

Package: calibrar (via r-universe)

October 21, 2026

Version 0.9.0.9011

Title Automated Parameter Estimation for Complex Models

Description General optimisation and specific tools for the parameter estimation (i.e. calibration) of complex models, including stochastic ones. It implements generic functions that can be used for fitting any type of models, especially those with non-differentiable objective functions, with the same syntax as 'stats::optim()'. It supports multiple phases estimation (sequential parameter masking), constrained optimization (bounding box restrictions) and automatic parallel computation of numerical gradients. Some common maximum likelihood estimation methods and automated construction of the objective function from simulated model outputs is provided. See <https://roliveros-ramos.github.io/calibrar/> for more details.

Depends R (>= 3.5.0)

Imports BB, cmaes, DEoptim, dfoptim, GenSA, graphics, minqa, optimx, foreach, lbfgsb3c, parallel, pso, rgenoud, soma, stats, stringr, utils

Suggests deSolve, ibm, knitr, rmarkdown, testthat (>= 3.0.0)

License GPL-2

Encoding UTF-8

VignetteBuilder knitr

URL <https://roliveros-ramos.github.io/calibrar/>

BugReports <https://github.com/roliveros-ramos/calibrar/issues>

ByteCompile TRUE

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

Config/Needs/website rmarkdown

Config/roxygen2/version 8.0.0

Config/pak/sysreqs cmake make libicu-dev

Repository <https://roliveros-ramos.r-universe.dev>
Date/Publication 2026-10-21 14:46:43 UTC
RemoteUrl <https://github.com/roliveros-ramos/calibrar>
RemoteRef HEAD
RemoteSha 9e333f327b34e8c09246e81ed9251b13e4633c18

Contents

calibrar-package	2
.get_command_argument	4
.read_configuration	5
ahres	6
ahres.results	9
calibrar.results	10
calibrar_demo	13
calibrate	14
calibration_data	17
calibration_objFn	19
calibration_setup	22
gaussian_kernel	24
gradient	25
objFn	26
optim2	30
optimh	33
sphereN	35
spline_par	36
Index	38

calibrar-package	<i>Automated Calibration for Complex Models</i>
------------------	---

Description

Automated Calibration for Complex Models

Details

calibrar package: Automated Calibration for Complex Models

This package allows the parameter estimation (i.e. calibration) of complex models, including stochastic ones. It implements generic functions that can be used for fitting any type of models, especially those with non-differentiable objective functions, with the same syntax as `base::optim`. It supports multiple phases estimation (sequential parameter masking), constrained optimization (bounding box restrictions) and automatic parallel computation of numerical gradients. Some common maximum likelihood estimation methods and automated construction of the objective function from simulated model outputs is provided. See <https://roliveros-ramos.github.io/calibrar/> for more details.

Author(s)

Ricardo Oliveros-Ramos Maintainer: Ricardo Oliveros-Ramos ricardo.oliveros@gmail.com

References

calibrar: an R package for the calibration of ecological models (Oliveros-Ramos and Shin 2014)

See Also

Useful links:

- <https://roliveros-ramos.github.io/calibrar/>
- Report bugs at <https://github.com/roliveros-ramos/calibrar/issues>

Examples

```
## Not run:
require(calibrar)
set.seed(880820)
path = NULL # NULL to use the current directory
# create the demonstration files
demo = calibrar_demo(model="PoissonMixedModel", L=5, T=100)
# get calibration information
calibrationInfo = calibration_setup(file=demo$path)
# get observed data
observed = calibration_data(setup=calibrationInfo, path=demo$path)
# read forcings for the model
forcing = read.csv(file.path(demo$path, "master", "environment.csv"), row.names=1)
# Defining 'runModel' function
runModel = function(par, forcing) {
  output = calibrar:::PoissonMixedModel(par=par, forcing=forcing)
  # adding gamma parameters for penalties
  output = c(output, list(gammas=par$gamma))
  return(output)
}
# real parameters
cat("Real parameters used to simulate data\n")
print(demo$par)
# objective functions
obj = calibration_objFn(model=runModel, setup=calibrationInfo,
                        observed=observed, forcing=forcing)
cat("Starting calibration...\n")
control = list(weights=calibrationInfo$weights, maxit=3.6e5) # control parameters
cat("Running optimization algorithms\n", "\t", date(), "\n")
cat("Running optim AHR-ES\n")
ahr = calibrate(par=demo$guess, fn=obj, lower=demo$lower, upper=demo$upper, control=control)
summary(ahr)

## End(Not run)
```

.get_command_argument *Extract a named command-line argument*

Description

Parse command-line arguments and return the value associated with a named option. This helper is primarily intended for scripting workflows that call R from the shell and pass arguments such as `--config=file.txt`.

Usage

```
.get_command_argument(  
  x,  
  argument,  
  prefix = "--",  
  default = FALSE,  
  verbose = FALSE  
)
```

Arguments

<code>x</code>	Character vector of command-line arguments, typically obtained from <code>commandArgs()</code> .
<code>argument</code>	Name of the argument to retrieve, without the prefix.
<code>prefix</code>	Prefix used for arguments of interest. The default is <code>--</code> .
<code>default</code>	Value returned when the argument is not found. The default is <code>FALSE</code> .
<code>verbose</code>	Logical. If <code>TRUE</code> , a warning is issued when the requested argument is not found.

Details

The function looks for entries matching either `prefix + argument` or `prefix + argument = value`. For example, with `prefix = "--"` and `argument = "config"`, both `--config` and `--config=file.txt` are recognised.

This function is exported because it can be useful in external packages or scripts that build command-line interfaces around `calibrar` workflows.

Value

The parsed argument value. If the argument is present without an explicit `=value` part, `TRUE` is returned. If the argument is absent, `default` is returned. Parsed values are passed through an internal type guesser, so logical and numeric values may be returned with their corresponding R types.

See Also

[`.read\_configuration\(\)`](#)

Examples

```
.get_command_argument(commandArgs(), "interactive")  
.get_command_argument(commandArgs(), "RStudio")  
.get_command_argument(commandArgs(), "RStudio", prefix = "")  
.get_command_argument(commandArgs(), "vanilla")  
.get_command_argument("--control.file=baz.txt", "control.file")
```

<code>.read_configuration</code>	<i>Read a simple key-value configuration file</i>
----------------------------------	---

Description

Read a plain-text configuration file in which each non-empty, non-comment line defines a key and one or more values separated by a recognised delimiter. Parsed values are returned as a named list.

Usage

```
.read_configuration(  
  file,  
  recursive = TRUE,  
  keep.names = TRUE,  
  conf.key = NULL,  
  ...  
)
```

Arguments

<code>file</code>	Path to the configuration file.
<code>recursive</code>	Logical. If TRUE, keys matching <code>conf.key</code> are treated as references to additional configuration files and are read recursively.
<code>keep.names</code>	Logical. If TRUE, keys are kept as written. If FALSE, they are converted to lower case.
<code>conf.key</code>	Optional pattern identifying keys that should be interpreted as additional configuration files.
<code>...</code>	Reserved for future extensions.

Details

The reader accepts simple lines of the form `key separator value`, where the separator may vary across lines. The current parser guesses the separator among common choices such as `=`, `;`, `,`, `:`, and `tab`.

Lines beginning with punctuation after trimming are ignored, and trailing comments are removed. When `conf.key` is supplied, matching entries are treated as file references. Their paths are resolved relative to the parent configuration file, and the referenced files can be merged recursively.

This helper is intentionally lightweight. It is useful for small calibration or script configuration files, but it is not intended to replace richer configuration formats such as YAML.

Value

A named list containing the parsed configuration values. Scalars are returned as length-one vectors where appropriate, and simple values are converted with `type.convert()`. If file does not exist, NULL is returned with a warning.

See Also

[.get_command_argument\(\)](#)

Examples

```
tf = tempfile(fileext = ".txt")
writeLines(c(
  "ncores = 4",
  "verbose = TRUE",
  "method = AHR-ES"
), tf)

cfg = .read_configuration(tf)
str(cfg)
```

ahres

Adaptive Hierarchical Recombination Evolutionary Strategy

Description

`ahres()` runs the AHR-ES optimiser directly through a dedicated interface. It is primarily intended for bounded, derivative-free optimisation of expensive or black-box objective functions, including calibration problems in which the objective may be noisy or obtained from an external model.

