initialisation
//                         (sans discrétisation ni combinatoire)
posMin=[0,0] ;
posMax=[1,1] ;
function init=plotInitPos(posMin,posMax,randOption, popSize)
for i=1 :popSize
pos=initPos(posMin,posMax,randOption,[-1,-1],%F) ;
init(i,1)=pos(1) ; init(i,2)=pos(2) ;
end
scf() ;
plot2d(init( :,1),init( :,2),axesflag=2,rect=[posMin(1,1),...
     posMin(1,2),posMax(1,1),posMax(1,2)]) ;
a=get("current_axes") ;
cpd=a.children ;
p=cpd.children ; // Polyline
p.line_mode="off" ;
p.mark_mode="on" ;
p.mark_style=0 ;
p.mark_size=4 ;
p.mark_foreground=2 ;
p.mark_background=2 ;
endfunction
//---------------------------------  Affichage des positions successives
// Note : uniquement les deux premières coordonnées
// Le tableau "positions" contient une position par ligne
function plotPos(positions)
scf() ;
plot2d(positions( :,1),positions( :,2),axesflag=0) ;
a=get("current_axes") ;
cpd=a.children ;
p=cpd.children ; // Polyline
p.line_mode="off" ;
p.mark_mode="on" ;
p.mark_style=0 ;
p.mark_size_unit="point" ;
p.mark_size=5 ;
p.mark_foreground=2 ;
p.mark_background=2 ;
```

```
endfunction
//------------------------------------------ DNPP PSO, en 2D
function dnppPSO(x,p,g,v,nbPoints,PSOtype)
  // Exemple :
  // x=[207,214] ; p=[102,294] ; g=[345,279] ; v=[350,87] ; v=v-x ;
  // dnppPSO (x,p,g,v, 1000, 0)
  select PSOtype
case 0  // SPSO
  w=0.73 ;
  c1=1.2 ;
  c2=1.2 ;
  for n=1 :nbPoints
    for d=1 :2
  depl=w*v(d)+c1*rand()*(p(d)-x(d))+c2*rand()*(g(d)-x(d)) ;
  dnpp(n,d)=depl+x(d) ;
end
  end
case 1 // microPSO
  w=0.1 ;
  c1=0.5 ;
  c2=1.5 ;
  for n=1 :nbPoints
for d=1 :2
  rd=c1+rand()*(c2-c1) ;
  xTemp(d)=x(d)+v(d)+rd*(g(d)-x(d)-v(d)) ;
  vTemp(d)=xTemp(d)-x(d) ;
  depl=w*v(d)+vTemp(d) ;
  dnpp(n,d)=depl+x(d) ;
end
  end
end
X=[x ;p ;g] ; // Les trois positions de base
    //            (courante, mémorisée, meilleure informatrice)
//plotPos(dnpp) ; // Affiche les points de la distribution
xmin=min(dnpp( :,1)) ; xmin=min(xmin, min(X( :,1))) ;
ymin=min(dnpp( :,2)) ; ymin=min(xmin, min(X( :,2))) ;
xmax=max(dnpp( :,1)) ; xmax=max(xmax, max(X( :,1))) ;
ymax=max(dnpp( :,2)) ; ymax=max(xmax, max(X( :,2))) ;
extend=0.1 ;
dx=xmax-xmin ; dy=ymax-ymin ;
xmin=xmin-extend*dx ; ymin=ymin-extend*ymin ;
    xmax=xmax+extend*dy ; ymax=ymax+extend*dy ;
scf() ;
plot2d(dnpp( :,1),dnpp( :,2),axesflag=0,rect=[xmin,ymin,xmax,ymax]) ;
a=get("current_axes") ;
cpd=a.children ;
pl=cpd.children ; // Polyline
pl.line_mode="off" ;
pl.mark_mode="on" ;
pl.mark_style=0 ;
pl.mark_size=0 ;
pl.mark_foreground=0 ;
```

