Estimating local extrema using derivative Gaussian processes

In this vignette, we provide worked examples on simulated data to demonstrates how to use the package `dgp` to compute the posterior density of local extrama using derivative Gaussian processes, which is discussed in the paper "Semiparametric Bayesian inference for local extrema of functions in the presence of noise".

Required Packages

The code and analysis use some packages, and we load them before doing simulation study. The package `KernSmooth` is for implementing the nonparametric kernel smoothing (NKS) of Song et. al, 2006, and `ftnonpar` is used for implementing the smoothed taut string methood (STS) in (Kovac, 2007).

```
## ============================================================================
## load packages
## ============================================================================
packages <- c("KernSmooth", "ftnonpar", "emulator", "doParallel", "Rsolnp", "mvnfast")
invisible(lapply(packages, library, character.only = TRUE))
```

```
## KernSmooth 2.23 loaded
## Copyright M. P. Wand 1997-2009
```

```
## Loading required package: mvtnorm
```

```
## Loading required package: foreach
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
##
## Attaching package: 'mvnfast'
```

```
## The following objects are masked from 'package:mvtnorm':
##
##     dmvt, rmvt
```

```
library(dgp)
```

```
##
## Attaching package: 'dgp'
```

```
## The following object is masked from 'package:ftnonpar':
##
##     smqreg
```

Simulated Data

DGP Method

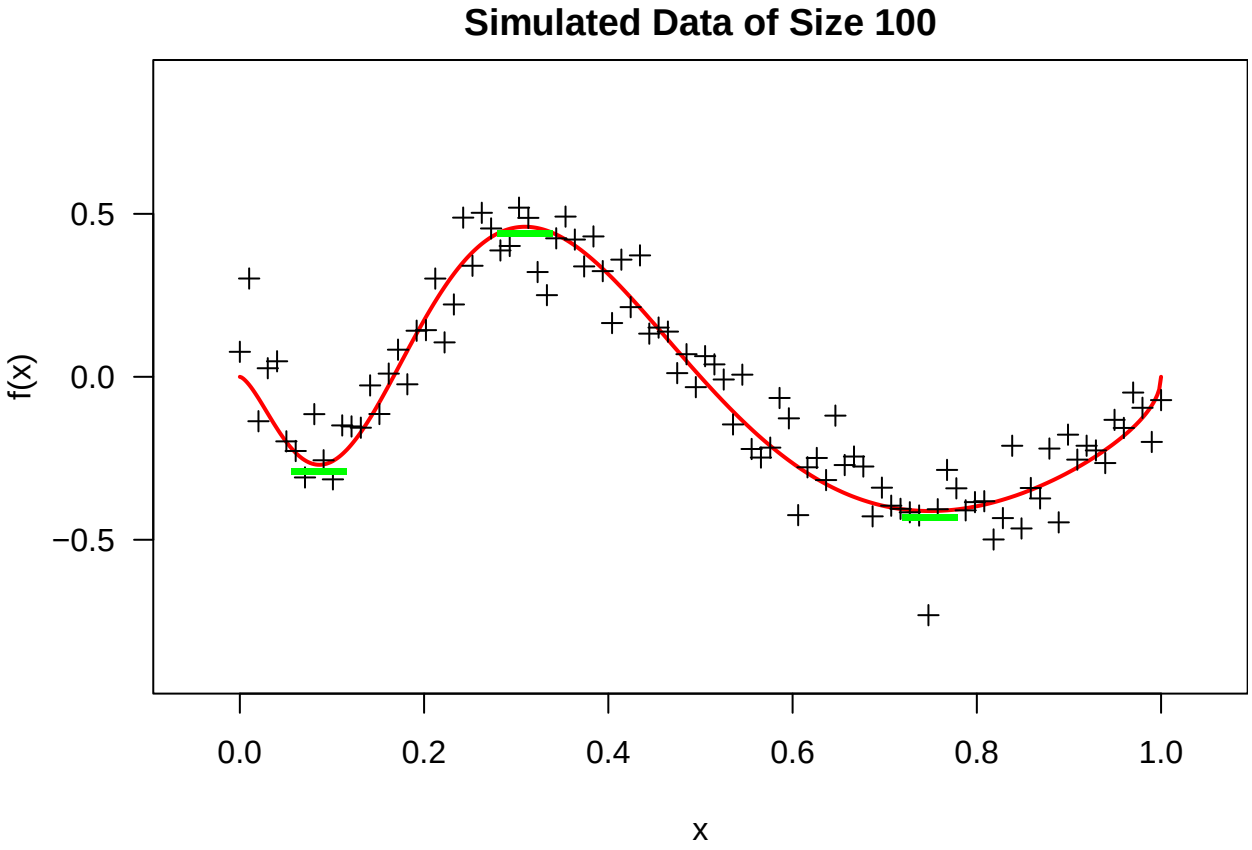
All 100 simulated data sets for analysis in the paper are stored in `./data/sim_data_n100.RData`. The simulated data are generated and plotted using the script `./data-raw/gen_sim_data.R` as shown below. Here, we show the result of data with $n = 100$. The data with $n = 500$ and $n = 1000$ can be generated as `sim_data_n100.RData`, and their data sets are saved in `./data/sim_data_n500.RData` and `./data/sim_data_n1000.RData`.

```
## ============================================================================
## Load the data
## ============================================================================
load("../data/sim_data_n100.RData", verbose = TRUE)
```

```
## Loading objects:
##   YY
##   sig
##   x_a
##   x_b
##   cri_pts
##   no_data
```

