A new sample-based algorithm to compute the total sensitivity index: R code

Arnald Puy

Contents

1 Preliminary steps

```
# LOAD ALL THE REQUIRED PACKAGES ------------------------------------------

# Define function to read in all required libraries in one go:
loadPackages <- function(x) {
  for(i in x) {
    if(!require(i, character.only = TRUE)) {
      install.packages(i, dependencies = TRUE)
      library(i, character.only = TRUE)
    }
  }
}

# Load packages
loadPackages(c("sensobol", "data.table", "ggplot2", "parallel", "scales"))
```

2 Define functions

2.1 Test functions

```
# TEST FUNCTIONS ----------------------------------------------------------

# Function A1:
A1 <- function(X) {
  # Preallocate
  mat <- tmp <- vector(mode = "list", length = nrow(X))
  Y <- vector(mode = "numeric", length = nrow(X))
  for(i in 1:nrow(X)) {
    mat[[i]] <- matrix(rep(X[i, ], times = ncol(X)),
                       nrow = ncol(X),
                       ncol = ncol(X),
                       byrow = TRUE)
    mat[[i]][upper.tri(mat[[i]])] <- 1
    tmp[[i]] <- matrixStats::rowProds(mat[[i]])
    Y[[i]] <- sum(tmp[[i]] * (-1) ^ (1:ncol(X)))
  }
  return(Y)
}

# Function A2:
A2 <- function(X) {
  a <- c(0, 0.5, 3, 9, 99, 99)
  y <- 1
  for (j in 1:6) {
```

```r
    y <- y * (abs(4 * X[, j] - 2) + a[j])/(1 + a[j])
  }
  return(y)
}

# Function B1:
B1 <- function(X) {
  y <- 1
  for(j in 1:ncol(X)) {
    y <- y * (ncol(X) - X[, j]) / (ncol(X) - 0.5)
  }
  return(y)
}

# Function B2:
B2 <- function(X) {
  y <- 1
  for(j in 1:ncol(X)) {
    y <- y * ((1 + 1 / ncol(X)) ^ ncol(X)) * X[, j] ^ (1 / ncol(X))
  }
  return(y)
}

# Function B3:
B3 <- function(X) {
  a <- rep(6.52, 6)
  y <- 1
  for (j in 1:6) {
    y <- y * (abs(4 * X[, j] - 2) + a[j])/(1 + a[j])
  }
  return(y)
}

# Function C1:
C1 <- function(X) {
  y <- 1
  for (j in 1:ncol(X)) {
    y <- y * (abs(4 * X[, j] - 2))}
  return(y)
}

# Function C2:
C2 <- function(X) {
  2 ^ ncol(X) * matrixStats::rowProds(X)
}
```

