Package: macpan2 (via r-universe)

November 15, 2024

Title Fast and Flexible Compartmental Modelling

Version 1.11.1

Description Fast and flexible compartmental modelling with Template Model Builder.

License GPL-3

Depends R (>= 4.1.0)

Imports TMB (>= 1.9.0), Matrix, oor, jsonlite, MASS, memoise, stats, utils

Suggests covr, knitr, rmarkdown, dplyr, tidyr, ggplot2, cowplot, lubridate, testthat (>= 3.0.0), htmltools, visNetwork, macpan2helpers, kableExtra, broom.mixed, outbreaks, tmbstan, numDeriv, parallel, iidda.api

LinkingTo TMB, RcppEigen

VignetteBuilder knitr

BugReports https://github.com/canmod/macpan2/issues

Additional_repositories https://canmod.r-universe.dev

Config/testthat/edition 3

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2.9000

URL https://canmod.github.io/macpan2/, https://github.com/canmod/macpan2

Repository https://canmod.r-universe.dev

RemoteUrl https://github.com/canmod/macpan2

RemoteRef HEAD

RemoteSha 22a5583be733e1862e5210bf42f78066defc4c1d

Contents

`BinaryOperator` *Binary Operator*

Description

Convert a function that represents an elementwise binary operator into one that is consistent with the C++ engine. This function is intended to clarify how **macpan2** treats binary operators, which is a little different from base R. The difference is clarified in `vignette("elementwise_binary_operators")`, and `BinaryOperator` is primarily used as a resource for that vignette.

Usage

```
BinaryOperator(operator)
```

Arguments

operator	A binary operator. Valid binary operations have the following characteristics. - They are functions. - They have exactly two arguments. - They do not have any ... arguments

Value

A binary operator consistent with the C++ engine.

Examples

```
set.seed(1L)
A = matrix(abs(rnorm(6)), 3, 2)  # 3 by 2 matrix
x = matrix(abs(rnorm(3)))        # 3 by 1 matrix
y = t(abs(rnorm(2)))             # 1 by 2 matrix
times = BinaryOperator(`*`)
pow = BinaryOperator(`^`)
identical(times(A, x), times(x, A))  ## TRUE
identical(pow(A, y), pow(y, A))  ## FALSE
```

---

comparison *Comparison Functions*

Description

Comparison Functions

Usage

```
all_equal(x, y)

all_consistent(x, y)

not_all_equal(x, y)

all_not_equal(x, y)
```

Arguments

x	character object
y	character object

Functions

- `all_equal()`: Is it true that all corresponding elements of `x` and `y` are equal, have the same shape, and have no missing values?
- `all_consistent()`: Is it true that all corresponding elements of `x` and `y` are either equal or at least one is a blank string, have the same shape, and have no missing values?
- `not_all_equal()`: Complement of `all_equal`.
- `all_not_equal()`: Do not know yet. Currently unused; should we remove?

`distribution` *Distributions*

Description

Distributions which can be used to specify prior or likelihood components in model calibration.

- Uniform Distribution (Improper), only appropriate for prior components - `mp_uniform`
- Normal Distribution - `mp_normal`
- Log-Normal Distribution - `mp_log_normal`
- Logit-Normal Distribution - `mp_logit_normal`
- Poisson Distribution - `mp_poisson`
- Negative Binomial Distribution - `mp_neg_bin`

Usage

```
mp_uniform(trans_distr_param = list())

mp_normal(
  location = mp_distr_param_null("location"),
  sd,
  trans_distr_param = list(location = mp_identity, sd = mp_log)
)

mp_log_normal(
  location = mp_distr_param_null("location"),
  sd,
  trans_distr_param = list(location = mp_identity, sd = mp_identity)
)

mp_logit_normal(
  location = mp_distr_param_null("location"),
  sd,
  trans_distr_param = list(location = mp_identity, sd = mp_identity)
```

```
)

mp_poisson(
  location = mp_distr_param_null("location"),
  trans_distr_param = list(location = mp_identity)
)

mp_neg_bin(
  location = mp_distr_param_null("location"),
  disp,
  trans_distr_param = list(location = mp_identity, disp = mp_log)
)
```

Arguments

`trans_distr_param`	Named list of transformations for each distributional parameter. See `transform_distr_param` for a list of available transformations.
`location`	Location parameter. Specifying the `location` parameter is only necessary when the distribution is used as a prior distribution. If it is used as a likelihood component the location parameter will be taken as the simulated variable being fitted to data, and so this `location` parameter should be left to the default.
`sd`	Standard deviation parameter.
`disp`	Dispersion parameter.

Details

All distributional parameter arguments can be specified either as a numeric value, a character string giving the parameter name, or a distributional parameter object (See `fit_distr_params`).

`empty_matrix` *Empty Matrix*

Description

Empty matrices are useful when defining matrices that do not need to be initialized because they will get computed before they are required by other expressions. They can also provide a useful placeholder for matrices that should only have a value after a certain phase in the simulation.

Usage

```
empty_matrix
```

Format

A `numeric` `matrix` with zero rows and zero columns.

Examples

```
spec = mp_tmb_model_spec(during = list(x ~ time_step(0)))
identical(spec$empty_matrices()$x, empty_matrix) ## TRUE
```

`engine_eval` *Engine Evaluation*

Description

Evaluate an expression in the TMB-based C++ simulation and objective function engine. This function is useful for trying out the `engine_functions` that can be used to define **macpan2** models.

Usage

```
engine_eval(
  .expr,
  ...,
  .matrix_to_return,
  .tmb_cpp = getOption("macpan2_dll"),
  .structure_labels = NullLabels()
)
```

Arguments

`.expr`	Expression as a one-sided formula, the right-hand-side of which is treated as the expression to be evaluated.
`...`	Named objects that can be coerced into numeric matrices.
`.matrix_to_return`	Optional name of one of the matrices given in `...` to be returned. If this argument is missing, the matrix that will be returned is the matrix returned by the expression on the right-hand-side of the formula.
`.tmb_cpp`	Name of a `C++` program defining the engine. Typically you just want to use the default, which is `macpan2`, unless you are extending the engine yourself.
`.structure_labels`	Deprecated.

Value

Matrix being produced on the right-hand-side or matrix given in `.matrix_to_return` if it is provided.

Examples

```
engine_eval(~ exp(y), y = pi) # ~ 23.14069

# It is not currently possible to assign values to a subset of
# a matrix in a natural way (e.g. you cannot do things like x[1] = exp(y)),
# but you can achieve this functionality using the assign function.
engine_eval(~ assign(x, 1, 0, exp(y))
  , x = rep(0, 2)
  , y = pi
  , .matrix_to_return = "x"
)
```

`engine_functions` *Engine Functions*

Description

Functions currently supported by the C++ TMB engine for constructing expressions for defining model simulations.

Details

The quickest way to experiment with these functions is to use the `engine_eval` function, as in the following example that calculates a force of infection.

```
engine_eval(~ beta * I / N
  , beta = 0.25
  , I = 1e3
  , N = 1e7
)
```

To produce a simulation using these engine functions, one may use `simple_sims`.

```
simple_sims(
  iteration_exprs = list(x ~ x - 0.9 * x),
  time_steps = 5,
  mats = list(x = 1)
)
```

Elementwise Binary Operators:

Elementwise binary operators take two matrix-valued arguments and apply a binary operator (e.g. `+`, `*`) to each set of corresponding elements, and return the corresponding matrix-valued output containing the resulting elements. What does 'corresponding' mean? If the two matrix-valued arguments have the same shape (same number of rows and columns), then two elements correspond if they occur in the same row and column position in the two matrices. If the two matrices are not of the same shape but there is one row and/or one column in either matrix, then the singleton rows and columns are recycled sufficiently many times so that they match the shape of the other matrix. If after recycling singleton rows and columns the matrices are still of different shape, then an error is thrown and the matrices are said to be incompatible.

Functions:

- `x + y`
- `x - y`
- `x * y`
- `x / y`
- `x ^ y`

Arguments:

- `x` – Any matrix with dimensions compatible with `y`.
- `y` – Any matrix with dimensions compatible with `x`.

Return:

- A matrix with the binary operator applied elementwise after any necessary recycling of rows and/or columns.

Examples:

```
engine_eval(~ 1 + 2)
engine_eval(~ y * z, y = 1:3, z = matrix(1:6, 3, 2))
engine_eval(~ 1 / (1 - y), y = 1/4)
```

Unary Elementwise Math:

Functions:

- `log(x)` – Natural logarithm
- `exp(x)` – Exponential function
- `cos(x)` – Cosine function
- `proportions(x, limit, eps)` – matrix of `x / sum(x)` or `rep(limit, length(x))` if `sum(x) < eps`

Arguments:

- `x` – Any matrix
- `limit` – numeric value to return elementwise from `proportions` if `sum(x) < eps`
- `eps` – numeric tolerance for `sum(x)`

Return:

- A matrix with the same dimensions as `x`, with the unary function applied elementwise.

Examples:

```
engine_eval(~ log(y), y = c(2, 0.5))
```

Integer Sequences:

Functions:

- `from:to` – Inclusive and ordered sequence of integers between two bounds.
- `seq(from, length, by)` – Ordered sequence of integers with equal spacing between adjacent values.

Arguments:

- `from` – Scalar integer giving the first integer in the sequence.
- `to` – Scalar integer giving the last integer in the sequence.
- `length` – Number of integers in the sequence.

- by – Scalar giving the difference between adjacent values in the sequence.

Return:

- Column vector with a sequence of integers.

Details:
The colon operator works much like the base R version `:`. It takes two scalar-valued integers and returns a column vector with all integers between the two inputs.
The `seq` function is a little different from the base R default, `seq`, in that it allows the user precise control over the length of the output through the `length` argument. The base R function gives the user this option, but not as the default.

Examples:

```
engine_eval(~ 1:10)
engine_eval(~ seq(1, 10, 2))
```

Replicate Elements

Functions:

- `rep(x, times)` – Replicate a column vector a number of times, by repeatedly stacking it on top of itself.
- `rep_each` – Not yet developed.
- `rep_length` – Not yet developed.

Arguments:

- `x` – A scalar-valued variable to repeat.
- `times` – A scalar-valued integer variable giving the number of times to repeat `x`.

Return:

- Column vector with `times` copies of `x` stacked on top of each other.

Examples:

```
engine_eval(~ rep(1, 10))
```

Matrix Multiplication:

Functions:

- `x %*% y` – Standard matrix multiplication.
- `x %x% y` – Kronecker product

Arguments:

- `x` – A matrix. For the standard product, `x` must have as many columns as `y` has rows.
- `y` – A matrix. For standard product, `y` must have as many rows as `x` has columns.

Return:

- The matrix product of `x` and `y`.

Examples:

```
engine_eval(~ (1:10) %*% t(1:10))
engine_eval(~ (1:10) %x% t(1:10))
```

Parenthesis:

The order of operations can be enforced in the usual way with round parentheses, `(`.

Reshaping and Combining Matrices:

Functions:

- `c(...)` – Stack columns of arguments into a single column vector.
- `cbind(...)` – Create a matrix containing all of the columns of a group of matrices with the same number of rows.
- `rbind(...)` – Create a matrix containing all of the rows of a group of matrices with the same number of columns.
- `matrix(x, rows, cols)` – Reshape a matrix to have `rows` rows and `cols` columns. The input `x` must have `rows * cols` elements.
- `t(x)` – Standard matrix transpose.

