Simulation Scenario Design

Contents

Here we show the common initialization script used for all scenarios throughout our simulations.

Set Scenario Parameters

We begin by defining the following named parameters in the global environment:

- `size_or_power`: value is `"size"` if the scenario corresponds to H_0; value is `"power"` if scenario corresponds to H_1
- `alpha`: the significance level to use for testing
- `num_permutation`: the number of permutation statistics used to compute each AMKAT P-value
- `num_replicates`: the number of data set replicates to simulate under the current scenario
- `x_type`: value is `"cts"` if $\boldsymbol{X}$ is simulated with continuous components; value is `"snp"` if $\boldsymbol{X}$ is simulated as SNP set data
- `error_distribution`: value is `"normal"` if random error vectors are simulated as multivariate normal; value is `"cauchy"` if simulated as multivariate Cauchy
- `error_correlation_strength`: the absolute value of the pairwise correlation between each component of $\boldsymbol{Y}$ (directions vary across pairs of components)
- `n`: the sample size for each replicate data set
- `p`: the dimension of $\boldsymbol{X}$
- `signal_strength`: a multiplier for the strength of the signal, where a value of 1 is baseline. Only relevant under H_1.
- `signal_density`: a character string indicating the number of signal variables; value is `"sparse"` or `"dense"`, with the corresponding number of signal variables depending upon the value of `x_type`. Only relevant under H_1.

Together, these parameters fully define the scenario to be simulated.

We define an example set of scenario parameter values which we will use to demonstrate the remainder of the setup:

```
size_or_power <- "power"
alpha <- 0.05
num_permutations <- 1000
num_replicates <- 1000
x_type <- "snp"
error_distribution <- "normal"
```

```
signal_strength <- 1
signal_density <- "sparse"
signal_correlation <- "low"
error_correlation_strength <- 0.5
n <- 150
p <- 3000
```

Run Initialization Script

Once scenario parameters have been defined in the global environment, we source the script `source_scripts/initialize_simulation_scenario.R` in order to create the remaining objects used to simulate data for the scenario. We now go through the contents of the script, which can be viewed at **(insert link)**.

Candidate Kernels

After calling a script to define the filename for the scenario, the script defines the set of candidate kernels used by each test.

```
# Set of candidate kernels to use in testing
candidate_kernels <- c('gau', 'lin', 'quad', 'exp')
if (x_type == 'snp') candidate_kernels <- c(candidate_kernels, 'IBS')
```

