Package 'compositional.mle'

October 28, 2026

Type Package

Title Compositional Maximum Likelihood Estimation

Version 2.0.1

Description Provides composable optimization strategies for maximum likelihood estimation (MLE). Solvers are first-class functions that combine via sequential chaining, parallel racing, and random restarts. Implements gradient ascent, Newton-Raphson, quasi-Newton (BFGS), and derivative-free methods with support for constrained optimization and tracing. Returns 'mle' objects compatible with 'algebraic.mle' for downstream analysis. Methods based on Nocedal J, Wright SJ (2006) ``Numerical Optimization'' <doi:10.1007/978-0-387-40065-5>.

License MIT + file LICENSE

Encoding UTF-8

ByteCompile true

Depends R (>= 3.5.0), algebraic.mle (>= 2.0.0)

Imports MASS, numDeriv

Suggests rmarkdown, dplyr, knitr, ggplot2, tibble, testthat (>= 3.0.0), cli, future, hypothesize, likelihood.model

URL https://github.com/queelius/compositional.mle, https://queelius.github.io/compositional.mle/

BugReports https://github.com/queelius/compositional.mle/issues

VignetteBuilder knitr

Config/testthat/edition 3

RoxygenNote 7.3.3

NeedsCompilation no

Author Alexander Towell [aut, cre] (ORCID: <https://orcid.org/0000-0001-6443-9897>)

Maintainer Alexander Towell <queelius@gmail.com>

Repository CRAN

Date/Publication 2026-10-28 07:40:09 UTC

Contents

bfgs . . . 3
chain . . . 4
clear_cache . . . 5
compose_transforms . . . 5
coordinate_ascent . . . 6
fisher_scoring . . . 7
get_fisher . . . 8
get_score . . . 9
gradient_ascent . . . 10
grid_search . . . 11
is_converged . . . 12
is_mle_constraint . . . 13
is_mle_problem . . . 13
is_solver_result . . . 14
is_tracing . . . 14
lbfgsb . . . 15
mle_constraint . . . 16
mle_problem . . . 17
mle_trace . . . 19
nelder_mead . . . 20
newton_raphson . . . 21
normal_sampler . . . 22
num_iterations . . . 23
optimization_path . . . 23
penalty_elastic_net . . . 24
penalty_l1 . . . 25
penalty_l2 . . . 26
plot.mle_trace_data . . . 26
plot.solver_result . . . 27
print.mle_trace_data . . . 28
race . . . 28
race_operator . . . 29
random_search . . . 30
sim_anneal . . . 31
uniform_sampler . . . 32
unless_converged . . . 33
update.mle_problem . . . 33
with_penalty . . . 34
with_restarts . . . 34
with_subsampling . . . 35
%>>% . . . 36

Index **38**

bfgs *BFGS Solver*

Description

Creates a solver using the BFGS quasi-Newton method via `optim()`. BFGS approximates the Hessian from gradient information, providing second-order-like convergence without computing the Hessian directly.

Usage

```
bfgs(max_iter = 100L, tol = 1e-08, report = 0L)
```

Arguments

`max_iter`	Maximum number of iterations
`tol`	Convergence tolerance (passed to optim's reltol)
`report`	Reporting frequency (0 = no reporting)

Details

BFGS is often a good default choice: it's more robust than Newton-Raphson (no matrix inversion issues) and faster than gradient ascent (uses curvature information).

The solver automatically uses the score function from the problem if available, otherwise computes gradients numerically.

Value

A solver function with signature (problem, theta0, trace) -> mle_result

Examples

```
set.seed(42)
x <- rnorm(50, 5, 2)
problem <- mle_problem(
  loglike = function(theta) sum(dnorm(x, theta[1], theta[2], log = TRUE)),
  constraint = mle_constraint(support = function(theta) theta[2] > 0,
                              project = function(theta) c(theta[1], max(theta[2], 1e-8)))
)
# Basic usage
result <- bfgs()(problem, c(4, 1.5))

# Race BFGS against gradient ascent
strategy <- bfgs() %|% gradient_ascent()
```

chain *Chain Solvers with Early Stopping*

Description

Chains multiple solvers sequentially with optional early stopping. More flexible than %>>% operator.