2.2 Creation of the sample matrices

```r
# SAMPLE MATRICES ------------------------------------------------------------

# Function to split a matrix into N parts
CutBySize <- function(m, block.size, nb = ceiling(m / block.size)) {
  int <- m / nb
  upper <- round(1:nb * int)
  lower <- c(1, upper[-nb] + 1)
  size <- c(upper[1], diff(upper))
  cbind(lower, upper)
}

# Function to create an A and AB matrix
scrambled_sobol <- function(A, B) {
  X <- rbind(A, B)
  for(i in 1:ncol(A)) {
    AB <- A
    AB[, i] <- B[, i]
    X <- rbind(X, AB)
  }
  AB <- rbind(A, X[((2*nrow(A)) + 1):nrow(X), ])
  return(AB)
}

# Function to create replicas of the A, B and AB matrices
scrambled_replicas <- function(N, k, replicas) {
  df <- randtoolbox::sobol(n = N * replicas, dim = k * 2)
  indices <- CutBySize(nrow(df), nb = replicas)
  X <- A <- B <- out <- list()
  for(i in 1:nrow(indices)) {
    lower <- indices[i, "lower"]
    upper <- indices[i, "upper"]
    X[[i]] <- df[lower:upper, ]
  }
  for(i in seq_along(X)) {
    A[[i]] <- X[[i]][, 1:k]
    B[[i]] <- X[[i]][, (k + 1) : (k * 2)]
  }
  for(i in seq_along(A)) {
    out[[i]] <- scrambled_sobol(A[[i]], B[[i]])
  }
  return(out)
}

# Separate matrices
separate_matrices <- function(data) {
```

```
  indices <- CutBySize(nrow(data), nb = k + 1)
  X <- list()
  for(i in 1:nrow(indices)) {
    lower <- indices[i, "lower"]
    upper <- indices[i, "upper"]
    X[[i]] <- data[lower:upper, ]
  }
  return(X)
}

# Define the sample sizes of the sample matrices
# for the new algorithm
computations <- function(x, k) {
  Nb <- x
  # Total number of runs
  Nc <- x * (k + 1)
  # Warm up sample size (one fourth)
  Ntot <- Nc / 4
  # Base sample when using 1/4 of initial sample
  Nts <- Ntot / (k + 1)
  # Number of saved runs
  Nsa <- Nc - Ntot
  # Initial sample size of the saved runs
  Nin <- Nsa / (k + 1)
  # Runs to estimate the remaining 3/4 factors
  Nrem <- Nin * (4 + 1)
  # Runs saved
  Nsaved <- Nsa - Nrem
  # Number of extra runs per factor
  Nextra <- Nsaved / 4
  df <- data.frame(Nb, Nc, Ntot, Nts, Nsa, Nin, Nrem, Nsaved, Nextra)
  return(df)
}
```

```
# SOBOL' STi INDICES -----------------------------------------------------

sobol_compute_Ti <- function(Y_A, Y_AB) {
  n <- length(Y_A[!is.na(Y_A)])
  f0 <- (1 / n) * sum(Y_A, na.rm = TRUE)
  VY <- 1 / n * sum((Y_A - f0) ^ 2, na.rm = TRUE)
  STi <- ((1 / (2 * n)) * sum((Y_A - Y_AB) ^ 2, na.rm = TRUE)) / VY
  return(STi)
}

sobol_Mapply_Ti <- function(d) {
  return(mapply(sobol_compute_Ti,
                d[, "Y_A"],
```

```
                d[, "Y_AB"]))
}

sobol_Ti <- function(Y, params) {
  # Calculate the number of parameters
  k <- length(params)
  # Calculate the length of the A matrix
  p <- length(1:(length(Y) / (k + 1)))
  # Extract the model output of the A matrix
  Y_A <- Y[1:p]
  # Extract the model output of the AB matrix
  Y_AB <- Y[(p+1):length(Y)]
  # Create vector with parameters
  parameters <- rep(params, each = length(Y_A))
  # merge vector with data table
  vec <- cbind(Y_A, Y_AB)
  out <- data.table::data.table(vec, parameters)
  # Compute Sobol'indices
  output <- out[, sobol_Mapply_Ti(.SD), by = parameters]
  return(output)
}
```

3 Load analytical values

```
# READ IN THE ANALYTICAL VALUES, COMPUTED BY SAMUELE LO PIANO -----------------

# Read the .csv file
AE <- fread("AE_df.csv")

# Re-arrange
analytical <- melt(AE, id.vars = "Function") %>%
  split(., .$Function) %>%
  lapply(., function(x) x[, .(value)])
```

4 Run the model

We compare the performance of the Jansen (1999) estimator with the new algorithm at n sample sizes, for $n = 2^4, 2^5, ..., 2^{13}$. At each sample size, we create a Sobol' matrix $(50n, 2k)$: from row 1 to n is the first replica, from row $n + 1$ to $2n$ is the second replica, and so on until the fifty replica. The first k matrix is labelled $\mathbf{A}$ and the second k matrix is labelled $\mathbf{B}$.

For each replica, we create k additional matrices $\mathbf{A}_\mathbf{B}^i$, $i = 1, 2, ..., k$, where the k matrix is composed of all columns of the $\mathbf{A}$ matrix except the i-th column, which is the i-th column of the $\mathbf{B}$ matrix.