Arguments:

- `...` – Any number of dimensionally consistent matrices. The definition of dimensionally consistent depends on the function.
- `x` – Can be any matrix for `t`, but for `matrix` it must have `rows * cols` elements.
- `rows` – Scalar integer giving the number of rows in the output.
- `cols` – Scalar integer giving the number of columns in the output.

Return:

- A combined or reshaped matrix.

Details:
Any number of column vectors can be combined into a bigger column vector.
Column and row vectors of the same length can be combined using the `cbind` and `rbind` functions respectively
The `matrix` function can be used to redefine the numbers of rows and columns to use for arranging the values of a matrix. It works similarly to the base R `matrix` function in that it takes the same arguments. On the other hand, this function differs substantially from the base R version in that it must be filled by column and there is no `byrow` option.
Matrices can be transposed with the usual function, `t`.

Examples:

```
engine_eval(~ c(a, b, c), a = 1, b = 10:13, c = matrix(20:25, 3, 2))
engine_eval(~ cbind(a, 10 + a), a = 0:3)
engine_eval(~ rbind(a, 10 + a), a = t(0:3))
engine_eval(~ matrix(1:12, 4, 3))
engine_eval(~ t(1:3))
```

Matrix Diagonals:

Functions:

- `to_diag(x)` – Create a diagonal matrix by setting the diagonal to a column vector, `x`.
- `from_diag(x)` – Extract the diagonal from a matrix, `x`, and return the diagonal as a column vector.

Arguments:

- `x` – Any matrix (for `from_diag`) or a column vector (for `to_diag`). It is common to assume that `x` is square for `from_diag` but this is not required.

Return:

- `to_diag(x)` – Diagonal matrix with `x` on the diagonal.
- `from_diag(x)` – Column vector containing the diagonal of `x`. A value is considered to be on the diagonal if it has a row index equal to the column index.

Details:
The `to_diag` function can be used to produce a diagonal matrix by setting a column vector equal to the desired diagonal. The `from_diag` does (almost) the opposite, which is to get a column vector containing the diagonal of an existing matrix.

Examples:

```
engine_eval(~from_diag(matrix(1:9, 3, 3)))
engine_eval(~to_diag(from_diag(matrix(1:9, 3, 3))))
engine_eval(~from_diag(to_diag(from_diag(matrix(1:9, 3, 3)))))
```

Summarizing Matrix Values:

Functions:

- `sum(...)` – Sum all of the elements of all of the matrices passed to `...`.
- `col_sums(x)` – Row vector containing the sums of each column.
- `row_sums(x)` – Column vector containing the sums of each row.
- `group_sums(x, f, n)` – Column vector containing the sums of groups of elements in `x`. The groups are determined by the integers in `f` and the order of the sums in the output is determined by these integers.

Arguments:

- `...` – Any number of matrices of any shape.
- `x` – A matrix of any dimensions, except for `group_sums` that expects `x` to be a column vector.
- `f` – A column vector the same length as `x` containing integers between `0` and `m-1`, given `m` unique groups. Elements of `f` refer to the indices of `x` that will be grouped and summed.
- `n` – A column vector of length `m`. If `f` does not contain group `k` in `[0, m-1]`, `group_sums` skips this group and the output at index `k+1` is `n[k+1]`.

Return:

- A matrix containing sums of subsets of the inputs.

Details:
The `row_sums` and `col_sums` are similar to the base R `rowSums` and `colSums` functions, but with slightly different behaviour. In particular, the `row_sums` function returns a column vector and the `col_sums` function returns a row vector. If a specific shape is required then the transpose `t` function must be explicitly used.

Examples:

```
x = 1
y = 1:3
A = matrix(1:12, 4, 3)
engine_eval(~ sum(y), y = y)
engine_eval(~ sum(x, y, A), x = x, y = y, A = A)
engine_eval(~ col_sums(A), A = A)
engine_eval(~ row_sums(A), A = A)
engine_eval(~ group_sums(x, f, n), x = 1:10, f = rep(0:3, 1:4), n = c(1:4))
```

Extracting Matrix Elements:

Functions:

- `x[i,j]` – Matrix containing a subset of the rows and columns of `x`.
- `block(x,i,j,n,m)` – Matrix containing a contiguous subset of rows and columns of `x` https://eigen.tuxfamily.org/dox/group__TutorialBlockOperations.html

Arguments:

- `x` – Any matrix.
- `i` – An integer column vector (for `[]`) or integer scalar (for `block`) containing the indices of the rows to extract (for `[]`) or the index of the first row to extract (for `block`).
- `j` – An integer column vector (for `[]`) or integer scalar (for `block`) containing the indices of the columns to extract (for `[]`) or the index of the first column to extract (for `block`).
- `n` – Number of rows in the block to return.
- `m` – Number of columns in the block to return.

Return:

- A matrix containing a subset of the rows and columns in `x`.

Details:
Note that zero-based indexing is used so the first row/column gets index, `0`, etc.

Examples:

```
engine_eval(~ A[c(3, 1, 2), 2], A = matrix(1:12, 4, 3))
engine_eval(~ block(x,i,j,n,m), x = matrix(1:12, 4, 3), i=1, j=1, n=2, m=2)
```

Accessing Past Values in the Simulation History:

For matrices with their simulation history saved, it is possible to bind the rows or columns of past versions of such matrices into a single matrix.

Functions:

- `rbind_lag(x, lag, t_min)` – Bind the rows of versions of `x` that were recorded at the end of all simulation iterations corresponding to time lags given by integers in `lag`.
- `rbind_time(x, t, t_min)` – Bind the rows of versions of `x` that were recorded at the end of all simulation iterations corresponding to integers in `t`.
- `cbind_lag(x, lag, t_min)` – Bind the columns of versions of `x` that were recorded at the end of all simulation iterations corresponding to time lags given by integers in `lag`. (TODO – cbind_lag is not developed yet)
- `cbind_time(x, t, t_min)` – Bind the columns of versions of `x` that were recorded at the end of all simulation iterations corresponding to integers in `t`. (TODO – cbind_lag is not developed yet)

Arguments:

- `x` – Any matrix with saved history such that the number of columns (for `rbind_*`) or rows (for `cbind_*`) does not change throughout the simulation.
- `lag` – Integer vector giving numbers of time steps before the current step to obtain past values of `x`.
- `t` – Integer vector giving time steps at which to obtain past values of `x`.
- `t_min` – Integer giving the minimum time step that is allowed to be accessed. All time-steps in `t` or implied by `lag` that are before `t_min` are ignored.

Return:

- A matrix containing values of `x` from past times.

Time Indexing:

Get or update the index of the current or lagged time step or the index of the current time group. A time group is a contiguous set of time steps defined by two change points.

Functions:

- `time_step(lag)`: Get the time-step associated with a particular lag from the current time-step. If the lagged time-step is less than zero, the function returns zero.
- `time_group(index, change_points)`: Update the `index` associated with the current time group. The current group is defined by the minimum of all elements of `change_points` that are greater than the current time step. The time group `index` is the index associated with this element. Please see the examples below, they are easier to understand than this explanation.
- `time_var(x, change_points)`: An improvement to `time_group`. Returns values in `x` at time steps in `change_points`, return value remains constant between `change_points`.

Arguments:

- `x`: Column vector representing a time series. `time_var` will return the value of `x` corresponding to element in `change_points` that contains the current time.
- `lag`: Number of time-steps to look back for the time-step to return.
- `change_points`: Increasing column vector of time steps giving the lower bound of each time group.

Return:

A 1-by-1 matrix with the time-step `lag` steps ago, or with zero if `t+1 < lag`

Examples:

```
simple_sims(
  iteration_exprs = list(x ~ time_step(0)),
  time_steps = 10,
  mats = list(x = empty_matrix)
)
sims = simple_sims(
  iteration_exprs = list(
    j ~ time_group(j, change_points),
    time_varying_parameter ~ time_variation_schedule[j]
  ),
  mats = list(
    j = 0,
    change_points = c(0, 4, 7),
    time_variation_schedule = c(42, pi, sqrt(2)),
    time_varying_parameter = empty_matrix
  ),
  time_steps = 10,
)
set.seed(1L)
change_points = c(0,2,5)
x_val = rnorm(length(change_points))
```

```
simple_sims(
    iteration_exprs = list(x ~ time_var(x_val,change_points))
  , int_vecs = list(change_points = change_points)
  , mats = list(x = empty_matrix, x_val=x_val)
  , time_steps = 10
)
```

Convolution:

One may take the convolution of each element in a matrix, x, over simulation time using a kernel, k. There are two arguments of this function.

Functions:

- `convolution(x, k)`

Arguments:

- `x` – The matrix containing elements to be convolved.
- `k` – A column vector giving the convolution kernel.

Return:

A matrix the same size as `x` but with the convolutions, y_{ij}, of each element, x_{ij}, given by the following.

$$y_{ij} = \sum_{\tau=0} x_{ij}(t-\tau)k(\tau)$$

unless $t < \tau$, in which case,

$$y_{ij} =$$

where y_{ij} is the convolution, $x_{ij}(t)$ is the value of x_{ij} at time step, t, $k(\tau)$ is the value of the kernel at lag, τ, and λ is the length of the kernel.

Details:

If any empty matrices are encountered when looking back in time, they are treated as matrices with all zeros. Similarly, any matrices encounte of `x`

Clamp:

Smoothly clamp the elements of a matrix so that they do not get closer to 0 than a tolerance, `eps`, with a default of 1e-12. The output of the `clamp` function is as follows.

Functions:

- `clamp(x, eps)`

Arguments:

- `x` : A matrix with elements that should remain positive.
- `eps` : A small positive number giving the theoretical minimum of the elements in the returned matrix.

Probability Densities:

All probability densities have the same first two arguments.

- `observed`
- `simulated`

The simulated argument gives a matrix of means for the observed values at which the densities are being evaluated. Additional arguments are other distributional parameters such as the standard deviation or dispersion parameter. All densities are given as log-densities, so if you would like the density itself you must pass the result through the exp function.

If the simulated matrix or the additional parameter matrices have either a single row or single column, these singleton rows and columns are repeated to match the number of rows and columns in the observed matrix. This feature allows one to do things like specify a single common mean for several values.

Functions:

- dpois(observed, simulated) – Log of the Poisson density based on this dpois TMB function.
- dnbinom(observed, simulated, over_dispersion) – Log of the negative binomial density based on this dnbinom TMB function. To get the variance that this function requires we use this expression, simulated + simulated^2/over_dispersion, following p.165 in this book
- dnorm(observed, simulated, standard_deviation) – Log of the normal density based on this dnorm TMB function.

Arguments:

- observed – Matrix of observed values at which the density is being evaluated.
- simulated – Matrix of distributional means, with singleton rows and columns recycled to match the numbers of rows and columns in observed.
- over_dispersion – Over-dispersion parameter given by (simulated/standard_deviation)^2 - simulated).
- standard_deviation – Standard deviation parameter.

Pseudo-Random Number Generators:

All random number generator functions have mean as the first argument. Subsequent arguments give additional distributional parameters. Singleton rows and columns in the matrices passed to the additional distributional parameters are recycled so that all arguments have the same number of rows and columns. All functions return a matrix the same shape as mean but with pseudo-random numbers deviating from each mean in the mean matrix.

Functions:

- rpois(mean) – Pseudo-random Poisson distributed values.
- rnbinom(mean, over_dispersion) – Pseudo-random negative binomially distributed values.
- rnorm(mean, standard_deviation) – Pseudo-random normal values.

Arguments:

- mean – Matrix of means about which to simulate pseudo-random variation.
- over_dispersion – Matrix of over-dispersion parameters given by (simulated/standard_deviation)^2 - simulated).
- standard_deviation – Matrix of standard deviation parameters.

