# Package: paramix (via r-universe)

January 9, 2025

**Title** Aggregate and Disaggregate Continuous Parameters for Compartmental Models

**Version** 0.0.1

**Description** A convenient framework for aggregating and disaggregating continuously varying parameters (for example, case fatality ratio, with age) for proper parametrization of lower-resolution compartmental models (for example, with broad age categories) and subsequent upscaling of model outputs to high resolution (for example, as needed when calculating age-sensitive measures like years-life-lost).

**License** MIT + file LICENSE

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.2

**Depends** R (>= 3.5.0)

**Suggests** lintr, knitr, rmarkdown, ggplot2, roxygen2, wpp2019, testthat (>= 3.0.0), patchwork

**VignetteBuilder** knitr

**Imports** data.table

**Config/testthat/edition** 3

**URL** `https://cmmid.github.io/paramix/`

**Repository** https://cmmid.r-universe.dev

**RemoteUrl** https://github.com/cmmid/paramix

**RemoteRef** HEAD

**RemoteSha** 7bd6278b9ed1ff7e7ae97ff352a9782a0574680d

## Contents

## `alembic` *Create the Blending and Distilling Object*

### Description

Create the Blending and Distilling Object

### Usage

```
alembic(
  f_param,
  f_pop,
  model_partition,
  output_partition,
 pars_interp_opts = interpolate_opts(fun = stats::splinefun, kind = "point", method =
    "natural"),
 pop_interp_opts = interpolate_opts(fun = stats::approxfun, kind = "integral", method =
    "constant", yleft = 0, yright = 0)
)
```

### Arguments

| | |
|---|---|
| `f_param` | a function, `f(x)` which transforms the feature (e.g. age), to yield the parameter values. Alternatively, a `data.frame` where the first column is the feature and the second is the parameter; see `xy.coords()` for details. If the latter, combined with `pars_interp_opts`, and defaulting to spline interpolation. |
| `f_pop` | like `f_param`, either a density function (though it does not have to integrate to 1 like a pdf) or a `data.frame` of values. If the latter, it is treated as a series of populations within intervals, and then interpolated with `pop_interp_opts` to create a density function. |
| `model_partition` | a numeric vector of cut points, which define the partitioning that will be used in the model |
| `output_partition` | the partition of the underlying feature |
| `pars_interp_opts` | a list, minimally with an element `fun`, corresponding to an interpolation function. Defaults to `splinefun()` "natural" interpolation. |
| `pop_interp_opts` | ibid, but for density. Defaults to `approxfun()` "constant" interpolation. |

### Value

a `data.table` with columns `model_partition`, `output_partition`, `weight` and `relpop`. The first two columns identify partition lower bounds, for both the model and output, the other values are associated with; the combination of `model_partition` and `output_partition` forms a unique identifier, but individually they may appear multiple times. Generally, this object is only useful as an input to the `blend()` and `distill()` tools.

### Examples

```
ifr_levin <- function(age_in_years) {
  (10^(-3.27 + 0.0524 * age_in_years))/100
}
age_limits <- c(seq(0, 69, by = 5), 70, 80, 101)
age_pyramid <- data.frame(
  from = 0:101, weight = ifelse(0:101 < 65, 1, .99^(0:101-64))
)
age_pyramid$weight[102] <- 0
# flat age distribution, then 1% annual deaths, no one lives past 101
ifr_alembic <- alembic(ifr_levin, age_pyramid, age_limits, 0:101)
```

## `blend` *Blend Parameters*

### Description

`blend` extracts aggregate parameters from an `alembic` object.

### Usage

```
blend(alembic_dt)
```

### Arguments

| | |
|---|---|
| `alembic_dt` | an `alembic()` return value |

### Value

a `data.table` of with two columns: `model_partition` (partition lower bounds) and `value` (parameter values for those partitions)