Model Parameters

Next, the true population parameters for the model under the current scenario are defined, excluding those related to the effect functions $h_1(\cdot), \ldots, h_4(\cdot)$.

```
### Y (RESPONSE VECTOR) ###

# Fix number of Y variables
num_y_variables <- 4



### EPSILON (RANDOM ERROR VECTOR) ###

epsilon_mean_vector <- rep(0, times = num_y_variables)

epsilon_covariance_matrix <- # initialize with all entries equal to rho
  matrix(error_correlation_strength,
         nrow = num_y_variables, ncol = num_y_variables)

# assign checkerboard +/- pattern to random error covariance matrix
for (i in 1:num_y_variables) for (j in 1:num_y_variables) if ((i + j) %% 2 == 1)
  epsilon_covariance_matrix[i, j] <- -error_correlation_strength

# set variance of Y components to 1 (covariances now equal correlations)
diag(epsilon_covariance_matrix) <- rep(1, times = num_y_variables)

# invert covariance matrix if random errors are multivariate Cauchy
if (error_distribution == 'cauchy') # uses precision matrix
  epsilon_precision_matrix <-
  as.matrix(Matrix::forceSymmetric(solve(epsilon_covariance_matrix)))
```

```r
### X (VECTOR OF PREDICTORS) ###

# Continuous X
if (x_type == 'cts') {

  x_mean_vector <- rep(0, times = p)

  x_covariance_matrix <- diag(p) # initialize

  # apply short-range correlation structure to covariance matrix
  for (i in 1:p) for (j in i:p) x_covariance_matrix[i, j] <- 0.6 ^ (abs(i - j))
  x_covariance_matrix <- as.matrix(Matrix::forceSymmetric(x_covariance_matrix))

}

# Discrete X (SNP-set data simulated using sim1000G)
if (x_type == 'snp') {

  # directory containing reference SNP-set data file from sim1000G package
  dir_snp_reference <- system.file('examples', package = 'sim1000G')

  # reference data file
  vcf_file <- file.path(dir_snp_reference, 'region.vcf.gz')

  # create reference data object from reference file using sim1000G
  vcf_reference_object <-
    sim1000G::readVCF(vcf_file, maxNumberOfVariants = 2000, min_maf = 0.05)

}
```

```
## [#.......] Reading VCF file..
```

```
## Rows: 569 Columns: 104
## -- Column specification --------------------------------------------------------
## Delimiter: "\t"
## chr (101): ID, REF, ALT, FILTER, INFO, FORMAT, NA06984, NA06989, NA12347, NA...
## dbl   (3): #CHROM, POS, QUAL
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## [##......] Chromosome:   4  Mbp:  77.35628  Region Size:  347.154 kb  Num of individuals: 95
## [##......] Before filtering  Num of variants: 567 Num of individuals: 95
## [###.....] After filtering  Num of variants: 567 Num of individuals: 95
```

```r
### W (COVARIATE VECTOR) ###

# matrix containing the covariate coefficient vector for each Y component
covariate_coefficient_matrix <-
  cbind(c(0.3, 0.1),    # coefficient vector for Y_1
        c(0.2, -0.2),   # coefficient vector for Y_2
        c(-0.2, -0.3),  # coefficient vector for Y_3
        c(-0.1, 0.2))   # coefficient vector for Y_4
```