Following Saltelli et al. (2010), we use only the $\mathbf{A}$ and the $\mathbf{A}_{\mathbf{B}}^{i}$ matrices as we only compute the total sensitivity index S_{Ti}.

At a given sample size n, our algorithm proceeds as follows: we run a test function in the $\mathbf{A}$ matrix $[f(\mathbf{A})]$ and in each $\mathbf{A}_{\mathbf{B}}^{i}$ matrix $[f(\mathbf{A}_{\mathbf{B}}^{i})]$, thus obtaining $1+k$ vectors of size n with the model output Y. The total number of runs to compute S_{Ti} is thus $n(1+k)$.

From each of these vectors of size n, we retrieve the first $\frac{1}{4}$ values, and compute S_{Ti}. This is the "warm-up" sample in our paper. We observe the results of the sensitivity analysis, and freeze $\frac{1}{4}$ of the parameters (those 2 with the lowest S_{Ti}; they will be retrieved later on). The other $\frac{3}{4}$ of each vector is the "saved sample".

Let's recapitulate at this point to see how our algorithm allows to save some model runs. For instance, let's assume that $n = 16$. At this stage, we have used a total of 88 model runs: $\frac{1}{4}n$ have been applied to the $\mathbf{A}$ matrix and to each $\mathbf{A}_{\mathbf{B}}^{i}$ $(4 * 7 = 28)$; plus $\frac{3}{4}n$ have been applied to the $\mathbf{A}$ matrix and the $\mathbf{A}_{\mathbf{B}}^{i}$ matrices of the 4 most important parameters $(12 * 5 = 60)$. This means that we have 24 extra runs ($[n(k+1)] - 88 = 24$; $112 - 88 = 24$) to divide into the 4 most important parameters to improve their estimation (6 extra model runs per parameter).

We then retrieve the $\mathbf{A}_{\mathbf{B}}^{i}$ matrices of those parameters that have not been fixed, and for each one of them, we select the 6 rows where we will compute the extra model runs. The $i-th$ column of each $\mathbf{A}_{\mathbf{B}}^{i}$ matrix is substituted by a vector of size $\frac{1}{4}n$ formed by randomly distributed numbers within [0, 1]. We will call this new matrices $\mathbf{A}_{\mathbf{X}}^{i}$ matrices. We compute the test function in each $\mathbf{A}_{\mathbf{X}}^{i}$ matrix $[f(\mathbf{A}_{\mathbf{X}}^{i})]$, and obtain 4 vectors of size $\frac{1}{4}n$ with the model output Y obtained using the new sampled points; this is the "extra sample" in our paper.

At this stage, and for each non-fixed parameter, we can compute n effects $f(\mathbf{A}) - f(\mathbf{A}_{\mathbf{B}}^{i})$; a total of 64 if $n = 16$. With the new effects $f(\mathbf{A}) - f(\mathbf{A}_{\mathbf{X}}^{i})$ and $f(\mathbf{A}_{\mathbf{B}}^{i}) - f(\mathbf{A}_{\mathbf{X}}^{i})$, we can add $6 * 2$ (12) extra effects to each non-fixed model input (a gross gain of 48). If added to the previously computed effects $f(\mathbf{A}) - f(\mathbf{A}_{\mathbf{B}}^{i})$ ($n = 16$ per factor), this means that each of the four important factor receives a total of 28 effects. For a total number of model runs of $n(k+1) = 112$, we are therefore able to compute $(4 * 2) + (28 * 4) = 120$ effects.

Once S_{Ti} has been computed for the four most important factors (using the 112 abovementioned effects), we merge the results with the S_{Ti} values obtained from the two less important parameters (which were calculated using 8 effects).