```
pl.mark_background=2 ;
// Affiche les trois positions
//plot2d(X( :,1),X( :,2),style=9, axesflag=0,rect=[xmin,ymin,xmax,ymax]) ;
    plot2d(X( :,1),X( :,2),rect=[xmin,ymin,xmax,ymax]) ;
a=get("current_axes") ;
cpd=a.children ;
cpd1=cpd(1) ;
pl=cpd1.children ;
pl.line_mode="off" ;
pl.mark_mode="on" ;
pl.mark_style=0 ;
pl.mark_size=5 ;
pl.mark_foreground=0 ;
pl.mark_background=2 ;
dnpp=resume(dnpp) ;
endfunction
//------------------------------------------------ Deux courbes
// Les titres sont à adapter à chaque cas
function  plot2(C1,C2,absc)
  scf() ;
  [nx,dummy]=size(absc) ;
  plot2d(absc,[C1,C2],style=[2,3]) ;
  xtitle(" ","Nombre d'évaluations","Médiane") ;
  legends(["Mersenne-Twister" ;"ANSI C"],[2,3],1) ;
endfunction
//------------------------------------------------ Trois courbes
// Les titres sont à adapter à chaque cas
function  plot3(C1,C2,C3,absc)
  scf() ;
  [nx,dummy]=size(absc) ;
  plot2d(absc,[C1,C2,C3],style=[2,3,5]) ;
  xtitle(" ","Résultat","Probabilité") ;
  legends(["40" ;"80" ;"80 aléatoire"],[2,3,5],1) ;
endfunction
//------------------------------------------- Courbe Sphere/Cube
function plotSphereOverCube(Dmax,in)
  absc=1 :Dmax ;
  select in
case 1 // Cube de côté 1
  // Sphère inscrite, de rayon 0,5
  for d=1 :Dmax
rate(d)=volumSphere(0.5,d)/volumCube(1,d) ;
  end
case 0 // Cube de côté 1
   // Sphère exinscrite de rayon racine(d)/2
  for d=1 :Dmax
rate(d)=volumCube(1,d)/volumSphere(0.5*sqrt(d),d) ;
  end
end
scf() ;
plot2d(absc,rate,style=2) ;
select in
```

```
  case 1
xtitle(" ","Dimension","Sphère inscrite/Cube") ;
  case 0
xtitle(" ","Dimension","Cube/Sphère exinscrite") ;
end
  [absc,rate]=resume(absc,rate) ;
endfunction
```

13.14. Tests (biais, corrélation)

```
//------------------- Test de bordures pour signatures
// (normalement faites sur la fonction Plateau)
function bound=boundary(positions)
[nPoints,D]=size(positions); // D=2
bound=0;
for n=1 :nPoints
if abs(positions(n,1))>=1 | abs(positions(n,2))>=1 then
bound=bound+1;
end
end
bound=bound/nPoints;
printf("\nTaux de points sur la frontière : %f",bound);
endfunction
//---------------------------------------- Test de corrélation positive
function nisbAround(pb,iterMax,randOption,distanceType)
//  nisbAround(1002,5000,0,0)
// Note : distanceType n'est utile qu'en cas de problème combinatoire
//  0 => Kendall-Tau
// 1 => Cayley
// nisb =  nearer is better
rand("seed",123456789); // Initialisation de la graine,
//                              pour résultats reproductibles
 [D,posMin,posMax, quantis,normalise]=pbDef(pb);
printf("\n Problème %i, dimension %i",pb,D);
if pb>1000 then combin=1;
else combin=0; end
iter=0;
betterInside=0;
while iter<iterMax
select combin
case 0  // Problème non combinatoire
// On tire au hasard deux points
// et s'arrange pour que f(x1)< f(x2)
for pop=1 :2
  for d=1 :D
    pos(pop,d)=alea(posMin(d),posMax(d),randOption);
  end
  cost(pop)=pbEval(pb,pos(pop, :),posMin,posMax,normalise);
end
  if cost(1)<=cost(2) then
x1=pos(1, :); x2=pos(2, :);
f2=cost(2);
  else
    x1=pos(2, :); x2=pos(1, :);
```