Usage

```
chain(..., early_stop = NULL)
```

Arguments

...	Solver functions to chain
early_stop	Optional function that takes a result and returns TRUE to stop the chain early. Default is NULL (no early stopping).

Details

The chain runs solvers in order, passing each result's theta.hat to the next solver. If early_stop is provided and returns TRUE for any intermediate result, the chain stops early.

Common early stopping conditions:

- Stop when converged: function(r) r$converged
- Stop when gradient is small: function(r) sqrt(sum(score^2)) < 1e-6
- Stop after reaching target: function(r) r$loglike > -100

Value

A new solver function that runs solvers in sequence

Examples

```
# Chain with early stopping when converged
strategy <- chain(
  grid_search(lower = c(-10, 0.1), upper = c(10, 5), n = 5),
  gradient_ascent(max_iter = 50),
  newton_raphson(max_iter = 20),
  early_stop = function(r) isTRUE(r$converged)
)

# Standard chain (no early stopping)
strategy <- chain(gradient_ascent(), newton_raphson())
```

clear_cache *Clear derivative cache*

Description

Clears the cached numerical derivatives (score and Fisher) from an mle_problem. This is useful when you want to force recomputation, for example after modifying data that the log-likelihood depends on.

Usage

```
clear_cache(problem)
```

Arguments

problem	An mle_problem object

Value

The problem object (invisibly), modified in place

Examples

```
loglike <- function(theta) -sum((theta - c(1, 2))^2)
problem <- mle_problem(loglike, cache_derivatives = TRUE)
# ... run some optimization ...
clear_cache(problem)  # Force fresh derivative computation
```

compose_transforms *Compose Multiple Function Transformations*

Description

Applies transformations right-to-left (like mathematical composition). This allows building complex log-likelihood transformations from simple ones.

Usage

```
compose_transforms(...)
```

Arguments

...	Transformer functions

Details

Note: For composing solvers, use `chain` instead.

Value

Composed transformer function

Examples

```
# Create a composition of transformations
transform <- compose_transforms(
  function(f) with_penalty(f, penalty_l1(), lambda = 0.01),
  function(f) with_penalty(f, penalty_l2(), lambda = 0.05)
)

# Apply to log-likelihood
loglike <- function(theta) -sum((theta - c(1, 2))^2)
loglike_transformed <- transform(loglike)
loglike_transformed(c(1, 2))
```

`coordinate_ascent` *Coordinate Ascent Solver*

Description

Creates a solver that optimizes one parameter at a time while holding others fixed. This is useful when parameters have different scales or when the likelihood decomposes nicely along coordinate directions.

Usage

```
coordinate_ascent(
  max_cycles = 50L,
  tol = 1e-08,
  line_search = TRUE,
  learning_rate = 0.1,
  cycle_order = c("sequential", "random"),
  verbose = FALSE
)
```

Arguments

`max_cycles`	Maximum number of full cycles through all parameters
`tol`	Convergence tolerance on log-likelihood change
`line_search`	Use line search for each coordinate (slower but more robust)
`learning_rate`	Step size for non-line-search mode (default 0.1)

cycle_order	Order of cycling: "sequential" (1,2,...,p) or "random"
verbose	Logical; if TRUE and the **cli** package is installed, display progress during optimization. Default is FALSE.

Details

Each cycle consists of optimizing each coordinate in turn using a simple golden section search. The algorithm converges when the log-likelihood improvement in a full cycle is less than `tol`.

Coordinate ascent can be effective when:

- Parameters are on very different scales
- The likelihood has axis-aligned ridges
- Computing the full gradient is expensive

However, it may converge slowly for problems with strong parameter correlations.

Value

A solver function with signature (problem, theta0, trace) -> mle_result

See Also

`gradient_ascent` for gradient-based optimization, `nelder_mead` for another derivative-free method

Examples

```
# Basic coordinate ascent
solver <- coordinate_ascent()

# With more cycles for difficult problems
solver <- coordinate_ascent(max_cycles = 100)

# Random cycling to avoid systematic bias
solver <- coordinate_ascent(cycle_order = "random")
```

---

fisher_scoring *Fisher Scoring Solver*

Description

Variant of Newton-Raphson that uses the expected Fisher information instead of the observed Fisher. Can be more stable for some problems.

