Package 'ggdmcModel'

May 8, 2026

Title Model Builders for 'ggdmc' Package

Version 0.2.9.0

Date 2025-07-15

Maintainer Yi-Shin Lin <yishinlin001@gmail.com>

Description A suite of tools for specifying and examining experimental designs related to choice response time models (e.g., the Diffusion Decision Model). This package allows users to define how experimental factors influence one or more model parameters using R-style formula syntax, while also checking the logical consistency of these associations. Additionally, it integrates with the 'ggdmc' package, which employs Differential Evolution Markov Chain Monte Carlo (DE-MCMC) sampling to optimise model parameters. For further details on the model-building approach, see Heathcote, Lin, Reynolds, Strickland, Gretton, and Matzke (2019) <doi:10.3758/s13428-018-1067-y>.

License GPL (>= 2)

URL https://github.com/yxlin/ggdmcModel

Imports Rcpp (>= 1.0.7), methods

Depends R (>= 3.5.0)

LinkingTo Rcpp (>= 1.0.7), RcppArmadillo (>= 0.10.7.5.0), ggdmcHeaders

RoxygenNote 7.3.2

Encoding UTF-8

NeedsCompilation yes

Author Yi-Shin Lin [aut, cre]

Repository CRAN

Date/Publication 2025-07-19 08:30:02 UTC

Contents

`BuildDMI` *Build Data Model Instance*

Description

Constructs a Data Model Instance (DMI) from data and model specifications. The DMI builder can handle different model types including the Linear Ballistic Accumulator, the Diffusion Decision and hyperparameter. The process of building a 'hyperparameter' DMI amounts to constructing a joint distribution over conventional statistical models.

Usage

```
BuildDMI(data, model)
```

Arguments

`data`	A data frame to be converted to a DMI object.
`model`	A model specification object of class `model` containing parameters, and other model-specific information. This is typically created using the 'BuildModel' function.

Value

A 'dmi' object or a list of 'dmi' objects (multiple subjects), with structure:

- For choice RT models: Returns a named list of 'dmi' objects (one per subject)
- For hyperparameter models: Returns a single 'dmi' object

Each 'dmi' object contains:

- 'model' - The model specification
- 'data' - The processed data (a list)
- 'node_1_index' - Index mapping for first nodes (LBA only)
- 'is_positive_drift' - A logical vector indicating drift directions. For the LBA model, each element corresponds to an accumulator. For the DDM, each element represents a condition. In the DDM, a positive drift direction corresponds to a correct response (i.e., the accumulator reaches the upper bound), and vice versa.

Model Types Supported

‘"lba"‘ Linear Ballistic Accumulator model

‘"hyper"‘ Hyperparameter model

‘"fastdm"‘ Diffusion Decision model

Examples

```
# Hyperparameter model example
hyper_model <- BuildModel(
    p_map = list(A = "1", B = "1", mean_v = "M", sd_v = "1", st0 = "1", t0 = "1"),
    match_map = list(M = list(s1 = "r1", s2 = "r2")),
    factors = list(S = c("s1", "s2")),
    constants = c(sd_v = 1, st0 = 0),
    accumulators = c("r1", "r2"),
    type = "hyper",
    verbose = FALSE
)

# LBA model example
model <- BuildModel(
    p_map = list(A = "1", B = "1", t0 = "1", mean_v = "M", sd_v = "1", st0 = "1"),
    match_map = list(M = list(s1 = "r1", s2 = "r2")),
    factors = list(S = c("s1", "s2")),
    constants = c(st0 = 0, sd_v = 1),
    accumulators = c("r1", "r2"),
    type = "lba"
)

dat <- data.frame(
  RT = c(0.7802726, 0.7890208, 1.3222672, 0.8376305, 0.7144698),
  R  = c("r1", "r1", "r2", "r1", "r1"),
  s  = c(1, 1, 1, 1, 1),
  S  = c("s1", "s1", "s1", "s1", "s1"),
  stringsAsFactors = FALSE
)

sub_dmis <- BuildDMI(dat, model)
```

BuildModel *Build a model object*

Description

The function performs a series of syntax checks to ensure the user enters strings/values conforming the C++ internal setting.