The object `YY` save the 100 simulated data sets. `x_a` and `x_b` are the end points of the domain of `x`. `cri_pts` stores the true stationary points, and `no_data` is the number of replicated data.

```
## ============================================================================
## Generate the simulated data
## ============================================================================
source("../data-raw/seed.R")
source("../data-raw/gen_sim_data.R")
```



Simulation study

Here we show how to use the functions in the `dgp` package to implement the method proposed in the paper.

First create objects needed. `H0_diff` is the distance matrix in the powered exponential kernel function.

```
## ============================================================================
## Creating needed objects
## ============================================================================
H0_diff <- lapply(YY, function(d) {
    outer(as.vector(d$x), as.vector(d$x),
          FUN = function(x1, x2) (x1 - x2))
})
# idx <- seq(x_a, x_b, length.out = 500)
n_test <- 100
x_test <- sort(seq(x_a, x_b, length.out = n_test))
n_grid <- 400
grid_t <- seq(x_a, x_b, length.out = n_grid)
```

Then estimate σ, and the hyperpameters τ and lengthscale h in the kernel by maximizing the marginal log likelihood written as the function `dgp::log_mar_lik_gp()`.

```
## ============================================================================
## Selecting parameters
## ============================================================================
cl <- makeCluster(6, type = "FORK")
registerDoParallel(cl)
EB_gp <- foreach(k = 1:no_data) %dopar% {
    res <- Rsolnp::solnp(pars = c(1, 1, 1), fun = log_mar_lik_gp,
                         LB = c(0.0001, 0.0001, 0.0001),
                         UB = c(1 / 0.0001, 1 / 0.0001, 1 / 0.0001),
                         control = list(TOL = 1e-5, trace = 0),
                         y = YY[[k]]$y, H0 = H0_diff[[k]])
    res$par
}
stopCluster(cl)
EB_gp[[1]]
```

```
## [1] 0.1155130 0.2896112 0.1300747
```

Given the hyperparameters, we apply the function `dgp::log_post_t_theory()` to compute the log posterior density of local extrema. The function requires the following arguments:

- `t`: The grid of t for evaluating the density value.
- `y`: the response value vector
- `x`: the input value vector
- `Kff`: The covariance matrix formed by the kernel function
- `A`: The matrix $K_{ff} + n\lambda I$ needed for computing the density
- `lambda`: the parameter $\lambda = \sigma^2/(n\tau^2)$
- `h`: the lengthscale parameter
- `sig`: the noise scale parameter
- `shape1`, `shape2`: the shape parameters of the beta prior on the stationary point.
- `a`, `b`: input domain.

```
## ============================================================================
## Computing log posterior by DGP beta(1, 1) and DGP beta(2, 3)
## ============================================================================
log_post <- matrix(0, no_data, n_grid)
log_post23 <- matrix(0, no_data, n_grid)
for (k in 1:no_data) {
    lambda_gp <- EB_gp[[k]][1] ^ 2 / (n * EB_gp[[k]][2] ^ 2)
    Kff_gp <- se_ker(H0 = H0_diff[[k]], tau = 1, h = EB_gp[[k]][3])
```