Usage

```
fisher_scoring(...)
```

Arguments

...	Arguments passed to newton_raphson

Details

Fisher scoring is identical to Newton-Raphson when the expected and observed Fisher information are equal (e.g., exponential families). For other models, it may have different convergence properties.

Value

A solver function with signature (problem, theta0, trace) -> mle_result

Examples

```
set.seed(42)
x <- rnorm(50, 5, 2)
problem <- mle_problem(
  loglike = function(theta) sum(dnorm(x, theta[1], theta[2], log = TRUE)),
  constraint = mle_constraint(
    support = function(theta) theta[2] > 0,
    project = function(theta) c(theta[1], max(theta[2], 1e-8))
  )
)
solver <- fisher_scoring()
result <- solver(problem, c(4, 1.5))
```

---

get_fisher *Get Fisher information function from problem*

Description

Returns the Fisher information matrix function, computing numerically if not provided. If `cache_derivatives = TRUE` was set in the problem and Fisher is computed numerically, results are cached using a single-value cache.

Usage

```
get_fisher(problem)
```

Arguments

problem	An mle_problem object

Value

Fisher information function that takes a parameter vector and returns the Fisher information matrix (negative Hessian of log-likelihood).

Examples

```
problem <- mle_problem(
  loglike = function(theta) -sum((theta - c(1, 2))^2)
)
fisher_fn <- get_fisher(problem)
fisher_fn(c(1, 2))  # Fisher information at the optimum
```

---

get_score *Get score function from problem*

Description

Returns the score (gradient) function, computing numerically if not provided. If `cache_derivatives = TRUE` was set in the problem and score is computed numerically, results are cached using a single-value cache.

Usage

```
get_score(problem)
```

Arguments

`problem`	An mle_problem object

Value

Score function that takes a parameter vector and returns the gradient of the log-likelihood.

Examples

```
problem <- mle_problem(
  loglike = function(theta) -sum((theta - c(1, 2))^2)
)
score_fn <- get_score(problem)
score_fn(c(0, 0))  # Gradient at (0, 0)
```

`gradient_ascent` *Gradient Ascent Solver*

Description

Creates a solver that uses gradient ascent (steepest ascent) to find the MLE. Optionally uses backtracking line search for adaptive step sizes.

Usage

```
gradient_ascent(
  learning_rate = 1,
  line_search = TRUE,
  max_iter = 100L,
  tol = 1e-08,
  backtrack_ratio = 0.5,
  min_step = 1e-12,
  verbose = FALSE
)
```

Arguments

`learning_rate`	Base learning rate / maximum step size
`line_search`	Use backtracking line search for adaptive step sizes
`max_iter`	Maximum number of iterations
`tol`	Convergence tolerance (on parameter change)
`backtrack_ratio`	Step size reduction factor for line search ($0 < r < 1$)
`min_step`	Minimum step size before giving up
`verbose`	Logical; if TRUE and the **cli** package is installed, display progress during optimization. Default is FALSE.

Details

Gradient ascent iteratively moves in the direction of the score (gradient of log-likelihood). With line search enabled, the step size is adaptively chosen to ensure the log-likelihood increases.

The solver respects constraints defined in the problem via projection.

Value

A solver function with signature (problem, theta0, trace) -> mle_result

See Also

`newton_raphson` for second-order optimization, `bfgs` for quasi-Newton, `%>>%` and `%|%` for solver composition

Examples

```
# Create a solver with default parameters
solver <- gradient_ascent()

# Create a solver with custom parameters
solver <- gradient_ascent(
  learning_rate = 0.5,
  max_iter = 500,
  tol = 1e-10
)

# Without line search (fixed step size)
solver <- gradient_ascent(learning_rate = 0.01, line_search = FALSE)
```

---

grid_search *Grid Search Solver*

Description

Creates a solver that evaluates the log-likelihood on a grid of points and returns the best. Useful for finding good starting points or for low-dimensional problems.

Usage

```
grid_search(lower, upper, n = 10L)
```

Arguments

`lower`	Lower bounds for the grid
`upper`	Upper bounds for the grid
`n`	Number of points per dimension (scalar or vector)

Details

Grid search is deterministic and exhaustive within its bounds. It's most useful for 1-3 dimensional problems or as the first stage of a multi-stage strategy (e.g., grid_search

The theta0 argument is ignored; the grid is determined by lower/upper/n. Points outside the problem's constraint support are skipped.