Usage

```
BuildModel(
  p_map = list(A = "1", B = "1", mean_v = "M", sd_v = "1", st0 = "1", t0 = "1"),
  accumulators = c("r1", "r2"),
  factors = list(S = c("s1", "s2")),
  match_map = list(M = list(s1 = "r1", s2 = "r2")),
  constants = c(sd_v = 1, st0 = 0),
  type = "lba",
  print_method = "head",
  verbose = TRUE
)
```

Arguments

p_map	Descibes the association between the parameter and the experimental factor.
accumulators	Specifies the response names and their levels.
factors	Specifies a list of factors along with their levels or conditions.
match_map	Maps stimulus conditions to response levels, indicating correctness.
constants	Allows the user to fix certain model parameters at constant values.
type	The model type used in the package, "fastdm", "hyper", or "lba".
print_method	a string indicating how you want the function to print model information. • head prints the first few elements. • sample samples and prints a handful of elements. • all prints all elements. . Default to head method.
verbose	Logical; if TRUE, prints design information.

Value

A S4 'model' object containing the following slots:

- parameter_map Stores the assocation between model parameters and the factors.
- accumulators Names of internal accumulators or manifested responses.
- factors Names of the factors.
- match_map Mapping between stimuli and responses.
- constants Specifies which model parameters are fixed to constant values.
- cell_names Names of the experimental conditions aora a cells.
- parameter_x_condition_names Parameter names after associated with conditions.
- model_boolean A 3D Boolean array guiding the allocation of model parameters to conditions.
- pnames Names of the model parameter associated with conditons.
- npar Numbers of parameters.
- type a string indicating the model type.

Examples

```
## A diffusion decision model
model <- BuildModel(
    p_map = list(
        a = c("S", "COLOUR"), v = c("NOISE"), z = "1", d = "1", sz = "1", sv = "1",
        t0 = "1", st0 = "1", s = "1", precision = "1"
    ),
    match_map = list(M = list(left = "z_key", right = "x_key")),
    factors = list(
        S = c("left", "right"), COLOUR = c("red", "blue"),
        NOISE = c("high", "moderate", "low")
    ),
    constants = c(d = 0, s = 1, st0 = 0, sv = 0, precision = 3),
    accumulators = c("z_key", "x_key"),
    type = "fastdm"
)

## A LBA model
model <- BuildModel(
    p_map = list(
        A = "1", B = c("S", "COLOR"), t0 = "1", mean_v = c("NOISE", "M"),
        sd_v = "M", st0 = "1"
    ),
    match_map = list(M = list(left = "z_key", right = "x_key")),
    factors = list(
        S = c("left", "right"),
        COLOR = c("red", "blue"),
        NOISE = c("high", "moderate", "low")
    ),
    constants = c(st0 = 0, sd_v.false = 1),
    accumulators = c("z_key", "x_key"),
    type = "lba"
)
```

---

build_cell_names_r *Find All Possible Conditions*

Description

Constructs all possible condition combinations (i.e., cells) based on experimental factors, parameter mappings, and response definitions. Returns both cell names and sorted factor definitions.

Usage

```
build_cell_names_r(parameter_map_r, factors_r, responses_r)
```

Arguments

parameter_map_r	An Rcpp::List where each element is a character vector mapping parameters to conditions. Names should correspond to parameters.
factors_r	An Rcpp::List where each element is a character vector of factor levels. Names should correspond to factor names.
responses_r	A character vector (std::vector<std::string>) of response/accumulator names.

Details

The function:

1. Converts R lists to 'C++' maps for efficient processing
2. Generates all condition combinations via Cartesian product
3. Handles special parameter mappings (like mapping accumulators to conditions)
4. Returns both cell names and the factor structure used

Value

An Rcpp::List with two elements:

- `cell_names`: Character vector of all possible condition combinations
- `sortedFactors`: The processed factor structure used to generate cells

Typical Workflow

This function is typically used to:

1. Establish the full experimental design space
2. Verify factor/parameter compatibility
3. Generate condition labels for model specification

This function primarily is to debug the internal process of model building.

Examples

```
# A simple example
p_map <- list(A = "1", B = "1", t0 = "1", mean_v = "M", sd_v = "1",
              st0 ="1")
factors <- list(S = c("s1", "s2"))
responses <- c("r1", "r2")
result <- build_cell_names_r(p_map, factors, responses)

# cat("B (2 factors), t0, mean_v (3 factors), sd_v (2 factors)")
p_map <- list(
    A = "H", B = c("S", "G"), t0 = "E", mean_v = c("D", "H", "M"),
    sd_v = c("D", "M"), st0 = "1"
)
factors <- list(
```

```
    S = c("s1", "s2", "s3"), D = c("d1", "d2"), E = c("e1", "e2"),
    G = c("g1", "g2", "g3"), H = c("h1", "h2", "h3", "h4", "h5")
)
responses <- c("r1", "r2", "r3")
result <- build_cell_names_r(p_map, factors, responses)
```

`build_model_boolean_r` *Build Model Boolean*

Description

Constructs a 3D boolean array indicating parameter-condition-response association to represent the experimental design.