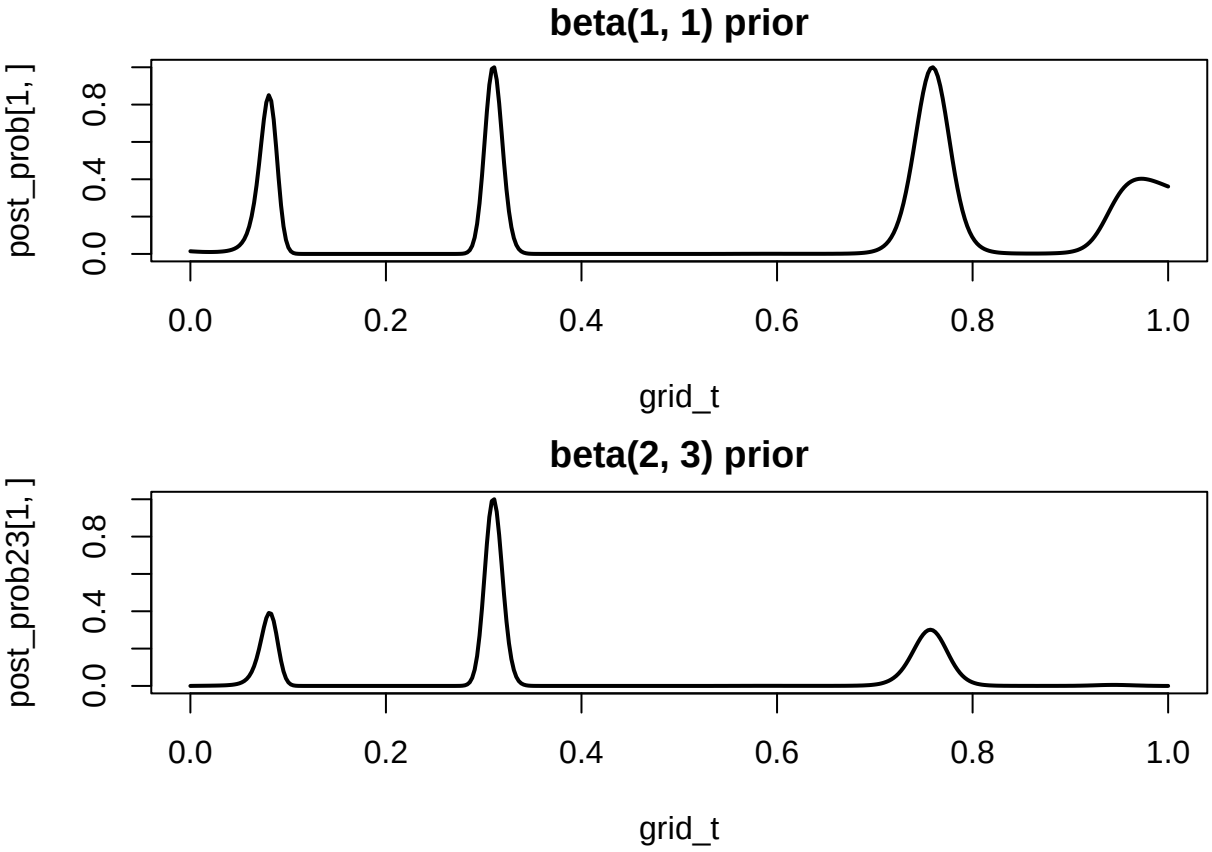
```
    A_gp <- Kff_gp + diag((n * lambda_gp), n)
    for (i in 1:n_grid) {
        log_post[k, i] <-
            log_post_t_theory(t = grid_t[i], y = YY[[k]]$y, x = YY[[k]]$x,
                              Kff = Kff_gp, A = A_gp, lambda = lambda_gp,
                              h = EB_gp[[k]][3], sig2 = EB_gp[[k]][1] ^ 2,
                              shape1 = 1, shape2 = 1, a = x_a, b = x_b)
        log_post23[k, i] <-
            log_post_t_theory(t = grid_t[i], y = YY[[k]]$y, x = YY[[k]]$x,
                              Kff = Kff_gp, A = A_gp, lambda = lambda_gp,
                              h = EB_gp[[k]][3], sig2 = EB_gp[[k]][1] ^ 2,
                              shape1 = 2, shape2 = 3, a = x_a, b = x_b)
    }
}
```

