Supplementary Material for 'Adaptive Batching Design with Application in Noisy Level Set Estimation'

This RMarkdown document supplements the article "Adaptive Batching for Gaussian Process Surrogates with Application in Noisy Level Set Estimation". We provide

- Code to reproduce Figure 6 in Section 6 of the paper that deals with Bermudan option pricing.
- Implementation in R for several key functions related to adaptive batching
- Definitions of the rescaled 2D Branin-Hoo function and the 6D Hartman function, which are used for synthetic experiments in Section 5.

The R code is part of a larger GitHub library for Bermudan option pricing which can be provided upon request (withheld to maintain author identity for double-blind review process).

A: Visualizing Adaptive Batching for Bermudan Option Pricing

We consider adaptive batching for a Gaussian GP metamodel for a two-dimensional basket Put Bermudan option. The asset follows log normal dynamics

$$Z_{t+\Delta t} = Z_t \cdot \exp\left((r - \frac{1}{2}diag\Xi)\Delta t + \sqrt{\Delta t} \cdot \Xi(\Delta W_t)\right)$$

where ΔW_t are independent Gaussians, and the payoff is $h_{Put}(t,z) = e^{-rt}(\mathcal{K} - z^1 - z^2)_+$.

The code below sets up the model for an arithmetic Basket Put with parameters in Table 4 of the article, including the total simulations N_T stored in *total.budget*, initial design size k_0 in *init.size*, initial batch size r_0 in *batch.nrep* and kernel function K in *kernel.family*. The parameters of the model are initialized as a list, which is later used as input in the main function **osp.seq.batch.design.simplified**.

```
model2d <- list(x0 = rep(40,2),K=40,sigma=rep(0.2,2),r=0.06,
                div=0,T=1,dt=0.04,dim=2,sim.func=sim.gbm,
                payoff.func=put.payoff, look.ahead=1, pilot.nsims=1000)

model2d$cand.len <- 1000      # size of testing set m_0
model2d$max.lengthscale <- c(40,40)  # bounds on lengthscales for the squared-exp kernel function
model2d$min.lengthscale <- c(3,3)
model2d$seq.design.size <- 100  # K_T
model2d$batch.nrep <- 20      # initial replication r_0
model2d$total.budget <- 2000  # budget N_T = r_0 * k = 2000
model2d$init.size <- 20       # initial design size k_0
model2d$init.grid <- int_2d   # initial grid: pre-loaded from the library

model2d$kernel.family <- "gauss"  # GP kernel function: squared-exponential
model2d$update.freq <- 5      # frequency of re-fitting GP hyperparameters
model2d$r.cand <- c(20, 30,40,50,60, 80, 120, 160) # r_L
```

To implement adaptive batching we need to specify the batch heuristic via *batch.heuristic* and the sequential design framework with *ei.func*. The function **osp.seq.batch.design.simplified** then fits the GP simulator. Fitting is done through the `DiceKriging` library, namely the main `km` function there. Below we apply

Adaptive Design with Sequential Allocation, ADSA (which relies on the MCU acquisition function) and Adaptive Batching with Stepwise Uncertainty Reduction, ABSUR. The method parameter `trainkm` means that the GP metamodels are trained using the default `DiceKriging::km` MLE optimizer.

```
## GP + ADSA
set.seed(110)
model2d$batch.heuristic <- 'adsa'
model2d$ei.func <- 'amcu'
oos.obj.adsa <- osp.seq.batch.design.simplified(model2d, method="trainkm")

### GP + ABSUR
set.seed(122)
model2d$batch.heuristic <- 'absur'
model2d$ei.func <- 'absur'
oos.obj.absur <- osp.seq.batch.design.simplified(model2d, method="trainkm")
```

The objects `oos.obj.xxx` are lists that contain the GP metamodels for each time-step of the Bermudan option problem ($M = 25$ in the example above). We visualize the fitted timing values at $t = 0.6 = 15\Delta t$. The respective zero contour is the *exercise boundary*.

The function **plt.2d.surf.with.batch** plots the fitted exercise boundary together with its 95% credible interval (solid line and dashed curves, respectively). The inputs are shown as the dots and their color indicates the replication amounts r_i's. The first argument is the fitted GP emulator (including the fitted model and the input sites), and the second argument is the batch sizes corresponding to the selected input sites.