Usage

```
build_model_boolean_r(parameter_map_r, factors_r, accumulators_r, match_map_r)
```

Arguments

`parameter_map_r`	An Rcpp::List where each element maps parameters to conditions (character vector). The element names indicates the model parameter. The element content is the factor name that assocaites with a model parameter. `1` represents no assocation.
`factors_r`	An Rcpp::List where each element defines factor levels (character vector). Names should be factor names.
`accumulators_r`	A character vector (std::vector<std::string>) of accumulator names. I use 'accumulator' to remind the difference of the implicit accumulator and the manifested response. Mostly, you may mix the two; however, sometimes, merging the two concepts may result in conceptual errors.
`match_map_r`	An Rcpp::List that defines the mapping between stimuli and responses, specifying which response are considered correct or incorrect. (This is a nested list structure).

Details

The function:

1. Converts all R inputs to C++ maps for efficient processing
2. Builds experimental design cells using `build_cell_names`
3. Processes parameter-condition mappings with `add_M`
4. Applies match map constraints to determine valid combinations
5. Returns results as a 3D logical array compatible with R

Value

An R logical array with dimensions:

- 1st dimension: Parameters (column)
- 2nd dimension: Conditions (row)
- 3rd dimension: Responses (slice)

Where ‘TRUE‘ indicates the model assumes that a model parameter (1st dimension) affects a condition (2nd dimension) at a particular response (3rd dimension).

Typical Use Case

Used when you need to:

- Validate experimental design completeness
- Generate design matrices for model fitting
- Check response-condition constraints

Examples

```
p_map <- list(A = "1", B = "1", mean_v = "M", sd_v = "1", st0 = "1",
              t0 = "1")
match_map <- list(M = list(s1 = "r1", s2 = "r2"))
factors <- list(S = c("s1", "s2"))
accumulators <- c("r1", "r2")
result <- build_model_boolean_r(p_map, factors, accumulators, match_map)
```

---

dmi-class *An S4 Class Representing a Data-Model Instance*

Description

The Data-Model Instance, 'dmi', class binds a model specification object with a corresponding experimental dataset. This structure provides a unified container used for fitting cognitive models, simulating responses, or conducting posterior predictive checks.

Details

Unlike the previous version that used data frames, the ‘dmi‘ class expects the data input as a **list** of trial-level records, optimised for internal modelling functions.

Value

An object of class 'dmi' to be passed to functions for model fitting, likelihood evaluation, simulation, or diagnostics.

Slots

`model` An object of class 'model' that defines the structure of the cognitive model, including parameter mappings, accumulators, and condition associations.

`data` A list representing the observed dataset. Each element typically corresponds to a condition or trial grouping, containing relevant variables (e.g., RTs, responses).

`node_1_index` An internal integer matrix used in LBA-type models to indicate the accumulator or node associated with the correct response (e.g., index of node 1).

`is_positive_drift` A logical flag used in models where drift direction matters, indicating whether the modelled drift rate is constrained to be positive (in the LBA model) or is going to the positive direction (in the DDM). This can be important for proper interpretation of parameters.

Structure

An object of class 'dmi' has the following slots:

Purpose

This class provides a complete representation of the modelling context by combining the experimental data and model structure. It serves as the standard input to fitting algorithms, allowing for parameter estimation, simulation, and model checking in accumulator-based cognitive models (e.g., LBA, DDM).

`get_node_1_index_r` *Get Index Mapping for the Node 1 Accumulator*

Description

Generates an integer matrix mapping experimental design cells to their corresponding indexes of the node 1 accumulator. The node 1 accumulator is the theoretical accumulator that reaches the threshold first. This function is primarily used for the LBA model.

Usage

```
get_node_1_index_r(parameter_map_r, factors_r, accumulators_r)
```

Arguments

`parameter_map_r` An Rcpp::List where each element is a character vector mapping parameters to conditions. Names should correspond to parameters.