4.1 Define settings

```
# DEFINE GENERAL SETTINGS ----------------------------------------------------

# Create vector with the name of the test functions
test_functions <- c("A1", "A2", "B1", "B2", "B3", "C1", "C2")

# Vector power of two
x <- seq(4, 13, 1)

# Get the initial sample sizes
Nb <- sapply(x, function(x) 2 ^ x)
```

```r
# Set number of factors
k <- 6
params <- paste("X", 1:k, sep = "")

# Set number of sample matrix replicas
replicas <- 50
```

4.2 New algorithm

```r
# DEFINE THE NEW ALGORITHM ------------------------------------------------

# Define the new estimator
new_estimator <- function(A, sample.size, params, model, type) {
  settings <- computations(sample.size, length(params))
  # Compute model output on the initial sample size
  Y.initial <- model(A)
  if(type == "old") { # RUN THE TRADITIONAL APPROACH --------------
    ##############################################################
    output <- sobol_Ti(Y.initial, params) %>%
      .[, model.runs := settings$Nc] %>%
      .[, algorithm:= "old"]
  }
  if(type == "new") { # RUN THE NEW ALGORITHM ---------------------
    ##############################################################
    # Add a column with the parameters and the model output
    A <- data.table(A)[, Y:= cbind(Y.initial)] %>%
      .[, parameters:= rep(c("A", params), each = settings$Nb)] %>%
      setnames(., paste("V", 1:k, sep = ""), paste("X", 1:k, sep = ""))

    # STi ON THE WARM UP SAMPLE ---------------------------------------
    STi.warmup <- sobol_Ti(A[, .SD[1:settings$Nts], parameters][, Y], params)
    # Retrieve a vector with the non-fixed parameters
    STi.non.fixed <- STi.warmup[V1 > quantile(V1, probs = 1 - 75 / 100)][, parameters]
    # Vector with the A and the non-fixed parameters
    A.STi.non.fixed <- c("A", STi.non.fixed)
    # Retrieve the STi indices of the fixed parameters
    STi.fixed <- STi.warmup[V1 < quantile(V1, probs = 1 - 75 / 100)]

    # RETRIEVE ALL RUNS FOR THE MOST IMPORTANT PARAMETERS-------------------------
    YA <- A[parameters %in% A.STi.non.fixed]
    Y <- YA[, Y]

    # RUN EXTRA RUNS ----------------------------------------------------------
    # Retrieve 1/4 of the rows of the AB matrices of the most
    # important parameters to compute the extra model runs
```

```r
AX <- A[parameters %in% STi.non.fixed][
  , .SD[1:((1+settings$Nextra) - 1)], parameters][, !"Y"]
# Substitute the column of the j parameter for a random numer
set.seed(666)
for(j in colnames(AX[, 2:6])) {
  AX[parameters == j, j] <- runif(settings$Nextra)
}
# Compute model output
Y.extra <- model(as.matrix(AX[, -1]))
# Add model output
AX <- data.table(AX)[, Y_AX:= cbind(Y.extra)]
# Create a vector with the new points located at the proper place
AY.extra <- AX[, c(Y_AX, rep(NA, settings$Nts+settings$Nextra)), parameters]
# ADD NA to cover the A matrix
temp <- rbind(data.table(parameters = "A", V1 = rep(NA, settings$Nb)),
              AY.extra) %>%
  .[, .(V1)] %>%
  setnames(., "V1", "Y_AX")
# Bind
Y_AX <- cbind(YA, temp)[!parameters == "A"][, Y_AX]
Yprove <- c(Y, Y_AX)

# Calculate the number of parameters
k <- length(STi.non.fixed)
# Extract the model output of the A matrix
p <- length(1:(length(Yprove) / ((2 * k) + 1)))
# Extract the model output of the A matrix
Y_A <- Y[1:p]
# Extract the model output of the AB matrix
Y_AB <- Y[(p+1): (p * (k + 1))]
# Extract the model output of the AX matrix
Y_AX <- Yprove[(p * (k + 1) + 1): length(Yprove)]
# Create vector with parameters
parameters <- rep(STi.non.fixed, each = length(Y_A))
# merge vector with data table
vec <- cbind(Y_A, Y_AB, Y_AX)
out <- data.table::data.table(vec, parameters)

# Define parameters
n <- length(Y_A[!is.na(Y_A)])
f0 <- (1 / n) * sum(Y_A, na.rm = TRUE)
VY <- 1 / n * sum((Y_A - f0) ^ 2, na.rm = TRUE)

# Compute
first <- out[, .(parameters, Y_A, Y_AB)] %>%
  setnames(., c("Y_A", "Y_AB"), c("one", "two"))
second <- out[, .(parameters, Y_A, Y_AX)] %>%
```

```
      na.omit() %>%
      setnames(., c("Y_A", "Y_AX"), c("one", "two"))
    third <- out[, .(parameters, Y_AB, Y_AX)] %>%
      na.omit() %>%
      setnames(., c("Y_AB", "Y_AX"), c("one", "two"))
    # Calculate output
    output <- rbind(first, second, third) %>%
      .[order(parameters)] %>%
      .[, ((1 / (2 * .N)) * sum((one - two) ^ 2, na.rm = TRUE)) / VY, parameters] %>%
      rbind(., STi.fixed) %>%
      .[order(parameters)] %>%
      .[, model.runs := settings$Nc] %>%
      .[, algorithm:= "new"]
  }
  return(output)
}

# Define the new algorithm
new_algorithm <- function(sample.size, params, model, type, replicas) {
  settings <- computations(sample.size, length(params))
  A <- scrambled_replicas(settings$Nb, k = length(params), replicas)
  out <- lapply(A, function(x) new_estimator(x, sample.size, params, model, type))
  return(out)
}
```