We then can transform the log density to posterior density.

```
## ============================================================================
## Transform from log posterior to posterior
## ============================================================================
post_prob <- matrix(0, no_data, n_grid)
post_prob23 <- matrix(0, no_data, n_grid)
for (k in 1:no_data) {
    post_prob[k, ] <- exp(log_post[k, ] - max(log_post[k, ]))
    post_prob23[k, ] <- exp(log_post23[k, ] - max(log_post23[k, ]))
}
```

The posterior density with beta(1, 1) prior and beta(1, 1) prior are shown below. This is the Figure 2(a)(b) in the paper when $n = 100$. The density for $n = 500$ and 1000 can be generated using the same approach.

```
par(mar = c(4, 4, 2, 1), mfrow = c(2, 1))
plot(grid_t, post_prob[1, ], type = "l", lwd = 2, main = "beta(1, 1) prior")
plot(grid_t, post_prob23[1, ], type = "l", lwd = 2, main = "beta(2, 3) prior")
```



To obtain the $(1-\alpha)100\%$ highest posterior density (HPD) intervals from the density, we can apply the function `dgp::get_hpd_interval_from_den()`. The result of the first data set is shown below.

When beta(1, 1) is used, there are four separated HPD intervals with the first three capturing one of the true critical points. As papers shows, the boundary issue can be mitigated when an informative prior such as beta(2, 3) is used or when n gets large. When beta(2, 3) is used, there are three separated HPD intervals, each capturing one of the true critical points.

```
## ============================================================================
## Compute HPD interval
## ============================================================================
## beta(1, 1)
hpdi_lst <- apply(post_prob, 1, get_hpd_interval_from_den,
                  grid_t = grid_t, target_prob = 0.95)
hpdi_1_lst <- lapply(hpdi_lst, function(x) x$ci_lower)
hpdi_2_lst <- lapply(hpdi_lst, function(x) x$ci_upper)
hpdi_1_2_lst <- list()
for (k in 1:no_data) hpdi_1_2_lst[[k]] <- cbind(hpdi_1_lst[[k]], hpdi_2_lst[[k]])

## beta(2, 3)
hpdi_lst23 <- apply(post_prob23, 1, get_hpd_interval_from_den,
                    grid_t = grid_t, target_prob = 0.95)
hpdi_1_lst23 <- lapply(hpdi_lst23, function(x) x$ci_lower)
hpdi_2_lst23 <- lapply(hpdi_lst23, function(x) x$ci_upper)
hpdi_1_2_lst23 <- list()
for (k in 1:no_data) {
    hpdi_1_2_lst23[[k]] <- cbind(hpdi_1_lst23[[k]], hpdi_2_lst23[[k]])
}
```

```r
## first data result
hpdi_1_2_lst[[1]]
```

```
##            [,1]      [,2]
## [1,] 0.05764411 0.0952381
## [2,] 0.29323308 0.3283208
## [3,] 0.71929825 0.7994987
## [4,] 0.92481203 1.0000000
```

```r
hpdi_1_2_lst23[[1]]
```

```
##            [,1]      [,2]
## [1,] 0.06015038 0.0952381
## [2,] 0.28822055 0.3333333
## [3,] 0.71929825 0.7919799
```

```r
cri_pts
```

```
## [1] 0.08632681 0.30955769 0.74905641
```