Signal Variable Indices

The indices of $\boldsymbol{X}$ corresponding to signal variables contributing to the effect on $\boldsymbol{Y}$ are defined according to the data type of $\boldsymbol{X}$ and the signal density:

```
# Case 1: X is continuous
#   In this case, X exhibits a short-range correlation structure,
#   with the correlation between component i and j equal to 0.6^(i-j).
if (x_type == 'cts') {

  # Subcase A: Sparse signal set (7 signal variables)
  if (signal_density == 'sparse') {

    signal_indices_shared <- c(2, 4, 6, 8) # shared signals for all Y components
    signal_indices_y1 <- 3                 # additional signals for Y_1
    signal_indices_y2_only <- 5            # additional signals for Y_2
    signal_indices_y3 <- 7                 # additional signals for Y_3
    signal_indices_y4 <- 7                 # additional signals for Y_4

  }

  # Subcase B: Dense signal set (80 signal variables)
  if (signal_density == 'dense') {

    # shared signals for all Y components
    #   clustered in groups of 4 with a 4-variable gap between each group
    signal_indices_shared <- c((5:8), (13:16), (21:24), (29:32), (37:40),
                               (45:48), (53:56), (61:64), (69:72), (77:80))

    # additional signals for Y_2
    #   clustered as with the shared components, but offset by 4 variables,
    #   so that the clusters for Y_2 occupy the gaps between the shared clusters
    signal_indices_y2_only <-
      c((3:4), (9:12), (17:20), (25:28), (33:36), (41:44), (49:52), (57:60),
        (65:68), (73:76), (81:82))

    # additional signals for Y_1
    #   subset of the additional signals for Y_2, located on cluster boundaries;
    #   half of the signals also affect Y_3; the other half also affect Y_4
    signal_indices_y1 <- c(28, 33, 36, 41, 44, 49, 52, 57, 60, 65)

    # additional signals for Y_3
    #   subset of the additional signals for Y_2, located on cluster boundaries;
    #   five of the signals also affect Y_1; the other ten also affect Y_4
    signal_indices_y3 <- c(28, 33, 36, 41, 44,
                           4, 9, 12, 17, 20, 25, 68, 73, 76, 81)

    # additional signals for Y_4
    #   subset of the additional signals for Y_2, located on cluster boundaries;
    #   five of the signals also affect Y_1; the other ten also affect Y_3
    signal_indices_y4 <- c(49, 52, 57, 60, 65,
                           4, 9, 12, 17, 20, 25, 68, 73, 76, 81)

  }
```

```
}

# Case 2: X is discrete SNP-set data
if (x_type == 'snp') {

  ### BASE ORIGINATING SET FOR SIGNAL VARIABLES ###

  # We begin by defining three regions of the reference SNP-set data that
  #  exhibit relatively low correlation among components
  region1 <- 132:(132 + 46)
  region2 <- 213:(213 + 30)
  region3 <- 505:(505 + 44)
  # Sparse signal set (28 signals)
  if (signal_density == 'sparse') {
    if (signal_correlation == 'high') {
      # Use the first 28 components of X as the base signal set
      signal_set <- 1:28
    } else {
      # Select signal variables from among the low-correlation regions
      signal_set <- c(region1[c(39, 41:46)],
                      region2[c(2:8)],
                      region3[c(9, 10, 11, 13, 15, 16, 17, 36:40, 42, 43)])
    }
  }
  # Dense signal set (122 signals)
  # Note: originating set will have 123 elements; only 122 will be used
  # for the actual signal set
  if (signal_density == 'dense') {
    if (signal_correlation == 'high') {
      # Use the first 123 components of X as the base signal set
      signal_set <- 1:123
    } else {
      # Use the entirety of the three low-corr regions as the signal set
      signal_set <- c(region1, region2, region3)
    }
  }

  ### Indices of Signal Set Used By Each Effect Function ###

  # Descriptions of spatial patterns of indices pertain to their distribution
  # within the signal set; note that the signal set may not correspond to a
  # contiguous region of X

  # shared signals for all Y components
  #   clustered in groups of 2 with a 2-variable gap between each group
  signal_indices_shared <-
    signal_set[sort(union(seq(from = 1, to = length(signal_set), by = 4),
                          seq(from = 2, to = length(signal_set), by = 4)))]

  # additional signals for Y_2
  #   clustered as with the shared components, but offset by 4 variables,
  #   so that the clusters for Y_2 occupy the gaps between the shared clusters
  signal_indices_y2_only <-
```

```
    signal_set[sort(union(seq(from = 3, to = length(signal_set), by = 4),
                          seq(from = 4, to = length(signal_set), by = 4)))]

  # additional signals for Y_1
  #   subset of the additional signals for Y_2, located in left of each cluster
  signal_indices_y1 <-
    signal_set[seq(from = 3, to = length(signal_set), by = 4)]

  # additional signals for Y_3
  #   subset of the additional signals for Y_2, located in right of each cluster
  signal_indices_y3 <-
    signal_set[seq(from = 4, to = length(signal_set), by = 4)]

  # additional signals for Y_4
  #   same as the additional signals for Y_2
  signal_indices_y4 <- signal_indices_y2_only

  # ensure that lengths of shared signal set and set of remaining signals match
  if (length(signal_indices_shared) > length(signal_indices_y2_only)) {
    signal_indices_shared <-
      signal_indices_shared[c(1:length(signal_indices_y2_only))]
  }
  if (length(signal_indices_y2_only) > length(signal_indices_shared)) {
    signal_indices_y2_only <-
      signal_indices_y2_only[c(1:length(signal_indices_shared))]
  }
}

# all signals affecting Y_1 (including shared signals)
signal_indices_y1 <- union(signal_indices_shared, signal_indices_y1)

# all signals affecting Y_3 (including shared signals)
signal_indices_y3 <- union(signal_indices_shared, signal_indices_y3)

# all signals affecting Y_4 (including shared signals)
signal_indices_y4 <- union(signal_indices_shared, signal_indices_y4)
```

Sample correlation heatmaps for the sparse and dense signal sets using simulated SNP-set data can be viewed here.