Assign:

Assign values to a subset of the elements in a matrix.

Functions:

- `assign(x, i, j, v)`

Arguments:

- `x` – Matrix with elements that are to be updated by the values in `v`.
- `i` – Column vector of row indices pointing to the elements of `x` to be updated. These indices are paired with those in `v`. If the length of `i` does not equal that of `v`, then it must have a single index that gets paired with every element of `v`. Indices are zero-based, `i=0` corresponds to the first row.
- `j` – Column vector of column indices pointing to the elements of `x` to be updated. These indices are paired with those in `v`. If the length of `j` does not equal that of `v`, then it must have a single index that gets paired with every element of `v`. Indices are zero-based, `j=0` corresponds to the first column.
- `v` – Column vector of values to replace elements of `x` at locations given by `i` and `j`.

Return:
The `assign` function is not called for its return value, which is an `empty_matrix`, but rather to modify `x` but replacing some of its components with those in `v`.

Examples:

```
x = matrix(1:12, 3, 4)
engine_eval(~ x + 1, x = x)
engine_eval(~ x + 1, x = x, .matrix_to_return = "x")
engine_eval(~ assign(x, 2, 1, 100), x = x, .matrix_to_return = "x")
engine_eval(~ assign(x
  , c(2, 1, 0)
  , 0
  , c(100, 1000, 10000)
), x = x, .matrix_to_return = "x")
```

Unpack:

Unpack elements of a matrix into smaller matrices.

Functions:

- `unpack(x, ...)`

Arguments:

- `x` – Matrix with elements to be distributed to the matrices passed through `...`.
- `...` – Matrices with elements to be replaced by the values of elements in `x` in column-major order. These matrices must be named matrices and not computed on the fly using expressions. Note that even subsetting (e.g. `unpack(x, y[0], y[3])`) counts as an expression. This use-case would require the `assign` function `assign(y, c(0, 3), 0, x)`.

Return:
The `unpack` function is not called for its return value, which is an `empty_matrix`, but rather to modify the matrices in `...` by replacing at least some of its components with those in `x`.

Examples:
Here we fill a matrix with integers from 1 to 12 and then unpack them one-at-a-time into two column vectors, `x` and `y`. By returning `y` we see the integers after the first three were used up by `x`.

```
engine_eval(~unpack(matrix(1:12, 3, 4), x, y)
  , x = rep(0, 3)
  , y = rep(1, 5)
  , .matrix_to_return = "y"
)
```

Print Matrix:

Print out the value of a matrix.

Functions:

- `print(x)`

Arguments:

- `x` – Name of a matrix in the model.

Return:

An `empty_matrix`.

Examples:

```
simple_sims(
     list(dummy ~ print(x), x ~ x / 2)
   , time_steps = 10
   , mats = list(x = 2)
)
```

`finalizer` *Finalizers*

Description

Finalizers

Usage

```
finalizer_char(x)

finalizer_index(x)
```

Arguments

`x`	Raw parsed expression.

Functions

- `finalizer_char()`: Finalize parsed expression so that the parse table is a little more human readable.
- `finalizer_index()`: Finalize parsed expression so that the parse table can be passed to the C++ engine.

find_all_paths *Find all Paths*

Description

Find all paths through a compartmental model.

Usage

```
find_all_paths(edges_df, start_node_guesses = character(0L))
```

Arguments

edges_df	A data frame with a from and a to column.
start_node_guesses	Optional guesses for nodes that start paths. This is useful for models that are not directed acyclic graphs (DAGs).

Value

List of character vectors of state variable names, each vector describing a path through the model.

fit_distr_params *Fitting Distributional Parameters*

Description

Distributional parameters can be added to the list of parameters that are fit during calibration. By default, distributional parameters in priors and likelihoods are not fit. Use mp_nofit to exclude distributional parameters from being fit and mp_fit to fit them.

Usage

```
mp_fit(x, trans = DistrParamTransDefault())

mp_nofit(x, trans = DistrParamTransDefault())
```

Arguments

x	numeric starting value of the distributional parameter to fit, or character name of an existing variable in the model with a default starting value to use.
trans	transformation to apply to the distributional parameter. By default, distributional parameters inherit a default transformation from the associated distribution. For example, the standard deviation parameter sd in the mp_normal distributions has a default log transformation specified using mp_log.

Value

A distributional parameter object.

Examples

```
# First we call the SIR model spec, and generate some data for calibration.
spec = mp_tmb_library("starter_models", "sir", package = "macpan2")
data = mp_simulator(spec, 50, "infection") |> mp_trajectory()

# Suppose we want to specify a Normal prior on the transmission parameter
# beta, and we are interested in estimating the prior standard deviation.
# Here we use `mp_fit` to estimate the standard deviation, `sd`, and we
# provide a numeric starting value for `sd` in the optimization.
cal = mp_tmb_calibrator(
    spec
  , data
  , traj = "infection"
  , par = list(beta = mp_normal(location = 0.35, sd = mp_fit(0.1)))
  , default = list(beta = 0.25)
)

# When viewing the calibration objective function we can see the additional
# prior density term added for beta. The standard deviation parameter has
# been automatically named 'distr_params_log_sd_beta'.
cal$simulator$tmb_model$obj_fn$obj_fn_expr

# Next we optimize and view the fitted parameters. We can see the
# distributional parameter in the coefficient table with a default value
# equal to the numeric value we provided to `mp_fit` above.
mp_optimize(cal)
mp_tmb_coef(cal)

# If instead we want control over the name of the new fitted distributional
# parameter, we can add a new variable to our model specification with the
# default value set to the desired optimization starting value.
updated_spec = spec |> mp_tmb_insert(default = list(sd_var = 0.1))

# In the calibrator, we use the name of this newly added variable, "sd_var",
# as input to `mp_fit`.
cal = mp_tmb_calibrator(
    updated_spec
  , data
  , traj = "infection"
  , par = list(beta = mp_normal(location = 0.35, sd = mp_fit("sd_var")))
  , default = list(beta = 0.25)
)

# We can see this distributional parameter get propogated to the objective
# function and the fitted parameter table.
cal$simulator$tmb_model$obj_fn$obj_fn_expr
mp_optimize(cal)
mp_tmb_coef(cal)
```

initial_valid_vars *Initial Valid Variables*

Description

Initial Valid Variables

Usage

```
initial_valid_vars(valid_var_names)
```

Arguments

valid_var_names	Character vector of variable names.

LedgerDefinition *Ledgers*

Description

A ledger is a table with rows that identify specific instances of a functional form used to define a mp_dynamic_model. Ledgers are most commonly created using the mp_join function as in the following example.

```
age = mp_index(Age = c("young", "old"))
state = mp_cartesian(
  mp_index(Epi = c("S", "I", "R")),
  age
)
mp_join(
  from = mp_subset(state, Epi = "S"),
  to = mp_subset(state, Epi = "I"),
  by = list(from.to = "Age")
)
#>     from      to
#>  S.young I.young
#>    S.old   I.old
```

`make_expr_parser` *Generate an Arithmetic Expression Parser*

Description

Generate an Arithmetic Expression Parser

Usage

```
make_expr_parser(parser_name = NULL, finalizer = force)
```

Arguments

`parser_name`	Name of the parsing function as a character string. No longer used, but still present for back-compatibility.
`finalizer`	Function used to post-process a parsed formula. The default is the identity finalizer, which returns the parsed formula itself. Other good choices are `finalizer_char`, which can be used to understand how the formula has been parsed, and `finalizer_index`, which can be passed to the C++ engine. The result of this function is another function that takes a single argument, `x`. This resulting function is recursive. The `x` argument should be a one-sided formula the first time this recursive function is called. In subsequent evaluations of the recursion, `x` will be a list with the following structure. When `x` is a formula, it must contain a named list of functions called `valid_funcs` and a named list of variables called `valid_vars`. **x** list of names and numeric objects that represent each leaf of the parse tree **n** integer vector the same length as `x` that give the number of arguments of the associated functions in `x` or `0` otherwise **i** index identifying the element of `x` corresponding to the first argument of the associated function or `0` if this is not a function **valid_funcs** named list of valid functions that was extracted from the environment of the formula being parsed **valid_vars** named list of default values of valid variables extracted from the environment of the formula being parsed **input_expr_as_string** the input formula stored as a string

Examples

```
parser = make_expr_parser(finalizer = finalizer_char)
foi = ~ beta * I / 100
valid_funcs = setNames(
  list(`*`, `/`),
  c("*", "/")
)
valid_vars = list(beta = 0.1, I = 30)
parser(foi)
```

mp_aggregate *Aggregate an Index*

Description

Create a one-column ledger (see LedgerDefinition) with rows identifying instances of an aggregation.

Usage

```
mp_aggregate(index, by = "Group", ledger_column = "group")
```

Arguments

index	An index to aggregate over.
by	A column set label to group by. By default a dummy and constant "Group" column is created.
ledger_column	Name of the column in the output ledger that describes the groups.

See Also

Other functions that return ledgers mp_join()

mp_cartesian *Cartesian Product of Index Tables*

Description

Produce a new index table by taking all possible pairwise combinations of the input tables. This is useful for producing product models that expand model components through stratification.

Usage

```
mp_cartesian(...)
```

Arguments

...	Index tables (see mp_index).

See Also

Other functions that return index tables mp_index(), mp_rename(), mp_subset(), mp_union()

Other functions that take products of index tables and return one index tables mp_linear(), mp_square(), mp_symmetric(), mp_triangle()

Examples

```
mp_cartesian(
  mp_index(Epi = c("S", "I")),
  mp_index(Age = c("young", "old"))
)

si = mp_index(Epi = c("S", "I"))
age = mp_index(Age = c("young", "old"))
loc = mp_index(City = c("hamilton", "toronto"))
vax = mp_index(Vax = c("unvax", "vax"))
(si
  |> mp_cartesian(age)
  |> mp_cartesian(loc)
  |> mp_cartesian(vax)
)

flow_rates = mp_index(Epi = c("infection", "recovery"))
mp_union(
  mp_cartesian(
    mp_subset(flow_rates, Epi = "infection"),
    age
  ),
  mp_subset(flow_rates, Epi = "recovery")
)
```

`mp_default` *Default Values*

Description

Default Values

Usage

```
mp_default(model)

mp_default_list(model)
```

Arguments

`model`	A model object from which to extract default values.

Value

A long-format data frame with default values for matrices required as input to model objects. The columns of this output are `matrix`, `row`, `col`, and `value`. Scalar matrices do not have any entries in the `row` or `col` columns.

Functions

- `mp_default_list()`: List of the default variables as matrices.

`mp_dynamic_model` *Dynamic Model*

Description

This is an 'old' model specification function that was tested out at a workshop. Currently it still drives the engine-agnostic-grammar vignette, but we plan to replace this function with `mp_tmb_model_spec` and other model specification functions.

Usage

```
mp_dynamic_model(
  expr_list = ExprList(),
  ledgers = list(),
  init_vecs = list(),
  unstruc_mats = list()
)
```

Arguments

`expr_list`	Expression list.
`ledgers`	Ledgers.
`init_vecs`	Initial structured vectors.
`unstruc_mats`	Initial unstructured matrices.

`mp_dynamic_simulator` *TMB Simulator from Dynamic Model*

Description

This is an 'old' function that was tested out at a workshop. Currently it still drives the engine-agnostic-grammar vignette, but we plan to replace this function with `mp_simulator`.