STS method

The STS method is implemented by the function `smqreg()`.

```r
## ============================================================================
## STS by Davies, P. L. and Kovac, A. (2001) and Kovac (2006)
## ============================================================================
smqreg_fit_lst <- list()
sts_est <- list()
for (k in 1:no_data) {
    smqreg_fit_lst[[k]] <- smqreg(YY[[k]]$y, verbose = FALSE)
    sts_est[[k]] <- YY[[k]]$x[smqreg_fit_lst[[k]]$loc + 1]
}
```

NKS method

The code for implementing the NKS method involves the functions in the `KernSmooth` package, and functions provided by Dr. Song including `LQfit()` and `AddCI()`. The function `nks_ci()` obtains the information about all stationary points including their index in the input sequence, point estimate, as well as confidence interval.

```r
## ============================================================================
## NKS by Song (2006)
## ============================================================================
bw_vec <- rep(0, no_data)
beta_lst <- list()
ci_song_lst_bon <- list()
all_info_lst <- list()
type1err <- 0.05
for (k in 1:no_data) {
    bw_vec[k] <- KernSmooth::dpill(x = YY[[k]]$x, y = YY[[k]]$y)
    beta_lst[[k]] <- LQfit(x = YY[[k]]$x, y = YY[[k]]$y, h = bw_vec[k])
    ci_song_lst_bon[[k]] <- AddCI(x = YY[[k]]$x, y = YY[[k]]$y, h = bw_vec[k],
                                  beta = beta_lst[[k]], alpha = type1err / 3)
    all_info_lst[[k]] <- cbind(pos = YY[[k]]$x, beta_lst[[k]],
                               ci_song_lst_bon[[k]])
}
```

```
song_der_zero_idx_lst <- find_song_der_zero_idx(all_info_lst, no_data,
                                                is.print = FALSE)
ci_pts_song_lst <- nks_ci(no_data = no_data,
                          all_info_lst = all_info_lst,
                          song_der_zero_pt_lst = song_der_zero_idx_lst)
imp_lst <- list()
for(k in 1:no_data) {
    position <- song_der_zero_idx_lst[[k]]
    x_val <- YY[[k]]$x[position]
    lCI <- ci_pts_song_lst[[k]][, 1]
    uCI <- ci_pts_song_lst[[k]][, 2]
    imp_lst[[k]] <- list(position = position,
                         x = x_val,
                         lCI = lCI,
                         uCI = uCI)
}
```

Method Comparison

Number of local extrema

In this section we compare DGP with STS and NKS in terms of number of local extrema that reproduces the results of Figure 3 in the paper for $n = 100$. The results for $n = 500$ can be reproduced by using the data `./data/sim_data_n500.RData`.

The function `get_map_lst()` obtains the maximum a posteriori (MAP) estimate from the posterior density derived from DGP.

```
## ============================================================================
## Number of local extrema
## ============================================================================
# -----------------
## DGP-beta(1, 1)
# -----------------
map_est <- get_map_lst(post_prob, grid_t, hpdi_1_2_lst)
table(unlist(lapply(map_est, length)))
```

```
##
##  3  4  5
## 47 48  5
```

```
# -----------------
## DGP-beta(2, 3)
# -----------------
map_est23 <- get_map_lst(post_prob23, grid_t, hpdi_1_2_lst23)
table(unlist(lapply(map_est23, length)))
```

```
##
##  3  4  5
## 86 13  1
```

```
# -----------------
## STS
# -----------------
table(sapply(smqreg_fit_lst, function(x) x$nmax))
```

```
##
```

```
##  2  3  5
## 50 49  1
```

```
# -----------------
## NKS
# -----------------
table(unlist(lapply(song_der_zero_idx_lst, function(x) length(x))))
```

```
##
##  3  4  5  6  7  8
## 45 10 29 11  4  1
```