`factors_r` An Rcpp::List where each element is a character vector of factor levels. Names should correspond to factor names.

`accumulators_r` A character vector of response accumulator names.

Details

The function:

1. Computes node indices for each condition-response pair
2. Returns results as an R-compatible integer matrix

Value

An integer matrix with dimensions:

- Rows: Experimental conditions (cells)
- Columns: Accumulators (responses)

Where values represent parameter indices for each condition-response combination.

Examples

```
cat("Flexible stimulus name")
p_map <- list(A = "1", B = "S", t0 = "E", mean_v = c("D", "M"),
              sd_v = "M", st0 = "1")
factors <- list(S = c("sti_1", "sti_2", "sti_3", "sti_4"),
                D = c("d1", "d2"), E = c("e1", "e2"))
responses <- c("resp_1", "resp_2", "resp_3", "resp_4")

# Get node indices
result <- get_node_1_index_r(p_map, factors, responses)
print(dim(result)[[1]])
# 64
```

---

get_pnames *Get Free Parameter Names from Model*

Description

Extracts the names of free parameters from an S4 model object, with optional debugging output to inspect both free and constant parameters.

Usage

```
get_pnames(model_r, debug = FALSE)
```

Arguments

model_r	An S4 object containing the model specification and design
debug	Logical flag indicating whether to print debugging information about both free and fixed parameters (default: FALSE)

Details

The function:

1. Creates a new design object from the model
2. Optionally prints debugging information about all parameters
3. Returns only the names of free (non-constant) parameters

Value

A character vector of free parameter names in the model

Debugging Output

When 'debug = TRUE', the function prints:

- Free parameters (those being estimated)
- Constants (fixed parameters)

Examples

```
model <- BuildModel(
  p_map = list(A = "1", B = "1", mean_v = "M", sd_v = "1", st0 = "1",
               t0 = "1"),
  match_map = list(M = list(s1 = "r1", s2 = "r2")),
  factors = list(S = c("s1", "s2")),
  constants = c(A = 0.75, mean_v.false = 1.5, sd_v = 1, st0 = 0),
  accumulators = c("r1", "r2"),
  type = "lba")

pnames <- get_pnames(model)
```

---

is_core_parameter_x_condition

Parameter Mapping and Condition Processing Utilities

Description

A set of helper functions for processing parameter mappings across experimental conditions. These functions are used internally for building the model Boolean array.

Usage

```
is_core_parameter_x_condition(parameter_map_r, factors_r)

is_parameter_x_condition(parameter_map_r, factors_r)

get_stimulus_level_r(parameter_map_r, factors_r, accumulators_r)

get_factor_cells_r(parameter_map_r, factors_r, accumulators_r)
```

Arguments

`parameter_map_r`
: A named list mapping parameters to conditions and factors. Example structure: `list(A = "1", B = "1", t0 = "1", mean_v = "M", sd_v = "1", st0 = "1")` Where:

  - '1' indicates this parameter is constant across conditions
  - "M" indicates this parameter is associated with the internal matching factor. It changes depends on whether it is a match (i.e., correct) response or a mismatched (i.e., incorrect) response.
  - Other strings indicate factor dependencies

`factors_r`
: A named list of experimental factors and their levels. Example: `list(S = c("red", "blue"))`

`accumulators_r`
: A character vector of accumulator names. Example: `c("r1", "r2")`

Details

These functions work together to:

- Analyse parameter mappings across experimental conditions
- Identify which parameters vary by conditions
- Generate appropriate stimulus levels and factor combinations

Value

is_core_parameter_x_condition Logical vector indicating whether core parameters (before associating with any conditions) are factor-dependent

is_parameter_x_condition Logical vector indicating whether parameters are factor-dependent

get_stimulus_level_r Character vector of stimulus levels for each accumulator

get_factor_cells_r List of factor combinations for each accumulator

Examples

```
p_map <- list(A = "1", B = "1", t0 = "1", mean_v = "M", sd_v = "1", st0 = "1")
factors <- list(S = c("red", "blue"))
accumulators <- c("r1", "r2")

# Check which parameters are core (not condition-dependent)
is_core_parameter_x_condition(p_map, factors)

# Get stimulus levels for each accumulator
get_stimulus_level_r(p_map, factors, accumulators)
```

`model-class` *An S4 class Representing a Cognitive Model Object.*

Description

The 'model' class stores information that defines how parameters in a cognitive model are associated with experimental conditions, responses, and other design factors. This object is typically created as part of the model specification process and is used as input to fitting functions or simulation routines.

