# Question Bonus

Essayegh Nour

Pour cette question on pourra essayer de trouver un juste milieu entre les algorithmes d'optimisation globale et locale et ainsi combiner les avantages des deux types de méthodes. En effet, les méthodes locales mènent à une solution locale. Tandis que les méthodes globales consomment énormément de temps, il parait clair qu'un compromis entre exploitation et exploration doit être trouvé d'où l'utilité de rechercher une sorte de solution hybride.

## 1- Combinaision recherche globale et locale

Une première méthode assez intuitive est de s'approcher de l'optimum global par une méthode aléatoire puis affiner le résultat en appliquant successivement une méthode locale. Cette méthode est assez naïve puisqu'on combine successivement les deux méthodes pour une résultat plus précis mais au détriment du temps de calcul. C'est le cas de NS_lbfgs et de RS_lbfgs qui correspond à l'algorithme Random_search suivi de l'optimisateur local lbfgs. Nous pouvons aussi utiliser d'autres optimisateurs locaux.

```
Random_lbfgs <- function(test_fun, param) {
  res_RS <- random_search(ofwrapper, param)
  param_lbfgs <- param
  #commencer par le meilleur point de random_search
  param_lbfgs$xinit <- res_RS$x_best
  res_lbfgs <- lbfgs(ofwrapper, param_lbfgs)
  res <- list()
  x_hist<- rbind(res_NS$xhist,res_lbfgs$xhist)
  y_hist <- rbind(res_NS$fhist,res_lbfgs$fhist)
  best_par <- res_lbfgs$x_best
  best_value <- res_lbfgs$f_best
  res <- list(xhist=x_hist, fhist=y_hist, x_best=best_par, f_best=best_value)
  return(res)
}
```

## 2- Optimisation locale itérée

Une idée d'algorithme inspiré par le cours de plan d'expérience est de construire un plan d'expérience en respectant le critère Maximin et générer un ensemble de points qui seront dans la suite considéré les points initiaux pour un optimiseur local. Par la suite, le meilleur minimum est considéré comme le minimum global de la fonction.

Cette méthode est en théorie assez simple et nous pouvons essayer de la coder sur R.

```
library(DiceDesign)
```

```
## Warning: package 'DiceDesign' was built under R version 4.0.3
```

```
lbfgsglob <- function (test_fun, param){
  budget <- param$budget
  xinit <- param$xinit
  dim <- length(xinit)

  current_par <- xinit
  current_value <- test_fun(xinit)
  best_par <- xinit
  best_value <- test_fun(xinit)

  PlanExpLHS <- lhsDesign(budget,dim)$design
  PlanOp<-maximinSA_LHS(PlanExpLHS)$design

  param_lbfgs <- param
  for (i in 1:budget){
      param_lbfgs$xinit <- PlanOp[i,]
      res_lbfgs <- lbfgs(ofwrapper, param_lbfgs)
      if (res_lbfgs$f_best< best_value){
        best_value<-res_lbfgs$f_best
        best_par <- res_lbfgs$x_best
        xhist<-res_lbfgs$xhist
        fhist<-res_lbfgs$fhist
      }
    }

res <- list()
res$xhist <- rbind(xhist)
res$fhist <- rbind(fhist)
res$x_best <- best_value
res$f_best <- best_par
return(res)
}
```

## 3- L'algorithme EGO

### x_best null ?

Concernant l'algorithme EGO, nous avons remarqué lors du lancement de l'algorithme pour des fonction multimodale complexe comme ackley que dans quelques cas le programme s'arrêtait à cause d'une erreur. Cette erreur consistait à avoir un x_best null. C'est-à-dire que l'approximation avec la fonction cmaes à échouer. Pour résoudre ce problème qui nous ralentie puisque les calculs devait être repris dès le début nous avons intégré une boucle while lors de l'optimisation avec cmaes de la forme suivante.