Value

A solver function with signature (problem, theta0, trace) -> mle_result

Examples

```
set.seed(42)
x <- rnorm(50, 5, 2)
problem <- mle_problem(
  loglike = function(theta) sum(dnorm(x, theta[1], theta[2], log = TRUE)),
  constraint = mle_constraint(support = function(theta) theta[2] > 0,
                              project = function(theta) c(theta[1], max(theta[2], 1e-8)))
)
# Simple grid search
solver <- grid_search(lower = c(-10, 0.1), upper = c(10, 5), n = 20)
result <- solver(problem, c(0, 1))

# Coarse-to-fine: grid then gradient
strategy <- grid_search(c(-10, 0.1), c(10, 5), n = 5) %>>% gradient_ascent()
```

---

is_converged *Check if solver converged*

Description

Check if solver converged

Usage

```
is_converged(x, ...)
```

Arguments

x	An mle result object
...	Additional arguments (unused)

Value

Logical indicating convergence

Examples

```
problem <- mle_problem(
  loglike = function(theta) -sum((theta - c(1, 2))^2)
)
result <- gradient_ascent(max_iter = 50)(problem, c(0, 0))
is_converged(result)
```

is_mle_constraint *Check if object is an mle_constraint*

Description

Check if object is an mle_constraint

Usage

```
is_mle_constraint(x)
```

Arguments

x	Object to test

Value

Logical indicating whether x is an mle_constraint.

Examples

```
constraint <- mle_constraint(support = function(theta) all(theta > 0))
is_mle_constraint(constraint)  # TRUE
is_mle_constraint(list())      # FALSE
```

is_mle_problem *Check if object is an mle_problem*

Description

Check if object is an mle_problem

Usage

```
is_mle_problem(x)
```

Arguments

x	Object to test

Value

Logical indicating whether x is an mle_problem.

Examples

```
problem <- mle_problem(
  loglike = function(theta) -sum((theta - c(1, 2))^2)
)
is_mle_problem(problem)  # TRUE
is_mle_problem(list())   # FALSE
```

---

is_solver_result *Check if object is a solver result*

Description

Check if object is a solver result

Usage

```
is_solver_result(x)
```

Arguments

x	Object to test

Value

Logical indicating whether x inherits from `solver_result`.

Examples

```
problem <- mle_problem(
  loglike = function(theta) -sum((theta - c(1, 2))^2)
)
result <- gradient_ascent(max_iter = 20)(problem, c(0, 0))
is_solver_result(result)  # TRUE
is_solver_result(list())  # FALSE
```

---

is_tracing *Check if tracing is enabled*

Description

Check if tracing is enabled

Usage

```
is_tracing(trace)
```

Arguments

trace	An mle_trace object

Value

Logical indicating if any tracing is enabled

Examples

```
# Tracing disabled (default)
trace <- mle_trace()
is_tracing(trace)  # FALSE

# Tracing enabled
trace <- mle_trace(values = TRUE)
is_tracing(trace)  # TRUE
```

---

lbfgsb *L-BFGS-B Solver (Box Constrained)*

Description

Creates a solver using L-BFGS-B, a limited-memory BFGS variant that supports box constraints (lower and upper bounds on parameters).

Usage

```
lbfgsb(lower = -Inf, upper = Inf, max_iter = 100L, tol = 1e-08)
```

Arguments

lower	Lower bounds on parameters (can be -Inf)
upper	Upper bounds on parameters (can be Inf)
max_iter	Maximum number of iterations
tol	Convergence tolerance

Details

Unlike the constraint system in mle_problem (which uses projection), L-BFGS-B handles box constraints natively within the algorithm. Use this when you have simple bound constraints.

Value

A solver function

Examples

```
set.seed(42)
x <- rnorm(50, 5, 2)
problem <- mle_problem(
  loglike = function(theta) sum(dnorm(x, theta[1], theta[2], log = TRUE))
)
# Positive sigma via box constraint
solver <- lbfgsb(lower = c(-Inf, 0.01), upper = c(Inf, Inf))
result <- solver(problem, c(4, 1.5))
```

---

mle_constraint *Create domain constraint specification*

Description

Specifies domain constraints for optimization. The support function checks if parameters are valid, and the project function maps invalid parameters back to valid ones.