Usage

```
ahres(
  par,
  fn,
  gr = NULL,
  ...,
  lower = -Inf,
  upper = +Inf,
  active = NULL,
  control = list(),
  hessian = FALSE,
  parallel = FALSE
)
```

Arguments

<code>par</code>	A numeric vector or list. The length of <code>par</code> defines the number of parameters to be estimated (i.e., the dimension of the problem).
<code>fn</code>	The objective function to be minimised. It should accept a parameter vector (or list, depending on the wrapper) as first argument and return either a scalar value (single-objective) or a numeric vector (multi-objective).
<code>gr</code>	Function computing the gradient of <code>fn</code> . Ignored by <code>ahres()</code> .
<code>...</code>	Additional arguments passed to <code>fn</code> and <code>gr</code> .
<code>lower</code>	Lower bounds for parameters. One value or a vector of the same length as <code>par</code> . NA is treated as $-\text{Inf}$. Default is unconstrained.
<code>upper</code>	Upper bounds for parameters. One value or a vector of the same length as <code>par</code> . NA is treated as Inf . Default is unconstrained.
<code>active</code>	Boolean vector of the same length as <code>par</code> , indicating if the parameter is used in the optimisation (TRUE) or held at a fixed value (FALSE).
<code>control</code>	A list of control options. Common options include <code>ncores</code> , <code>run</code> , <code>master</code> , <code>verbose</code> , <code>REPORT</code> , <code>restart.file</code> , <code>gradient</code> , and <code>gr.method</code> . Additional solver-specific options may be passed through to the underlying optimiser.
<code>hessian</code>	Logical. Should a numerically differentiated Hessian matrix be returned? Currently not implemented.
<code>parallel</code>	Logical. Enable parallel computation (e.g., for replicated evaluations and/or numerical gradients) using up to <code>control\$ncores</code> cores.

Details

`ahres()` accepts the same broad interface as the other optimisation entry points in **calibrar**, but is specialised to the "AHR-ES" algorithm. Parameters may be supplied as a vector or list. Internally, list-structured parameters are flattened for optimisation and then restored before the result is returned.

The function supports box constraints through `lower` and `upper`, and parameter masking through `active`, so only selected parameters are evolved while the others are held fixed at their supplied starting values.

If `control$parscale` and/or `control$fnscale` are supplied, optimisation is performed on the corresponding scaled parameter space and objective scale, and results are transformed back before being returned.

`gr` is accepted only for interface compatibility and is ignored by this method. In the current implementation, `control$maxit` or `control$maxgen` must be supplied; otherwise the optimisation stops with an error.

For scalar objectives, the default settings are usually sufficient. For vector-valued objectives, advanced users may also need to provide suitable `control` entries such as `weights`, `nvar`, and `aggFn`, depending on how the objective has been constructed.

Value

An object of class `"ahres.results"`, typically with components:

par Best parameter values found, returned on the original scale.

value Objective value at par, returned on the original objective scale.

counts Number of function and gradient evaluations.

convergence Convergence code (0 usually indicates a regular stop, 1 that the maximum number of generations/evaluations was reached).

message A short message describing the stopping condition.

generations Number of generations performed.

partial Optional vector of objective components when the objective function returns more than one value.

trace Optional tracing information when tracing is enabled.

active Information about active parameters used during optimisation.

Parallel execution

AHR-ES parallelism is controlled through entries inside `control`, notably `control$parallel` and `control$ncores`. The top-level `parallel` argument is currently retained for interface consistency but is not used directly inside `ahres()`.

Author(s)

Ricardo Oliveros-Ramos

See Also

[optimh\(\)](#), [optim2\(\)](#), [calibrate\(\)](#), [summary.ahres.results\(\)](#)

Other optimisers: [calibrate\(\)](#), [optim2\(\)](#), [optimh\(\)](#)

Examples

```
## Not run:
ahres(
  par = rep(0.5, 5),
  fn = sphereN,
  lower = -5,
  upper = 5,
  control = list(maxit = 200)
)

## End(Not run)
```

ahres.results

Evolutionary-strategy result objects

Description

Objects of class `ahres.results` are returned by `ahres()`. They store the fitted parameters and the main diagnostics produced by the AHR-ES optimiser.

`coef()` returns the fitted parameter values stored in an `ahres.results` object.

`plot()` is currently a placeholder method for `ahres.results` objects. No graphical summary is produced at present.

Usage

```
## S3 method for class 'ahres.results'
print(x, short = FALSE, ...)

## S3 method for class 'ahres.results'
coef(object, ...)

## S3 method for class 'ahres.results'
plot(x, ...)

## S3 method for class 'ahres.results'
summary(object, ...)

## S3 method for class 'summary.ahres.results'
print(x, ...)
```

Arguments

<code>x</code>	An <code>ahres.results</code> object, or a <code>summary.ahres.results</code> object for <code>print.summary.ahres.results()</code>
<code>short</code>	Logical. If TRUE, print a compact summary without listing the full parameter vector.
<code>...</code>	Additional arguments passed to the underlying method.
<code>object</code>	An <code>ahres.results</code> object.

Format

A list, usually with the following components:

par Best parameter values found by the AHR-ES algorithm.

value Aggregated objective-function value at `par`.

counts Named integer vector giving the number of function and gradient evaluations recorded by the optimiser.

convergence Integer convergence code. 0 indicates that the stopping criterion was reached before the generation limit; 1 indicates that the maximum number of generations or function evaluations was reached.

message Character string describing the stopping condition.

generations Number of generations completed.

partial Optional vector of partial objective-function values, present when the objective function returns multiple components.

trace Optional tracing information collected during the run.

active Information on which parameters were active in the optimisation, as stored by [ahres\(\)](#).

Details

These objects are specific to the AHR-ES optimisation workflow. They contain the final parameter vector, the aggregated objective value, convergence and iteration information, and, when relevant, partial objective values and tracing information.

The main accessors are [print\(\)](#), [summary\(\)](#), and [coef\(\)](#). A [plot\(\)](#) method is currently exported as a placeholder but does not yet produce a graphical summary.

Value

`coef.ahres.results()` returns the fitted parameter vector.

`plot.ahres.results()` returns NULL, invisibly.

`summary.ahres.results()` returns an object of class `summary.ahres.results`.

See Also

[ahres\(\)](#), [summary\(\)](#), [coef\(\)](#)

<code>calibrar.results</code>	<i>Calibration result objects</i>
-------------------------------	-----------------------------------

Description

Objects of class `calibrar.results` are returned by [calibrate\(\)](#). They store the final parameter values together with the objective value, convergence information, the set of calibrated parameters, and tracing information from the calibration run.

`coef()` returns the fitted parameter values stored in a `calibrar.results` object.

`plot()` is currently a placeholder method for `calibrar.results` objects. No graphical summary is produced at present.

`dim()` returns the number of fitted parameters after unlisting the parameter structure stored in the result object.