### Examples

```
ifr_levin <- function(age_in_years) {
  (10^(-3.27 + 0.0524 * age_in_years))/100
}

age_limits <- c(seq(0, 69, by = 5), 70, 80, 101)
age_pyramid <- data.frame(
  from = 0:101, weight = ifelse(0:101 < 65, 1, .99^(0:101-64))
)
age_pyramid$weight[102] <- 0
# flat age distribution, then 1% annual deaths, no one lives past 101

alembic_dt <- alembic(ifr_levin, age_pyramid, age_limits, 0:101)

ifr_blend <- blend(alembic_dt)
# the actual function
plot(
  60:100, ifr_levin(60:100),
  xlab = "age (years)", ylab = "IFR", type = "l"
)
# the properly aggregated blocks
lines(
  age_limits, c(ifr_blend$value, tail(ifr_blend$value, 1)),
  type = "s", col = "dodgerblue"
)
# naively aggregated blocks
ifr_naive <- ifr_levin(head(age_limits, -1) + diff(age_limits)/2)
lines(
  age_limits, c(ifr_naive, tail(ifr_naive, 1)),
  type = "s", col = "firebrick"
)
# properly aggregated, but not accounting for age distribution
bad_alembic_dt <- alembic(
  ifr_levin,
  within(age_pyramid, weight <- c(rep(1, 101), 0)), age_limits, 0:101
)
ifr_unif <- blend(bad_alembic_dt)
lines(
  age_limits, c(ifr_unif$value, tail(ifr_unif$value, 1)),
  type = "s", col = "darkgreen"
)
```

## distill *Distill Outcomes*

### Description

distill takes a low-age resolution outcome, for example deaths, and proportionally distributes that outcome into a higher age resolution for use in subsequent analyses like years-life-lost style calculations.

### Usage

```
distill(alembic_dt, outcomes_dt, groupcol = names(outcomes_dt)[1])
```

## Arguments

| | |
|---|---|
| `alembic_dt` | an `alembic()` return value |
| `outcomes_dt` | a long-format `data.frame` with a column either named `from` or `model_from` and a column `value` (other columns will be silently ignored) |
| `groupcol` | a string, the name of the outcome model group column. The `outcomes_dt[[groupcol]]` column must match the `model_partition` lower bounds, as provided when constructing the `alembic_dt` with `alembic()`. |

## Details

When the `value` column is re-calculated, note that it will aggregate all rows with matching `groupcol` entries in `outcomes_dt`. If you need to group by other features in your input data (e.g. if you need to distill outcomes across multiple simulation outputs or at multiple time points), that has to be done by external grouping then calling `distill()`.

## Value

a `data.frame`, with `output_partition` and recalculated `value` column

## Examples

```
ifr_levin <- function(age_in_years) {
  (10^(-3.27 + 0.0524 * age_in_years))/100
}

age_limits <- c(seq(0, 69, by = 5), 70, 80, 101)
age_pyramid <- data.frame(
  from = 0:101, weight = ifelse(0:101 < 65, 1, .99^(0:101-64))
)
age_pyramid$weight[102] <- 0
# flat age distribution, then 1% annual deaths, no one lives past 101

alembic_dt <- alembic(ifr_levin, age_pyramid, age_limits, 0:101)

results <- data.frame(model_partition = head(age_limits, -1))
results$value <- 10
distill(alembic_dt, results)
```

---

## `distill_summary` *Distillation Calculation Comparison Summary*

---

## Description

Implements several approaches to imputing higher resolution outcomes, then tables them up for convenient plotting.

## Usage

```
distill_summary(alembic_dt, outcomes_dt, groupcol = names(outcomes_dt)[1])
```

## Arguments

| | |
|---|---|
| `alembic_dt` | an `alembic()` return value |
| `outcomes_dt` | a long-format `data.frame` with a column either named `from` or `model_from` and a column `value` (other columns will be silently ignored) |
| `groupcol` | a string, the name of the outcome model group column. The `outcomes_dt[[groupcol]]` column must match the `model_partition` lower bounds, as provided when constructing the `alembic_dt` with `alembic()`. |

## Value

a `data.table`, columns:

- `partition`, the feature point corresponding to the value
- `value`, the translated `outcomes_dt$value`
- `method`, a factor with levels indicating how feature points are selected, and how `value` is weighted to those features:
  - `f_mid`: features at the `alembic_dt` outcome partitions, each with value corresponding to the total value of the corresponding model partition, divided by the number of outcome partitions in that model partition
  - `f_mean`: the features at the model partition means
  - `mean_f`: the features distributed according to the relative density in the outcome partitions
  - `wm_f`: the `alembic()` approach

## Examples

```
library(data.table)
f_param <- function(age_in_years) {
  (10^(-3.27 + 0.0524 * age_in_years))/100
}

model_partition <- c(0, 5, 20, 65, 101)
density_dt <- data.table(
  from = 0:101, weight = c(rep(1, 66), exp(-0.075 * 1:35), 0)
)
alembic_dt <- alembic(
  f_param, density_dt, model_partition, seq(0, 101, by = 1L)
)

# for simplicity, assume a uniform force-of-infection across ages =>
# infections proportion to population density.
model_outcomes_dt <- density_dt[from != max(from), .(value = sum(f_param(from) * weight)),
  by = .(model_from = model_partition[findInterval(from, model_partition)])
]
```

```
ds_dt <- distill_summary(alembic_dt, model_outcomes_dt)
```

## `interpolate_opts` *Interpolation Options*

### Description

Creates and interpolation options object for use with `alembic()`.

