TP optimisation - Partie 2

AIT BAALI Hamza, AMINI Nada, ESSAYEGH Nour, LEFDALI Rida

1- fichier *ego.Simple*.R

a) Création du plan d'expériences initial et construction du processus gaussien

In [ ]:
```
# create a LHS DoE
  ninit <- 10*zdim
  # Xinit <- data.frame(LB + (UB-LB)*lhsDesign(n = ninit,dimension = zdim)$design) # this version
  Xinit <- data.frame(matrix(LB,nrow=ninit,ncol=zdim,byrow=T) + matrix(UB-LB,nrow=ninit,ncol=zdim,
  # calculate associated objective function
  Yinit <- apply(X = Xinit,MARGIN = 1,FUN = test_fun)
  imin <- which.min(Yinit)
  fmin <- Yinit[imin]
  xmin <- as.numeric(Xinit[imin,])
  x_hist[1:ninit,]<-as.matrix(Xinit)
  y_hist[1:ninit,1]<-Yinit
  # make a kriging model.
  # I use a fairly high lower bound on the thetas here to prevent degenerated models
  if (!debugMode) {
    capture.output(GPmodel <-km(design = Xinit,response = Yinit,covtype="matern3_2",lower = rep(0.
  else {
    GPmodel <-km(design = Xinit,response = Yinit,covtype="matern3_2",lower = rep(0.8,zdim),multist
  }
```

b) Définition du critère d'acquisition.

In [ ]:
```
internalFun <- meanofGP # meanofGP, mEI
```

c) Optimisation du critère d'acquisition

In [ ]:
```
# optimize with CMA-ES the acquisition criterion to define the next iterate
    paramCMA <- list(LB=LB,UB = UB,budget = 3000, dim=zdim, xinit=runif(n = zdim,min = LB,max = UB
    optresCMA <- cmaes(internalFun, paramCMA)
    fnext <- test_fun(optresCMA$x_best) # the one call to the true objective function
    if (fnext<fmin){
      fmin <- fnext
      xmin <- optresCMA$x_best
    }
    x_hist[i+ninit,]<-optresCMA$x_best
    y_hist[i+ninit]<-fnext
```

d) Appel à la vraie fonction coût

In [ ]:
```
fnext <- test_fun(optresCMA$x_best) # the one call to the true objective function
```

e) Mise à jour du critère d'acquisition

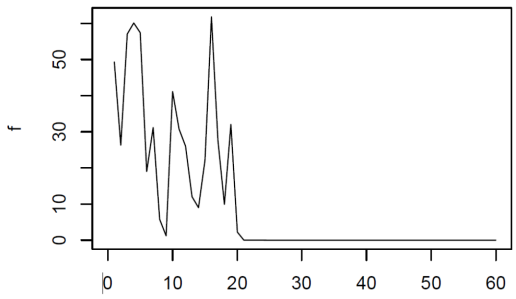
In [ ]:
```
# update DoE
    newX <- rbind(GPmodel@X,optresCMA$x_best)
    newY <- rbind(GPmodel@y,fnext)
    # build a new GP (which involves another internal optimization, typically a likelihood maximiz
    # the printout of the km function is thrown to the garbage for better lisibility, remove the c
    if (!debugMode) {
      capture.output(GPmodel <-km(design = newX,response = newY,covtype="matern3_2",lower = rep(0.
    else {
      GPmodel <-km(design = newX,response = newY,covtype="matern3_2",lower = rep(0.5,zdim),multist
    }
```

2- Test de l'optimiseur ego

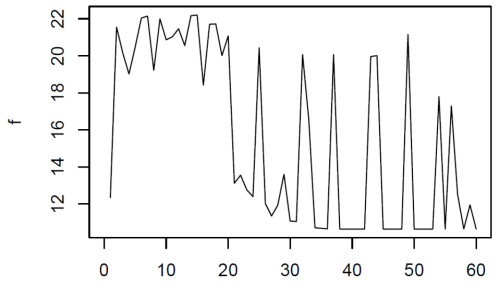
On commence d'abord par se familiariser avec la fonction egoSimple avant de fixer les paramètres sur 1 pour *no_test* et 2 pour *zdim*. On test cette optimiseur global sur une fonction sans optima locaux comme *sphere* et sur une fonction avec optima locaux comme *ackley*.

On remarque que le temps d'exécution est relativement long par rapport aux autres optimiseurs.

Sphère



Ackley



Les 20 premières itérations correspondent (Ackley comme dans Sphère) aux évaluations correspondant aux plans d'expérience. À partir de là on commence à avoir les évaluations qui sont des minimisations de la moyenne.