Usage

```
mp_dynamic_simulator(
  dynamic_model,
  time_steps = 0L,
  vectors = NULL,
  unstruc_mats = NULL,
  mats_to_save = NULL,
  mats_to_return = NULL,
  params = OptParamsList(0),
  random = OptParamsList(),
  obj_fn = ObjectiveFunction(~0),
  log_file = LogFile(),
  do_pred_sdreport = TRUE,
  tmb_cpp = "macpan2",
  initialize_ad_fun = TRUE,
  ...
)
```

Arguments

dynamic_model	Object product by mp_dynamic_model.
time_steps	Number of time steps to simulate.
vectors	Named list of named vectors as initial values for the simulations that are referenced in the expression list in the dynamic model.
unstruc_mats	= Named list of objects that can be coerced to numerical matrices that are used in the expression list of the dynamic model.
mats_to_save	TODO
mats_to_return	TODO
params	TODO
random	TODO
obj_fn	TODO
log_file	TODO
do_pred_sdreport	TODO
tmb_cpp	TODO
initialize_ad_fun	TODO
...	TODO

mp_effects_descr *Describe Statistical Effects*

Description

Additional information that can be joined to the output of the `tidy.TMB` or `tidy.stanfit` functions in the `broom.mixed` package.

Usage

```
mp_effects_descr(model)

mp_add_effects_descr(coef_table, model)
```

Arguments

`model`	A model in the TMB engine that can be used to compute tables of statistical effects.
`coef_table`	Coefficient table that was probably generated using `mp_tmb_coef` or `mp_tmbstan_coef`, but also perhaps generated directly using the `tidy.TMB` or the `tidy.stanfit` methods in the `broom.mixed` package.

Functions

- `mp_add_effects_descr()`: Convenience function for adding coefficient descriptions from a calibrated model to `coef_tables` generated by `mp_tmb_coef` or `mp_tmbstan_coef`.

mp_euler *State Updates*

Description

Use these functions to update a model spec so that the state variables are updated according to different numerical methods.

Usage

```
mp_euler(model)

mp_rk4(model)

mp_euler_multinomial(model)

mp_hazard(model)
```

Arguments

model	Object with quantities that have been explicitly marked as state variables.

Details

The default update method for model specifications produced using mp_tmb_model_spec is mp_euler. This update method yields a difference-equation model where the state is updated once per time-step using the absolute flow rate as the difference between steps.

These state update functions are used to modify a model specification to use a particular kind of state update. To see these modified models for a particular example one may use the mp_expand function (see examples).

Functions

- mp_rk4(): ODE solver using Runge-Kutta 4. Any formulas that appear before model flows in the during list will only be updated with RK4 if they do contain functions in getOption("macpan2_non_iterable_fu and if they do not make any state variable assignments (i.e., the left-hand-side does not contain state variables). Each formula that does not meet these conditions will be evaluated only once at each time-step before the other three RK4 iterations are taken. By default, the time_var function and functions that generate random numbers (e.g., rbinom) are not iterable. Functions that generate random numbers will only be called once with state update methods that do not repeat expressions more than once per time-step (e.g., mp_euler), and so repeating these functions with RK4 could make it difficult to compare methods. If you really do want to regenerate random numbers at each RK4 iteration, you can do so by setting the above option appropriately. The time_var function assumes that it will only be called once per time-step, and so it should never be removed from the list of non-iterable functions. Although in principle it could make sense to update state variables manually, it currently causes us to be confused. We therefore require that all state variables updates are set explicitly (e.g., with mp_per_capita_flow) if any are explicit.
- mp_euler_multinomial(): Update state with process error given by the Euler-multinomial distribution.
- mp_hazard(): Update state with hazard steps, which is equivalent to taking the step given by the expected value of the Euler-multinomial distribution.

Examples

```
sir = mp_tmb_library("starter_models", "sir", package = "macpan2")
sir
sir |> mp_euler()             |> mp_expand()
sir |> mp_rk4()               |> mp_expand()
sir |> mp_euler_multinomial() |> mp_expand()
```

mp_expand *Expand Model*

Description

Expand a structured model so that it is represented in an unstructured format requiring a more verbose description. Currently, this is only applicable for `mp_tmb_model_spec` objects that have explicit flows (e.g. `mp_per_capita_flow`). For such models, `mp_expand` produces a model with expression lists composed entirely of plain R formulas.

Usage

```
mp_expand(model)
```

Arguments

`model`	A model object.

Examples

```
sir = mp_tmb_library("starter_models", "sir", package = "macpan2")
print(sir)
print(mp_expand(sir))
```

mp_extract *Extract Index*

Description

Extract the index for a particular dimension in a ledger from a ledger or from an object containing one or more ledgers.

Usage

```
mp_extract(x, dimension_name)
```

Arguments

`x`	Object
`dimension_name`	Name of a dimension used in a ledger.

mp_factors *Factor an Index*

Description

Factor an Index

Usage

```
mp_factors(index, unpack = c("no", "maybe", "yes"))
```

Arguments

index	An index to be factored.
unpack	Place factors in the global environment?

mp_final *Final Values*

Description

Return the values of variables after the simulation loop has finished and the `final` set of expressions have been evaluated.

Usage

```
mp_final(model)

mp_final_list(model)
```

Arguments

model	Object that can be used to simulate.

Functions

- `mp_final_list()`: Final values formatted as a list of matrices.

mp_flow_frame *Flow Frame (experimental)*

Description

Get a data frame representing the flows in a model specification.

Usage

```
mp_flow_frame(spec, topological_sort = TRUE, loops = "^$")
```

Arguments

spec	A mp_tmb_model_spec.
topological_sort	Should the states be topologically sorted to respect the main direction of flow?
loops	Pattern for matching the names of flows that make the flow model not a DAG, which is a critical assumption when topologically sorting the order of states and flows in the output. This is only relevant if topological_sort is used.

Value

A data frame that gives information provided in calls to mp_per_capita_flow and mp_per_capita_inflow.

mp_group *Group an Index*

Description

Create a new index with fewer columns to create names for an aggregated vector that is labelled by the input index.

Usage

```
mp_group(index, by)
```

Arguments

index	Index to group rows.
by	Column set label to group by.

mp_index *Model Quantity Index Table*

Description

Make an index table to enumerate model quantity labels by category. These objects generalize and wrap data.frames, where each column is a label category and each row is an index. Indices must contain only letters, numbers, and underscores. Blank empty string entries are allowed, but missing values (NAs) are not.

Usage

```
mp_index(..., labelling_column_names)

## S3 method for class 'Index'
print(x, ...)

## S3 method for class 'Index'
names(x)

## S3 method for class 'Index'
labelling_column_names(x)

## S3 method for class 'Index'
labels(object, ...)
```

Arguments

...	Character vectors to combine to produce an index. Alternatively, any number of data frames of character-valued columns. If data frames are supplied, their rows will be bound and the result converted to an index if possible.
labelling_column_names	A character vector of the names of the index that will be used to label the model components (i.e. rows) being described. The labelling_column_names cannot have duplicates and must contain at least one name. The index given by the labelling_column_names must uniquely identify each row. The default NULL gives the set of columns, in order starting with the first column, that are required to uniquely identify each row.
x	An index.
object	An index.

Details

For example, the following index table describes the state variables of the model:

```
sir = mp_index(Epi = c("S", "I", "R"))
print(sir)
#>  Epi
#>    S
#>    I
#>    R
```

Here, the column `Epi` denotes that the category of these labels is epidemiological. There is nothing special about this specific choice of category name; we could have also used another name like `Compartment`.

However, in more complicated models, it is good to think carefully about choosing descriptive category names. For example, in an age-structured SIR model, we could add an `Age` column to generate an index table as follows:

```
sir_age = mp_index(
 Epi = rep(c("S", "I", "R"), 2),
 Age = rep(c("young", "old"), each = 3)
)
print(sir_age)
#>  Epi   Age
#>    S young
#>    I young
#>    R young
#>    S   old
#>    I   old
#>    R   old
```

Here, having the first column in the index table labeled `Compartment` would be somewhat misleading, as the compartments aren't actually just "S", "I", and "R", they are each of the epidemiological states stratified by the age groups "young" and "old".

This index table could also be generated by first specifying individual index tables for the `Epi` and `Age` columns, and then using a `macpan2` product function that combines the tables into a single index table:

```
sir = mp_index(Epi = c("S", "I", "R"))
age = mp_index(Age = c("young", "old"))
prod = mp_cartesian(sir, age)
prod
#>  Epi   Age
#>    S young
#>    I young
#>    R young
#>    S   old
#>    I   old
#>    R   old
```

The `mp_cartesian()` function will produce a table with entries that are all possible combinations of the individual index tables. The "See Also" section of the `mp_cartesian()` help page catalogues all available product functions.

We can produce the full labels of model quantities, which are simply dot-concatenated indices, one for each entry in the index table, using the `labels()` function:

```
#> [1] "S.young" "I.young" "R.young" "S.old"   "I.old"   "R.old"
```

Dots are not allowed in indices so that the labels can be inverted to reproduce the original index table (provided that the column names can be retrieved).

It is recommended to use UpperCamelCase for the columns of index tables and single uppercase characters ("S", "I"), all lowercase character strings ("gamma"), and/or snake_case strings ("aging_rate") for indices. This convention helps when reading code that contains references to both column names and indices.

Functions

- `print(Index)`: Print an index.
- `names(Index)`: Get the names of the columns of an index.
- `labelling_column_names(Index)`: Retrieve the `labelling_column_names` of an index. These are the names of the columns that are used to label the model components.
- `labels(Index)`: Convert an index into a character vector giving labels associated with each model component (i.e. row) being described.

See Also

`mp_structured_vector()`

`mp_set_numbers()`

Other functions that return index tables `mp_cartesian()`, `mp_rename()`, `mp_subset()`, `mp_union()`

Examples

```
state = mp_index(
  Epi = c("S", "I", "S", "I"),
  Age = c("young", "young", "old", "old")
)
print(state)
labels(state)
mp_cartesian(state, mp_index(City = c("hamilton", "toronto")))
```

`mp_initial` *Initial Values*

Description

Return a data frame containing the values of variables at the end of the `before` phase, right before the simulation loop begins (i.e. right before the `during` phase).

Usage

```
mp_initial(model)

mp_initial_list(model)
```

Arguments

model	A model specification object or model simulator object from which to extract initial values.

Value

A long-format data frame with initial values for matrices. The columns of this output are matrix, time, row, col, and value. Scalar matrices do not have any entries in the row or col columns. The before phase corresponds to a time value of 0.

Functions

- mp_initial_list(): List of the initial variables as matrices.

---

mp_join *Join Indexes*

Description

Join two or more index tables (see mp_index) to produce a ledger (see LedgerDefinition).

Usage

```
mp_join(..., by = empty_named_list())
```

Arguments

...	Named arguments giving indexes created by mp_index or another function that manipulates indexes. Each argument will become a position vector used to subset or expand numeric vectors in archetype formulas.
by	What columns to use to join the indexes. See below on how to specify this argument.

Details

When two index tables are passed to ..., mp_join behaves very much like an ordinary inner join. When more than two tables are passed to ..., mp_join iteratively joins pairs of tables to produce a final ledger. For example, if index tables A B, and C are passed to mp_join, an inner join of A and B is performed and the result is joined with C. In each of these successive internal joins. The properties of inner joins ensures that the order of tables does not affect the set of rows in the final table (SW states without proof!).