Signal Strength Coefficients

Multipliers are set for the effect functions $h_1, \ldots, h_4$, including a global multiplier and additional individual multipliers for each function. Multiplier values depend on the data type of $\boldsymbol{X}$, the signal density, the distribution of the random error vector, and the scenario-specific signal strength multiplier value set by the user via the `signal_strength` variable.

```
# Case 1: X is continuous
if (x_type == 'cts') {

  # baseline global multiplier for signal strength
  baseline_signal_strength <- switch(signal_density, sparse = 1, dense = 1 / 3)

  # signal strength multipliers for specific Y components
```

```
  strength_modifier_y1 <- 1
  strength_modifier_y2 <- switch(signal_density, sparse = 3 / 4, dense = 4 / 5)
  strength_modifier_y3 <- switch(signal_density, sparse = 4 / 5, dense = 1 / 4)
  strength_modifier_y4 <- switch(signal_density, sparse = 8, dense = 6)

}

# Case 2: X is discrete SNP-set data
if (x_type == 'snp') {

  # baseline global multiplier for signal strength
  baseline_signal_strength <-
    switch(signal_density, sparse = 0.55, dense = 0.08)
  if (signal_correlation == 'high')
    baseline_signal_strength <- baseline_signal_strength / 2

  # signal strength multipliers for specific Y components
  strength_modifier_y1 <- switch(signal_density, sparse = 0.54, dense = 0.72)
  strength_modifier_y2 <- switch(signal_density, sparse = 0.6, dense = 0.5)
  strength_modifier_y3 <- 0.1
  strength_modifier_y4 <- 0.75
}

# double signal strength if random errors are distributed as multivariate Cauchy
if (error_distribution == 'cauchy')
  baseline_signal_strength <- 2 * baseline_signal_strength

# global multiplier for signal strength (including scenario-specific multiplier)
overall_signal_strength <- baseline_signal_strength * signal_strength
```

