Statistica Sinica: Supplement

Sequential design of experiments for estimating quantiles of black-box functions

T. Labopin-Richard and V. Picheny

Institut de Mathématiques de Toulouse, Université Paul Sabatier, Toulouse, France

MIAT, Université de Toulouse, INRA, Castanet-Tolosan, France

Supplementary Material

This supplementary material contains the R codes used to generate the experiments. The beginning of the code may be changed to use it on a different test problem.

S1 Main function

```
library(KrigInv);library(DiceOptim);library(DiceDesign);library(Rcpp)
```

Setting; fun is the black-box function, Sigma the covariance of X

```
fun<-hartman4; d<-4

Sigma<-matrix(rep(.05, d*d), d,d)

diag(Sigma)<-.1
```

Algorithm settings

```
alpha<-.95; type<-"variance"
```

```
n.init<-15; n.ite<-45

n<-1e3; n.large<-1e5; n.cand<-1e5; n.cand.red<-300
```

Initial design of experiments

```
x.init<-rep(.5,d) + qnorm(lhsDesign(n.init,d)$design)%*%chol(Sigma)

y.init<-as.numeric(apply(x.init, 1, fun))
```

Initial kriging model

```
model<-km(~.,design=data.frame(x.init),response=y.init,

            lower=rep(.05,d),upper=rep(1,d),control=list(trace=FALSE))
```

Main loop

```
for (ite in 1:n.ite) {
```

renew integration points

```
x<-mvrnorm(n=n, mu=rep(0.5, d), Sigma)
```

Kriging prediction on x.large to compute quantile

```
p.large<-predict(model,data.frame(x),"UK",checkNames=F,light.return=T)

qn<-p.large$mean[order(p.large$mean)[round(alpha*n.large)]]
```

renew integration points

```
x<-mvrnorm(n=n, mu=rep(0.5, d), Sigma)
```

Precalculations

```
precalc.data.x<-precomputeUpdateData(model, integration.points=x)
```

```
pred.x<-predict(model,data.frame(x),"UK",checkNames=F,light.return=T)

data.x<-list(x=x,mean=pred.x$mean,sd=pred.x$sd,precalc.data=precalc.data.x)
```

Generating x.cand

```
x.cand<-mvrnorm(n=n.cand, mu=rep(0.5, d), Sigma)

p.cand<- predict(model,data.frame(x.cand),"UK",checkNames=F)

dens<-dnorm((qn - p.cand$mean)/p.cand$sd)

prob.n<-pmax(dens/sum(dens), 1e-3/n.cand)

prob.n.cum<-cumsum(c(0, prob.n/sum(prob.n)))

my.indices<-findInterval(runif(n.cand.red),prob.n.cum,all.inside=T)

x.cand<-x.cand[my.indices,]
```

Computing criterion for all candidate points

```
Vn<-apply(x.cand, 1, compute_crit, model=model,

            data.x=data.x, x=x, alpha=alpha, type=type)
```

Handling NAs, local descent for the variance criterion

```
if (type=="proba") {

  Vn[is.na(Vn)]<-max(Vn, na.rm = TRUE)

  newX<-x.cand[which.min(Vn),,drop=FALSE]

} else {

  Vn[is.na(Vn)]<-min(Vn, na.rm = TRUE)

  newX<-x.cand[which.max(Vn),,drop=FALSE]

  res<-optim(par=newX,fn=compute_crit,model=model,

       data.x=data.x,x=x,alpha=alpha,type=type,
      lower=rep(min(x),d),upper=rep(max(x),d),
      control=list(fnscale=-1,maxit=5),method="L-BFGS-B")
  newX<-res$par
}
```

New observation

```
newy<-fun(newX)
```

Model update

```
model<-update(object=model, newX=newX, newy=newy)
}
```

End of main loop.

S2 Auxiliary functions

S2.1 Function compute_crit()

This function computes the criterion to be optimized. It takes as inputs:

- xnew : candidate point
- model : a kriging model (class km)
- data.x : precalculations at integration points
- x : integration points (vector if 1D, matrix or data.frame)
- type : either “proba” or “variance”.

```
compute_crit<-function(xnew, model, data.x, x, alpha, type="proba"){

if (is.null(dim(x)))  x<- matrix(x, ncol=1)

if (is.null(dim(xnew))) xnew<-matrix(xnew, nrow=1)

if (checkPredict(x=xnew, model=list(model))){

return(NA)

} else {
```

Get precalculations

```
m.x<-data.x$mean

s.x<-data.x$sd

precalc.data.x<-data.x$precalc.data

n.x<-length(m.x)

k<-round(alpha*n.x)
```