When two index tables are passed to ..., the by argument is just a character vector of column names on which to join (as in standard R functions for joining data frames), or the dot-concatenation of these column names. For example,

```
state = mp_index(
  Epi = c("S", "I", "S", "I"),
  Age = c("young", "young", "old", "old")
)
mp_join(
  from = mp_subset(state, Epi = "S"),
  to = mp_subset(state, Epi = "I"),
  by = "Age"
)
#>     from      to
#>  S.young I.young
#>    S.old   I.old
```

If there are more than two tables then the by argument must be a named list of character vectors, each describing how to join the columns of a pair of tables in .... The names of this list are dot-concatenations of the names of pairs of tables in .... For example,

```
rates = mp_index(
  Epi = c("lambda", "lambda"),
  Age = c("young", "old")
)
mp_join(
  from = mp_subset(state, Epi = "S"),
  to = mp_subset(state, Epi = "I"),
  rate = mp_subset(rates, Epi = "lambda"),
  by = list(
    from.to = "Age",
    from.rate = "Age"
  )
)
#>     from      to         rate
#>  S.young I.young lambda.young
#>    S.old   I.old   lambda.old
```

If the by columns have different names in two tables, then you can specify these using formula notation where the left-hand-side is a dot-concatenation of columns in the first table and the right-hand-side is a dot-concatenation of the columns in the second table. For example,

```
contact = mp_index(
  AgeSusceptible = c("young", "young", "old", "old"),
  AgeInfectious = c("young", "old", "young", "old")
)
mp_join(
  sus = mp_subset(state, Epi = "S"),
```

```
  inf = mp_subset(state, Epi = "I"),
  con = contact,
  by = list(
    sus.con = "Age" ~ "AgeSusceptible",
    inf.con = "Age" ~ "AgeInfectious"
  )
)
#>      sus     inf         con
#>  S.young I.young young.young
#>    S.old I.young   old.young
#>  S.young   I.old   young.old
#>    S.old   I.old     old.old
```

See Also

Other functions that return ledgers `mp_aggregate()`

`mp_labels` *Index Labels*

Description

Return a character vector of labels for each row of an index (or a ledger?? FIXME: what does this mean for ledgers??).

Usage

```
mp_labels(x, labelling_column_names)
```

Arguments

`x`	Object
`labelling_column_names`	What index columns should be used for generating the labels. If missing then defaults will be used. (FIXME: clarify how the defaults are used.)

`mp_layout_grid` *Flow Diagram Grid Layout*

Description

Create a grid on which to layout the flow diagram of a model specification.

Usage

```
mp_layout_grid(
  spec,
  east = "",
  south = "^$",
  north = "^$",
  west = "^$",
  loops = north,
  x_gap = 0.3,
  y_gap = 0.3,
  north_south_sep = 0,
  east_west_sep = 0,
  trim_states = character()
)
```

Arguments

spec	A model specification made with mp_tmb_model_spec or related function.
east	Regular expression for matching the names of flows that will be connected eastward in the layout.
south	Regular expression for matching the names of flows that will be connected southward in the layout.
north	Regular expression for matching the names of flows that will be connected northward in the layout.
west	Regular expression for matching the names of flows that will be connected westward in the layout.
loops	Regular expression for matching the names of flows that cause loops in the flow model, and so should be ignored when building the layout.
x_gap	Size of the gap to the left and right of the 1-by-1 space provided for a node.
y_gap	Size of the gap above and below the 1-by-1 space provided for a node.
north_south_sep	Horizontal separation between north and south flow arrows.
east_west_sep	Vertical separation between east and west flow arrows.
trim_states	List of states to remove from the diagram

---

mp_layout_paths *Flow Diagram Grid Layout*

Description

Layout the flow diagram of a model specification so that each row is one of the paths through the model (ignoring loops).

Usage

```
mp_layout_paths(
  spec,
  sort_paths = TRUE,
  combine_columns = TRUE,
  deduplicate_edges = TRUE,
  loops = "^$",
  ignore = "^$",
  x_gap = 0.3,
  y_gap = 0.3,
  north_south_sep = 0,
  east_west_sep = 0,
  trim_states = character()
)
```

Arguments

spec	A model specification made with mp_tmb_model_spec or related function.
sort_paths	Should the paths/rows be sorted to minimize the number times an edge must go through a node that it is not connected with?
combine_columns	Should each state/node get its own column in the layout (FALSE) or should the algorithm try to place branching states in the same column (TRUE, default).
deduplicate_edges	Should each row have all of the edges in the path or should duplicate edges be removed?
loops	Regular expression for matching the names of flows that cause loops in the flow model, and so should be ignored when building the layout.
ignore	Regular expression for matching the names of flows that should be removed from the layout analysis entirely. These will be isolated in a data frame for custom drawing of 'difficult' edges.
x_gap	Size of the gap to the left and right of the 1-by-1 space provided for a node.
y_gap	Size of the gap above and below the 1-by-1 space provided for a node.
north_south_sep	Horizontal separation between north and south flow arrows.
east_west_sep	Vertical separation between east and west flow arrows.
trim_states	List of states to remove from the diagram

---

mp_ledgers *Bundle up Ledgers*

Description

Bundle up several ledgers (see LedgerDefinition) to pass to mp_dynamic_model.

Usage

```
mp_ledgers(...)
```

Arguments

...	Ledgers to bundle up.

---

mp_linear *Linear Chain Product*

Description

TODO: what does this mean?

Usage

```
mp_linear(x, y_labelling_column_names)
```

Arguments

x	An index.
y_labelling_column_names	TODO

See Also

Other functions that take products of index tables and return one index tables `mp_cartesian()`, `mp_square()`, `mp_symmetric()`, `mp_triangle()`

---

mp_lookup *Lookup*

Description

Lookup a subset or factor index associated with a symbol, and return the index associated with that symbol.

Usage

```
mp_lookup(index, symbol)
```

Arguments

index	Index table (see `mp_index`).
symbol	Character string that could possibly be associated with a subset or factor of `index`.

mp_model_starter *Model Starter*

Description

Create a directory with a template model definition.

Usage

```
mp_model_starter(starter_name, dir)
```

Arguments

starter_name	Currently can only be sir.
dir	String giving the path to a directory for copying the template model definition.

mp_optimize *Optimize*

Description

Optimize

Usage

```
mp_optimize(model, optimizer, ...)

## S3 method for class 'TMBCalibrator'
mp_optimize(model, optimizer = c("nlminb", "optim"), ...)
```

Arguments

model	A model object capable of being optimized. See below for model types that are supported.
optimizer	Name of an implemented optimizer. See below for options for each type of model.
...	Arguments to pass to the optimizer.

Value

The output of the optimizer. The model object is modified and saves the history of optimization outputs. These outputs can be obtained using mp_optimizer_output.

Methods (by class)

- mp_optimize(TMBCalibrator): Optimize a TMB calibrator.

mp_optimizer_output *Optimizer Output*

Description

Get the output from an optimizer used in model calibration.

Usage

```
mp_optimizer_output(model, what = c("latest", "all"))
```

Arguments

model	An object that has been optimized.
what	A string indicating whether to return the results of the "latest" optimization attempt or a list with "all" of them.

Details

When objects created by mp_tmb_calibrator are successfully passed to mp_optimize, they build up an optimization history. This history is recorded as a list of the output produced by the underlying optimizer (e.g. nlminb). This mp_optimizer_output function returns the latest output by default or the entire history list.

mp_par *Fit Parameters*

Description

Define the prior distributions for parameters and random effects to be passed to par argument of the mp_tmb_calibrator function.

Usage

```
mp_par(param, random)
```

Arguments

param	Named list of distributional specifications for the fixed effects.
random	Named list of distributional specifications for the random effects.

mp_per_capita_flow *Flow*

Description

Specify different kinds of flows between compartments.

Usage

```
mp_per_capita_flow(from, to, rate, abs_rate = NULL)

mp_per_capita_inflow(from, to, rate, abs_rate = NULL)

mp_per_capita_outflow(from, rate, abs_rate = NULL)

mp_absolute_flow(from, to, rate, rate_name = NULL)
```

Arguments

from	String giving the name of the compartment from which the flow originates.
to	String giving the name of the compartment to which the flow is going.
rate	String giving the expression for the per-capita or absolute flow rate. Alternatively for per-capita flows, and for back compatibility, a two-sided formula with the left-hand-side giving the name of the absolute flow rate per unit time-stepand the right-hand-side giving an expression for the per-capita rate of flow from from to to.
abs_rate	String giving the name for the absolute flow rate, which will be computed as from * rate. If a formula is passed to rate (not recommended), then this abs_rate argument will be ignored.
rate_name	String giving the name for the absolute flow rate.

Details

The examples below can be mixed and matched in mp_tmb_model_spec() to produce compartmental models. Note that the symbols used below must be used in an appropriate context (e.g., if N is used for total population size, then there must be an expression like N ~ S + I + R somewhere in the model or for models with constant population size there must be a default variable, N, with a numerical value).

Functions

- mp_per_capita_inflow(): Only flow into the to compartment, and do not flow out of the from compartment. The from compartment can even be a function of a set of compartments, because it will not be updated. A common construction is mp_per_capita_inflow("N", "S", "birth_rate", "birth") for adding a birth process, which involves the total population size, N, rather than a single compartment.

- `mp_per_capita_outflow()`: Only flow out of the `from` compartment, without going anywhere. This is useful for removing individuals from the system (e.g., death). To keep track of the total number of dead individuals one can use `mp_per_capita_flow` and set `to` to be a compartment for these individuals (e.g., `to = "D"`).
- `mp_absolute_flow()`: Experimental

Examples

```
# infection by mass action
# https://github.com/canmod/macpan2/blob/main/inst/starter_models/si
mp_per_capita_flow("S", "I", "beta * I / N", "infection")

# recovery
# https://github.com/canmod/macpan2/blob/main/inst/starter_models/sir
mp_per_capita_flow("I", "R", "gamma", "recovery")

# disease progression with different severity
# https://github.com/canmod/macpan2/blob/main/inst/starter_models/macpan_base
mp_per_capita_flow("E", "I_mild", "alpha * phi"      , "progression_mild")
mp_per_capita_flow("E", "I_sev" , "alpha * (1 - phi)", "progression_sev")

# birth
# https://github.com/canmod/macpan2/blob/main/inst/starter_models/sir_demog
mp_per_capita_inflow("N", "S", "nu", "birth")

# death
# https://github.com/canmod/macpan2/blob/main/inst/starter_models/sir_demog
mp_per_capita_outflow("S", "mu", "death_S")
mp_per_capita_outflow("I", "mu", "death_I")
mp_per_capita_outflow("R", "mu", "death_R")

# vaccination
# https://github.com/canmod/macpan2/blob/main/inst/starter_models/shiver
mp_per_capita_flow("S", "V", "((a * S)/(b + S))/S",  "vaccination")

# importation (experimental)
# mp_absolute_inflow("I", "delta", "importation")
```

`mp_positions` *Position Vectors*

Description

Return an integer vector of positions of `x` in `table`. Currently this is a simple wrapper around `match`.

Usage

```
mp_positions(x, table, zero_based = TRUE)
```

Arguments

x	Character vector
table	Character vector
zero_based	Use zero-based indexing? Defaults to TRUE, otherwise standard R one-based indexing is used.

---

mp_rbf *Fit a Time-Varying Parameter with Radial Basis Functions*

Description

Pass the output of this function to the tv argument of mp_tmb_calibrator to model time variation of a parameter with flexible radial basis functions.

