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

This is the supplementary document for paper "Adaptive Batching for Gaussian Process Surrogates with Application in Noisy Level Set Estimation". We firstly provided the example code to regenerate Figure 6 in Section 6 of the paper. Then we included the implementation examples for several main functions. Last but not least, we specify the detailed definitions for the rescaled 2D Branin-Hoo function and 6D Hartman function, which are used for synthetic experiments in Section 5.

Figure Regeneration

This is a demo code to generate Figure 6. The plot shows the fitted exercise boundary with its 95% credible interval (solid line and dashed line) obtained with Gaussian Process for two-dimensional basket put Bermudan option. Two batch heuristics, ABSUR and ADSA, are used to select the inputs and their associated replications, which are shown as the dots and their color

Firstly we load the required libraries. Note that all implementations for estimating the optimal stopping criteria for Bermudan option with adaptive batching design are included in library `mlOSP`, which is maintained on Github by the authors of the paper. Link will be provided upon request.

```
knitr::opts_chunk$set(echo = TRUE)
library(ks)
library(fields) # for plotting purposes, use quilt.plot in 2D
library(mlOSP)
library(DiceKriging)
library(tgp)  # use lhs from there
library(randtoolbox)  # use sobol and halton QMC sequences
library(hetGP)
library(laGP)
library(ggplot2)
library(pander)
library(RColorBrewer)
library(scales)
data("int_2d")
```

Then we set up the model for basket put with parameters in Table 4, including the total simulations N_T by *total.budget*, initial design size k_0 by *init.size*, initial batch size r_0 by *batch.nrep* and kernal function K by *kernel.family*. The parameters of the model is 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)
model2d$pilot.nsims <- 1000
model2d$look.ahead <- 1
model2d$cand.len <- 1000   # size of testing set m_0
model2d$max.lengthscale <- c(40,40)
model2d$min.lengthscale <- c(3,3)
model2d$tmse.eps <- 0
```

```
model2d$ucb.gamma <- 1.96

model2d$seq.design.size <- 100
model2d$batch.nrep <- 20  # initial replication r_0
model2d$total.budget <- 2000  # budget N = r_0 * k = 2000
model2d$init.size <- 20   # initial design size k_0
model2d$init.grid <- int_2d