Usage

```
mle_constraint(support = function(theta) TRUE, project = function(theta) theta)
```

Arguments

support	Function testing if theta is in support (returns TRUE/FALSE)
project	Function projecting theta onto support

Value

An mle_constraint object

Examples

```
# Positive parameters only
constraint <- mle_constraint(
  support = function(theta) all(theta > 0),
  project = function(theta) pmax(theta, 1e-8)
)

# Parameters in [0, 1]
constraint <- mle_constraint(
  support = function(theta) all(theta >= 0 & theta <= 1),
  project = function(theta) pmax(0, pmin(1, theta))
)

# No constraints (default)
constraint <- mle_constraint()
```

`mle_problem` *Create an MLE Problem Specification*

Description

Encapsulates a maximum likelihood estimation problem, separating the statistical specification from the optimization strategy.

Usage

```
mle_problem(loglike, ...)

## Default S3 method:
mle_problem(
  loglike,
  score = NULL,
  fisher = NULL,
  constraint = NULL,
  theta_names = NULL,
  n_obs = NULL,
  cache_derivatives = FALSE,
  ...
)

## S3 method for class 'likelihood_model'
mle_problem(
  loglike,
  data,
  constraint = NULL,
  theta_names = NULL,
  cache_derivatives = FALSE,
  ...
)

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

Arguments

`loglike`	Log-likelihood function taking parameter vector theta, or a `likelihood_model` object.
`...`	Additional arguments (unused).
`score`	Score function (gradient of log-likelihood). If NULL, computed numerically via numDeriv::grad when needed.
`fisher`	Fisher information matrix function. If NULL, computed numerically via numDeriv::hessian when needed.

constraint	Domain constraints as mle_constraint object
theta_names	Character vector of parameter names for nice output
n_obs	Number of observations (for AIC/BIC computation)
cache_derivatives	Logical; if TRUE and score/fisher are computed numerically, cache the most recent result to avoid redundant computation. This is particularly useful during line search where the same point may be evaluated multiple times. Default is FALSE.
data	Data frame (or matrix/vector) of observations
x	An mle_problem object.

Details

When passed a `likelihood_model` object (from the **likelihood.model** package), automatically extracts log-likelihood, score, and Fisher information functions.

The problem object provides lazy evaluation of derivatives. If you don't provide analytic score or fisher functions, they will be computed numerically when requested.

When `cache_derivatives = TRUE`, numerical derivatives are cached using a single-value cache (stores the most recent theta and result). This is efficient for optimization where consecutive calls often evaluate at the same point (e.g., during line search or convergence checking). Use `clear_cache` to manually clear the cache if needed.

Value

An mle_problem object

The input object, invisibly (for method chaining).

See Also

`mle_constraint` for constraint specification

Examples

```
# Direct specification
problem <- mle_problem(
  loglike = function(theta) -sum((theta - c(1, 2))^2)
)

# With analytic derivatives
problem <- mle_problem(
  loglike = function(theta) sum(dnorm(data, theta[1], theta[2], log = TRUE)),
  score = function(theta) {
    c(sum(data - theta[1]) / theta[2]^2,
      -length(data)/theta[2] + sum((data - theta[1])^2) / theta[2]^3)
  },
  constraint = mle_constraint(
    support = function(theta) theta[2] > 0,
    project = function(theta) c(theta[1], max(theta[2], 1e-8))
```

```
  ),
  theta_names = c("mu", "sigma")
)

# Without analytic derivatives (computed numerically)
problem <- mle_problem(
  loglike = function(theta) sum(dnorm(data, theta[1], theta[2], log = TRUE)),
  constraint = mle_constraint(
    support = function(theta) theta[2] > 0
  )
)
```

`mle_trace` *Create a Trace Configuration*

Description

Specifies what information to track during optimization.

Usage

```
mle_trace(
  values = FALSE,
  path = FALSE,
  gradients = FALSE,
  timing = FALSE,
  every = 1L
)

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

Arguments

`values`	Track log-likelihood values at each iteration
`path`	Track parameter values at each iteration
`gradients`	Track gradient norms at each iteration
`timing`	Track wall-clock time
`every`	Record every nth iteration (1 = all iterations)
`x`	An mle_trace object.
`...`	Additional arguments (unused).