```
f2=cost(1);
  end
// x1 est le centre de la D-sphère et son rayon est dist(x1,x2)
radius=distance(x1,x2);
// On tire au hasard un point x3 à l'intérieur de la D-sphère
// et de l'espace de recherche
inside=0;
while inside ~=D
  x3=randSphere(x1,radius,0,randOption);
  inside=0;
  for d=1 :D
if x3(d)<posMin(d) | x3(d)>posMax(d) then
  break;
else inside=inside+1;
end
  end
end
// On incrémente le compte de réussite
if   f2>pbEval(pb,x3,posMin,posMax,normalise) then
  betterInside=betterInside+1;
end
case 1  // Problème combinatoire. Une position est une permutation
// On définit au hasard deux permutations   différentes
        // et on s'arrange pour que
// la première soit meilleure que la seconde
radius=0;
while radius<2
  x1=permut(D,randOption);
  x2=permut(D,randOption);
  // On calcule une distance entre les deux permutations
  radius=permutDist(x1,x2,distanceType);
end // Fin "while  radius==0
// Le "centre" est la meilleure permutation
f1=pbEval(pb,x1,posMin,posMax,normalise);
f2=pbEval(pb,x2,posMin,posMax,normalise);
if f1<f2 then centre=x1, costOther=f2;
else centre=x2; costOther=f1;
end
// On effectue  des transpositions au hasard sur le "centre";
// mais de façon à rester à une distance  à l'autre permutation
// telle que  0 < dist < radius
dist=%inf;
while dist>=radius  | dist==0
  newPos=centre; // Position initiale
  // On choisit au hasard un nombre de transpositions ...
  nbTrans=1+floor(alea(0,radius,randOption));
  // ... on les définit, toujours au hasard, et on les applique
  for nb=1 :nbTrans
p1=0; p2=0;
while p1==p2
  p1=1+floor(alea(0,D,randOption));
  p2=1+floor(alea(0,D,randOption));
end
temp=newPos(p1); newPos(p1)=newPos(p2); newPos(p2)=temp;
```

```
  end
  //La méthode est sans garantie, alors, éventuellement, on boucle
   dist=permutDist(centre,newPos,distanceType);
end
// On teste la valeur de la nouvelle position
  costNewPos=pbEval(pb,newPos,posMin,posMax,normalise);
  if  costNewPos<costOther then
betterInside=betterInside+1;
  end
end // Fin de select combin
// Pour un suivi progressif
iter=iter+1;
absc(iter)=iter;
p=betterInside/iter; // Probabilité
nisb(iter)=2*p-1;// Corrélation
printf("\n niter %i => NisB %f",iter, nisb(iter));
 end
printf("\n Problème %i, dimension %i", pb,D);
if combin then  printf("\ndistanceType %i",distanceType); end
printf("\n %i itérations", iterMax);
itMin=floor(0.9*iterMax);
corrMean=mean(nisb(itMin :iterMax));
printf("\n Corrélation moyenne sur les 10%% dernières : %f", corrMean);
scf();
plot2d(absc,nisb,style=2);
xtitle(" ","Nombre d'itérations","Corrélation");
[absc,nisb]=resume(absc,nisb);
endfunction
//----------- Test corrélation positive, sur la ligne entre deux positions
function nisbLine(pb,iterMax,randOption,distanceType)
// Pour une fonction unimodale, le coefficient est 1
// Attention : la réciproque est fausse (contre-exemple Rosenbrock 2D).
// Inversement, si la fonction est multimodale, le coefficient est inférieur à 1.
// Mais il ne s'agit que d'une estimation expérimentale.
// Par exemple Rosenbrock 4D est bimodale, mais la plupart
// des estimations donnent cependant 1
// (sauf à faire un nombre énorme d'estimations).
// Exemples :
// nisbLine(4,5000,0,1)
// nisbLine(204,10000,0)
// nisbLine(1002,5000,0,1)
// Note : distanceType n'est utile qu'en cas de problème combinatoire
// Attention : en fait, avec Kendall-Tau (distanceType=0),
//  c'est sans intérêt. Utiliser plutôt distanceType=1.
// nisb =  nearer is better
rand("seed",123456789); // Initialisation de la graine,
//                              pour résultats reproductibles
 [D,posMin,posMax,quantis,normalise]=pbDef(pb);
printf("\n Problème %i, dimension %i",pb,D);
  if pb>1000 then combin=1;
else combin=0; end
iter=0;
betterInside=0;
betterOutside=0;
```