4.3 Run the model

```
# RUN THE MODEL ------------------------------------------------------------

out <- list()
run_model <- as.list(c(test_functions))
names(run_model) <- test_functions
estimators <- c("new", "old")
for(i in names(run_model)) {
  if(i == "A1") {
    test_F <- A1
  } else if(i == "A2") {
    test_F <- A2
  } else if(i == "B1") {
    test_F <- B1
  } else if(i == "B2") {
    test_F <- B1
  } else if(i == "B3") {
    test_F <- B3
  } else if(i == "C1") {
    test_F <- C1
```

```
  } else if(i == "C2") {
    test_F <- C2
  }
  out[[i]] <- mclapply(estimators, function(x)
    lapply(Nb, function(Nb) new_algorithm(Nb,
                                          params = params,
                                          model = test_F,
                                          type = x,
                                          replicas = replicas)),
    mc.cores = detectCores() - 1)
}
```

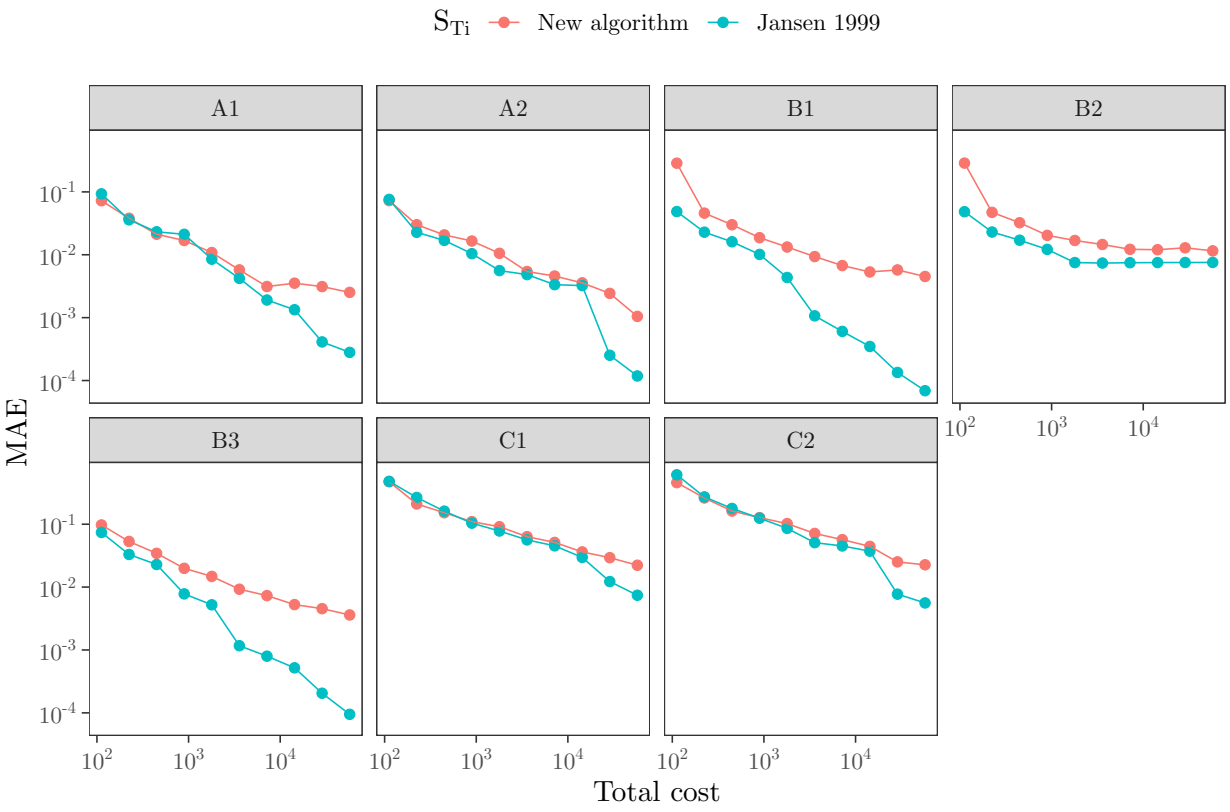
4.4 Compute the Mean Absolute Error (MAE)

```
# COMPUTE MAE ------------------------------------------------------------------

# Arrange data
temp <- lapply(out, function(x) lapply(x, function(y)
  lapply(y, function(z) rbindlist(z, idcol = "replicas")))) %>%
  lapply(., function(x) lapply(x, function(y) rbindlist(y, idcol = "N"))) %>%
  lapply(., function(x) rbindlist(x))

# Merge indices with analytical values
for(i in names(temp)) {
  temp[[i]] <- cbind(temp[[i]], analytical[[i]])
}

# Compute the MAE
MAE <- rbindlist(temp, idcol = "Function") %>%