First we plot the figure for results obtained with ADSA.

```
oos.obj.adsa$ndesigns[15]  # number of unique designs
```

```
## [1] 37
```

```
### plot Figure 6 right panel - ADSA
plt.2d.surf.with.batch(oos.obj.adsa$fit[[15]],
                       oos.obj.adsa$batches[1:oos.obj.adsa$ndesigns[15] - 1, 15])
```

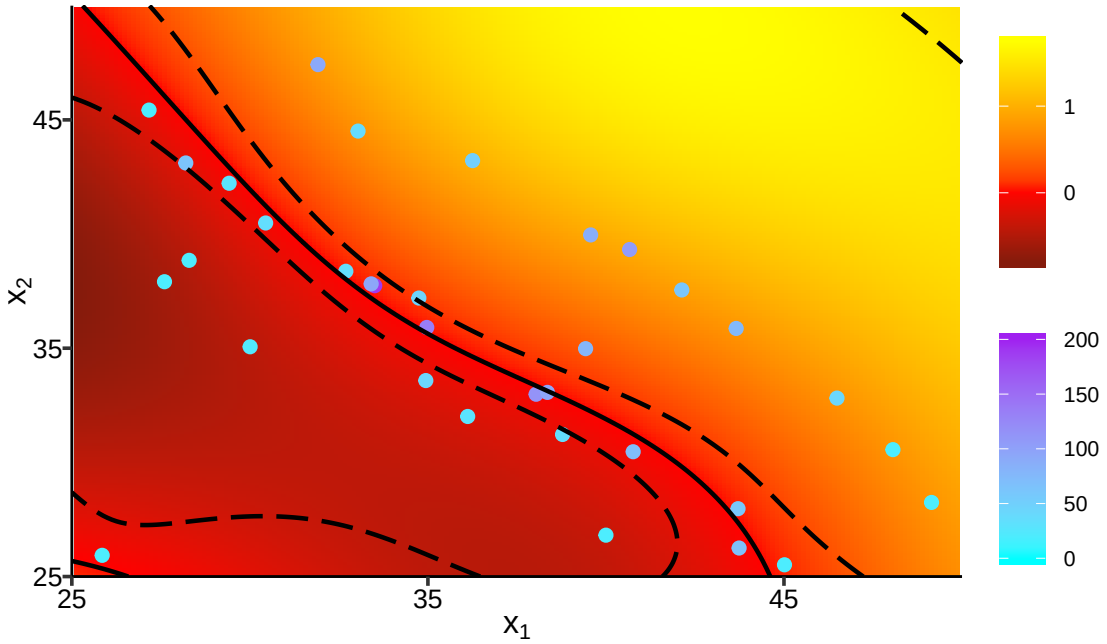

Figure 1: Timing Value Exercise Boundary using ADSA

The second one is for ABSUR

```
oos.obj.absur$ndesigns[15]  # number of unique designs
```

```
## [1] 40
```

```
### plot Figure 6 left panel - ABSUR
plt.2d.surf.with.batch(oos.obj.absur$fit[[15]],
                       oos.obj.absur$batches[1:oos.obj.absur$ndesigns[15] - 1, 15])
```