Usage

```
mp_rbf(
  tv,
  dimension,
  initial_weights,
  seed,
  prior_sd = 1,
  fit_prior_sd = TRUE,
  sparse_tol = 0.01
)
```

Arguments

tv	String giving the name of the parameter.
dimension	Number of bases.
initial_weights	Optional vector with dimensions elements. These are the parameters that are fitted and determine how tv varies with time.
seed	Optional random seed to use to generate the initial_weights if they are not provided.
prior_sd	Prior standard deviation default value for radial basis function coefficients, defaults to 1.
fit_prior_sd	Should the prior sd be be fitted.
sparse_tol	Tolerance below which radial basis function outputs are set exactly to zero. Small values are more accurate but slower. Lack of accuracy can be visually apparent as jumps in graphs of the time-varying parameter.

mp_reduce *Reduce Model*

Description

Reduce a model by removing any model structure (e.g. mp_per_capita_flow), so that expression lists are plain R formulas.

Usage

```
mp_reduce(model)
```

Arguments

model	A model object.

mp_reference *Reference Index*

Description

Extract the index used as a reference for generating position vectors.

Usage

```
mp_reference(x, dimension_name)
```

Arguments

x	Object
dimension_name	Name of a dimension used in a ledger if applicable.

mp_rename *Rename Index Columns*

Description

Rename Index Columns

Usage

```
mp_rename(x, ...)
```

Arguments

x	An index with columns to be renamed.
...	Name-value pairs. The name gives the new name and the value is a character vector giving the old name.

See Also

Other functions that return index tables mp_cartesian(), mp_index(), mp_subset(), mp_union()

mp_simulator *Simulator*

Description

Construct a simulator from a model specification object.

Usage

```
mp_simulator(model, time_steps, outputs, default = list())
```

Arguments

model	A model specification object.
time_steps	How many time steps should be simulated when simulations are requested?
outputs	Character vector of names of model quantities that will be outputted when simulations are requested.
default	Named list of numerical objects that will update the default values defined in the model specification object. Any number of objects can be updated or not.

mp_sim_bounds *Simulation Bounds (Experimental)*

Description

Set the simulation bounds (start time and end time) for a calibration. This is used to override the default simulation bounds taken from the observed data passed to `mp_tmb_calibrator`. the first date is when the first simulated time step (chosen to be before the first data point so that infectious individuals can be built up) and the second date is the last simulated time step (chosen to be after the last data point so that there can be a forecast period). the last argument gives the scale of a single time step (in this case it should always be daily).

Usage

```
mp_sim_bounds(sim_start, sim_end, time_scale)
```

Arguments

`sim_start`	Start time of each simulation.
`sim_end`	End time of each simulation.
`time_scale`	Qualitative description of the size of a time step. currently only `"steps"`, `"daily"`, and `"weekly"` are allowed, and but `"steps"` is the only recommended version as the other two are poorly tested and will throw a warning. The recommended `"steps"` option assumes that positive integers are used to indicate a particular point in the simulation.

mp_slices *Slice an index*

Description

Slice an index

Usage

```
mp_slices(index, unpack = c("no", "maybe", "yes"))
```

Arguments

`index`	Index to slice up.
`unpack`	Place factors in the global environment?

mp_square *Self Cartesian Product*

Description

Self Cartesian Product

Usage

```
mp_square(x, suffixes = c("A", "B"))
```

Arguments

x	An index.
suffixes	Length-2 character vector giving suffixes that disambiguate the column names in the output.

See Also

Other functions that take products of index tables and return one index tables mp_cartesian(), mp_linear(), mp_symmetric(), mp_triangle()

mp_state_dependence_frame *State Dependence Frame*

Description

Data frame giving states that per-capita flow rates directly depend on. This is intended for plotting diagrams and not for mathematical analysis, in that it does not describe indirect dependence for flow rates on state variables.

Usage

```
mp_state_dependence_frame(spec)
```

Arguments

spec	Model specification from spec.

mp_state_vars *State Variables*

Description

Get the state variables in a model specification.

Usage

```
mp_state_vars(spec)
```

Arguments

spec	Model specification (mp_tmb_model_spec).

Value

Character vector of names of all state variables that have been explicitly represented in the model using functions like mp_per_capita_flow.

mp_structured_vector *Structured Vectors*

Description

This documentation was originally in mp_index() and should be cleaned up See issue #131. Also this is an experimental feature.

Usage

```
mp_structured_vector(x, ...)

mp_set_numbers(vector, ...)
```

Arguments

x	An index.
...	Passed on to S3 methods.
vector	An index.

Functions

- mp_set_numbers(): Update numerical values of a structured vector. TODO: details on syntax.

Examples

```
state = mp_index(
  Epi = c("S", "I", "S", "I"),
  Age = c("young", "young", "old", "old")
)
state_vector = (state
  |> mp_structured_vector()
  |> mp_set_numbers(Epi = c(S = 1000))
  |> mp_set_numbers(Epi = c(I = 1), Age = "old")
)
print(state_vector)
```

`mp_subset` *Subset of Indexes*

Description

Take a subset of the rows of an index table (see `mp_index`) to produce another index table. The `mp_subset` function gives rows that match a certain criterion and `mp_setdiff` gives rows that do not match.

Usage

```
mp_subset(x, ...)

mp_setdiff(x, ...)
```

Arguments

`x`	Model index.
`...`	Name-value pairs. The names are columns (or sets of columns using dot-concatenation) in `x` and the values are character vectors that refer to labels with respect to those columns. These values determine the resulting subset.

See Also

Other functions that return index tables `mp_cartesian()`, `mp_index()`, `mp_rename()`, `mp_union()`

mp_symmetric *Symmetric Self Cartesian Product*

Description

Symmetric Self Cartesian Product

Usage

```
mp_symmetric(x, y_labelling_column_names, exclude_diag = TRUE)
```

Arguments

x	An index.
y_labelling_column_names	TODO
exclude_diag	Should 'diagonal' commponents be excluded from the output.

See Also

Other functions that take products of index tables and return one index tables `mp_cartesian()`, `mp_linear()`, `mp_square()`, `mp_triangle()`

mp_time_scale *Time Scale*

Description

Time Scale

Usage

```
mp_time_scale(start, end, time_step_scale = c("steps", "daily", "weekly"), ...)
```

Arguments

start	First date or time in the first time step
end	Last date or time in the last time step
time_step_scale	TODO
...	TODO

mp_tmb *Get Underlying TMB Object*

Description

Get the result of `TMB::MakeADFun` underlying a TMB-based model in `macpan2`.

Usage

```
mp_tmb(model)
```

Arguments

`model`	An object based on TMB.

mp_tmbstan_coef *Model Coefficient Table with stan*

Description

Leverages the `tmbstan` and `broom.mixed` packages to generate MCMC-based coefficient tables.

Usage

```
mp_tmbstan_coef(model, tmbstan_args = list(), ...)
```

Arguments

`model`	Object that contains information about model coefficients.
`tmbstan_args`	Arguments to pass on to `tmbstan`, which is used to generate an `rstan` object from the underlying `TMB` object.
`...`	Arguments to pass onto the `broom.mixed::tidy.stanfit` method.

`mp_tmb_calibrator` *Make TMB Calibrator*

Description

Construct an object that can get used to calibrate an object produced by `mp_tmb_model_spec` or `mp_tmb_library`, and possibly modified by `mp_tmb_insert` or `mp_tmb_update`.

Usage

```
mp_tmb_calibrator(
  spec,
  data,
  traj,
  par,
  tv = character(),
  outputs = traj,
  default = list(),
  time = NULL
)
```

Arguments

`spec`	An TMB model spec to fit to data. Such specs can be produced by `mp_tmb_model_spec` or `mp_tmb_library`, and possibly modified with `mp_tmb_insert` and `mp_tmb_update`.
`data`	A data frame containing trajectories to fit to and possibly time-varying parameters. The data must be of the same format as that produced by `mp_trajectory`.
`traj`	A character vector giving the names of trajectories to fit to data, or a named list of likelihood distributions specified with `distribution` for each trajectory.
`par`	A character vector giving the names of parameters, either time-varying or not, to fit using trajectory match.
`tv`	A character vector giving the names of parameters to make time-varying according to the values in `data`, or a radial basis function specified with `mp_rbf`.
`outputs`	A character vector of outputs that will be generated when `mp_trajectory`, `mp_trajectory_sd`, or `mp_trajectory_ensemble` are called on the optimized calibrator. By default it is just the trajectories listed in `traj`.
`default`	A list of default values to use to update the defaults in the `spec`. By default nothing is updated. Alternatively one could use `mp_tmb_update` to update the spec outside of the function. Indeed such an approach is necessary if new expressions, in addition to default updates, need to be added to the spec (e.g. seasonally varying transmission).
`time`	Specify the start and end time of the simulated trajectories, and the time period associated with each time step. Currently the only valid choice is `NULL`, which takes simulation bounds from the `data`.

Examples

```
spec = mp_tmb_library("starter_models", "sir", package = "macpan2")
sim = mp_simulator(spec, 50, "infection")
data = mp_trajectory(sim)
cal = mp_tmb_calibrator(
    spec
  , data
  , traj = "infection"
  , par = "beta"
  , default = list(beta = 0.25)
)
mp_optimize(cal)
mp_tmb_coef(cal)  ## requires broom.mixed package
```

`mp_tmb_coef` *TMB Model Coefficient Table*

Description

TMB Model Coefficient Table

Usage

```
mp_tmb_coef(model, back_transform = TRUE, ...)
```

Arguments

`model`	Object that contains information about model coefficients.
`back_transform`	A boolean to indicate if model coefficients should be back transformed to display their defaults, estimates, and confidence intervals on the original scale. Coefficient names are also stripped of their transformation identifier. Currently, this back transformation only applies to log transformed coefficients that have been named with "log_" prefix or logit transformed coefficients that have been named with "logit_" prefix. Back transformation also applies to time varying parameters and distributional parameters that get automatic prefixes when used. `back_transform` defaults to `TRUE`.
`...`	Arguments to pass onto the `broom.mixed::tidy.TMB` method.

Value

A data frame that describes the fitted coefficients.

mp_tmb_expr_list *Expression List*

Description

Create a list of expressions for defining a compartmental model in TMB.

Usage

```
mp_tmb_expr_list(
  before = list(),
  during = list(),
  after = list(),
  .simulate_exprs = character(0L)
)
```

Arguments

before	List of formulas to be evaluated in the order provided before the simulation loop begins. Each formula must have a left hand side that gives the name of the matrix being updated, and a right hand side giving an expression containing only the names of matrices in the model, functions defined in macpan2.cpp, and numerical literals (e.g. 3.14). The available functions are described in engine_functions. Names can be provided for the components of before, and these names do not have to be unique. These names are used by the .simulate_exprs argument.
during	List of formulas to be evaluated at every iteration of the simulation loop, with the same rules as before.
after	List of formulas to be evaluated after the simulation loop, with the same rules as before.
.simulate_exprs	Character vector of names of expressions to be evaluated within TMB simulate blocks. This is useful when an expression cannot be evaluated during the computation of the objective function and its gradients (e.g. if the expression contains randomness or other discontinuities that will break the automatic differentiation machinery of TMB).

Value

Object of class ExprList with the following methods.

Methods:

- $data_arg(...): Return the following components of the data structure to pass to C++.
  - expr_output_id – Indices into the list of matrices identifying the matrix being produced.
  - expr_sim_block – Identified whether or not the expression should be evaluated inside a simulate macro within TMB.

- `expr_num_p_table_rows` – Number of rows associated with each expression in the parse table (`p_table_*`)
- `eval_schedule` – Vector giving the number of expressions to evaluate in each phase (before, during, or after) of the simulation.
- `p_table_x` – Parse table column giving an index for looking up either function, matrix, or literal.
- `p_table_n` – Parse table column giving the number of arguments in functions.
- `p_table_i` – Parse table column giving indices for looking up the rows in the parse table corresponding with the first argument of the function.