Precalculations for xnew

```
p.xnew<-predict_nobias_km(model,data.frame(xnew),"UK",checkNames=F)

m.xnew<-p.xnew$mean

s.xnew<-p.xnew$sd

kn<-as.numeric(computeQuickKrigcov(model=model,integration.points=x,

      X.new=data.frame(x=xnew),precalc.data=precalc.data.x,

      F.newdata=p.xnew$F.newdata,c.newdata=p.xnew$c))

data.xnew<-list(mean=m.xnew, sd=s.xnew, F.newdata=p.xnew$F.newdata,

                  c.newdata=p.xnew$c)
```

99% confidence intervals on z

```
ynew.range<-c(m.xnew - 4*s.xnew, m.xnew + 4*s.xnew)
```

```
z.range<- (ynew.range - m.xnew) / s.xnew^2
```

Get quantile points

```
i2<-order(m.x + kn * z.range[1])[k]

.J2<-i2

.I3<-i3<-z.range[1]

while (T) {

  v<-get_indices(kn, m.x, i2, i3)

  if (v[2] > z.range[2])  break;

  i3<-(m.x[i2] - m.x[v[1]]) / (kn[v[1]] - kn[i2])

  .J2<-c(.J2, i2<-v[1])

  .I3<-c(.I3, i3)

}

.I3<-c(.I3, z.range[2])

xQ <-x[.J2,,drop=FALSE]
```

Precalculations for quantile points

```
precalc.data.xQ<-precomputeUpdateData(model,integration.points=xQ)

pred.xQ<-predict_nobias_km(model,data.frame(xQ),"UK",checkNames=F)

data.xQ<-list(mean=pred.xQ$mean, sd=pred.xQ$sd,

                precalc.data=precalc.data.xQ)

if (type=="proba") {

  Gamma<-compute_gamma(xnew=xnew, xQ=xQ, model=model,

                         data.x=data.x, data.xQ=data.xQ, I=.I3)

  return(abs( mean(Gamma) - (1-alpha) ))
} else if (type=="variance") {

  return(compute_varQ(xnew=xnew, xQ=xQ, model=model,

                 data.xnew=data.xnew, data.xQ=data.xQ, I=.I3))

} } }
```

S2.2 Function compute_gamma()

This function computes the probability of exceeding the quantile. It takes as inputs:

- xnew: candidate point
- xQ: quantile points
- model: a kriging model (class km)
- data.x: precalculations at integration points
- data.xQ: precalculations at quantile points
- I: critical intervals

```
compute_gamma<-function(xnew, xQ, model, data.x, data.xQ, I){
```

Predict at xnew

```
pred.xnew<-predict_nobias_km(model,data.frame(xnew),"UK",checkNames=F)

s.xnew<-pred.xnew$sd
```

Compute conditional covariances

```
k.x.xnew<-computeQuickKrigcov(model=model,X.new=data.frame(x=xnew),
integration.points=as.matrix(data.x$x),c.newdata=pred.xnew$c,
precalc.data=data.x$precalc.data,F.newdata=pred.xnew$F.newdata)
k.xQ.xnew<-computeQuickKrigcov(model=model,X.new=data.frame(x=xnew),
integration.points=as.matrix(xQ),c.newdata=pred.xnew$c,
precalc.data=data.xQ$precalc.data,F.newdata=pred.xnew$F.newdata)
n.x<-length(data.x$mean)
Gamma<-rep(0, n.x)
for (i in 1:length(data.xQ$mean)) {
  varW<-pmax(1e-21, data.x$sd^2 + k.xQ.xnew[i]^2/s.xnew^2
               - 2*k.xQ.xnew[i]*k.x.xnew/s.xnew^2)
  ai<-(data.x$mean - data.xQ$mean[i]) / sqrt(varW)
bip<-I[i+1]*s.xnew
  bim<-I[i]*s.xnew
  ri<-(k.xQ.xnew[i] - k.x.xnew) / ( sqrt( varW) * s.xnew )
  ri<-pmin(pmax(ri, -1), 1)
  Gamma<-Gamma+pbivnorm(ai,rep(bip,n.x),ri)
  -pbivnorm(ai,rep(bim,n.x),ri)
}
return(Gamma)
}
```

S2.3 Function compute_varQ()

This function computes the quantile variance. It takes as inputs:

- xnew: candidate point
- xQ: quantile points
- model: a kriging model (class km)
- data.x: precalculations at integration points
- data.xQ: precalculations at quantile points
- I: critical intervals.

```
compute_varQ<-function(xnew, xQ, model, data.xnew, data.xQ, I){
```

Compute conditional covariance

```
k.xQ.xnew<-computeQuickKrigcov(model=model,X.new=data.frame(x=xnew),
integration.points=data.frame(xQ),precalc.data=data.xQ$precalc.data,
F.newdata=data.xnew$F.newdata, c.newdata=data.xnew$c.newdata)
```