On arrive à voir que pour la fonction sphère les 20 itérations suffisent pour trouver le minimum global. Cela est dû à la simplicité de cette fonction.

3 - la fonction mEI

On code la fonction qui retourne l'opposé de l'expected improvement, et on change internalFun pour qu'elle pointe sur celle-ci:

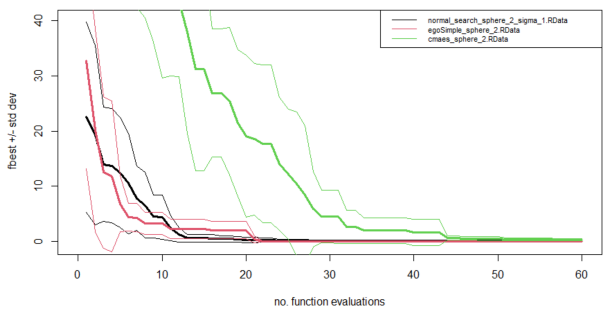
In [ ]:
```
#on définit la fonction mEI d'apres la formule vue en cours.
  mEI <- function(x) {
    x <- matrix(x,ncol=zdim)
    y <- predict(object = GPmodel,newdata=data.frame(x),type="UK")
    EI<-(fmin-y$mean)*pnorm((fmin-y$mean)/y$sd)+y$sd*dnorm((fmin-y$mean)/y$sd)
    mEI<--EI
    mEI
    }
#on initialise le critère d'acquisition
  internalFun <- mEI
```

Fonction sphère

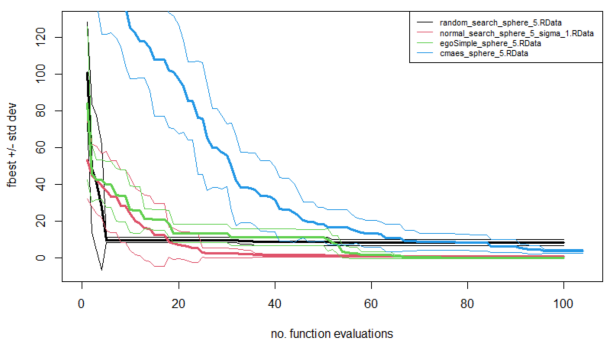
Nous avons testé cet optimiseur en 2D puis en 5D sur la fonction sphère pour pouvoir comparer ce résultat avec les optimisateurs vus précédemment comme normal_search et cmaes. Nous allons prendre 6 tests au total.

Pour abréger le temps de calcul nous avons choisi de diminuer le budget à 60 pour les fonctions en 2D et à 100 itérations pour les fonctions en 5D au lieu de 150.

2D



5D

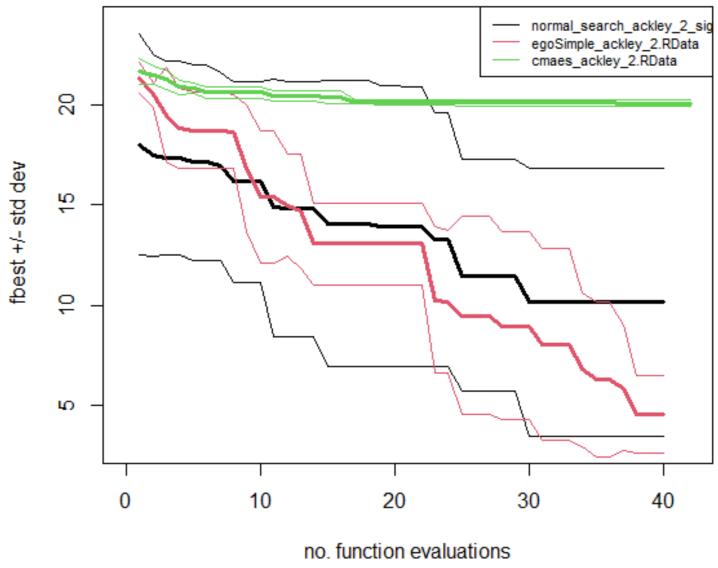


On remarque que dans les deux cas l'algorithme EGO est très efficace pour trouver l'unique minimum de cette fonction. Il est en effet plus précis dans ces calculs puisqu'il n'y a pas de composante aléatoire. Le seul inconvénient de cet algorithme est que son temps d'exécution est considérable par rapport aux algorithmes vus dans la première partie du TP.