model2d$tmse.eps <- 0
model2d$kernel.family <- "gauss"  # kernel function for Gaussian Process
model2d$ucb.gamma <- 1.96
model2d$update.freq <- 5   # number of sequential design steps to update the GP surrogate
nReps <- 2
model2d$r.cand <- c(20, 30,40,50,60, 80, 120, 160) # r_L
```

We specify the batch heuristic with *batch.heuristic* and the sequential design framework with *ei.func*, and then fit GP simulators for the option with function **osp.seq.batch.design.simplified**. This function is a simplified version of **osp.seq.batch.design**, where the user can specify the model from a full list of *km* (kriging in package `DiceKriging`), *trainkm* (train parameters in kriging), *hetgp* (heteroscedastic Gaussian Process) and *homtp* (homoscedastic Student-t Process). The last two models are implemented in package `hetGP`. The user can also specify the adaptive batching algorithm from a full list of *fb* (Fixed Batching), *mlb* (Multi-Level Batching), *rb* (Rachet Batching), *adsa* (Adaptive Batching Design with Stepwise Allocation) and *ddsa* (Deterministic Batching Design with Stepwise Allocation), and the sequential design framework from a full list of *amcu* (Adaptive Maximum Contour Uncertainty), *mcu* (Maximum Contour Uncertainty), *csur* (Contour Stepwise Uncertainty Reduction), *icu* (Integrated Contour Uncertainty) and *absur* (used together when *batch.heuristic* is *absur*).

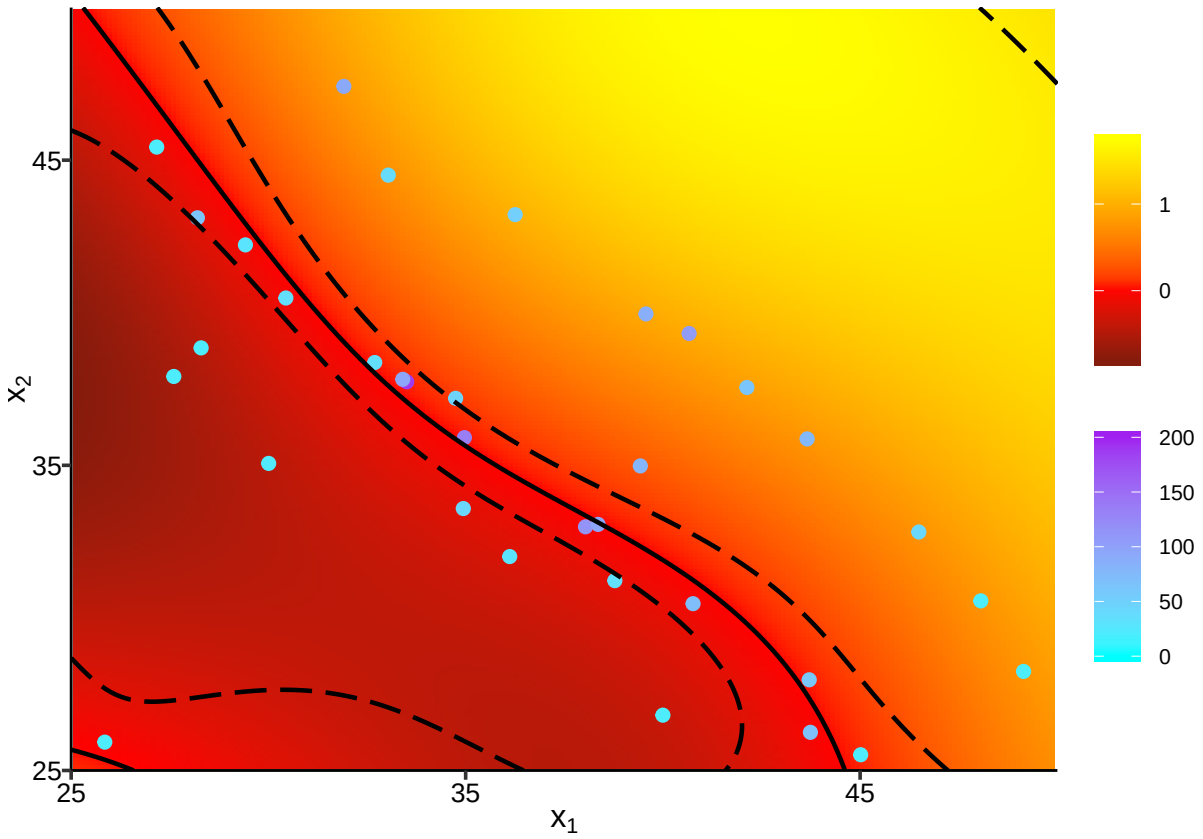
```
## GP + ADSA
set.seed(110)
model2d$batch.heuristic <- 'adsa'
model2d$ei.func <- 'amcu'
oos.obj.adsa <- osp.seq.batch.design.simplified(model2d, method="trainkm")

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

In the end, we plot the fitted exercise boundary at $t = 0.6$ with function **plt.2d.surf.with.batch**. The first argument is the fitted GP emulator (including the fitted model and the input sites), and the second argument is the batch size corresponding to the selected input sites.