Value

An object of class 'model', used to configure and fit cognitive decision models to experimental data.

Slots

`parameter_map` A named list or structure indicating how each model parameter varies with experimental factors (e.g., which parameters depend on which conditions).

`accumulators` A character vector naming the accumulators in the model (e.g., for racing models or diffusion models with multiple response alternatives).

`factors` A named list where each element is a factor in the experimental design, and each value is a vector of levels for that factor.

`match_map` A list specifying which responses are considered correct or incorrect for each condition. Typically used in decision models to differentiate match/non-match.

`constants` A named list of model parameters that are fixed to user-defined values, rather than estimated.

`cell_names` A character vector giving the names of each condition cell in the design Boolean array (e.g., 's1.d1.r1', 's1.d1.r2', 's1.d2.r1', etc.), derived from crossing factor levels.

`parameter_x_condition_names` A character vector naming how each parameter is associated with particular condition cells.

`model_boolean` A 3D logical array. Its dimensions are:

- slice: accumulators,
- row: cells (i.e., conditions),
- column: free parameters

, indicating whether a parameter is free to vary for a given accumulator and condition.

`pnames` A character vector listing the names of all free parameters in the model.

`npar` An integer giving the total number of free parameters in the model.

`type` A character string indicating the type of model (e.g., 'fastdm' for the diffusion model described in Voss, Rothermund, and Voss (2004) <doi:10.3758/BF03196893>.)

Structure

An object of class 'model' contains the following slots:

Purpose

This class object encapsulates all necessary mappings and constraints required for model fitting. It is used by the fitting engine to determine which parameters vary, what parameters are fixed, and how each condition affects the model structure.

`split_parameter_x_condition`
Map Experimental Conditions to Model Parameters

Description

Binds experimental conditions to model parameters by combining parameter mappings and experimental factors, automatically handling the 'M' (matching) factor, specifically for the Linear Ballisitic Accumulation Model. `split_parameter_x_condition` separates bound parameters and conditions.

Usage

```
split_parameter_x_condition(parameter_M_r)

bind_condition2parameters_r(parameter_map_r, factors_r)
```

Arguments

`parameter_M_r`	a string vector of parameter x condition.
`parameter_map_r`	A named list received from R (converted to Rcpp::List) where: • Names correspond to parameter names • Elements are character vectors mapping conditions to parameter
`factors_r`	A named list of experimental factors where: • Names are factor names • Elements are character vectors of factor levels

Details

This function:

1. Converts R lists to C++ std::map containers for efficient lookup
2. Processes the parameter mapping through 'add_M()' to handle response mappings
3. Returns human-readable parameter-condition pairs

Value

A character vector where each element represents a parameter-condition binding in the format 'parameter.condition'. The special 'M' factor is to represent matching and non-matching true/false in the LBA model.

C++ Implementation

The function uses:

- Rcpp::List to take the 'list' from R and convert it to C++ std::map for efficient key-value lookups
- std::vector for storing the resulting parameter-condition pairs
- Rcpp::CharacterVector for returning the result to R

Examples

```
p_map <- list(A = "1", B = "1", t0 = "1", mean_v = c("M", "S"), sd_v = "1",
          st0 = "1")
factors <- list(S = c("s1", "s2"))
parameter_M <-bind_condition2parameters_r(p_map, factors)
# [1] "A"               "B"               "mean_v.s1.false" "mean_v.s1.true"
# [5] "mean_v.s2.false" "mean_v.s2.true"  "sd_v"            "st0"
# [9] "t0"

result <- split_parameter_x_condition(parameter_M)
# [[1]]
# [1] "A"
#
# [[2]]
# [1] "B"
#
# [[3]]
# [1] "mean_v" "s1"     "false"
#
# [[4]]
# [1] "mean_v" "s1"     "true"
#
# [[5]]
# [1] "mean_v" "s2"     "false"
#
# [[6]]
# [1] "mean_v" "s2"     "true"
#
# [[7]]
# [1] "sd_v"
#
# [[8]]
# [1] "st0"
#
# [[9]]
# [1] "t0"
```

`table_parameters` *Tabulate Model Parameter*

Description

Functions for inspecting and displaying parameter structures in models built with ‘ggdmcModel‘.