  setnames(., c("V1", "value"), c("estimated", "analytical")) %>%
  .[, .(MAE = mean(abs(estimated - analytical))),
    .(Function, model.runs, algorithm)] %>%
  .[, model.runs:= as.numeric(model.runs)]
```

4.5 Plot results

```
# PLOT MAE ---------------------------------------------------------------------

ggplot(MAE, aes(model.runs, MAE, color = algorithm)) +
  geom_point() +
  geom_line() +
  scale_color_discrete(name = expression(S[Ti]),
                       labels = c("New algorithm", "Jansen 1999")) +
  scale_y_log10(labels = trans_format("log10", math_format(10^.x))) +
  scale_x_log10(labels = trans_format("log10", math_format(10^.x))) +
```

```
  labs(x = "Total cost",
       y = "MAE") +
  facet_wrap(~Function,
             ncol = 4) +
  theme_bw() +
  theme(aspect.ratio = 1,
        legend.position = "top",
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        legend.background = element_rect(fill = "transparent",
                                         color = NA),
        legend.key = element_rect(fill = "transparent",
                                  color = NA))
```



References

Jansen, M. 1999. "Analysis of variance designs for model output." *Computer Physics Communications* 117 (1-2): 35–43. https://doi.org/10.1016/S0010-4655(98)00154-4.

Saltelli, A., P. Annoni, I. Azzini, F. Campolongo, M. Ratto, and S. Tarantola. 2010. "Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index." *Computer Physics Communications* 181 (2): 259–70. https://doi.org/10.1016/j.cpc.2009.09.018.