Value

An mle_trace configuration object

The input object, invisibly (for method chaining).

Examples

```
# Track everything
trace <- mle_trace(values = TRUE, path = TRUE, gradients = TRUE)

# Minimal tracing (just convergence path)
trace <- mle_trace(values = TRUE)

# Sample every 10th iteration for long runs
trace <- mle_trace(values = TRUE, path = TRUE, every = 10)
```

---

nelder_mead *Nelder-Mead Solver (Derivative-Free)*

Description

Creates a solver using the Nelder-Mead simplex method via `optim()`. This is a derivative-free method useful when gradients are unavailable or unreliable.

Usage

```
nelder_mead(max_iter = 500L, tol = 1e-08)
```

Arguments

`max_iter`	Maximum number of iterations
`tol`	Convergence tolerance

Details

Nelder-Mead doesn't use gradient information, making it robust but potentially slower. It's useful as a fallback when gradient-based methods fail, or for problems with non-smooth likelihoods.

Value

A solver function

Examples

```
set.seed(42)
x <- rnorm(50, 5, 2)
problem <- mle_problem(
  loglike = function(theta) sum(dnorm(x, theta[1], theta[2], log = TRUE)),
  constraint = mle_constraint(support = function(theta) theta[2] > 0,
                              project = function(theta) c(theta[1], max(theta[2], 1e-8)))
)
# Use when gradients are problematic
result <- nelder_mead()(problem, c(4, 1.5))
```

```
# Race against gradient methods
strategy <- gradient_ascent() %|% nelder_mead()
```

`newton_raphson` *Newton-Raphson Solver*

Description

Creates a solver that uses Newton-Raphson (second-order) optimization. Uses the Fisher information matrix to scale the gradient for faster convergence near the optimum.

Usage

```
newton_raphson(
  line_search = TRUE,
  max_iter = 50L,
  tol = 1e-08,
  backtrack_ratio = 0.5,
  min_step = 1e-12,
  verbose = FALSE
)
```

Arguments

`line_search`	Use backtracking line search for stability
`max_iter`	Maximum number of iterations
`tol`	Convergence tolerance (on parameter change)
`backtrack_ratio`	Step size reduction factor for line search
`min_step`	Minimum step size before giving up
`verbose`	Logical; if TRUE and the **cli** package is installed, display progress during optimization. Default is FALSE.

Details

Newton-Raphson computes the search direction as $I(\theta)^{-1}s(\theta)$ where I is the Fisher information and s is the score. This accounts for parameter scaling and typically converges faster than gradient ascent when near the optimum.

Requires the problem to have a Fisher information function (either analytic or computed numerically).

Value

A solver function with signature (problem, theta0, trace) -> mle_result

See Also

gradient_ascent for first-order optimization, fisher_scoring (alias), %>>% for chaining

Examples

```
set.seed(42)
x <- rnorm(50, 5, 2)
problem <- mle_problem(
  loglike = function(theta) sum(dnorm(x, theta[1], theta[2], log = TRUE)),
  constraint = mle_constraint(support = function(theta) theta[2] > 0,
                              project = function(theta) c(theta[1], max(theta[2], 1e-8)))
)
# Basic usage
solver <- newton_raphson()
result <- solver(problem, c(4, 1.5))

# Often used after gradient ascent for refinement
strategy <- gradient_ascent(max_iter = 50) %>>% newton_raphson(max_iter = 20)
```

---

normal_sampler *Normal Sampler Factory*

Description

Creates a sampler function for use with with_restarts that generates normally distributed starting points around a center.

Usage

```
normal_sampler(center, sd = 1)
```

Arguments

center	Mean of the normal distribution
sd	Standard deviation (scalar or vector)

Value

A sampler function

Examples

```
sampler <- normal_sampler(c(0, 1), sd = c(5, 0.5))
strategy <- with_restarts(gradient_ascent(), n = 20, sampler = sampler)
```

num_iterations *Get number of iterations*

Description

Get number of iterations

Usage

```
num_iterations(x, ...)
```

Arguments

x	An mle result object
...	Additional arguments (unused)

Value

Number of iterations, or NA_integer_ if not available.