Main loop

```
m.xnew<-data.xnew$mean; s.xnew<-data.xnew$sd
varX<-EX<-PA<-a<-b<-rep(0, length(data.xQ$mean))
a<-k.xQ.xnew / s.xnew^2
b<-data.xQ$mean - k.xQ.xnew / s.xnew^2*m.xnew
all.d<-s.xnew^2*I + m.xnew
all.u<-(all.d-m.xnew)/s.xnew
phi.u<-dnorm(all.u)
Phi.u<-pnorm(all.u)
```

```
PA<-diff(Phi.u)

I<-which(PA>0)

term1<-- diff(all.u*phi.u)

term2<-- diff(phi.u)

EX[I]<-m.xnew + s.xnew * term2[I] / PA[I]

varX[I]<-s.xnew^2*(1+term1[I]/PA[I]-(term2[I]/PA[I])^2)

varQ<-sum(a^2*varX*PA) + sum((b + a*EX)^2*(1-PA)*PA)

if (length(data.xQ$mean) >1) {

  baEXPA<-matrix((b + a*EX)*PA, nrow=1)

  Q<-2*crossprod(baEXPA)

  varQ<-varQ - sum(Q[upper.tri(Q)])

}

return(varQ)

}
```

S2.4 Function get_indices()

This function computes the indices related the quantile point, as described in the appendix of the article. For efficiency purpose, it is coded in `C++`, and interfaced with `R` using the package `Rcpp`.

```
sourceCpp(code='

#include <Rcpp.h>

// [[Rcpp::export]]

int find_nth(const Rcpp::NumericVector& xa, const int middle)

{
  Rcpp::NumericVector sort(middle);

  /*std::cout << "middle=" << middle << std::endl;*/

  std::partial_sort_copy(xa.begin(), xa.end(), sort.begin(), sort.end());

  /*for (int i = 0; i < sort.size(); ++i) {*/

  /*std::cout << \' \' << i << \':\' << xa[i] << \'/\' << sort[i];*/

  /*}*/

  /*std::cout << std::endl;*/

  return std::find(xa.begin(), xa.end(), sort[sort.size() - 1]) - xa.begin() + 1;

}

// [[Rcpp::export]]

int

c_iter_kk(const Rcpp::NumericVector& c1, const Rcpp::NumericVector& c2,

const int i2, int imax, int value)

{

  int i = i2;

  for (; i < imax && c1[i] != value && c2[i] != value; ++i);

  return i + 1;

}

template <typename Scalar>

struct gen_idx {

Scalar i;

gen_idx() : i(0) {}

Scalar operator () () { return i += 1; }

};

struct comp_idx {
```

```
const Rcpp::NumericVector* ref;

comp_idx(const Rcpp::NumericVector& r) : ref(&r) {}

bool operator () (double d1, double d2) const

{

  int i1 = (int) d1;

  int i2 = (int) d2;

  return (*ref)[i1 - 1] < (*ref)[i2 - 1];

}

bool operator () (int i1, int i2) const

{

  return (*ref)[i1 - 1] < (*ref)[i2 - 1];

}

};

// [[Rcpp::export]]

Rcpp::NumericVector

c_order_nv(const Rcpp::NumericVector& vec)

{

  Rcpp::NumericVector idx(vec.size());

  std::generate(idx.begin(), idx.end(), gen_idx<double>());

  std::sort(idx.begin(), idx.end(), comp_idx(vec));

  return idx;

}

// [[Rcpp::export]]

std::vector<int>

c_order(const Rcpp::NumericVector& vec)
```

```
{
  std::vector<int> idx(vec.size());
  std::generate(idx.begin(), idx.end(), gen_idx<int>());
  std::sort(idx.begin(), idx.end(), comp_idx(vec));
  return idx;
}
// [[Rcpp::export]]
Rcpp::NumericVector
get_indices(const Rcpp::NumericVector& vec_a,
const Rcpp::NumericVector& vec_b, int k, double Imin)
{
  --k;
  // cherche argmin(-(b[k] - b) / (a[k] - a) > Imin)
  int min_idx = -1;
  double min_val = std::numeric_limits<double>::infinity();
  for (int idx = 0; idx < vec_a.size(); ++idx) {
  if (idx == k) { continue; }
  double val = (vec_b[idx] - vec_b[k]) / (vec_a[k] - vec_a[idx]);
  if (val < min_val && val > Imin) {
  min_idx = idx;
  min_val = val;
  }
  }
  return Rcpp::NumericVector::create(min_idx + 1, min_val);
}
```

```
')
```