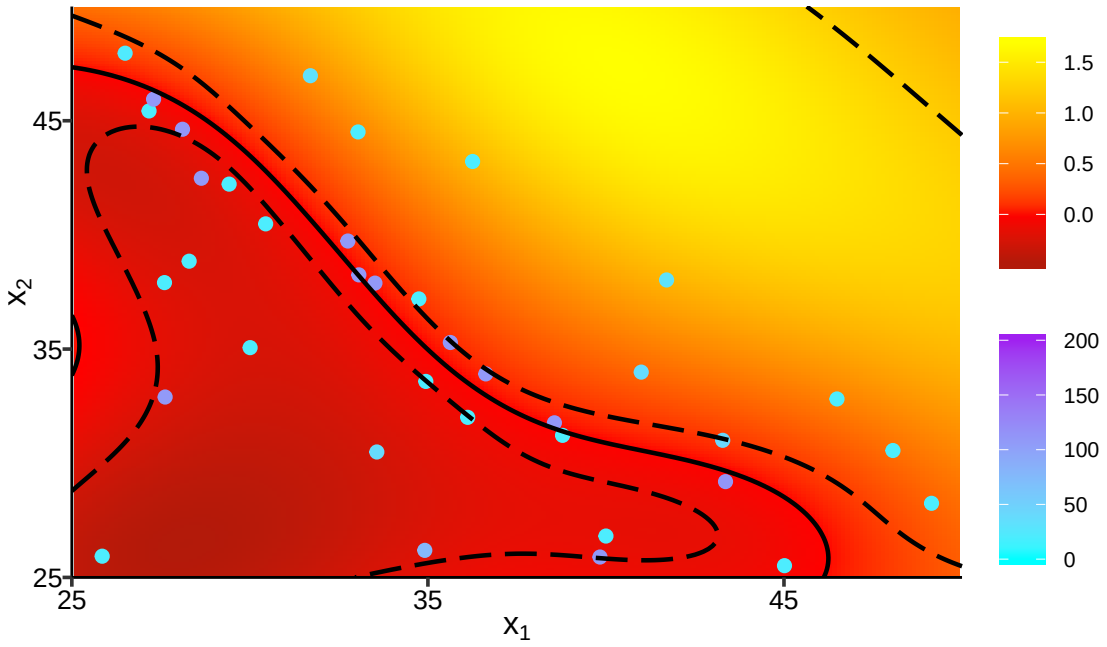

Figure 2: Timing Value Exercise Boundary using ABSUR

B: Implementations of adaptive batching design functions

Below we provide the R implementation of a simplified version of adaptive batching algorithms for estimating the stopping criteria for Bermudan options. The full version (which is hidden to maintain the double-blind review) handles a variety of Gaussian Process metamodels (specified via *method*) and implements all the proposed batching heuristics:

- `fb`: Fixed Batching
- `mlb`: Multi-Level Batching
- `rb`: Ratchet Batching
- `adsa`: Adaptive Batching Design with Stepwise Allocation
- `ddsa`: Deterministic Batching Design with Stepwise Allocation

The function's main input is *model* which is the list defining the Bermudan option and GP parameters. *t0* is one of the overhead parameters in ABSUR.

During the backward dynamic programming loop indexed by i we iterate over $t = T, T - \Delta t, \ldots, 0$ (a total of $M = T/\Delta t$ steps), and the simulator of $T(t, x)$ returns the difference between the pathwise payoff (*fsim$payoff* below) along a trajectory of $(\mathbf{X}_{t:T})$ that is based on the forward exercise strategy summarized by the forward-looking $\{\hat{\mathfrak{S}}_s, s > t\}$, and $h(t, x)$ (`immPayoff` in the function).

To accomplish this, the main variables are:

- `fits`: list of GP simulators at each time step; the hyper-parameters of the GP metamodel are re-trained every `model$update.freq` steps;
- `all.X`: matrix of designs; the first d dimensions are the input sites $\mathbf{x}_{1:k_n}$, the $d+1$ dimension is the observation mean $\bar{\mathbf{y}}_{1:k_n}$, and the $d+2$ dimension is the observation variance;
- `batch_matrix`: the batch size $\mathbf{r}_{1:k_n}$ for each input.

```
osp.seq.batch.design.simplified <- function(model, method="km", t0 = 0.01)
{
  M <- model$T/model$dt

  # parameters in absur
  r_lower = model$r.cand[1]
  r_upper = min(model$r.cand[length(model$r.cand)], 0.1 * model$total.budget)
  r_interval = seq(r_lower, r_upper, length = 1000)
  theta_for_optim = c(0.1371, 0.000815, 1.9871E-6)  # c_{ovh} in eq. (13)
  batch_matrix <- matrix(rep(0, M*model$seq.design.size), ncol=M)

  # batch parameter in adsa and ddsa
  c_batch = 20 / model$dim

  fits <- list()   # list of emulator objects at each step
  pilot.paths <- list()
  # when to refit the whole GP
  update.kernel.iters <- seq(0,model$seq.design.size,by=model$update.freq)

  # set-up a skeleton to understand the distribution of X
  pilot.paths[[1]] <- model$sim.func( matrix(rep(model$x0[1:model$dim],
                                                 5*model$init.size),
                                             nrow=5*model$init.size, byrow=T),
                                      model, model$dt)
  for (i in 2:(M-1))
    pilot.paths[[i]] <- model$sim.func( pilot.paths[[i-1]], model, model$dt)

  pilot.paths[[1]] <- pilot.paths[[3]]
  init.grid <- pilot.paths[[M-1]]
  budget.used <- rep(0,M-1)

  ############ step back in time
  for (i in (M-1):1)
  {
    all.X <- matrix( rep(0, (model$dim+2)*model$seq.design.size), ncol=model$dim+2)

    # candidates are based on a rectangle defined by empirical quantiles of the init.grid
    lhs.rect <- matrix(rep(0, 2*model$dim), ncol=2)
    for (jj in 1:model$dim)
        lhs.rect[jj,] <- quantile( init.grid[,jj], c(0.02, 0.98) )

    # Candidate grid of potential NEW sites to add; only keep in-the-money sites
    # Will be ranked using the EI acquisition function
    ei.cands <- lhs( model$cand.len, lhs.rect )  # from tgp package
    ei.cands <- ei.cands[ model$payoff.func( ei.cands,model) > 0,,drop=F]
```

```
# initial replicated design
init.grid <- model$init.grid
K0 <- dim(init.grid)[1]
big.grid <- init.grid[ rep(1:K0, model$batch.nrep),]
fsim <- forward.sim.policy( big.grid, M-i,fits[i:M],model, compact=T, offset=0)

# payoff at t
immPayoff <- model$payoff.func( init.grid, model)

# batched mean and variance at each unique input site
for (jj in 1:K0) {
  all.X[jj,model$dim+1] <- mean( fsim$payoff[jj + seq(from=0,len=model$batch.nrep,by=K0)])
              - immPayoff[ jj]
  all.X[jj,model$dim+2] <- var( fsim$payoff[ jj + seq(from=0,len=model$batch.nrep,by=K0)])
}
all.X[1:K0,1:model$dim] <- init.grid
batch_matrix[1:K0, i] <- model$batch.nrep

# create the km object
fits[[i]] <- km(y~0, design=data.frame(x=init.grid),
                response=data.frame(y=all.X[1:K0,model$dim+1]),
                noise.var=all.X[1:K0,model$dim+2]/model$batch.nrep,
                covtype=model$kernel.family,
                control=list(trace=F),
                lower=model$min.lengthscale, upper=model$max.lengthscale)

r_batch = model$batch.nrep

add.more.sites <- TRUE
k <- K0 + 1
running_budget = model$batch.nrep*K0
n <- K0 + 1

# active learning loop
while(add.more.sites)
{
  # predict on the candidate grid, using GP mean and posterior GP variance
  pred.cands <- predict(fits[[i]],data.frame(x=ei.cands), type="UK")
  cand.mean <- pred.cands$mean
  cand.sd <- pred.cands$sd
  nug <- sqrt(mean(all.X[1:(k-1), model$dim+2]))
  nug <- rep(nug, length(cand.mean))

  losses <- cf.el(cand.mean, cand.sd)

  # multiply by weights mu(x) based on the distribution of pilot paths
  dX_1 <- pilot.paths[[i]]; dX_2 <- dX_1
  for (dd in 1:model$dim) {
    dX_1[,dd] <- dX_1[,dd]/(lhs.rect[dd,2]-lhs.rect[dd,1])
    dX_2[,dd] <- dX_2[,dd]/(lhs.rect[dd,2]-lhs.rect[dd,1])
  }
  ddx <- distance( dX_1, dX_2)
  x.dens <- apply( exp(-ddx*dim(ei.cands)[1]*0.01), 2, sum)
```

```
  # use active learning measure to select new sites and associated replication
  if (model$ei.func == 'absur') {
    overhead = 3 * model$dim * CalcOverhead(theta_for_optim[1],
                                            theta_for_optim[2],
                                            theta_for_optim[3],
                                            k + 1)
    al.weights <- cf.absur(cand.mean, cand.sd, nug,
                           r_interval, overhead, t0)

    # select site and replication with highest EI score
    x.dens.matrix <- matrix(x.dens, nrow=length(x.dens), ncol=length(r_interval))
    ei.weights <- x.dens.matrix * al.weights

    # select next input location
    max_index <- which.max(ei.weights)
    x_index <- max_index %% length(x.dens)
    x_index <- ifelse(x_index, x_index, length(x.dens))
    add.grid <- ei.cands[x_index,,drop=F]

    # select associated batch size
    r_index <- (max_index - 1) / model$cand.len + 1
    r_batch <- min(round(r_interval[r_index]), model$total.budget - running_budget)
  } else {
    # use active learning measure to select new sites
    adaptive.gamma <- (quantile(cand.mean, 0.75, na.rm = T) -