Usage

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

## S3 method for class 'calibrar.results'
coef(object, ...)

## S3 method for class 'calibrar.results'
predict(object, replicates = 1, na.rm = FALSE, ...)

## S3 method for class 'calibrar.results'
plot(x, ...)

## S3 method for class 'calibrar.results'
summary(object, ..., show_par = NULL, par.only = FALSE)

## S3 method for class 'summary.calibrar.results'
print(x, digits = 3, ...)

## S3 method for class 'calibrar.results'
dim(x)
```

Arguments

<code>x</code>	A <code>calibrar.results</code> object, or a <code>summary.calibrar.results</code> object for <code>print.summary.calibrar.results</code> .
<code>...</code>	Additional arguments passed to the underlying method.
<code>object</code>	A <code>calibrar.results</code> object.
<code>replicates</code>	Number of replicate evaluations of the fitted parameter set to perform. If <code>replicates = 1</code> , the underlying model function is evaluated once. If <code>replicates > 1</code> , repeated evaluations are combined by taking the mean across replicates for each returned output.
<code>na.rm</code>	Logical. Passed to the internal averaging step used when <code>replicates > 1</code> .
<code>show_par</code>	Optional character vector of parameter names or integer vector of positions indicating which parameters should be shown in the summary. By default, all fitted parameters are shown.
<code>par.only</code>	Logical. If <code>TRUE</code> , return only the method name and selected parameter values. This is mainly useful when summarising several <code>calibrar.results</code> objects side by side.
<code>digits</code>	Number of significant digits to use when printing summary tables or parameter values.

Format

A list, usually with the following components:

par Best parameter values found, returned with the same structure as the input `par` supplied to `calibrate()`.

value Objective-function value at par.

counts Named integer vector giving the number of function and gradient evaluations recorded for the last calibration phase.

convergence Integer convergence code returned by the optimiser used in the final phase.

message Character string with additional information returned by the optimiser.

method Name of the optimisation method used in the calibration.

fn Objective function used for the calibration.

active Logical vector indicating which parameters were active (estimated) in the full calibration problem.

elapsed Elapsed time for the complete calibration run.

trace A list containing tracing information from the last phase and additional run-level metadata. Depending on the control settings and the optimiser used, this may include per-phase outputs, generation traces, partial objective values, and other diagnostic information.

Details

A `calibrar.results` object represents the final output of a complete calibration workflow, potentially including multiple calibration phases.

The most useful accessors are:

- `print()` for a compact console summary;
- `summary()` for a more explicit summary, including selected parameters;
- `coef()` to extract the fitted parameter values;
- `predict()` to evaluate the fitted parameter set with the underlying model function when this is available; and
- `dim()` to report the number of parameters in the fitted solution.

The `predict()` method is currently only available when the objective function was built with `calibration_objFn()`, because it relies on metadata attached to that function.

Value

`coef.calibrar.results()` returns the fitted parameter values, with the same structure as the `par` component of the object.

`predict.calibrar.results()` returns the model outputs produced by the fitted parameter set. When `replicates > 1`, it returns the mean output across replicates and stores the full replicate-level outputs in the `.replicates` component of the returned object.

`plot.calibrar.results()` returns `NULL`, invisibly.

`summary.calibrar.results()` returns either:

- an object of class `summary.calibrar.results` when called on a single object; or
- a `data.frame` with one row per result object when several `calibrar.results` objects are supplied through `...`

The `print` method formats these summaries for console display.

`dim.calibrar.results()` returns a single integer.

See Also

[calibrate\(\)](#), [calibration_objFn\(\)](#), [summary\(\)](#), [coef\(\)](#), [predict\(\)](#)

calibrar_demo

Create small demonstration calibration projects

Description

Create self-contained example projects that can be used to learn the calibrar workflow, test installation, or prototype calibration code. Each demo writes a small set of files to disk and returns the information needed to read the setup table, load the observed data, define the model interface, and run a calibration.

Usage

```
calibrar_demo(path = NULL, model = NULL, ...)

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

Arguments

path	Directory where the demo project should be created. If NULL, the current working directory is used.
model	Name of the demo model to generate. Supported values currently include "PoissonMixedModel", "PredatorPrey", "SIR", and "IBMLotkaVolterra".
...	Further arguments passed to print() .
x	A calibrar_demo object returned by calibrar_demo() .

Details

The generated project is meant to be a compact, runnable example of the calibrar workflow rather than a realistic scientific application. The returned object typically contains enough information to proceed through the standard helper functions:

- [calibration_setup\(\)](#) to read the setup table,
- [calibration_data\(\)](#) to load the observed data,
- [calibration_objFn\(\)](#) to build the objective function, and
- [calibrate\(\)](#) or [optim2\(\)](#) to run the optimisation.

The exact files created depend on model, but they usually include a calibration settings table, one or more observation files, and any required forcing or master data files.

The currently available demos are:

"PoissonMixedModel" A Poisson autoregressive mixed model for the dynamics of a population observed at several sites over time.

"PredatorPrey" A deterministic Lotka–Volterra predator–prey system solved numerically.

"SIR" A simple susceptible–infected–recovered epidemic model solved numerically.

"IBMLotkaVolterra" A stochastic individual-based predator–prey example.

Value

An object of class `calibrar.demo`, which also inherits from `calibrar.results()`. The returned list contains the demo project path, the true parameter values used to generate the synthetic data, the path to the calibration setup file, suggested starting values and bounds, phase information, model-specific constants, and a precomputed objective value at the true parameter set.

See Also

`calibration_setup()`, `calibration_data()`, `calibration_objFn()`, `calibrate()`, `summary.calibrar.results()`, `print.calibrar.results()`

Examples

```
tmp = tempdir()
demo = calibrar_demo(path = tmp, model = "PoissonMixedModel", T = 8, L = 3)
print(demo)

setup = calibration_setup(demo$setup)
observed = calibration_data(setup, path = demo$path)
```

calibrate

Sequential parameter estimation for the calibration of complex models

Description

`calibrate()` minimises an objective function `fn` for the calibration of complex (possibly expensive and stochastic) models. It supports sequential calibration in multiple phases (progressively activating parameters), replicated evaluations for stochastic objectives, restartable runs, and parallel execution patterns for computationally intensive models.

Usage

```
calibrate(
  par,
  fn,
  gr,
  ...,
  method,
  lower,
  upper,
  phases,
  control,
```

```

    hessian,
    replicates,
    parallel
)

## Default S3 method:
calibrate(
  par,
  fn,
  gr = NULL,
  ...,
  method = NULL,
  lower = NULL,
  upper = NULL,
  phases = NULL,
  control = list(),
  hessian = FALSE,
  replicates = 1,
  parallel = FALSE
)

```

Arguments

par	A numeric vector or list. The length of par defines the number of parameters to be estimated (i.e., the dimension of the problem).
fn	The objective function to be minimised. It should accept a parameter vector (or list, depending on the wrapper) as first argument and return either a scalar value (single-objective) or a numeric vector (multi-objective).
gr	A function computing the gradient of fn. If NULL, a numerical approximation is used. Alternatively, a character string can specify the numerical gradient scheme: "central", "forward" (default), "backward", or "richardson".
...	Additional arguments passed to fn and gr.
method	Optimisation method(s) to be used. Can be a single method name or a vector of method names (e.g., one per phase). If NULL, a default is chosen based on replicates (see Details).
lower	Lower bounds for parameters. One value or a vector of the same length as par. NA is treated as $-\text{Inf}$. Default is unconstrained.
upper	Upper bounds for parameters. One value or a vector of the same length as par. NA is treated as Inf . Default is unconstrained.
phases	Optional integer vector of the same length as par, indicating the phase at which each parameter becomes active. If omitted, all parameters are active in a single phase.
control	A list of control options. Common options include ncores, run, master, verbose, REPORT, restart.file, gradient, and gr.method. Additional solver-specific options may be passed through to the underlying optimiser.
hessian	Logical. Should a numerically differentiated Hessian matrix be returned? Currently not implemented.

<code>replicates</code>	Integer or integer vector controlling the number of replicate evaluations of <code>fn</code> per phase. The default is 1 (deterministic objective).
<code>parallel</code>	Logical. Enable parallel computation (e.g., for replicated evaluations and/or numerical gradients) using up to <code>control\$ncores</code> cores.

Details

Sequential phases. When `phases` is provided, parameters are activated progressively across phases. In phase k , parameters with `phases` $\leq k$ are active and estimated, while all others are held fixed. If `phases` is omitted, all parameters are estimated in a single phase.

Replicated evaluations. The argument `replicates` controls the number of replicate evaluations of `fn` per phase. It can be a single integer (applied to all phases) or a vector of length equal to the number of phases. Replication is useful for stochastic models to reduce Monte Carlo noise and/or to target robust parameter sets.

Default method selection. If `method` is not provided, `calibrate()` defaults to "Rvmmmin" when `all(replicates == 1)` (deterministic objective), and to "AHR-ES" otherwise (replicated/stochastic objective).