RMSE

In this section we compare DGP with STS and NKS in terms of RMSE that reproduces the results of Figure 3 in the paper for $n = 100$. The results for $n = 500$ can be reproduced by using the data `./data/sim_data_n500.RData`.

```
## ============================================================================
## RMSE
## ============================================================================
b0 <- 0
b3 <- 1
b1 <- (cri_pts[1] + cri_pts[2]) / 2
b2 <- (cri_pts[2] + cri_pts[3]) / 2

# -----------------
## DGP-beta(1, 1)
# -----------------
map_mat <- matrix(0, nrow = no_data, 3)
for (i in 1:no_data) {
    x_1 <- map_est[[i]][map_est[[i]] > b0 & map_est[[i]] < b1]
    map_mat[i, 1] <- mean(unlist(x_1))
    x_2 <- map_est[[i]][map_est[[i]] > b1 & map_est[[i]] < b2]
    map_mat[i, 2] <- mean(unlist(x_2))
    x_3 <- map_est[[i]][map_est[[i]] > b2 & map_est[[i]] < b3]
    map_mat[i, 3] <- mean(unlist(x_3))
}

sqrt(mean((map_mat[, 1] - cri_pts[1]) ^ 2)) * 100
```

```
## [1] 0.668166
```

```
sqrt(mean((map_mat[, 2] - cri_pts[2]) ^ 2)) * 100
```

```
## [1] 0.8831689
```

```
sqrt(mean((map_mat[, 3] - cri_pts[3]) ^ 2)) * 100
```

```
## [1] 4.128234
```

```
# -----------------
## DGP-beta(2, 3)
# -----------------
map_mat23 <- matrix(0, nrow = no_data, 3)
for (i in 1:no_data) {
    x_1 <- map_est23[[i]][map_est23[[i]] > b0 & map_est23[[i]] < b1]
    map_mat23[i, 1] <- mean(unlist(x_1))
```

```
    x_2 <- map_est23[[i]][map_est23[[i]] > b1 & map_est23[[i]] < b2]
    map_mat23[i, 2] <- mean(unlist(x_2))
    x_3 <- map_est23[[i]][map_est23[[i]] > b2 & map_est23[[i]] < b3]
    map_mat23[i, 3] <- mean(unlist(x_3))
}

sqrt(mean((map_mat23[, 1] - cri_pts[1]) ^ 2)) * 100
```

```
## [1] 0.6480705
```

```
sqrt(mean((map_mat23[, 2] - cri_pts[2]) ^ 2)) * 100
```

```
## [1] 0.8759889
```

```
sqrt(mean((map_mat23[, 3] - cri_pts[3]) ^ 2)) * 100
```

```
## [1] 3.849926
```

```
# -----------------
## STS
# -----------------
sts_est_mat <- matrix(0, nrow = no_data, 3)
na_count_sts <- rep(0, 3)
multi_count_sts <- rep(0, 3)

for (i in 1:no_data) {
    x_1 <- sts_est[[i]][sts_est[[i]] > b0 & sts_est[[i]] < b1]
    if (length(x_1) == 0) na_count_sts[1] <- na_count_sts[1] + 1
    if (length(x_1) > 1) multi_count_sts[1] <- multi_count_sts[1] + 1
    sts_est_mat[i, 1] <- mean(x_1, na.rm = TRUE)

    x_2 <- sts_est[[i]][sts_est[[i]] > b1 & sts_est[[i]] < b2]
    if (length(x_2) == 0) na_count_sts[2] <- na_count_sts[2] + 1
    if (length(x_2) > 1) multi_count_sts[2] <- multi_count_sts[2] + 1
    sts_est_mat[i, 2] <- mean(x_2, na.rm = TRUE)

    x_3 <- sts_est[[i]][sts_est[[i]] > b2 & sts_est[[i]] < b3]
    if (length(x_3) == 0) na_count_sts[3] <- na_count_sts[3] + 1
    if (length(x_3) > 1) multi_count_sts[3] <- multi_count_sts[3] + 1
    sts_est_mat[i, 3] <- mean(x_3, na.rm = TRUE)
}

sqrt(mean((sts_est_mat[, 1][!is.nan(sts_est_mat[, 1])] - cri_pts[1]) ^ 2)) * 100
```

```
## [1] 1.645483
```

```
sqrt(mean((sts_est_mat[, 2] - cri_pts[2]) ^ 2)) * 100
```

```
## [1] 1.530308
```

```
sqrt(mean((sts_est_mat[, 3] - cri_pts[3]) ^ 2)) * 100
```

```
## [1] 3.136814
```

```
# -----------------
## NKS
# -----------------
nks_mat <- matrix(0, nrow = no_data, 3)
```

```r
multi_count_nks <- rep(0, 3)
# multi_count_2_nks <- 0
# multi_count_3_nks <- 0

for (i in 1:no_data) {
    x_1 <- imp_lst[[i]]$x[imp_lst[[i]]$x > b0 & imp_lst[[i]]$x < b1]
    if (length(x_1) > 1) multi_count_nks[1] <- multi_count_nks[1] + 1
    nks_mat[i, 1] <- mean(x_1)

    x_2 <- imp_lst[[i]]$x[imp_lst[[i]]$x > b1 & imp_lst[[i]]$x < b2]
    if (length(x_2) > 1) multi_count_nks[2] <- multi_count_nks[2] + 1
    nks_mat[i, 2] <- mean(x_2)

    x_3 <- imp_lst[[i]]$x[imp_lst[[i]]$x > b2 & imp_lst[[i]]$x < b3]
    if (length(x_3) > 1) multi_count_nks[3] <- multi_count_nks[3] + 1
    nks_mat[i, 3] <- mean(x_3)
}

c(sqrt(mean((nks_mat[, 1] - cri_pts[1]) ^ 2)),
  sqrt(mean((nks_mat[, 2] - cri_pts[2]) ^ 2)),
  sqrt(mean((nks_mat[, 3] - cri_pts[3]) ^ 2))) * 100
```

```
## [1] 1.632523 1.391695 4.981166
```