                         quantile(cand.mean, 0.25, na.rm = T))/mean(cand.sd)
    al.weights <- cf.smcu(cand.mean, cand.sd, adaptive.gamma)
    x.dens2 <- dlnorm( ei.cands[,1], meanlog=log(model$x0[1]) +
                         (model$r-model$div - 0.5*model$sigma[1]^2)*i*model$dt,
                       sdlog = model$sigma[1]*sqrt(i*model$dt) )
    x.dens2 <- x.dens2*dlnorm( ei.cands[,2],
                               meanlog=log(model$x0[2]) +
                                 (model$r-model$div-0.5*model$sigma[2]^2)*i*model$dt,
                           sdlog = model$sigma[2]*sqrt(i*model$dt) )

    # select site with highest EI score
    ei.weights <- x.dens*al.weights
    x_index <- which.max(ei.weights)
    add.grid <- ei.cands[x_index,,drop=F]
  }

  # use adaptive batching algorithms to select batch size
  if (model$batch.heuristic == 'fb') {
    r_batch = model$batch.nrep
  } else {
    r0 = min(model$total.budget - running_budget, round(c_batch*sqrt(k)))
    if (model$batch.heuristic == 'adsa') {
      adsa_batch <- batch.adsa(fits[[i]], batch_matrix[1:(n - 1), i],
                               ei.cands, x.dens2, add.grid, r0, nug, method)
      add.grid <- adsa_batch$x_optim
      r_batch <- adsa_batch$r_optim
    }
  }
```

```
  # if using up all budget, move on to next time step
  if (is.numeric(r_batch) && r_batch == 0) {
    add.more.sites <- FALSE
    next
  }

  if(is.null(add.grid) || model$batch.heuristic == 'ddsa' && k%%2) { # Reallocation
    # Indices for inputs which receive further allocation
    idx_diff = which(r_batch != batch_matrix[1:(n - 1), i])
    r_seq_diff = r_batch[idx_diff] - batch_matrix[idx_diff, i]

    ids <- seq(1, length(r_seq_diff))
    ids_rep <- unlist(mapply(rep, ids, r_seq_diff))

    newX <- all.X[idx_diff, 1:model$dim]
    newX <- matrix(unlist(mapply(rep, newX, r_seq_diff)), ncol = model$dim)

    # sample corresponding y-values
    fsim <- forward.sim.policy(newX, M-i, fits[i:M], model, compact=T, offset=0)

    # payoff at t
    immPayoff <- model$payoff.func(newX, model)
    newY <- fsim$payoff - immPayoff

    add.mean <- tapply(newY, ids_rep, mean)
    add.var <- tapply(newY, ids_rep, var)
    add.var[is.na(add.var)] <- 0.0000001

    y_new = (all.X[idx_diff, model$dim+1] * batch_matrix[idx_diff, i] +
               add.mean * r_seq_diff) / r_batch[idx_diff]
    var_new = (all.X[idx_diff, model$dim+2] * (batch_matrix[idx_diff, i] - 1) +
                 batch_matrix[idx_diff, i] * all.X[idx_diff, model$dim+1] ^ 2 +
                 add.var * (r_seq_diff - 1) +
                 r_seq_diff * add.mean ^ 2) / (batch_matrix[idx_diff, i] - 1)
    all.X[idx_diff, model$dim + 1] = y_new
    all.X[idx_diff, model$dim + 2] = var_new
    batch_matrix[1:(n - 1), i] = r_batch

    running_budget = running_budget + sum(r_seq_diff)

    if (k %in% update.kernel.iters) {
        fits[[i]] <- km(y~0, design=data.frame(x=all.X[1:n - 1, 1:model$dim]),
                        response=data.frame(y=all.X[1:n - 1, model$dim+1]),
                        noise.var=all.X[1:n - 1,model$dim+2]/r_batch,
                        covtype=model$kernel.family, control=list(trace=F),
                        lower=model$min.lengthscale, upper=model$max.lengthscale)
    } else {
        fits[[i]] <- update(fits[[i]],
                            newX=all.X[idx_diff, 1:model$dim, drop = F],
                            newy=y_new,
                            newnoise=var_new / r_batch, newX.alreadyExist=T,
                            cov.re=F)
    }
```

```
      } else { # add a new input
        add.grid <- matrix(rep(add.grid[1, ,drop=F], r_batch), nrow = r_batch, byrow=T)

        # compute corresponding y-values
        fsim <- forward.sim.policy( add.grid,M-i,fits[i:M],model,offset=0)

        immPayoff <- model$payoff.func(add.grid, model)
        add.mean <- mean(fsim$payoff - immPayoff)
        if (r_batch == 1) {
          add.var <- 0.00001
        } else {
          add.var <- var(fsim$payoff - immPayoff)
        }
        all.X[n,] <- c(add.grid[1,],add.mean,add.var)
        batch_matrix[n, i] <- r_batch

        if (k %in% update.kernel.iters) {
          fits[[i]] <- km(y~0, design=data.frame(x=all.X[1:n,1:model$dim]),
                          response=data.frame(y=all.X[1:n,model$dim+1]),
                          noise.var=all.X[1:n,model$dim+2]/r_batch,
                          covtype=model$kernel.family, control=list(trace=F),
                          lower=model$min.lengthscale, upper=model$max.lengthscale)
        } else {
          fits[[i]] <- update(fits[[i]],
                              newX=add.grid[1,,drop=F],
                              newy=add.mean,
                              newnoise=add.var / r_batch,
                              cov.re=F)
          }

        running_budget = running_budget + sum(r_batch)
        n <- n + 1
      }

      # resample the candidate set
      ei.cands <- lhs(model$cand.len, lhs.rect)
      ei.cands <- ei.cands[ model$payoff.func( ei.cands,model) > 0,,drop=F]
      if (n >= model$seq.design.size || running_budget >= model$total.budget) {
        add.more.sites <- FALSE
      }
      k <- k+1
    }
    budget.used[i] <- n
  }

  return (list(fit=fits, ndesigns=budget.used, batches = batch_matrix))
}
```