Method Arguments:

- `...`: Character vector containing the names of the matrices in the model.

`mp_tmb_fixef_cov` *Covariance of Fixed Effect Estimates*

Description

Covariance of Fixed Effect Estimates

Usage

```
mp_tmb_fixef_cov(model)
```

Arguments

`model`	Object that contains information about fitted model parameters.

Value

A covariance matrix.

`mp_tmb_insert` *Modify a TMB Model Spec*

Description

Insert, update, or delete elements of a TMB model spec, produced using `mp_tmb_library` or `mp_tmb_model_spec`, or `mp_tmb_delete`. The only difference between `mp_tmb_insert` and `mp_tmb_update` is that the former shifts the positions of existing expressions to make room for the new expressions, whereas the latter overwrites existing expressions using the new expressions. The treatment of new `default` values and `integers` is the same. The examples below clarify this difference. Note that `mp_tmb_delete` does not contain an `expressions` argument, because it is not necessary to specify new expressions in the case of deletion.

Usage

```
mp_tmb_insert(
  model,
  phase = "during",
  at = 1L,
  expressions = list(),
  default = list(),
  integers = list(),
  must_save = character(),
  must_not_save = character(),
  sim_exprs = character()
)

mp_tmb_update(
  model,
  phase = "during",
  at = 1L,
  expressions = list(),
  default = list(),
  integers = list(),
  must_save = character(),
  must_not_save = character(),
  sim_exprs = character()
)

mp_tmb_delete(
  model,
  phase,
  at,
  default = character(),
  integers = character(),
  must_save = character(),
  must_not_save = character(),
  sim_exprs = character()
)
```

Arguments

model	TMB model spec object produced using mp_tmb_library or mp_tmb_model_spec.
phase	At what phase should expressions be inserted, updated, or deleted.
at	Expression number, which can be identified by printing out model, at which the expressions should be inserted or updated. If inserted then the existing expressions with number at and higher are shifted after the new expressions are added. If updated, then the existing expressions with number from at to at + length(expressions) - 1 are replaced with the new expressions. For mp_tmb_delete, a numeric vector of integers identifying expressions to delete from the model.

expressions	Expressions to insert into the model spec or to replace existing expressions.
default	Named list of objects, each of which can be coerced into a numeric matrix. The names refer to variables that appear in before, during, and after. For mp_tmb_delete, a character vector of such objects to delete from the model.
integers	Named list of vectors that can be coerced to integer vectors. These integer vectors can be used by name in model formulas to provide indexing of matrices and as grouping factors in group_sums. For mp_tmb_delete, a character vector of such objects to delete from the model.
must_save	Character vector of the names of matrices that must have their values stored at every iteration of the simulation loop. For example, a matrix that the user does not want to be returned but that impacts dynamics with a time lag must be saved and therefore in this list.
must_not_save	Character vector of the names of matrices that must not have their values stored at every iteration of the simulation loop. For example, the user may ask to return a very large matrix that would create performance issues if stored at each iteration. The creator of the model can mark such matrices making it impossible for the user of the model to save their full simulation history.
sim_exprs	Character vector of the names of before, during, and after expressions that must only be evaluated when simulations are being produced and not when the objective function is being evaluated. For example, expressions that generate stochasticity should be listed in sim_exprs because TMB objective functions must be continuous.

Details

These modifications do not update the model specification in-place. Instead the output of mp_tmb_insert, mp_tmb_update, and mp_tmb_delete define a new model specification and should be saved if you want to use the new model (e.g., new_model = mp_tmb_insert(model, ...)).

Value

A new model spec object with updated and/or inserted information.

Examples

```
si = mp_tmb_library("starter_models", "si", package = "macpan2")
print(si)

## Update the mixing process to include
## optional phenomenological heterogeneity.
## We need mp_tmb_update here so that
## the previous infection expression is
## overwritten.
mp_tmb_update(si, phase = "during"
  , at = 1
  , expressions = list(infection ~ beta * I * (S/N)^zeta)
  , default = list(zeta = 1)
)
```

```
## Parameterize with log_beta in place of beta.
## We need mp_tmb_insert here so that the
## existing expression for computing the initial
## number of susceptible indiviudals is not
## overwritten.
mp_tmb_insert(si, phase = "before"
  , at = 1
  , expressions = list(beta ~ exp(log_beta))
  , default = list(log_beta = log(0.5))
)
```

`mp_tmb_insert_reports` *Insert Reports*

Description

A version of `mp_tmb_insert` making it more convenient to transform an incidence variable into a reports variable, which accounts for reporting delays and under-reporting. This new reports variable is a convolution of the simulation history of an incidence variable with a kernel that is proportional to a Gamma distribution of reporting delay times.

Usage

```
mp_tmb_insert_reports(
  model,
  incidence_name,
  report_prob,
  mean_delay,
  cv_delay,
  reports_name = sprintf("reported_%s", incidence_name),
  report_prob_name = sprintf("%s_report_prob", incidence_name),
  mean_delay_name = sprintf("%s_mean_delay", incidence_name),
  cv_delay_name = sprintf("%s_cv_delay", incidence_name)
)
```

Arguments

`model`	A model produced by `mp_tmb_model_spec`.
`incidence_name`	Name of the incidence variable to be transformed.
`report_prob`	Value to use for the reporting probability; the proportion of cases that get reported.
`mean_delay`	Mean of the Gamma distribution of reporting delay times.
`cv_delay`	Coefficient of variation of the Gamma distribution of reporting delay times.
`reports_name`	Name of the new reports variable.
`report_prob_name`	Name of the variable containing `report_prob`.

mean_delay_name
: Name of the variable containing mean_delay.

cv_delay_name
: Name of the variable containing cv_delay.

---

mp_tmb_library *TMB Library*

Description

Get a TMB model specification from a model library.

Usage

```
mp_tmb_library(..., package = NULL, alternative_specs = FALSE)
```

Arguments

...
: File path components pointing to a directory that contains an R script that creates an object called spec, which is produced by mp_tmb_model_spec.

package
: If NULL, file.path is used to put together the ... components but if package is the name of a package (as a character string) then system.file is used to put together the ... components.

alternative_specs
: If TRUE, return a list of alternative specification objects. For models without alternatives this will cause the return value to be a list with one element containing a spec object.

See Also

show_models()

Examples

```
mp_tmb_library(
    "starter_models"
  , "si"
  , package = "macpan2"
)
```

`mp_tmb_model_spec` *Specify a TMB Model*

Description

Specify a model in the TMB engine.

Usage

```
mp_tmb_model_spec(
  before = list(),
  during = list(),
  after = list(),
  default = list(),
  integers = list(),
  must_save = character(),
  must_not_save = character(),
  sim_exprs = character(),
  state_update = c("euler", "rk4", "euler_multinomial", "hazard")
)
```

Arguments

`before`	List of formulas to be evaluated (in the order provided) before the simulation loop begins. Each `formula` must have a left hand side that gives the name of the matrix being updated, and a right hand side giving an expression containing only the names of matrices in the model, functions defined in the TMB engine, and numerical literals (e.g. `3.14`). The available functions in the TMB engine can be described in `engine_functions`. Names can be provided for the components of `before`, and these names do not have to be unique. These names are used by the `sim_exprs` argument.
`during`	List of formulas to be evaluated at every iteration of the simulation loop, with the same rules as `before`.
`after`	List of formulas to be evaluated after the simulation loop, with the same rules as `before`.
`default`	Named list of objects, each of which can be coerced into a `numeric matrix`. The names refer to variables that appear in `before`, `during`, and `after`.
`integers`	Named list of vectors that can be coerced to integer vectors. These integer vectors can be used by name in model formulas to provide indexing of matrices and as grouping factors in `group_sums`.
`must_save`	Character vector of the names of matrices that must have their values stored at every iteration of the simulation loop. For example, a matrix that the user does not want to be returned but that impacts dynamics with a time lag must be saved and therefore in this list.

must_not_save	Character vector of the names of matrices that must not have their values stored at every iteration of the simulation loop. For example, the user may ask to return a very large matrix that would create performance issues if stored at each iteration. The creator of the model can mark such matrices making it impossible for the user of the model to save their full simulation history.
sim_exprs	Character vector of the names of `before`, `during`, and `after` expressions that must only be evaluated when simulations are being produced and not when the objective function is being evaluated. For example, expressions that generate stochasticity should be listed in `sim_exprs` because TMB objective functions must be continuous.
state_update	(experimental) Optional character vector for how to update the state variables when it is relevant. Options include `"euler"`, `"rk4"`, and `"euler_multinomial"`.

---

mp_traj *Trajectory Specification*

Description

Specify a set of trajectories to fit. The output of this function is intended to be passed to the `traj` argument of `mp_tmb_calibrator`.

Usage

```
mp_traj(likelihood = list(), condensation = list())
```

Arguments

likelihood	List of likelihood components. The names of the list identify the trajectory associated with each likelihood component.
condensation	List of condensation methods. The names of the list identify the trajectories produced by each condensation method.

---

mp_trajectory *Trajectory*

Description

Return simulations of the trajectory of the output variables of a dynamical model simulator. To see this functionality in context, please see `vignette("quickstart")`.

Usage

```
mp_trajectory(model, include_initial = FALSE)

mp_trajectory_sd(model, conf.int = FALSE, conf.level = 0.95)

mp_trajectory_ensemble(model, n, probs = c(0.025, 0.975))

mp_trajectory_sim(model, n, probs = c(0.025, 0.25, 0.5, 0.75, 0.975))

mp_trajectory_replicate(model, n)
```

Arguments

model	A dynamical model simulator produced by mp_simulator.
include_initial	Should the initial values of the simulation be included in the output? If TRUE this will include outputs for time == 0 associated with the initial values. See mp_initial for another approach to getting the initial values.
conf.int	Should confidence intervals be produced?
conf.level	If conf.int is TRUE, what confidence level should be used? For example, the default of 0.95 corresponds to 95% confidence intervals.
n	Number of random trajectories to simulate.
probs	Numeric vector of probabilities corresponding to quantiles for summarizing the results over the random realizations.

Value

A data frame with one row for each simulated value and the following columns.

matrix Name of the variable in the model. All variables are matrix-valued in macpan2 (scalars are technically 1-by-1 matrices), which explains the name of this field. In hindsight I would have called it variable.

time Time index of the simulated value, with time = 0 indicating initial values.

row The 0-based index of the row of the matrix, or the name of the row of the matrix if row names (or names for column vectors) are supplied for the default value of the matrix.

col The 0-based index of the column of the matrix, or the name of the column of the matrix if column names are supplied for the default value of the matrix. It is also possible that this column is blank if everything is either a scalar or column vector (a common case).

value (mp_trajectory and mp_trajectory_sd) Simulation values.

sd (for mp_trajectory_sd only) The standard deviations of the simulated values accounting for parameter estimation uncertainty.

conf.low (for mp_trajectory_sd only) The lower bounds of the confidence interval for the simulated values.

conf.high (for mp_trajectory_sd only) The upper bounds of the confidence interval for the simulated values.

n% (for mp_trajectory_[ensemble|sim]) The n-th quantiles of the simulation values over repeated simulations.