Multi-objective outputs. The objective function `fn` may return a scalar (single-objective) or a numeric vector of objective components (multi-objective). Multi-objective optimisation is currently supported only by methods in `multiMethods` (e.g., "AHR-ES"). For single-objective methods, `fn` must be scalar; alternatively, objective components can be aggregated into a scalar by defining an aggregated objective in `calibration_objFn(aggregate = TRUE)`.

Aggregation. When `fn` is created via `calibration_objFn()`, metadata such as the number of components (`nvar`) and component weights (`weights`) can be stored as attributes. If `aggregate = TRUE`, `fn` is scalar and can be used with any method. If `aggregate = FALSE` (vector output), only multi-objective methods can use it directly.

Parallel execution and run directories. If `parallel = TRUE`, `calibrate()` may distribute replicate evaluations and/or finite-difference gradient computations across multiple cores. The number of cores is controlled by `control$ncores`. For file-based or externally executed models, `control$master` and `control$run` can be used to manage a master/template directory and per-run working directories (created if needed).

Restart. Partial results can be written and used to resume a calibration via `control$restart.file`. Restart functionality is currently available only for "AHR-ES", "Rvmmmin", and "hjn".

Value

An object of class "calibrar.results" with components:

par Best parameter values found (returned with the same structure as input `par`).

value Objective value at `par`.

counts Number of calls to `fn` and `gr` (where applicable).

convergence Convergence code returned by the underlying optimiser.

message Additional information returned by the optimiser, if any.

method The optimisation method used in the final phase.

fn The objective function.

active Logical vector indicating which parameters were active (estimated).

elapsed Elapsed time for the full calibration run.

trace Tracing information and per-phase outputs (when available).

Choosing an optimisation method

For smooth deterministic objectives, gradient-based methods such as "Rvmmin", "L-BFGS-B", "LBFGSB3", or "nllminb" are often efficient (with box constraints when needed). For noisy or stochastic objectives (typically when replicates > 1), heuristic/global methods such as "AHR-ES" are generally more appropriate. Not all methods are directly comparable (local vs.\ global, deterministic vs.\ stochastic, scalar vs.\ vector-valued objectives); method choice should reflect the structure of fn and the presence of constraints.

Notes

"SANN" is included for compatibility with `stats::optim()`, but it is highly sensitive to tuning and often performs poorly on continuous problems under default settings. For stochastic objectives, "AHR-ES" is the intended default within `calibrate()`.

Author(s)

Ricardo Oliveros-Ramos

See Also

[calibration_setup](#), [calibration_data](#), [calibration_objFn](#)

Other optimisers: [ahres\(\)](#), [optim2\(\)](#), [optimh\(\)](#)

Examples

```
calibrate(par=rep(NA, 5), fn=sphereN)
## Not run:
calibrate(par=rep(NA, 5), fn=sphereN, replicates=3)
calibrate(par=rep(0.5, 5), fn=sphereN, replicates=3, lower=-5, upper=5)
calibrate(par=rep(0.5, 5), fn=sphereN, replicates=3, lower=-5, upper=5, phases=c(1,1,1,2,3))
calibrate(par=rep(0.5, 5), fn=sphereN, replicates=c(1,1,4), lower=-5, upper=5, phases=c(1,1,1,2,3))

## End(Not run)
```

calibration_data

Read observed data defined in a calibration setup table

Description

Build the observed object used by [calibration_objFn\(\)](#) from a setup table created with [calibration_setup\(\)](#). Each active row in the setup table points to a file containing observed data, and this function reads those files into a named list whose elements correspond to the calibration variables.

Usage

```
calibration_data(setup, path = ".", file = NULL, verbose = TRUE, ...)
```

Arguments

setup	A calibration setup table, usually created with <code>calibration_setup()</code> .
path	Base directory used to resolve relative paths from <code>setup\$file</code> . Each file is looked up with <code>base::file.path()</code> using path and the value stored in the corresponding file row.
file	Optional path to save the resulting observed object with <code>base::saveRDS()</code> . This is useful when observed data are expensive to read repeatedly.
verbose	Logical. If TRUE, print progress messages describing which variables were loaded.
...	Additional arguments passed to <code>utils::read.csv()</code> when reading text files.

Details

Rows with `use_data = FALSE` are kept in the output as NA. This makes it possible to include penalty-only components in the same calibration workflow: such rows still participate in `calibration_objFn()` provided the model returns a simulated component with the same name.

Only rows satisfying both `setup$calibrate` and `setup$use_data` are actually read from disk. All other rows are represented as NA in the returned list.

File reading currently depends on file type:

text files (`.csv`, `.txt`) Read with `utils::read.csv()`. If `varid` is supplied, that column is extracted and returned as a numeric vector. Otherwise the full table is converted to a numeric matrix after dropping the first `col_skip` columns, when applicable.

R objects (`.rds`) Read with `base::readRDS()` and returned as-is.

netCDF (`.nc`) Not currently supported by the implementation.

The interpretation of `varid`, `col_skip`, and `nrows` is controlled by the corresponding columns in `setup`, after defaults have been filled by `calibration_setup()`.

Value

A named list with one element per row of `setup`. Element names are taken from `setup$variable`.

Depending on the corresponding `setup` row, each element is typically one of:

- a numeric vector, when `varid` selects a single column from a text file;
- a numeric matrix, when a full table is read from a text file;
- an arbitrary R object read from an `.rds` file;
- NA, when `use_data = FALSE` or `calibrate = FALSE`.

Expected alignment with model outputs

The names of the returned list are used later by `calibration_objFn()` to match observed and simulated quantities. For each active calibration row, `setup$variable` must therefore match the name of the corresponding element in the list returned by the model function.

Saving and reusing observed data

When `file` is supplied, the created list is also saved as an `.rds` file. This can be helpful in iterative calibration workflows where the setup table is stable but reading the original data files repeatedly is inconvenient.

See Also

`calibration_setup()`, `calibration_objFn()`, `objFn()`

Examples

```
## Not run:
demo = calibrar_demo(model = "PredatorPrey")
setup = calibration_setup(file = demo$setup)
observed = calibration_data(setup = setup, path = demo$path)

names(observed)
str(observed)

## End(Not run)
```

<code>calibration_objFn</code>	<i>Create a calibration objective function from a model, setup table, and data</i>
--------------------------------	--

Description

Build an objective function that runs a model, compares its outputs with the observed data prepared by `calibration_data()`, and returns either a vector of partial objective values or a single aggregated objective value.

Usage

```
calibration_objFn(model, setup, observed, aggFn = NULL, aggregate = FALSE, ...)
```

Arguments

<code>model</code>	Function that runs the model for a candidate parameter set. Its first argument must be <code>par</code> . The function must return a named list of simulated outputs whose names match <code>setup\$variable</code> for all active calibration rows.
<code>setup</code>	A calibration setup table, usually created with <code>calibration_setup()</code> .

observed	Observed data as returned by <code>calibration_data()</code> . This is typically a named list whose elements correspond to <code>setup\$variable</code> .
aggFn	Function used to aggregate partial objective values when <code>aggregate = TRUE</code> . The default is an internal weighted sum. The function must accept arguments <code>x</code> (partial objective values) and <code>w</code> (weights).
aggregate	Logical. If <code>FALSE</code> (the default), the returned function produces one objective value per active calibration row. If <code>TRUE</code> , those partial values are combined into a single scalar using <code>aggFn</code> and the weights stored in <code>setup</code> .
...	Additional arguments passed to <code>model</code> each time the returned objective function is evaluated.

Details

This is the main bridge between a model-specific simulation function and the optimisation tools in `calibrar`. The resulting function can be passed directly to `calibrate()` and related optimisers.

For each row where `setup$calibrate` is `TRUE`, the function looks for a variable named `setup$variable` in both `observed` and the list returned by `model`. The comparison itself is delegated to the objective function named in `setup$type`, using the same convention as `objFn()`.

This means that the structure of the objects returned by `model` must already be compatible with the corresponding objective function. For example:

- pointwise criteria such as `"norm2"`, `"lnorm2"`, or `"pois"` usually expect vectors, matrices, or arrays with shapes matching the observed data;
- `"multinom"` expects matrix inputs;
- penalty terms such as `"normp"` may ignore `observed`, but still require the model output to contain the corresponding named element.