```
while iter<iterMax
  select combin
case 0  // Problème non combinatoire
// On tire au hasard deux points
// et s'arrange pour que f(x1)< f(x2)
for pop=1 :2
  for d=1 :D
pos(pop,d)=alea(posMin(d),posMax(d),randOption);
  end
  cost(pop)=pbEval(pb,pos(pop, :),posMin,posMax,normalise);
end
if cost(1)<=cost(2) then
  x1=pos(1, :); x2=pos(2, :);
  f2=cost(2);
else
  x1=pos(2, :); x2=pos(1, :);
  f2=cost(1);
end
// On tire au hasard un point x3 entre les deux
lambda=-1;
while lambda<=0  // On exclut la valeur 0
// Ceci pourrait aussi être fait en utilisant
//  l'epsilon-machine : lambda=alea(%eps,1,randOption);
  lambda=alea(0,1,randOption);
end
x3=x1+lambda*(x2-x1);
// On incrémente les comptes de réussite
f3=pbEval(pb,x3,posMin,posMax,normalise);
if   f3<f2 then
  betterInside=betterInside+1;
end
case 1  // Problème combinatoire. Une position est une permutation
// On définit au hasard deux permutations suffisamment différentes
        // et on s'arrange pour que la première soit meilleure que la seconde
  radius=0;
  while radius<2
    // x1=grand(1,'prm',identity);
    // x2=grand(1,'prm',identity);
    x1=permut(D,randOption);
    x2=permut(D, randOption);
    // On calcule une distance entre les deux permutations
    // et la séquence de transpositions pour passer de l'une à l'autre
    // Deux métriques possibles
    select distanceType
  case 0  // Juste pour tests. Pas intéressant en pratique
    transpo=permutDecompKT(x1,x2);
  case 1
    transpo=permutDecompCayley(x1,x2,[]);
    end
    [dummy,radius]=size(transpo);
  end // Fin "while  radius==0
f1=pbEval(pb,x1,posMin,posMax,normalise);
f2=pbEval(pb,x2,posMin,posMax,normalise);
costOther=max(f1,f2);
```

```
select distanceType
  case 0 // Kendall-Tau
dist=%inf ;
while dist>=radius
  newPos=x1 ; // Position initiale.
              //  Ce peut être la meilleure ou non, c'est sans importance
  // puisque de toute façon on cherche un point entre x1 et x2
  // et transpo donne la liste des transpositions pour aller de x1 à x2
  // On choisit au hasard un nombre de transpositions
  // et on les effectue pour être « entre » x1 et x2 ;
  nbTrans=1+floor(alea(0,radius,randOption)) ;
  for nb=1 :nbTrans
p1=transpo(1,nb) ;
p2=transpo(2,nb) ;
temp=newPos(p1) ; newPos(p1)=newPos(p2) ; newPos(p2)=temp ;
  end
  // La méthode est sans garantie,
              // alors on vérifie qu'on est bien « plus proche »
  // sinon, on boucle
  dist=permutDist(x2,newPos,distanceType) ;
end
  case 1  // Cayley
    newPos=x1 ; // Position initiale
    nbTrans=1+floor(alea(0,radius,randOption)) ;
    for nb=1 :nbTrans
  p1=transpo(1,nb) ;
  p2=transpo(2,nb) ;
          temp=newPos(p1) ; newPos(p1)=newPos(p2) ; newPos(p2)=temp ;
    end
// La méthode est sûre. Pas besoin de boucler
end // Fin de select distanceType
// On teste la valeur de la nouvelle position
if  pbEval(pb,newPos,posMin,posMax,normalise)<costOther then
  betterInside=betterInside+1 ;
end
  end // Fin de select combin
  // Pour un suivi progressif
  iter=iter+1 ;
  absc(iter)=iter ;
  p=betterInside/iter ;
  nisb(iter)=2*p-1 ;
  printf("\n niter %i => NisB %f",iter,nisb(iter)) ;
    end
  printf("\n Problème %i, dimension %i",pb,D) ;
  if combin then  printf("\ndistanceType %i",distanceType) ; end
  printf("\n %i itérations", iterMax) ;
  itMin=floor(0.9*iterMax) ;
  corrMean=mean(nisb(itMin :iterMax)) ;
  printf("\n Corrélation moyenne sur les 10%% dernières : %f",corrMean) ;
  scf() ;
  plot2d(absc,nisb,style=2) ;
  xtitle(" ","Nombre d'itérations","Corrélation") ;
  [absc,nisb]=resume(absc,nisb) ;
endfunction
```