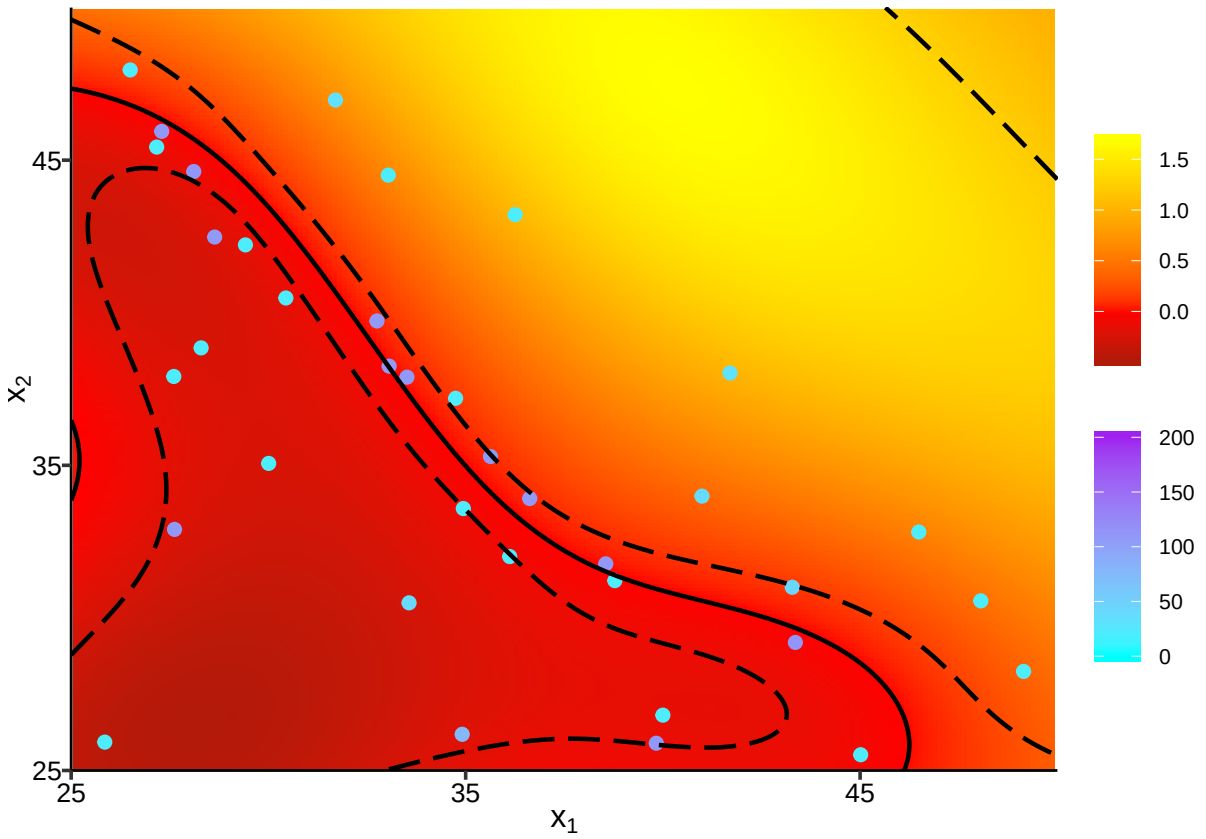
Firstly we plot the figure for results obtained with ADSA.

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



Secondly we plot the figure for results obtained with ABSUR.

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



Implementations for adaptive batching design functions

This is the implementation of a simplified function of adaptive batching algorithms for estimating the stopping criteria for Bermudan option, only for Gaussian Process with two batching heuristics: ADSA and ABSUR. The full version which handles variety of Gaussian Process based models and three other batching heuristics can be provided upon request. There are three inputs: *model* is the list defined the parameters. During the backward dynamic programming, we iterate over $t = T, T - \Delta t, \ldots, 0$ (in a total of $M = T/\Delta t$ steps), and the simulator of $T(t, x)$ returns the difference between the pathwise payoff (*fsim$payoff* in the function) along a trajectory of $(\mathbf{X}_{t:T})$ that is based on the forward exercise strategy summarized by the forward-looking $\{\hat{S}_s, s > t\}$, and $h(t, x)$ (*immPayoff* in the function). To accomplish this, several main variables are used: *fits* (list of GP simulators at each time point; the parameters of GP are re-estimated 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), and *batch_matrix* (the batch size $\mathbf{r}_{1:k_n}$ for each input).

```
detach("package:mlOSP", unload=TRUE)
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_{oh} in eq. (13)
  batch_matrix <- matrix(rep(0, M*model$seq.design.size), ncol=M)
```

```r
# 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)

  # construct the input domain where candidates will be looked for
  if (is.null(model$lhs.rect)) {
    model$lhs.rect <- 0.02
  }
  if (length(model$lhs.rect) == 1) {
    lhs.rect <- matrix(rep(0, 2*model$dim), ncol=2)
    # create a box using empirical quantiles of the init.grid cloud
    for (jj in 1:model$dim)
      lhs.rect[jj,] <- quantile( init.grid[,jj], c(model$lhs.rect, 1-model$lhs.rect) )
  } else { # already specified
    lhs.rect <- model$lhs.rect
  }

  # Candidate grid of potential NEW sites to add.
  # Will be ranked using the EI acquisition function
  # only keep in-the-money sites
  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 design
  init.grid <- model$init.grid
  K0 <- dim(init.grid)[1]
  # initial conditions for all the forward paths: replicated design with batch.nrep
  big.grid <- init.grid[ rep(1:K0, model$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
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)])
}
# use  first dim+1 columns for batched GP regression
all.X[1:K0,1:model$dim] <- init.grid
k <- K0
batch_matrix[1:K0, i] <- model$batch.nrep

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

# initialize gamma for RB and MLB
gamma <- sqrt(mean(all.X[1:K0,model$dim+2])/model$batch.nrep) / 2
r_batch = model$batch.nrep

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

# active learning loop:
# to do it longer for later points to minimize error build-up use: *(1 + i/M)
while(add.more.sites)
{
  # predict on the candidate grid.
  # Need predictive GP mean, posterior GP variance and similation Stdev
  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 based on the distribution of pilot paths
  dX_1 <- pilot.paths[[i]]; dX_2 <- ei.cands
  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])
  }
```

```
    # from package lagp
    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
      if (model$ei.func != 'icu') {
        ei.weights <- x.dens*al.weights
      } else {
        ei.weights<- al.weights # those are negative for ICU
      }

      x_index <- which.max(ei.weights)
      add.grid <- ei.cands[x_index,,drop=F]
    }

    # use active 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

      # Indexes for inputs which receives 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)

      # compute 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))
}
```