If `model` returns `NULL`, the generated objective function returns `NULL` after emitting a diagnostic message.

Value

A function of `par` suitable for optimisation.

When evaluated, the returned function:

1. runs `model(par, ...)`;
2. extracts the simulated outputs;
3. compares them with `observed` according to `setup$type`;
4. returns either a named numeric vector of partial objective values or a single aggregated numeric value.

The returned function has class `"objFn"` and carries metadata attributes used internally by `calibrate()`, including the number of objective components, their weights, and the names of the active variables.

Aggregated versus vector objectives

With `aggregate = FALSE`, the returned objective is multi-component: it produces one value per active calibration row. This can be useful for methods that exploit vector-valued objectives.

With `aggregate = TRUE`, the returned objective is scalar and can be used with optimisers that require a single value. Aggregation is performed with `aggFn(x = partial_values, w = setup$weight[setup$calibrate])`.

Expected model interface

In typical use, the model function has the form:

```
function(par, ...) -> named list
```

where `par` may be a numeric vector or a list, depending on how the optimisation is configured. The returned list should expose one element per calibration target, with names that match `setup$variable` exactly.

Workflow

The usual sequence is:

1. define a model-running function;
2. read the setup table with `calibration_setup()`;
3. read observed data with `calibration_data()`;
4. create the objective function with `calibration_objFn()`;
5. call `calibrate()` with the resulting function.

See Also

`calibration_setup()`, `calibration_data()`, `calibrate()`, `objFn()`

Examples

```
## Not run:
demo = calibrar_demo(model = "PredatorPrey")
setup = calibration_setup(file = demo$setup)
observed = calibration_data(setup = setup, path = demo$path)

run_model = calibrar:::PredatorPreyModel

obj_vec = calibration_objFn(
  model = run_model,
  setup = setup,
  observed = observed,
  T = demo$T,
  aggregate = FALSE
)

obj_scalar = calibration_objFn(
  model = run_model,
  setup = setup,
  observed = observed,
```

```

    T = demo$T,
    aggregate = TRUE
  )

  obj_vec(demo$guess)
  obj_scalar(demo$guess)

## End(Not run)

```

calibration_setup	<i>Read and validate a calibration setup table</i>
-------------------	--

Description

Read a calibration settings table from a text file and coerce it to the structure expected by `calibration_data()` and `calibration_objFn()`. The setup table defines which model outputs are compared with observations, which objective function is used for each output, whether a row contributes to the calibration, and how each component is weighted in the final objective.

Usage

```
calibration_setup(file, control = list(), ...)
```

Arguments

<code>file</code>	Path to the calibration setup file, typically a .csv file.
<code>control</code>	Named list of defaults used when optional columns are missing from the setup table. Currently recognised entries are: <code>col_skip</code> Default number of leading columns to discard when a text file is read as a full table by <code>calibration_data()</code>. Defaults to 2. <code>nrows</code> Default number of rows to read from each data file. Defaults to NA, which is interpreted downstream as “read all rows”.
<code>...</code>	Additional arguments passed to <code>utils::read.csv()</code> when reading file.

Details

The function is a validated wrapper around `utils::read.csv()`. It checks required column names, coerces standard column types, fills a few optional columns from `control`, and derives weight from `cv` when needed.

The setup table may contain the following columns:

<code>variable</code>	Name of the calibration component. This name must match the corresponding element returned by the model function supplied to <code>calibration_objFn()</code> .
<code>type</code>	Name of the objective function used for that component. In current workflows this is usually the name of a function accepted by <code>objFn()</code> , such as “norm2”, “lnorm2”, “pois”, “multinom”, “normp”, or “penalty”.

calibrate Logical flag indicating whether the component is active in the calibration. Only rows with `calibrate = TRUE` are included in the objective function returned by `calibration_objFn()`.

weight Positive weight used when partial objective values are aggregated. If `weight` is missing, or contains NA, it is derived from `cv` as $1 / (2 * cv^2)$ for the corresponding rows.

cv Coefficient of variation used to derive `weight` when an explicit weight is not supplied. At least one of `weight` or `cv` must be available in the table.

use_data Logical flag indicating whether an observed dataset must be read for that row. This is typically FALSE for pure penalties such as "normp", where the objective depends only on simulated values.

file Path to the observed data file for that component. Paths are interpreted relative to the `path` argument of `calibration_data()`. This column is still required when `use_data = FALSE`, although the file will not be read in that case.

varid Optional identifier used by `calibration_data()` when extracting a single column from a text file. If omitted, a full numeric table is read instead.

col_skip Optional number of leading columns to discard when a text file is read as a full table. Missing values are replaced by `control$col_skip`.

nrows Optional number of rows to read from each observed-data file. Missing values are replaced by `control$nrows`.

Required columns are `variable`, `type`, `calibrate`, `use_data`, and `file`. In addition, at least one of `weight` or `cv` must be supplied.

Value

A `data.frame` describing the calibration targets. The returned table is suitable for direct use in `calibration_data()` and `calibration_objFn()`. Standard columns are coerced to the expected types and missing optional columns are added where needed.

Weight handling

Aggregation weights are stored in the returned table regardless of whether the objective is later used in vector or scalar form.

If the table does not contain a `weight` column, weights are computed from `cv` as $1 / (2 * cv^2)$. If `weight` exists but contains missing values, those entries are filled using the same formula. All final weights must be strictly positive.

Relationship with the rest of the workflow

A typical file-based calibration workflow is:

1. read the setup table with `calibration_setup()`;
2. load observed data with `calibration_data()`;
3. build the objective function with `calibration_objFn()`;
4. pass the resulting function to `calibrate()` or another optimiser.

See Also

`calibration_data()`, `calibration_objFn()`, `calibrate()`, `objFn()`

Examples

```
## Not run:
demo = calibrar_demo(model = "PredatorPrey")
setup = calibration_setup(file = demo$setup)
setup

## End(Not run)
```

gaussian_kernel

Discretise a bivariate Gaussian kernel on a grid

Description

Evaluate a two-dimensional Gaussian density on a regular grid defined by lower and upper bounds in each dimension.

Usage

```
gaussian_kernel(par, lower, upper, n = 10, checkSymmetry = TRUE, ...)
```

Arguments

<code>par</code>	A list containing at least <code>mean</code> and <code>sigma</code> , where <code>mean</code> is a numeric vector of length two and <code>sigma</code> is a 2 x 2 covariance matrix.
<code>lower</code>	Numeric vector of length two giving the lower bounds of the grid.
<code>upper</code>	Numeric vector of length two giving the upper bounds of the grid.
<code>n</code>	Number of grid cells per dimension. This can be a scalar or a length-two numeric vector.
<code>checkSymmetry</code>	Logical. Passed to <code>mvtnorm::dmvnorm()</code> when checking the covariance matrix.
<code>...</code>	Further arguments reserved for future extensions.

Details

This helper is convenient when a smooth bivariate dispersal or movement kernel needs to be represented on a finite grid. The function uses `mvtnorm::dmvnorm()` internally and returns the density values without normalising them to sum to one over the discretised domain.

Value

A list with components `x`, `y`, and `z`, suitable for contour or image plotting. `x` and `y` contain the grid coordinates, and `z` contains the corresponding matrix of Gaussian density values.

See Also

`mvtnorm::dmvnorm()`

Examples

```
ker = gaussian_kernel(
  par = list(mean = c(0, 0), sigma = diag(2)),
  lower = c(-2, -2),
  upper = c(2, 2),
  n = c(10, 10)
)
str(ker)
```

gradient

Numerical gradient approximation

Description

`gradient()` computes a numerical approximation to the gradient of a scalar objective function at a given parameter vector. It provides several finite-difference schemes and can distribute the underlying function evaluations across a registered foreach backend when `parallel = TRUE`.