ABSUR details

The code below implements the ABSUR algorithm to select the new input x_{n+1} and its associated batch size r_{n+1}. Basically we start with expanding a candidate matrix for both input (row-wise) and batch size (column-wise). Then eq. (12) is calculated over the cross-combination of input and batch size as a weight

matrix returned by the function. In **osp.seq.batch.design.simplified**, the index of optimal candidate input and batch size is obtained by maximizing the weight function.

```
cf.absur <- function(objMean, objSd, nugget, r_cand, overhead, t0) {
  # expand mean and sd vectors to matrix of size len(x_cand) * len(r_cand)
  r_len <- length(r_cand)
  x_len <- length(objMean)
  objMean_matrix <- matrix(objMean, nrow=x_len, ncol=r_len)
  objSd_matrix <- matrix(objSd, nrow=x_len, ncol=r_len)
  r_matrix <- matrix(r_cand, nrow = x_len, ncol=r_len, byrow=TRUE)

  # EI at cross combination of candidate input and batch size
  nugget_matrix <- nugget / sqrt(r_matrix)
  d_cur = pnorm(-abs(objMean_matrix)/objSd_matrix)  # normalized distance to zero contour
  new_objSd2 <- nugget_matrix * objSd_matrix /
    sqrt(nugget_matrix ^ 2 + objSd_matrix ^ 2) # new posterior variance
  d_new <- pnorm(-abs(objMean_matrix)/new_objSd2)   # new distance to zero-contour

  # difference between next-step distance and current distance weighted by the overhead
  return( (d_cur-d_new) / (r_matrix * t0 + overhead) )
}
```

ADSA details

In ADSA, we must either calcuate the reallocation of replications over the existing inputs $\mathbf{x}_{1:k_n}$, or allocate all new replications to one new input x_{k_n+1}. The inputs are

- *fit*: the current GP simulator;
- *r_seq*: vector of batch sizes for existing inputs;
- *xtest* : test points to approximate the integral;
- *xt_dens*: density $\mu(\tilde{x}_i)$ at test points;
- *x_new*: the new selected input;
- *r0* : number of new simulations in the batch;
- *nugget*: estimated noise standard deviation $\tau(x_{k_n+1})$;
- *method*: the Gaussian Process metamodel, such as kriging, Student-t Process or heteroscedastic GP.