Usage

```
table_parameters(model_r, parameters_r)

print_parameter_map(model_r)
```

Arguments

`model_r`	An S4 model object created by `BuildModel`.
`parameters_r`	Numeric vector of parameter values (for ‘table_parameters‘ only)

Details

These functions help analyse whether the parameter and the factor are constructed as `BuildModel` specified:

- ‘table_parameters()‘ creates a tabular representation showing how parameters map to stimuli, responses, and other model components
- ‘print_parameter_map()‘ displays the model’s parameter mapping.

Value

table_parameters Returns a List in matrix form showing how parameters map to model parameters

print_parameter_map Prints the parameter mapping structure and returns invisibly as integer status (0 for success)

Examples

```
# Build a model first
model <- BuildModel(
    p_map = list(a = "1", v = "S", z = "1", d = "1", sz = "1", sv = "1", t0 = "1",
                 st0 = "1", s = "1"),
    match_map = list(M = list(s1 = "r1", s2 = "r2")),
    factors = list(S = c("s1", "s2")),
    constants = c(d = 1, s = 1, sv = 1, sz = 0.5, st0 = 0),
    accumulators = c("r1", "r2"),
    type = "fastdm"
)

# Tabulate a parameter vector to examine how the factor-dependent
```

```
# drift rate maps to the condition, s1 and s2.
p_vector <- c(a = 1, sv = 0.2, sz = 0.25, t0 = 0.15, v.s1 = 4, v.s2 = 2, z = .38)

pmat <- table_parameters(model, p_vector)
# Transpose the result to get a more readable format
result <- lapply(pmat, function(x) {
    t(x)
})

print(result)
# $s1.r1
#    a d s st0 sv  sz  t0    v z
# r1 1 1 1   0  1 0.5 0.2 0.25 4
# r2 1 1 1   0  1 0.5 0.2 0.25 4
#
# $s1.r2
#    a d s st0 sv  sz  t0    v z
# r1 1 1 1   0  1 0.5 0.2 0.25 4
# r2 1 1 1   0  1 0.5 0.2 0.25 4
#
# $s2.r1
#    a d s st0 sv  sz  t0    v z
# r1 1 1 1   0  1 0.5 0.2 0.15 4
# r2 1 1 1   0  1 0.5 0.2 0.15 4
#
# $s2.r2
#    a d s st0 sv  sz  t0    v z
# r1 1 1 1   0  1 0.5 0.2 0.15 4
# r2 1 1 1   0  1 0.5 0.2 0.15 4

# Print the parameter map
tmp <- print_parameter_map(model)
# All parameters: a       d       s       st0     sv      sz      t0
#                 v.s1    v.s2    z
# Core parameters: a      d       s       st0     sv      sz      t0
#                  v       z
# Free parameters: a      t0      v.s1    v.s2    z
# Constant values: d: 1   s: 1    st0: 0  sv: 1   sz: 0.5

# Parameter map:
#
# 1. When the second row is 1, it indicates that the parameter is fixed.
# The internal machinery goes to the 'constant' to find its value. Note
# the constant will be sorted alphabetically.
# 2. When the second row is 0, it indicates that the parameter is free.
# The internal machinery goes to the p_vector to find its value.
# When doing MCMC sampling, a new p_vector is proposed by the sampler at
# every iteration.

# Cell, s1.r1:
# Acc 0: 0 0 1 2 3 4 1 2 4 <- C++ index
#        1 0 0 0 0 0 1 1 1 <- Whether the parameter is fixed
# Acc 1: 0 0 1 2 3 4 1 2 4
```

```
#        1 0 0 0 0 0 1 1 1
#
# Cell, s1.r2:
# Acc 0: 0 0 1 2 3 4 1 2 4
#        1 0 0 0 0 0 1 1 1
# Acc 1: 0 0 1 2 3 4 1 2 4
#        1 0 0 0 0 0 1 1 1
#
# Cell, s2.r1:
# Acc 0: 0 0 1 2 3 4 1 3 4
#        1 0 0 0 0 0 1 1 1
# Acc 1: 0 0 1 2 3 4 1 3 4
#        1 0 0 0 0 0 1 1 1
#
# Cell, s2.r2:
# Acc 0: 0 0 1 2 3 4 1 3 4
#        1 0 0 0 0 0 1 1 1
# Acc 1: 0 0 1 2 3 4 1 3 4
#        1 0 0 0 0 0 1 1 1
#
# Cell (ncell =  4): s1.r1        s1.r2   s2.r1   s2.r2
```

Index