Usage

```
gradient(fn, x, method, control, parallel, ...)

## Default S3 method:
gradient(fn, x, method = NULL, control = list(), parallel = FALSE, ...)
```

Arguments

<code>fn</code>	Function whose gradient is to be approximated.
<code>x</code>	Numeric vector giving the point at which the gradient is evaluated.
<code>method</code>	Numerical differentiation scheme. If <code>NULL</code> , the default method used by <code>gradient.default()</code> is "forward".
<code>control</code>	A list of control arguments governing step-size construction and, for Richardson extrapolation, the extrapolation sequence.
<code>parallel</code>	Logical. Should the function evaluations needed for the numerical derivative be carried out in parallel via <code>foreach</code> ?
<code>...</code>	Additional arguments passed to <code>fn</code> .

Details

The currently implemented methods are:

- "forward" (default): one-sided forward finite differences.
- "backward": one-sided backward finite differences.
- "central": central finite differences.
- "richardson": central differences with Richardson extrapolation.

Step sizes are controlled through `control`. Common entries are:

- `step_method`: how the initial step size is computed. Implemented values are "simple", "optextras", and "start".
- `eps`: base perturbation size.
- `zero.tol`: threshold used to decide when a parameter value is treated as effectively zero for step-size construction.
- `d`: starting fraction of `x` used by `step_method = "start"`.
- `r` and `v`: Richardson iteration depth and reduction factor for `method = "richardson"`.

When `parallel = TRUE`, a suitable `foreach` backend must already be registered by the user. This function does not itself create or manage a parallel backend.

Value

A numeric vector of the same length as `x` containing the estimated gradient of `fn` at `x`.

See Also

`optim2()`, `optimh()`, `calibrate()`

Examples

```
gradient(fn = function(x) sum(x^3), x = c(1, 2), method = "forward")
gradient(fn = function(x) sum(x^3), x = c(1, 2), method = "central")
```

objFn

Objective function between observed and simulated data

Description

Compute an objective value (error, negative log-likelihood, and/or penalty) given observed data (`obs`) and model outputs (`sim`). This function is a thin dispatcher: it resolves `FUN` via `match.fun()` and returns `FUN(obs = obs, sim = sim, ...)`.

Usage

```
objFn(obs, sim, FUN, ...)

fitness(obs, sim, FUN, ...)
```

Arguments

obs	Observed data as expected by FUN. Typically a numeric vector, matrix, or array. Missing values (NA) are permitted; see <i>Missing values</i> below.
sim	Simulated data matching obs, in the sense expected by FUN. For most pointwise criteria (e.g. norm2, lnorm2, pois), sim should have the same shape as obs. For composition criteria (e.g. multinom), obs and sim are expected to be matrices with rows representing samples and columns representing classes.
FUN	Objective function to apply. Can be: • a function, e.g. FUN = norm2; • or a character string naming a function, e.g. FUN = "norm2". The function must accept arguments named obs and sim (additional arguments may be supplied via ...) and must return a single numeric scalar.
...	Additional arguments forwarded to FUN.

Details

The returned value is intended to be **minimised** by an optimiser: lower values indicate a better match between sim and obs (or a lower penalty).

Value

A numeric scalar: the value of FUN(obs = obs, sim = sim, ...). By convention this is an objective to be **minimised**.

Conventions and expectations

- **Minimisation:** all provided objectives are formulated so that lower values indicate a better fit (or weaker penalty).
- **Shapes:** objFn() does not reshape or recycle data. It is the caller's responsibility to supply obs and sim in a compatible form for the chosen FUN.
- **Scalar output:** FUN should return a length-one numeric value. Returning vectors is not supported by objFn() itself (although higher level workflows may aggregate vector-valued objectives elsewhere).

Missing values

Most built-in objectives stop if all(is.na(obs)). Otherwise, they ignore missing values using na.rm = TRUE inside sum()/mean(). This means:

- partial NAs in obs are ignored;
- NAs in sim will also be dropped from sums when na.rm = TRUE (possibly masking simulation failures if not checked upstream).

Numerical constraints and stability

Some objectives impose additional constraints:

- **Poisson** (pois): uses $\log(\text{sim})$; sim must be strictly positive wherever $\text{obs} > 0$ (otherwise $-\text{Inf}$ may occur). Consider flooring the simulated intensity, e.g. `sim <- pmax(sim, 1e-12)`.
- **Log-scale** (lnorm2, lnorm3, lnorm4, lnorm4b): apply $\log(x + \text{tiny})$; both $\text{obs} + \text{tiny}$ and $\text{sim} + \text{tiny}$ must be positive. If your data may contain zeros, `tiny` should be chosen accordingly.
- **Compositions** (multinom): expects matrix inputs and uses row sums to convert counts/weights into proportions. See details below.

Supported built-in objective functions

The following functions are available with their current behaviour. All are formulated as objectives to be minimised.

norm2 Sum of squared errors on the original scale:

$$\sum (\text{obs} - \text{sim})^2$$

Typical use: continuous observations with approximately additive errors.

lnorm2 Sum of squared errors on the log scale:

$$\sum (\log(\text{obs} + \text{tiny}) - \log(\text{sim} + \text{tiny}))^2$$

Arguments: `tiny` (default $1\text{e-}2$), added before the log to avoid $\log(0)$. Typical use: positive-valued data with multiplicative (lognormal) error structure.

lnorm3 Log-scale squared error with an estimated multiplicative scaling factor q . Internally:

- compute element-wise ratios `ratio <- obs/sim`;
- set NaN ratios to NA;
- estimate `q <- mean(ratio, na.rm = TRUE)`;
- return $\sum (\log(\text{obs} + \text{tiny}) - \log(\text{sim} + \text{tiny}) - \log(q))^2$.

Typical use: positive data where an overall multiplicative bias (scale mismatch) is expected and should not be fully penalised.

lnorm4 / lnorm4b Extensions of lnorm3 that add a penalty term to discourage extreme values of the scaling factor q (or extreme per-observation ratios). They rely on the helper `rangeq()`:

- compute `ratio <- obs/sim`, estimate `q <- mean(ratio, na.rm=TRUE)`;
- compute a penalty using parameters `b` and `c`:

$$\text{pen} = n \cdot (\max(|\log_2(q)|, b)^c - b^c)$$

when `dump = TRUE` (used by lnorm4), or

$$\text{pen} = \sum (\max(|\log_2(\text{ratio})|, b)^c - b^c)$$

when `dump = FALSE` (used by lnorm4b), where n is the number of non-missing ratios.

- add the penalty to the lnorm3-style objective.

Arguments: `tiny`, `b` (default 1), `c` (default 2). Typical use: log-scale fitting where scale drift must be controlled.

`pois` Poisson negative log-likelihood (up to constants):

$$-\sum (obs \log(sim) - sim)$$

Typical use: counts (or count-like rates) with Poisson observation error. Note: `sim` must be positive where `obs > 0`.

`multinom` A composition (multinomial-like) objective operating on matrices. Inputs are expected as `obs` and `sim` matrices with:

- rows = samples (e.g. time steps, hauls, sites),
- columns = classes (e.g. age/size bins, categories).

Internal steps (high-level):

- Let A be the number of classes ($A \leftarrow \text{ncol}(\text{sim})$).
- Rows of `sim` that are all zeros (excluding rows that are all NA) are replaced by 1 on that row (interpreted as a uniform prior).
- Row sums are used to compute proportions: $Psim = sim / \text{sum}(sim)$, $Pobs = obs / \text{sum}(obs)$ (row-wise).
- Rows with $\text{sum}(sim) == 0$ are set to NA for numerical convenience.
- Rows with $\text{sum}(obs) == 0$ are set to NA (interpreted as “no proportion data available”).
- A variance term `sigma2` and a small stabiliser `tiny` are used to define an objective that penalises discrepancies between `Pobs` and `Psim`.

Arguments: `size` (default 20) and `tiny` (default $1e-3$). Interpretation: `size` plays the role of an effective sample size (larger values typically increase the weight of the composition fit).

`normp / re` Pure penalty on simulated values:

$$\sum sim^2$$

This ignores `obs` and can be used as a regulariser, or when `sim` represents a residual vector or deviates already computed upstream. `re` is an alias of `normp`.