Examples

```
problem <- mle_problem(
  loglike = function(theta) -sum((theta - c(1, 2))^2)
)
result <- gradient_ascent(max_iter = 50)(problem, c(0, 0))
num_iterations(result)
```

optimization_path *Extract Optimization Path as Data Frame*

Description

Converts the trace data from an MLE result into a tidy data frame for custom analysis and plotting (e.g., with ggplot2).

Usage

```
optimization_path(x, ...)
```

Arguments

x	A solver_result with trace_data, or an mle_trace_data object
...	Additional arguments (unused)

Value

A data frame with columns:

- `iteration`: Iteration number
- `loglike`: Log-likelihood value (if traced)
- `grad_norm`: Gradient norm (if traced)
- `time`: Elapsed time in seconds (if traced)
- `theta_1`, `theta_2`, ...: Parameter values (if path traced)

Examples

```
# Get optimization path as data frame
problem <- mle_problem(
  loglike = function(theta) -sum((theta - c(3, 2))^2),
  constraint = mle_constraint(support = function(theta) TRUE)
)
trace_cfg <- mle_trace(values = TRUE, path = TRUE)
result <- gradient_ascent(max_iter = 30)(problem, c(0, 0), trace = trace_cfg)

path_df <- optimization_path(result)
head(path_df)
```

---

penalty_elastic_net *Elastic net penalty (combination of L1 and L2)*

Description

Creates a penalty combining L1 and L2 norms. The parameter alpha controls the balance: alpha=1 is pure LASSO, alpha=0 is pure Ridge.

Usage

```
penalty_elastic_net(alpha = 0.5, weights = NULL)
```

Arguments

`alpha`	Balance between L1 and L2 (numeric in [0,1], default: 0.5)
`weights`	Optional parameter weights (default: all 1)

Value

Penalty function

Examples

```
# Equal mix of L1 and L2
penalty <- penalty_elastic_net(alpha = 0.5)

# More L1 (more sparsity)
penalty <- penalty_elastic_net(alpha = 0.9)

# More L2 (more shrinkage)
penalty <- penalty_elastic_net(alpha = 0.1)
```

---

penalty_l1 *L1 penalty function (LASSO)*

Description

Creates a penalty function that computes the L1 norm (sum of absolute values). Used for sparsity-inducing regularization.

Usage

```
penalty_l1(weights = NULL)
```

Arguments

weights	Optional parameter weights (default: all 1)

Value

Penalty function

Examples

```
penalty <- penalty_l1()
penalty(c(1, -2, 3))  # Returns 6

# Weighted L1
penalty <- penalty_l1(weights = c(1, 2, 1))
penalty(c(1, -2, 3))  # Returns 1*1 + 2*2 + 1*3 = 8
```

penalty_l2 *L2 penalty function (Ridge)*

Description

Creates a penalty function that computes the L2 norm squared (sum of squares). Used for parameter shrinkage.

Usage

```
penalty_l2(weights = NULL)
```

Arguments

weights	Optional parameter weights (default: all 1)

Value

Penalty function

Examples

```
penalty <- penalty_l2()
penalty(c(1, -2, 3))  # Returns 14

# Weighted L2
penalty <- penalty_l2(weights = c(1, 2, 1))
penalty(c(1, -2, 3))  # Returns 1^2 + (2*2)^2 + 3^2 = 26
```

plot.mle_trace_data *Plot Trace Data Directly*

Description

Plot Trace Data Directly

Usage

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

Arguments

x	An mle_trace_data object
...	Arguments passed to plotting functions

Value

Called for side effects (generates a plot). Returns the input object invisibly.

`plot.solver_result` *Plot Optimization Convergence*

Description

Visualizes the optimization trajectory from an MLE result with tracing enabled. Shows log-likelihood progression, gradient norm decay, and optionally the parameter path (for 2D problems).

Usage

```
## S3 method for class 'solver_result'
plot(x, which = c("loglike", "gradient"), main = NULL, ...)
```

Arguments

`x`	A solver_result object with trace_data in solver_info
`which`	Character vector specifying which plots to show: "loglike" (log-likelihood), "gradient" (gradient norm), "path" (2D parameter path)
`main`	Optional title
`...`	Additional arguments passed to plot

Details

This function requires that the solver was run with tracing enabled via `mle_trace()`. Without trace data, the function will warn and return invisibly.