### Usage

```
interpolate_opts(fun, kind = c("point", "integral"), ...)
```

### Arguments

| | |
|---|---|
| `fun` | a function |
| `kind` | a string; either "point" or "integral". How to interpret the x, y values being interpolated. Either as point observations of a function OR as the integral of the function over the interval. |
| `...` | arbitrary other arguments, but checked against signature of `fun` |

### Details

This method creates the interpolation object for use with `alembic()`; this is a convenience method, which does basic validation on arguments and ensures the information used in `alembic()` to do interpolation is available.

The `...` arguments will be provided to `fun` when it is invoked to interpolate the tabular "functional" form of arguments to `alembic()`. If `fun` has an argument `kind`, that parameter will also be passed when invoking the function; if not, then the input data will be transformed to $\{x, z\}$ pairs, such that $x_{i+1} - x_i * z_i = y_i$ - i.e., transforming to a point value and a functional form which is assumed constant until the next partition.

### Value

a list, with `fun` and `kind` keys, as well as whatever other valid keys appear in `...`.

### Examples

```
interpolate_opts(
  fun = stats::splinefun, method = "natural", kind = "point"
)
interpolate_opts(
  fun = stats::approxfun, method = "constant", yleft = 0, yright = 0,
  kind = "integral"
)
```

## parameter_summary *Parameter Calculation Comparison Summary*

### Description

Implements several approaches to computing partition-aggregated parameters, then tables them up for convenient plotting.

### Usage

```
parameter_summary(f_param, f_pop, model_partition, resolution = 101L)
```

### Arguments

| | |
|---|---|
| `f_param` | a function, `f(x)` which transforms the feature (e.g. age), to yield the parameter values. Alternatively, a `data.frame` where the first column is the feature and the second is the parameter; see `xy.coords()` for details. If the latter, combined with `pars_interp_opts`, and defaulting to spline interpolation. |
| `f_pop` | like `f_param`, either a density function (though it does not have to integrate to 1 like a pdf) or a `data.frame` of values. If the latter, it is treated as a series of populations within intervals, and then interpolated with `pop_interp_opts` to create a density function. |
| `model_partition` | a numeric vector of cut points, which define the partitioning that will be used in the model |
| `resolution` | the number of points to calculate for the underlying `f_param` function. The default 101 points means 100 partitions. |

### Value

a `data.table`, columns:

- `model_category`, a integer corresponding to which of the intervals of `model_partition` the `x` value is in
- `x`, a numeric series from the first to last elements of `model_partition` with length `resolution`
- `method`, a factor with levels:
  - `f_val`: `f_param(x)`
  - `f_mid`: `f_param(x_mid)`, where `x_mid` is the midpoint x of the `model_category`
  - `f_mean`: `f_param(weighted.mean(x, w))`, where `w` defined by `densities` and `model_category`
  - `mean_f`: `weighted.mean(f_param(x), w)`, same as previous
  - `wm_f`: the result as if having used `paramix::blend()`; this should be very similar to `mean_f`, though will be slightly different since `blend` uses `integrate()`

## Examples

```
# COVID IFR from Levin et al 2020 https://doi.org/10.1007/s10654-020-00698-1
f_param <- function(age_in_years) {
  (10^(-3.27 + 0.0524 * age_in_years))/100
}

densities <- data.frame(
  from = 0:101,
  weight = c(rep(1, 66), exp(-0.075 * 1:35), 0)
)

model_partition <- c(0, 5, 20, 65, 101)

ps_dt <- parameter_summary(f_param, densities, model_partition)
ps_dt


ggplot(ps_dt) + aes(x, y = value, color = method) +
  geom_line(data = \(dt) subset(dt, method == "f_val")) +
  geom_step(data = \(dt) subset(dt, method != "f_val")) +
  theme_bw() + theme(
    legend.position = "inside", legend.position.inside = c(0.05, 0.95),
    legend.justification = c(0, 1)
  ) + scale_color_discrete(
    "Method", labels = c(
      f_val = "f(x)", f_mid = "f(mid(x))", f_mean = "f(E[x])",
      mean_f = "discrete E[f(x)]", wm_f = "integrated E[f(x)]"
    )
  ) +
  scale_x_continuous("Age", breaks = seq(0, 100, by = 10)) +
  scale_y_log10("IFR", breaks = 10^c(-6, -4, -2, 0), limits = 10^c(-6, 0))
```

# Index