`penalty` Scaled quadratic penalty:

$$n \cdot \text{mean}(sim^2)$$

Arguments: `n` (default 100). This assumes a fixed sample size and can be used to put the penalty on a comparable scale across datasets.

Writing your own objective function

You can supply any custom function via `FUN` provided it:

- accepts arguments named `obs` and `sim` (plus optional ...);
- returns a length-one numeric scalar to be minimised;
- defines its own parameter checks and missing-value policy.

Note

`fitness()` is provided as an exported alias of `objFn()`. Both names refer to the same underlying function, but `objFn()` is the preferred name in the current user-facing documentation.

See Also

match.fun

Examples

```
## Basic squared-error objective
obs <- c(1, 2, 3, NA, 5)
sim <- c(1.2, 1.9, 2.7, 4.0, 5.1)
objFn(obs, sim, FUN = "norm2")

## Log-scale objective (positive data)
obs <- c(0.1, 1, 10)
sim <- c(0.2, 0.9, 11)
objFn(obs, sim, FUN = lnorm2, tiny = 1e-2)

## Poisson objective (counts) with flooring for numerical safety
obs <- c(0, 3, 10, 2)
sim <- c(0, 2.5, 9.8, 1.9)
sim <- pmax(sim, 1e-12)
objFn(obs, sim, FUN = "pois")

## Composition objective (matrices: rows = samples, cols = classes)
obs <- rbind(c(10, 5, 0),
             c( 0, 0, 0), # interpreted as "no composition data"
             c( 2, 1, 7))
sim <- rbind(c( 9, 6, 1),
             c( 0, 0, 0), # replaced internally by sim+1 on that row
             c( 1, 2, 6))
objFn(obs, sim, FUN = "multinom", size = 20, tiny = 1e-3)

## Custom objective function
my_obj <- function(obs, sim, ...) {
  if (all(is.na(obs))) stop("All observed values are NA.")
  sum(abs(obs - sim), na.rm = TRUE) # L1 error
}
objFn(obs = c(1, 2, NA), sim = c(1.1, 1.7, 3), FUN = my_obj)
```

optim2

Unified optimisation interface with structured parameters and parallel numerical gradients

Description

optim2() provides a unified interface to multiple deterministic and stochastic optimisation methods, combining the algorithms available through stats::optim() with a small set of additional solvers accessed via **calibrar**'s internal dispatcher.

Usage

```

optim2(
  par,
  fn,
  gr = NULL,
  ...,
  method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN", "Brent", "nlm", "nlminb",
    "Rcgmin", "Rvmin", "hjn", "spg", "LBFGSB3", "AHR-ES"),
  lower = -Inf,
  upper = +Inf,
  active = NULL,
  control = list(),
  hessian = FALSE,
  parallel = FALSE
)

```

Arguments

par	A numeric vector or list. The length of par defines the number of parameters to be estimated (i.e., the dimension of the problem).
fn	The objective function to be minimised. It should accept a parameter vector (or list, depending on the wrapper) as first argument and return either a scalar value (single-objective) or a numeric vector (multi-objective).
gr	A function computing the gradient of fn. If NULL, a numerical approximation is used. Alternatively, a character string can specify the numerical gradient scheme: "central", "forward" (default), "backward", or "richardson".
...	Additional arguments passed to fn and gr.
method	Optimisation method(s) to be used. Can be a single method name or a vector of method names (e.g., one per phase). If NULL, a default is chosen based on replicates (see Details).
lower	Lower bounds for parameters. One value or a vector of the same length as par. NA is treated as -Inf. Default is unconstrained.
upper	Upper bounds for parameters. One value or a vector of the same length as par. NA is treated as Inf. Default is unconstrained.
active	Boolean vector of the same length as par, indicating if the parameter is used in the optimisation (TRUE) or held at a fixed value (FALSE).
control	A list of control options. Common options include ncores, run, master, verbose, REPORT, restart.file, gradient, and gr.method. Additional solver-specific options may be passed through to the underlying optimiser.
hessian	Logical. Should a numerically differentiated Hessian matrix be returned? Currently not implemented.
parallel	Logical. Enable parallel computation (e.g., for replicated evaluations and/or numerical gradients) using up to control\$ncores cores.

Details

Methods. The current selection includes (i) base `stats::optim()` methods, (ii) additional gradient-based solvers from external packages, and (iii) heuristic/global methods.

Comparability. Not all methods are directly comparable: some are deterministic local optimisers (e.g., quasi-Newton), others are derivative-free local searches, and others are stochastic/global heuristics. Method choice should reflect the objective function (smooth vs. non-smooth, deterministic vs. stochastic) and the presence of constraints.

Control arguments. `optim2()` standardises a set of common control arguments (e.g., iteration limits, tolerances, tracing, and scaling) where supported, while still allowing method-specific control parameters to be passed through to the underlying solver.

Parallel numerical gradients. When analytic gradients are not provided, `optim2()` can compute finite-difference numerical gradients and distribute function evaluations across multiple cores, which can substantially reduce wall-clock time for expensive objective functions.

Value

A list with components:

par The best set of parameters found.

value The value of `fn` corresponding to `par`.

counts A two-element integer vector giving the number of calls to `fn` and `gr` respectively. This excludes those calls needed to compute the Hessian, if requested, and any calls to `fn` to compute a finite-difference approximation to the gradient.

convergence An integer code. 0 indicates successful completion.

message A character string giving any additional information returned by the optimizer, or `NULL`.

hessian Only if argument `hessian` is `TRUE`. A symmetric matrix giving an estimate of the Hessian at the solution found. Note that this is the Hessian of the unconstrained problem even if the box constraints are active.

Choosing an optimisation method

For smooth deterministic objectives, quasi-Newton methods (e.g., "BFGS" or "L-BFGS-B" with box constraints) are often efficient when gradients are available or can be reliably approximated. For box-constrained problems, consider "L-BFGS-B" or the extended bounded solvers (e.g., "Rvmmin", "spg"). For noisy or stochastic objectives, heuristic/global methods (e.g., "AHR-ES") may be more appropriate.

Notes

"SANN" is included for compatibility with `stats::optim()`, but it is highly sensitive to tuning and often performs poorly on continuous problems under default settings. For noisy or rugged objective functions, "AHR-ES" is generally the recommended heuristic alternative within `optim2()`.

Author(s)

Ricardo Oliveros-Ramos

See Also

[optim](#), [nlm](#), [nlminb](#)

Other optimisers: [ahres\(\)](#), [calibrate\(\)](#), [optimh\(\)](#)

Examples

```
optim2(par=rep(NA, 5), fn=sphereN)
```

optimh	<i>General-purpose optimisation with heuristic and derivative-free methods</i>
--------	--

Description

`optimh()` provides a unified interface to the heuristic and derivative-free optimisation methods available through **calibrar**. It is intended for objective functions that are expensive, noisy, non-smooth, stochastic, or otherwise difficult to handle with standard gradient-based optimisers.

Usage

```
optimh(
  par,
  fn,
  gr = NULL,
  ...,
  method = c("AHR-ES", "Nelder-Mead", "SANN", "hjn", "bobyqa", "CMA-ES", "genSA", "DE",
    "soma", "genoud", "PSO", "hybridPSO", "mads", "hjk", "hjkb", "nmk", "nmkb"),
  lower = -Inf,
  upper = +Inf,
  active = NULL,
  control = list(),
  hessian = FALSE,
  parallel = FALSE
)
```

Arguments

<code>par</code>	A numeric vector or list. The length of <code>par</code> defines the number of parameters to be estimated (i.e., the dimension of the problem).
<code>fn</code>	The objective function to be minimised. It should accept a parameter vector (or list, depending on the wrapper) as first argument and return either a scalar value (single-objective) or a numeric vector (multi-objective).
<code>gr</code>	Function computing the gradient of <code>fn</code> . It is included for API consistency with optim2() and calibrate() , but is ignored by most methods available through <code>optimh()</code> .
<code>...</code>	Additional arguments passed to <code>fn</code> and <code>gr</code> .

method	Optimisation method to use. Available choices are "AHR-ES", "Nelder-Mead", "SANN", "hjn", "bobyqa", "CMA-ES", "genSA", "DE", "soma", "genoud", "PSO", "hybridPSO", "mads", "hjk", "hjkb", "nmk", and "nmkb".
lower	Lower bounds for parameters. One value or a vector of the same length as par. NA is treated as $-\text{Inf}$. Default is unconstrained.
upper	Upper bounds for parameters. One value or a vector of the same length as par. NA is treated as Inf . Default is unconstrained.
active	Boolean vector of the same length as par, indicating if the parameter is used in the optimisation (TRUE) or held at a fixed value (FALSE).
control	A list of control options. Common options include ncores, run, master, verbose, REPORT, restart.file, gradient, and gr.method. Additional solver-specific options may be passed through to the underlying optimiser.
hessian	Logical. Should a numerically differentiated Hessian matrix be returned? Currently not implemented.
parallel	Logical. Enable parallel computation (e.g., for replicated evaluations and/or numerical gradients) using up to control\$ncores cores.