Missing local extrema

```r
## ============================================================================
## Number of simulations with missing or multiple estimated local extrema in each interval
## ============================================================================

# -----------------
## DGP-beta(1, 1)
# -----------------
sum(unlist(lapply(map_est, function(x) sum(b0 < x & x < b1))) > 1)
```

```
## [1] 0
```

```r
sum(unlist(lapply(map_est, function(x) sum(b1 < x & x < b2))) > 1)
```

```
## [1] 0
```

```r
sum(unlist(lapply(map_est, function(x) sum(b2 < x & x < b3))) > 1)
```

```
## [1] 15
```

```r
# -----------------
## DGP-beta(2, 3)
# -----------------
sum(unlist(lapply(map_est23, function(x) sum(b0 < x & x < b1))) > 1)
```

```
## [1] 0
```

```r
sum(unlist(lapply(map_est23, function(x) sum(b1 < x & x < b2))) > 1)
```

```
## [1] 0
```

```r
sum(unlist(lapply(map_est23, function(x) sum(b2 < x & x < b3))) > 1)
```

```
## [1] 14
```

```r
# -----------------
## STS
# -----------------
multi_count_sts
```

```
## [1] 0 0 1
```

```r
# -----------------
## NKS
# -----------------
multi_count_nks
```

```
## [1] 18  0 47
```

Coverage

The coverage can be computed using the main functions `get_cover()` and `get_cover_all()`.

```r
## This part takes about 10 minutes!
cl <- makeCluster(10)
registerDoParallel(cl)
n_new <- 1000
x_new <- seq(0, 1, length.out = n_new + 2)[2:n_new + 1]
n <- 100
x <- seq(0, 1, length.out = n)
y0 <- sapply(x, f0)
cover_mat <- foreach(j = 1:100, .combine = 'rbind') %dopar% {
        set.seed(j)
        y <- y0 + sig * rnorm(n)
        return(dgp::get_cover_all(x, y, n, x_new, cri_pts))
}
stopCluster(cl)
colMeans(cover_mat[cover_mat[, 1] == 3, ])
```

```
##        hpd_n bonferroni_1 bonferroni_2 bonferroni_3     cov_01_1     cov_01_2
##    3.0000000    0.6603774    0.7169811    0.7735849    0.8113208    0.7547170
##     cov_01_3     cov_05_1     cov_05_2     cov_05_3    cov_001_1    cov_001_2
##    0.7358491    0.9056604    0.8113208    0.8113208    0.9811321    0.8679245
##    cov_001_3
##    0.8301887
```