Functions

- `mp_trajectory_sd()`: Simulate a trajectory that includes uncertainty information provided by the `sdreport` function in `TMB` with default settings.
- `mp_trajectory_ensemble()`: Simulate a trajectory that includes uncertainty information provided by repeatedly sampling from a normal approximation to the distribution of the fitted parameters, and generating one trajectory for each of these samples. The quantiles of the empirical distribution of these trajectories can be used to produce a confidence interval for the fitted trajectory.
- `mp_trajectory_sim()`: Generate quantiles over `n` realizations of the trajectory. Instead of a `value` column in the output data frame, there is one column for each of the quantiles defined in `probs`.
- `mp_trajectory_replicate()`: Generate a list of `n` simulation results.

Examples

```
spec = mp_tmb_library("starter_models"
  , "si"
  , package = "macpan2"
)
simulator = mp_simulator(spec
  , time_steps = 10L
  , outputs = c("infection", "I")
)
trajectory = mp_trajectory(simulator)
print(trajectory)
```

---

mp_triangle *Self Cartesian Product Excluding One Off-Diagonal Side*

Description

Self Cartesian Product Excluding One Off-Diagonal Side

Usage

```
mp_triangle(
  x,
  y_labelling_column_names,
  exclude_diag = TRUE,
  lower_tri = FALSE
)
```

Arguments

x	An index.
y_labelling_column_names	TODO
exclude_diag	Should 'diagonal' commponents be excluded from the output.
lower_tri	Should the lower triangular components be include from the output. If FALSE the result is upper triangular.

See Also

Other functions that take products of index tables and return one index tables mp_cartesian(), mp_linear(), mp_square(), mp_symmetric()

---

mp_union *Union of Indexes*

Description

Union of Indexes

Usage

```
mp_union(...)
```

Arguments

...	Indexes.

See Also

Other functions that return index tables mp_cartesian(), mp_index(), mp_rename(), mp_subset()

---

mp_zero_vector *Zero Vector*

Description

Create a numeric vector of all zeros with names given by x

Usage

```
mp_zero_vector(x, ...)
```

Arguments

x	Object representing the names of the output vector. Most commonly this will be a character vector.
...	Passed on to S3 methods.

`names_and_labels` *Names and Labels*

Description

This page describes functions for giving names and labels to entities in structured models.

Usage

```
to_labels(x)

to_names(x)

to_name(x)

to_name_pairs(x)

to_values(x)
```

Arguments

`x`	Object from which to extract its name, names, labels, name-pairs, or values. Not all types of objects will work for all functions.

Value

Character vector (or numeric vector in the case of `to_values`) that describes `x`.

Functions

- `to_labels()`: Extract a vector for describing the rows of a data frame or values of a numeric vector.
- `to_names()`: Extract a character vector for describing the character-valued columns in a data frame or the flattened structure of a numeric vector. Names obey the following restrictions: (1) they cannot have dots, (2) all values must start with a letter, (3) all characters must be letters, numbers, or underscore.
- `to_name()`: Extract a string (i.e. length-1 character vector) for describing the character-valued columns in a data frame or the flattened structure of a numeric vector. The name of an object is the dot-concatenation of its names.
- `to_name_pairs()`: A character vector with all possible pairwise dot-concatenations of a set of names.
- `to_values()`: Extract the `numeric` column from a data frame with only a single numerical column. This data frame might have more than one column, but only one of them can be numeric. This function will also turn numeric `matrix` and `array` objects with `dimnames` into a flattened numeric vector with labels produced by appropriately dot-concatenating the `dimnames`.

Context

A goal of macpan2 is to provide a mechanism for representing structured compartmental models. An example of such a model is to have each compartment in an SEIR model split into a set of spatial locations and into a set of age groups. It is crucial but difficult to assign meaningful and consistent names to the compartments, flow rates, transmission rates, contact rates, sub-population sizes, and other parameters determining these quantities. Such names should convey how the different quantities relate to one another. For example, the names should make clear that the rate of flow between two compartments is specific to, say, the age group and location of those compartments. The naming system should facilitate identifying model quantities and sets of quantities. For example, in a spatially structured model we might want to refer to all states in a particular location (e.g. Toronto) and a specific state within that location (e.g. susceptible individuals in Toronto).

Model entities (e.g. states, flow rates, transmission rates), can be described using a data frame of string-valued columns. The rows of these data frames represent the entities being represented. The columns of the data frame represent different ways to describe the rows.

```
EpiSympVax = data.frame(
 Epi = c(rep(c("S", "E", "I", "I", "R", "beta"), 2), "alpha", "gamma", "gamma", "infectiousness", "infe
 Symp = c(rep(c("", "", "mild", "severe", "", ""), 2), "", "mild", "severe", "mild", "severe", ""),
  Vax = c(rep(c("unvax", "vax"), each = 6), "", "", "", "", "", "dose_rate")
)
EpiSympVax
#>                Epi   Symp       Vax
#> 1                S            unvax
#> 2                E            unvax
#> 3                I   mild     unvax
#> 4                I severe     unvax
#> 5                R            unvax
#> 6             beta            unvax
#> 7                S              vax
#> 8                E              vax
#> 9                I   mild       vax
#> 10               I severe       vax
#> 11               R              vax
#> 12            beta              vax
#> 13           alpha
#> 14           gamma   mild
#> 15           gamma severe
#> 16 infectiousness   mild
#> 17 infectiousness severe
#> 18                       dose_rate
```

Non-empty values in each cell must contain only letters, numbers, underscores, and must start with a letter. Empty values are zero-length strings that can be used to indicate that some partitions are not applicable to some variables. The purpose for these restrictions is to facilitate the construction of strings and character vectors that summarize different aspects of the data frame. When taken together, these summaries can be inverted to restore the full labelled partition and so they represent zero information loss. This equivalence allows us to go back-and-forth between the two representations without loosing information, but perhaps gaining convenience.

There are three types of summaries: the names, the name, and the labels. The names of a data frame are the names of the string-valued columns.

```
to_names(EpiSympVax)
#> [1] "Epi"  "Symp" "Vax"
```

The name of a data frame is the dot-concatenation of the names.

```
to_name(EpiSympVax)
#> [1] "Epi.Symp.Vax"
```

The labels of a data frame is the row-wise dot-concatenation of the string-valued columns.

```
to_labels(EpiSympVax)
#>  [1] "S..unvax"               "E..unvax"               "I.mild.unvax"
#>  [4] "I.severe.unvax"         "R..unvax"               "beta..unvax"
#>  [7] "S..vax"                 "E..vax"                 "I.mild.vax"
#> [10] "I.severe.vax"           "R..vax"                 "beta..vax"
#> [13] "alpha.."                "gamma.mild."            "gamma.severe."
#> [16] "infectiousness.mild."   "infectiousness.severe." "..dose_rate"
```

These labels give a unique single character string for referring to each variable. With only the labels and one of either the names or the name, one may recover the labelled partition. The labels provide convenient names for the variables – i.e. rownames. By convention we use UpperCamelCase for partition names and a modified form of snake_case for variable labels. Our modification of snake case allows for single uppercase letters in order to accommodate the convention in epidemiology for using single uppercase letters to refer to state variables. For example, `S`, `I`, and `R`, as well as `I_mild` and `I_severe`, would be consistent with our modified snake case style.

`nlist` *Self Naming List*

Description

Self Naming List

Usage

```
nlist(...)
```

Arguments

`...`	Objects to put into the list

rbf *Radial Basis Functions*

Description

Compute a set of radial basis functions (`dimension` of them).

Usage

```
rbf(time_steps, dimension, scale = time_steps/dimension)
```

Arguments

`time_steps`	number of time steps in the model
`dimension`	number of gaussians in the basis
`scale`	width of the gaussians

Examples

```
matplot(rbf(100, 5), type = "l")
```

Reader *Reader*

Description

Construct objects for reading data.

Usage

```
Reader(...)

CSVReader(...)

JSONReader(...)

TXTReader(...)

RReader(...)

NULLReader(...)
```

Arguments

`...`	Character vectors giving path components to the file to be read.

Functions

- `CSVReader()`: Read CSV files.
- `JSONReader()`: Read JSON files.
- `TXTReader()`: Read TXT files.
- `RReader()`: Read R files.
- `NULLReader()`: Placeholder reader that always returns `NULL`.

---

show_models *Print a table of contents of available models*

Description

Collects information from the headers of the README files in the model directories and returns the results as a data frame

Usage

```
show_models(
  dir = system.file("starter_models", package = "macpan2"),
  show_missing = FALSE,
  for_markdown = FALSE
)
```

Arguments

`dir`	directory to list
`show_missing`	(logical) include entries for models with no README information?
`for_markdown`	(logical) format for rendering the table with markdown-formatted links to model readme files?

Value

a data frame containing entries `Directory` (model directory), `Title` (model title), `Description` (short description)

Examples

```
show_models(show_missing = TRUE)
```

simple_sims *Simple Iterated Simulation*

Description

Simple Iterated Simulation

Usage

```
simple_sims(iteration_exprs, time_steps, int_vecs = list(), mats = list())
```

Arguments

iteration_exprs	List of expressions to pass to the engine. The expressions are only allowed to use valid engine_functions. Each expression is evaluated in order, once for each iteration. The number of iterations is controlled by the time_steps argument.
time_steps	Number of time steps to iterate.
int_vecs	Named list of integer vectors.
mats	Named list of matrices.

Value

A data frame with the simulation results.

StringData *String Data*

Description

Create objects for representing names and labels in a dynamical model.

Usage

```
StringDataFromFrame(data)

StringDataFromDotted(labels, name)

## S3 method for class 'StringData'
print(x, ...)
```

Arguments

data	Data frame with names given by column names and labels by the elements of the columns.
labels	Character vector with (dot-separated) partition labels.
name	Character scalar with (dot-separated) partition name.
x	StringData object
...	Not used but present for S3 method consistency.

Methods (by generic)

- print(StringData): Print out a StringData object.

Functions

- StringDataFromFrame(): Construct object from a data frame without any dots in either the names or the values.
- StringDataFromDotted(): Construct object from a character scalar with (dot-separated) partition names and a character vector with (dot-separated) partition labels.

Examples

```
vars = (mp_cartesian(
      mp_index(Epi = c("S", "I", "R"))
    , mp_index(Age = c("young", "old"))
  )
  |> as.data.frame()
  |> StringDataFromFrame()
)
vars
vars$dot()
```

---

to_positions *To Positions*

Description

Return position vector of indices corresponding to the input object.

Usage

```
to_positions(x)
```

Arguments

x	An object of a class that can be converted to a position vector.

See Also

mp_positions()

---

to_string *To String*

Description

Convert an object to a string.

Usage

```
to_string(x)
```

Arguments

x	Object to convert to a string.

Value

A length-1 character vector.

Transform *Transform*

Description

Transform

Usage

```
Transform(variable, default = NULL, trans_variable = variable)

Identity(variable, default = NULL, trans_variable = variable)

Log(variable, default = NULL, trans_variable = sprintf("log_%s", variable))

Logit(
  variable,
  default = NULL,
  trans_variable = sprintf("logit_%s", variable)
)
```

Arguments

variable	Character string giving the name of a variable in the model.
default	Default value for the untransformed variable. If NULL (the default) this value is taken from the initial value in the model containing the transformation.
trans_variable	Character string to use as the name of the transformed version of the variable.

Functions

- Identity(): Identity transformation.
- Log(): Log transformation.
- Logit(): Logit transformation.

---

transform_distr_param *Distributional Parameter Transformation*

Description

Objects used

- mp_identity - Identity transformation
- mp_log - Log transformation
- mp_logit - Logit transformation
- mp_sqrt - Square-root transformation

Usage

```
mp_identity

mp_log

mp_logit

mp_sqrt
```

Index