The function **batch.ddsa** (available upon request) first computes the additional allocation at existing inputs using the pegging algorithm. Then we calculate the expected loss metric of the above reallocation compared to the loss from adding a new input as defined in equations (19) and (21).

If we choose to reallocate the replications over the existing inputs $\mathbf{x}_{1:k_n}$, then the function returns a `NULL` value for the new input *x_optim*, and the new batch size vector for the existing inputs as *r_optim*; otherwise, it returns the new input *x_optim* is *x_new*, and the associated batch size *r_optim* is *r0*.

```
batch.adsa <- function(fit, r_seq, xtest, xt_dens, x_new, r0, nugget, method) {
  x_optim = NULL
  ddsa.res = batch.ddsa(fit, r_seq, xtest, xt_dens, r0, method)
  r_new = ddsa.res$r_new
  K = ddsa.res$K
  L = ddsa.res$L
  # determine between allocating in existing samples or moving to a new location

  # i1 - reallocation
```

```
  Delta_R = 1/r_seq - 1/r_new
  Delta_R = diag(Delta_R)

  v = solve(t(L)) %*% (solve(L) %*% K)
  I1 = diag(t(v) %*% Delta_R %*% v)

  # i2 - adding a new input
  x_all = rbind(xtest, fit@X, x_new)
  C = covMatrix(fit@covariance, x_all)$C
  k = C[1:nrow(xtest), (nrow(xtest) + nrow(fit@X) + 1)]
  k_new = C[(nrow(xtest) + 1):(nrow(xtest) + nrow(fit@X)),
            (nrow(xtest) + nrow(fit@X) + 1)]

  ss = fit@covariance@sd2

  a = solve(t(L)) %*% (solve(L) %*% k_new)
  var_new = ss - t(k_new) %*% a
  cov = k - t(K) %*% a
  I2 = vec(cov) ^ 2 / (nugget^2 / r0 + var_new[1, 1])

  if (sum(I1 * xt_dens) > sum(I2 * xt_dens)) {
    # allocate to existing samples
    r_optim = r_new
  } else {
    # go for a new location
    r_optim = r0
    x_optim = x_new
  }
  return(list(x_optim = x_optim, r_optim = r_optim))
}
```

C: Definitions of Syntethic Benchmarks

Rescaled 2D Branin-Hoo Function

We swap the two dimensions x_1 and x_2 in the original Branin-Hoo function and rescale it so that:

- the function is monotonically decreasing along x_1
- the mean of the function is 0
- the variance is 1 (empirically, based on 100,000 designs randomly selected).

```
braninsc <- function(xx)
{
  x1 <- xx[,1]
  x2 <- xx[,2]

  x1bar <- 15*x1
  x2bar <- 15 * x2 - 5

  term1 <- x1bar - 5.1*x2bar^2/(4*pi^2) + 5*x2bar/pi - 20
  term2 <- (10 - 10/(8*pi)) * cos(x2bar)

  y <- (term1^2 + term2 - 181.47) / 178
```

```
  return(y)
}
```

Rescaled 6D Hartman Function

We rescale 6D Hartman function so that the mean of the function is 0, and the variance is 1 (based on 100,000 designs randomly selected). Also we change the values of parameters α and A to descrease the slope of the bumps and make the function more evenly distributed around zero.

```
hart6sc <- function(xx)
{
  alpha <- c(0.2, 0.22, 0.28, 0.3)
  A <- c(8, 3, 10, 3.50, 1.7, 6,
         0.5, 8, 10, 1, 6, 9,
         3, 3.5, 1.7, 8, 10, 6,
         10, 6, 0.5, 8, 1, 9)
  A <- matrix(A, 4, 6, byrow=TRUE)
  P <- 10^(-4) * c(1312, 1696, 5569, 124, 8283, 5886,
                   2329, 4135, 8307, 3736, 1004, 9991,
                   2348, 1451, 3522, 2883, 3047, 6650,
                   4047, 8828, 8732, 5743, 1091, 381)
  P <- matrix(P, 4, 6, byrow=TRUE)

  xxmat <- matrix(rep(xx,times=4), 4, 6, byrow=TRUE)
  inner <- rowSums(A[,1:6]*(xxmat-P[,1:6])^2)
  outer <- sum(alpha * exp(-inner))

  y <- -outer
  y <- (y + 0.1)/0.1
  return(y)
}
```