En effet, si mon x_best est null, le suivant est tiré suivant une loi uniforme.

```
library(DiceDesign)
library(DiceKriging)
```

```
## Warning: package 'DiceKriging' was built under R version 4.0.3
```

```
source("./cmaes.R")
```

```
egoSimple <- function(test_fun, param) {
#....

    # 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),sigma=2.)
    optresCMA <- cmaes(internalFun, paramCMA)

    while (is.null(optresCMA$x_best)) {
      paramCMA$xinit<-runif(n = dim,min = LB,max = UB)
      optresCMA <- cmaes(internalFun, paramCMA)
    }

#....
}
```

**normal_search ou cmaes ?**

Le deuxième moyen que l'on peut imaginer pour améliorer cet algorithme c'est de tester une optimisation avec un autre optimiseur global que cmaes et j'ai choisi le normal_search pour tester cela. On peut imaginer que ce changement d'optimiseur rendra l'execution un poil plus rapide puisque normal_search n'a pas besoin d'autant de budget pour donner une bonne approximation du point x_best.

```
source("./normal_search.R")

egoSimple <- function(test_fun, param) {
#...
    paramNS <- list(LB=LB,UB = UB,budget = 150, dim=zdim,
                    xinit=runif(n = zdim,min = LB,max = UB),sigma=1)
    optresNS <- normal_search(internalFun, paramNS)


    fnext <- test_fun(optresNS$x_best)
    if (fnext<fmin){
      fmin <- fnext
      xmin <- optresNS$x_best
    }
    x_hist[i+ninit,]<-optresNS$x_best
    y_hist[i+ninit]<-fnext

#...
}
```

Une première remarque concerne le temps de calcul. Pour avoir un ordre d'idées, le temps que prend ego_NS est de 2min 3s et celui d'ego est de 5min15s. cette différence assez significative marque un bon point pour notre optimiseur. Pour vérifier sa supériorité il faut comparer les résultats des tests pour les deux fonctions avec *postproc_tests_optimizers.R*

Sur la figure nous pouvons remarquer un léger avantage pour l'optimiseur ego_NS donc on peut en conclure que c'est une bonne façon d'améliorer les performances de l'algorithme EGO.

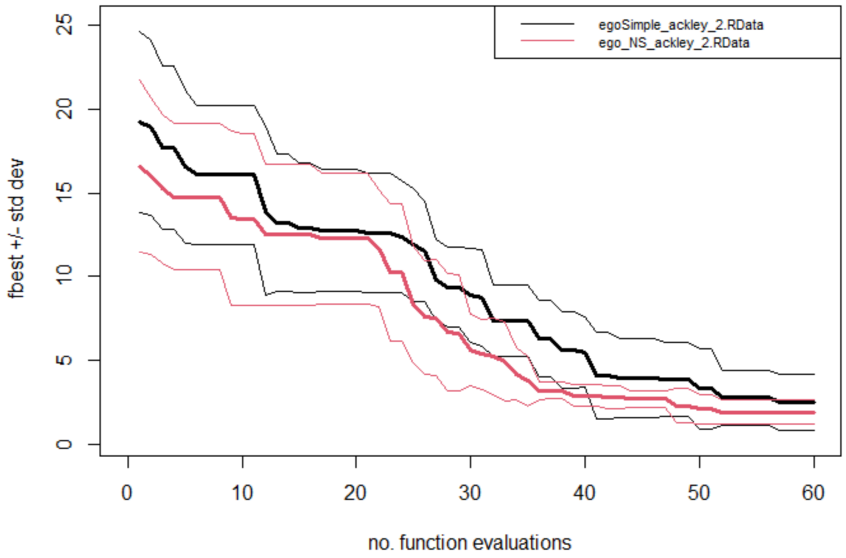


Figure 1: Ackley

**Autres fonctions d'acquisition**

Comme nous avons pu le voir lors de la partie 2 du Tp d'optimisation, la fonction d'acquisition peut être modifiée pour remplacer le critère de progrès espéré que nous avions codé précedement. En effet, parmis ces fonctions nous pouvons imaginer une combinaison entre la moyenne de krigeage et l'ecart type de krigeage. En adaptant cela à chaque itération.