Effect Functions

The functions $h_1, \ldots, h_4$ modeling the effect of $\boldsymbol{X}$ on each component of $\boldsymbol{Y}$ are defined according to the data type of $\boldsymbol{X}$ and the signal density.

```
### Helper functions ###
# Product: Degree 2 physicist's Hermite polynomial with exp(-x^2)
#   modified scale and shape
hermitePolyDeg2 <- function(x, scale, shape, coefficient) {
  ax2 <- (scale * x) ^ 2
  return((4 * ax2 - 2) * exp(-ax2 / shape) * coefficient)
}

# Product: Degree 3 physicist's Hermite polynomial with exp(-x^2)
#   modified scale and shape
hermitePolyDeg3 <- function(x, scale, shape, coefficient) {
  ax <- scale * x
  return((8 * ax ^ 3 - 12 * ax) * exp(-ax ^ 2 / shape) * coefficient)
}

# Product: Degree 4 physicist's Hermite polynomial with exp(-x^2)
#   modified scale and shape
hermitePolyDeg4 <- function(x, scale, shape, coefficient) {
  ax <- scale * x
```

```
  return((16 * ax ^ 4 - 48 * ax ^ 2 + 12) * exp(-ax ^ 2 / shape) * coefficient)
}

# Case 1: X is continuous
if (x_type == 'cts') {

  # Subcase A: Sparse signal set
  if (signal_density == 'sparse') {

    # h1: linear, main effects with alternating directions
    computeEffectOnY1 <- function(x, coefficient) {
      x_signals <- x[signal_indices_y1]
      return(coefficient * (x_signals[[5]] + x_signals[[1]] - x_signals[[2]] +
                              x_signals[[3]] - x_signals[[4]]))
    }

    # h2: quadratic functional form
    computeEffectOnY2 <- function(x, coefficient) {
      x_signals_shared <- x[signal_indices_shared]
      marginal_signal_for_y1 <- signal_indices_y1[[5]]
      marginal_signal_for_y3 <- signal_indices_y3[[5]]
      return(coefficient *
               (x[[signal_indices_y2_only]]^2 + x[[signal_indices_y2_only]] +
                  x[[signal_indices_y2_only]] * x[[marginal_signal_for_y1]] -
                  x[[signal_indices_y2_only]] * x[[marginal_signal_for_y3]] +
                  x[[marginal_signal_for_y3]] * x[[marginal_signal_for_y1]] +
                  x[[marginal_signal_for_y3]] * x_signals_shared[[1]] -
                  x_signals_shared[[2]] * x_signals_shared[[4]] +
                  x_signals_shared[[3]]))
    }

    # h3: nonlinear functional form (involving hermite polynomials)
    computeEffectOnY3 <- function(x, coefficient) {
      x_signals <- x[signal_indices_y3]
      return(coefficient *
               (hermitePolyDeg2(x_signals[[2]], scale = 3 / 4, shape = 1,
                                coefficient = 1) +
                  hermitePolyDeg3(x_signals[[5]], scale = 3 / 4, shape = 1,
                                  coefficient = 0.5) +
                  hermitePolyDeg3(x_signals[[1]], scale = 3 / 4, shape = 1,
                                  coefficient = 0.5) +
                  hermitePolyDeg4(x_signals[[3]], scale = 3 / 4, shape = 1,
                                  coefficient = -0.25) +
                  hermitePolyDeg4(x_signals[[4]], scale = 3 / 4, shape = 1,
                                  coefficient = -0.25)))
    }

    # h4: nonlinear functional form (involving trig functions)
    computeEffectOnY4 <- function(x, coefficient) {
      x_signals <- x[signal_indices_y4]
      return(coefficient * sum(cos(x_signals) * exp(-x_signals ^ 2 / 10)))
    }
  }
```

```
  # Subcase B: Dense signal set
  if (signal_density == 'dense') {

    # h1: linear, main effects with alternating directions
    alternating_signs <- rep(1, times = length(signal_indices_y1))
    for (i in seq(from = 2, to = length(signal_indices_y1), by = 2)) {
      alternating_signs[[i]] <- -1
    }
    computeEffectOnY1 <- function(x, coefficient) {
      return(coefficient * sum(alternating_signs * x[signal_indices_y1]))
    }

    # h2: quadratic functional form (dominated by pairwise interactions)
    computeEffectOnY2 <- function(x, coefficient) {
      return(coefficient *
               sum(x[signal_indices_shared] * x[signal_indices_y2_only]))
    }

    # h3: nonlinear functional form (involving hermite polynomials)
    computeEffectOnY3 <- function(x, coefficient) {
      x_signals <- 0.7 * x[signal_indices_y3]
      return(coefficient * sum((8 * x_signals ^ 3 - 12 * x_signals) *
                                 exp(-x_signals ^ 2 / 3)))
    }

    # h4: nonlinear functional form (involving trig functions)
    computeEffectOnY4 <- function(x, coefficient) {
      x_signals <- x[signal_indices_y4]
      return(coefficient * sum(cos(x_signals) * exp(-x_signals ^ 2 / 10)))
    }
  }
}

# Case 2: X is discrete SNP-set data
if (x_type == 'snp') {

  # h1: linear, main effects with alternating directions
  computeEffectOnY1 <- function(x, coefficient) {
    return(coefficient * sum(x[signal_indices_y1]))
  }

  # h2: quadratic functional form (dominated by pairwise interactions)
  computeEffectOnY2 <- function(x, coefficient) {
    return(coefficient *
             sum(x[signal_indices_shared] * x[signal_indices_y2_only]))
  }

  # h3: nonlinear functional form (involving hermite polynomials)
  computeEffectOnY3 <- function(x, coefficient) {
    x_signals <- 0.7 * x[signal_indices_y3]
    return(coefficient * sum((8 * x_signals ^ 3 - 12 * x_signals) *
                               exp(-x_signals ^ 2 / 3)))
  }
```

```
  # h4: nonlinear functional form (involving trig functions)
  computeEffectOnY4 <- function(x, coefficient) {
    x_signals <- x[signal_indices_y4]
    return(coefficient * sum(cos(x_signals) * exp(-x_signals ^ 2 / 10)))
  }
}

# vector-valued effect function (h1, h2, h3, h4) for effect on full Y vector
#   incorporates previously defined multipliers for signal strength
computeEffectOnY <- function(x) {

  # initialize output matrix
  out <- matrix(NA, nrow = n, ncol = num_y_variables)

  # iterate over sample observations/individuals
  for (i in seq_len(n)) { # ith observation (X_i, Y_i)

    out[i, 1] <- # h_1(X_i) (effect on Y_{i,1} of X_i)
      computeEffectOnY1(x[i, ], overall_signal_strength * strength_modifier_y1)
    out[i, 2] <- # h_2(X_i) (effect on Y_{i,2} of X_i)
      computeEffectOnY2(x[i, ], overall_signal_strength * strength_modifier_y2)
    out[i, 3] <- # h_3(X_i) (effect on Y_{i,3} of X_i)
      computeEffectOnY3(x[i, ], overall_signal_strength * strength_modifier_y3)
    out[i, 4] <- # h_4(X_i) (effect on Y_{i,4} of X_i)
      computeEffectOnY4(x[i, ], overall_signal_strength * strength_modifier_y4)
  }

  return(out)

}
```