Details

optimh() uses the same interface conventions as `optim2()` and `calibrate()`: parameters may be supplied as a vector or list, box constraints can be enforced through lower and upper, and subsets of parameters can be held fixed through active.

The available methods include both genuinely heuristic/global algorithms (for example "AHR-ES", "CMA-ES", "DE", "genSA", "PSO") and a few derivative-free local searches (for example "Nelder-Mead", "bobyqa", "hjn", "nmk"). The function name should therefore be read broadly: it is the package entry point for optimisation methods that are typically useful when classical smooth, gradient-based optimisation is not the best fit.

Analytic gradients are accepted for interface consistency, but many methods implemented through optimh() ignore gr. Likewise, not all methods return Hessian information even when hessian = TRUE is requested.

Value

A list in the same general format as `optim2()`, typically including:

par The best parameter values found.

value The value of fn at par.

counts Number of function and gradient evaluations reported by the underlying solver, when available.

convergence Convergence code returned by the underlying solver.

message Additional information returned by the optimiser, if any.

hessian A Hessian estimate when supported and requested; otherwise usually NULL.

Choosing a method

Heuristic and evolutionary methods are often useful for rugged, multi-modal, or stochastic objective functions. Derivative-free local methods may be a better first choice when the objective is deterministic but gradients are unavailable or unreliable. Method choice should still reflect the structure of the problem: locality versus global exploration, box constraints, stochasticity, and computational cost.

Author(s)

Ricardo Oliveros-Ramos

See Also

[optim2\(\)](#), [calibrate\(\)](#), [ahres\(\)](#), [gradient\(\)](#)

Other optimisers: [ahres\(\)](#), [calibrate\(\)](#), [optim2\(\)](#)

Examples

```
## Not run:
optimh(
  par = rep(0.5, 5),
  fn = sphereN,
  lower = -5,
  upper = 5,
  method = "AHR-ES",
  control = list(maxit = 200)
)

## End(Not run)
```

sphereN

Noisy sphere test function

Description

Evaluate a stochastic variant of the sphere test function often used in optimisation examples. The deterministic sphere function is the sum of the squared coordinates; here, Gaussian noise is added to each coordinate before the squared distance is computed.

Usage

```
sphereN(x, sd = 0.1, aggregate = TRUE)
```

Arguments

x	Numeric vector of coordinates.
sd	Standard deviation of the Gaussian noise added independently to each coordinate.
aggregate	Logical. If TRUE, return the summed squared distance. If FALSE, return the contribution of each coordinate separately.

Details

This function is mainly useful for examples, testing, and illustrating how `calibrar` behaves with stochastic objective functions.

Value

A numeric scalar if `aggregate = TRUE`, or a numeric vector of squared contributions if `aggregate = FALSE`.

See Also

`calibrate()`, `optim2()`, `ahres()`

Examples

```
set.seed(1)
sphereN(rep(0, 5))
sphereN(rep(0, 5), aggregate = FALSE)
```

spline_par

Interpolate time-varying parameters with a spline

Description

Build a smooth time series from values specified at spline knots. This is useful when a model parameter is allowed to vary through time but is represented by a smaller set of control values to be estimated.

Usage

```
spline_par(par, n, knots = NULL, periodic = FALSE, period = NULL)

## S3 method for class 'spline.par'
plot(x, ...)
```

Arguments

par	Numeric vector of parameter values at the knots.
n	Number of output time points.
knots	Optional numeric vector giving the knot positions associated with par. If NULL, equally spaced knots are used.
periodic	Logical. Should the spline be treated as periodic?
period	Optional period of the cycle when periodic = TRUE and custom knots are supplied.
x	A spline.par object returned by <code>spline_par()</code> .
...	Further graphical arguments passed to <code>graphics::plot()</code> .

Details

The function rescales knot positions internally to the unit interval before building the spline. If `periodic = TRUE`, the series is extended so that the interpolation wraps around smoothly at the boundaries.

The returned object is designed to be plotted with `plot()` and to provide both the discrete interpolated values used in a model and a smoother curve for visual inspection.

Value

An object of class `spline.par`: a list containing the interpolated values (`x`), the corresponding time points (`time`), the knot locations, the knot values, and auxiliary information used by the plotting method.

See Also

`stats::splinefun()`

Examples

```
sp = spline_par(par = c(0.2, 0.8, 0.4, 0.6), n = 12)
head(sp$x)
plot(sp)
```

Index

- * **calibration**
 - calibrar-package, 2
- * **classes**
 - ahres.results, 9
 - calibrar.results, 10
- * **optimisers**
 - ahres, 6
 - calibrate, 14
 - optim2, 30
 - optimh, 33
- .get_command_argument, 4
- .get_command_argument(), 6
- .read_configuration, 5
- .read_configuration(), 4
- ahres, 6
- ahres(), 9, 10, 17, 33, 35, 36
- ahres.results, 9
- base::file.path(), 18
- base::readRDS(), 18
- base::saveRDS(), 18
- calibrar (calibrar-package), 2
- calibrar-package, 2
- calibrar.results, 10
- calibrar.results(), 14
- calibrar_demo, 13
- calibrar_demo(), 13
- calibrate, 14
- calibrate(), 8, 10, 11, 13, 14, 20, 21, 23, 24, 26, 33–36
- calibration_data, 17, 17
- calibration_data(), 13, 14, 19–24
- calibration_objFn, 17, 19
- calibration_objFn(), 12–14, 17–19, 22–24
- calibration_setup, 17, 22
- calibration_setup(), 13, 14, 17–19, 21
- coef(), 10, 12, 13
- coef.ahres.results (ahres.results), 9
- coef.calibrar.results
 - (calibrar.results), 10
- dim(), 12
- dim.calibrar.results
 - (calibrar.results), 10
- fitness (objFn), 26
- gaussian_kernel, 24
- gradient, 25
- gradient(), 35
- graphics::plot(), 37
- match.fun(), 26
- mvtnorm::dmvnorm(), 24, 25
- nlm, 33
- nlminb, 33
- objFn, 26
- objFn(), 19–22, 24
- optim, 33
- optim2, 30
- optim2(), 8, 13, 17, 26, 33–36
- optimh, 33
- optimh(), 8, 17, 26, 33
- plot.ahres.results (ahres.results), 9
- plot.calibrar.results
 - (calibrar.results), 10
- plot.spline.par (spline_par), 36
- predict(), 12, 13
- predict.calibrar.results
 - (calibrar.results), 10
- print(), 10, 12, 13
- print.ahres.results (ahres.results), 9
- print.calibrar_demo (calibrar_demo), 13
- print.calibrar.results
 - (calibrar.results), 10
- print.calibrar.results(), 14

```
print.summary.ahres.results  
    (ahres.results), 9  
print.summary.calibrar.results  
    (calibrar.results), 10  
  
sphereN, 35  
spline_par, 36  
spline_par(), 37  
stats::splinefun(), 37  
summary(), 10, 12, 13  
summary.ahres.results (ahres.results), 9  
summary.ahres.results(), 8  
summary.calibrar.results  
    (calibrar.results), 10  
summary.calibrar.results(), 14  
  
utils::read.csv(), 18, 22
```