Implementations for ABSUR

This is the function used for ABSUR 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)
  a = 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
  a_new <- pnorm(-abs(objMean_matrix)/new_objSd2)   # new distance to zero-contour

  # difference between next-step ZC and current ZC weighted by the overhead
  return( (a-a_new) / (r_matrix * t0 + overhead) )
}
```

Implementations for ADSA

This is the function used for ADSA to either calcuate the reallocation of replications over the existing inputs $\mathbf{x}_{1:k_n}$, or allocate all replications to one new input x_{k_n+1}. The inputs are *fit* (the current GP simulator), *r_seq* (batch size vector for all existing inputs), *xtest* (testing points), *xt_dens* (density of testing points), *x_new* (the new selected input), *r0* (the total number of new replications), *nugget* (estimated noise standard deviation τ), and *method* (the Gaussian Process model, such as, kriging, Student-t Process or heteroscedastic GP). It firstly estimates the reallocation at existing inputs with function **batch.ddsa**. Then it calculates the loss for reallocation and adding a new input as defined in equations (19) and (21), and compares these two values. 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, 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
  if (method == "km" | method == "trainkm") {
    x_all = rbind(xtest, fit@X, x_new)
    C = covMatrix(fit@covariance, x_all)$C
    k = C[1:nrow(xtest), (nrow(xtest) + nrow(fit@X) + 1)]
    k_new = C[(nrow(xtest) + 1):(nrow(xtest) + nrow(fit@X)),
              (nrow(xtest) + nrow(fit@X) + 1)]
  } else {
    x_all = rbind(xtest, fit$X0, x_new)

    if (method == "homtp") {
      # Lower triangle matrix of covariance matrix
      C <- hetGP::cov_gen(x_all, theta=fit$theta, type="Gaussian") * fit$sigma2
    } else {
      C <- hetGP::cov_gen(x_all, theta=fit$theta, type="Gaussian") * fit$nu_hat
    }
    k = C[1:nrow(xtest), (nrow(xtest) + nrow(fit$X0) + 1)]
    k_new = C[(nrow(xtest) + 1):(nrow(xtest) + nrow(fit$X0)),
              (nrow(xtest) + nrow(fit$X0) + 1)]
  }

  if (method == "km" | method == "trainkm") {
    ss = fit@covariance@sd2
  } else {
    if (method == 'homtp') {
      ss = fit$sigma2
    } else {
      ss = fit$nu_hat
    }
  }

  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 on existing samples
    r_optim = r_new
  } else {
    # choose a new location
    r_optim = r0
    x_optim = x_new
  }
  return(list(x_optim = x_optim, r_optim = r_optim))
}
```

Rescaled functions

Rescaled 2D Branin-Hoo Function

We swap the two dimensions x_1 and x_2 in the original rescaled Branin-Hoo function and rescale it so that the function is monotonically decreasing along x_1, the mean of the function is 0, and the variance is 1 (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 smoothly distributed.

```
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)
}
```