Functions to Simulate Sample Observations of Model Components

Functions are defined for simulating observations from the model components $\boldsymbol{X}$, $\boldsymbol{W}$, $\boldsymbol{\epsilon}$ and $\boldsymbol{Y}$, with the definition of each function depending on the relevant scenario parameters.

```
### Simulate matrix of observations for X vector ###
#  each column is a component of X; each row is an observation

# Case 1: X is continuous
if (x_type == 'cts') simulateDataX <-
  function() mvrnorm(n, x_mean_vector, x_covariance_matrix)

# Case 2: X is discrete SNP-set data
if (x_type == 'snp') simulateDataX <-
  function() {

    # Initialize sim1000G using reference object
    sim1000G::startSimulation(vcf_reference_object,
                              totalNumberOfIndividuals = 1000)

    # Simulate individuals
    subject_ids <- sim1000G::generateUnrelatedIndividuals(n)
```

```
    # Simulate genotype data for SNP set
    return(sim1000G::retrieveGenotypes(subject_ids))

  }

### Simulate matrix of observations for covariate vector ###
#  each column is a covariate; each row is an observation
simulateDataCovariates <- function() {

  # First covariate (standard normal)
  covariates_column1 <- rnorm(n)

  # Second covariate (Bernoulli)
  covariates_column2 <- rbinom(n, size = 1, prob = 0.4)

  return(cbind(covariates_column1, covariates_column2))

}

### Simulate matrix of observations for random error vector ###
#  each column is a covariate; each row is an observation

# Case 1: Random error vector distribution is multivariate normal
if (error_distribution == 'normal') simulateDataEpsilon <-
  function() MASS::mvrnorm(
    n, epsilon_mean_vector, epsilon_covariance_matrix)

# Case 2: Random error vector distribution is multivariate Cauchy
if (error_distribution == 'cauchy') simulateDataEpsilon <-
  function() LaplacesDemon::rmvcp(
    n, epsilon_mean_vector, epsilon_precision_matrix)


### Simulate matrix of observations for response vector Y ###
#  using simulated data matrices for X and for W (covariates)
#  each column is a covariate; each row is an observation

# Case 1: Size simulation (H_0 is true)
if (size_or_power == 'size') {
  simulateDataY <- function(x, covariates) {
    covariate_effects_on_y <- covariates %*% covariate_coefficient_matrix
    epsilon <- simulateDataEpsilon()
    return(covariate_effects_on_y + epsilon)
  }
}

# Case 2: Power simulation (H_1 is true)
if (size_or_power == 'power') {
  simulateDataY <- function(x, covariates) {
    x_effects_on_y <- computeEffectOnY(x)
    covariate_effects_on_y <- covariates %*% covariate_coefficient_matrix
    epsilon <- simulateDataEpsilon()
    return(covariate_effects_on_y + x_effects_on_y + epsilon)
```

```
  }
}
```