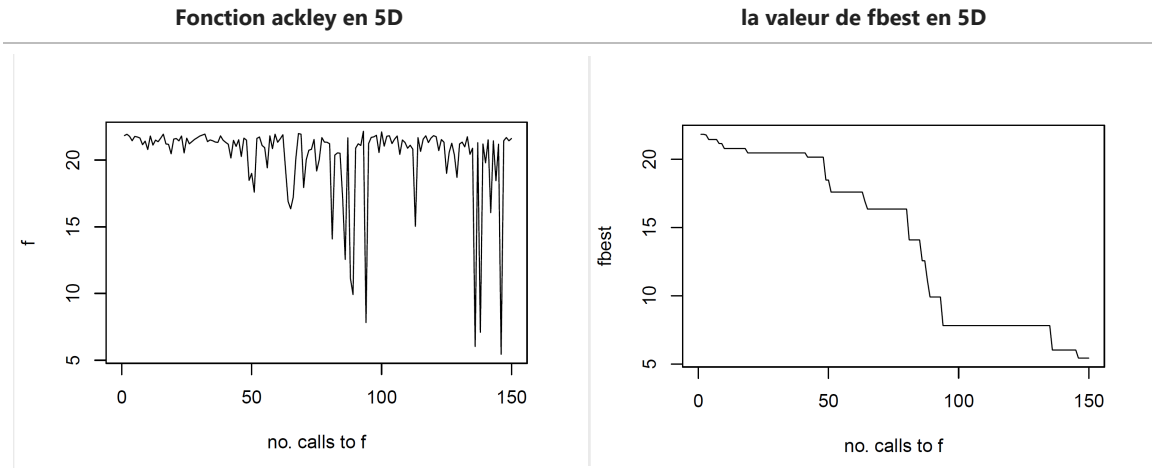
Fonction ackeley

Nous avons fait des tests sur la fonction Ackley. Cette fonction est assez complexe. Nous avons décidé de tester notre optimiseur en 2D et avec un budget de 40. Sur un budget aussi réduit, on voit clairement que l'optimiseur EGO est plus performant que les autres algorithmes que l'on a testés.

D'une part un algorithme normal_seach qui, en réglant son coefficient alpha, donne de bons résultats. D'autre part, cmaes qui est bon pour les grandes dimensions(5D et +) mais qui converge très lentement (il a besoin d'un budget très important). On peut voir sur la figure suivante que ce dernier ne converge pas du tout vers le minimum global et c'est bien évidemment à cause du budget très réduit que l'on a pris.

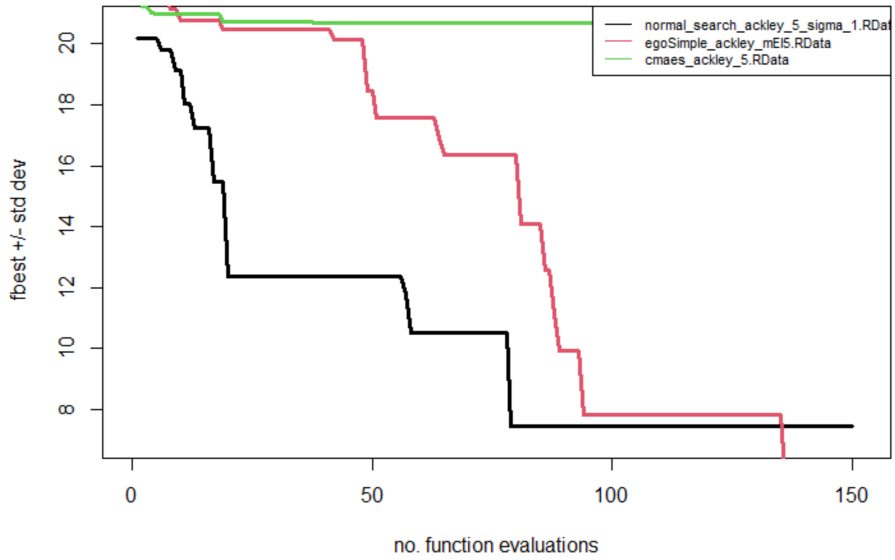


Nous avons aussi testé une fois EGO sur Ackley en 5D . On peut voir les résultats sur la figure suivante.



Ce que l'on peut dire est que, pour une fonction aussi compliquée que Ackley en 5D, l'approximation du minimum est très bonne et nous avons aussi la possibilité de suivre l'évolution plus ordonnée des valeurs que prend la fonction puisque c'est une méthode déterministe.

On peut aussi représenter un seul test pour la fonction Ackley en 5D en utilisant normal_search et cmaes pour mieux visualiser la performance des différents algorithmes et pouvoir les comparer entre eux.



Cette dernière figure confirme ce qui a été dit précédemment.