The "path" plot is only shown for 2D parameter problems.

Value

Invisibly returns the trace data

Examples

```
# Enable tracing when solving
problem <- mle_problem(
  loglike = function(theta) -sum((theta - c(3, 2))^2),
  constraint = mle_constraint(support = function(theta) TRUE)
)
trace_cfg <- mle_trace(values = TRUE, gradients = TRUE, path = TRUE)
result <- gradient_ascent(max_iter = 50)(problem, c(0, 0), trace = trace_cfg)

# Plot convergence diagnostics
plot(result)
```

print.mle_trace_data *Print MLE Trace Data*

Description

Print MLE Trace Data

Usage

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

Arguments

x	An mle_trace_data object.
...	Additional arguments (unused).

Value

The input object, invisibly (for method chaining).

race *Race Multiple Solvers*

Description

Runs multiple solvers (optionally in parallel) and returns the best result (highest log-likelihood). More flexible than %|% operator.

Usage

```
race(..., parallel = FALSE)
```

Arguments

...	Solver functions to race
parallel	Logical; if TRUE and the **future** package is installed, solvers are run in parallel using the current future plan. Default is FALSE.

Details

When parallel = TRUE, solvers are executed using future::future() and results collected with future::value(). The current future plan determines how parallelization happens (e.g., plan(multisession) for multi-process execution).

Failed solvers (those that throw errors) are ignored. If all solvers fail, an error is thrown.

Value

A new solver function that races all solvers and picks the best

Examples

```
# Race three methods sequentially
strategy <- race(gradient_ascent(), bfgs(), nelder_mead())

# Race with parallel execution (requires future package)
## Not run:
future::plan(future::multisession)
strategy <- race(gradient_ascent(), bfgs(), nelder_mead(), parallel = TRUE)

## End(Not run)
```

---

race_operator *Parallel Solver Racing (Operator)*

Description

Runs multiple solvers and returns the best result (highest log-likelihood). Useful when unsure which method will work best for a given problem.

Usage

```
s1 %|% s2
```

Arguments

s1	First solver function
s2	Second solver function

Details

For parallel execution or more than 2 solvers, use race.

Value

A new solver function that runs both and picks the best

See Also

race for parallel execution

Examples

```
# Race gradient-based vs derivative-free
strategy <- gradient_ascent() %|% nelder_mead()

# Race multiple methods
strategy <- gradient_ascent() %|% bfgs() %|% nelder_mead()
```

`random_search` *Random Search Solver*

Description

Creates a solver that evaluates the log-likelihood at random points and returns the best. Useful for high-dimensional problems where grid search is infeasible.

Usage

```
random_search(sampler, n = 100L)
```

Arguments

`sampler`	Function generating random parameter vectors
`n`	Number of random points to evaluate

Details

Unlike grid search, random search scales better to high dimensions. The sampler should generate points in a reasonable region; points outside the problem's constraint support are skipped.

Value

A solver function

Examples

```
# Create a random search solver with uniform sampling
solver <- random_search(
  sampler = uniform_sampler(c(-10, 0.1), c(10, 5)),
  n = 100
)
```

`sim_anneal` *Simulated Annealing Solver*

Description

Creates a solver using simulated annealing for global optimization. Simulated annealing can escape local optima by probabilistically accepting worse solutions, with the acceptance probability decreasing over time (controlled by a "temperature" parameter).

Usage

```
sim_anneal(
  temp_init = 10,
  cooling_rate = 0.95,
  max_iter = 1000L,
  neighbor_sd = 1,
  min_temp = 1e-10,
  verbose = FALSE
)
```

Arguments

`temp_init`	Initial temperature (higher = more exploration)
`cooling_rate`	Temperature reduction factor per iteration (0 < r < 1)
`max_iter`	Maximum number of iterations
`neighbor_sd`	Standard deviation for generating neighbor proposals
`min_temp`	Minimum temperature before stopping
`verbose`	Logical; if TRUE and the **cli** package is installed, display progress during optimization. Default is FALSE.

Details

At each iteration: 1. Generate a neighbor by adding Gaussian noise to current parameters 2. If the neighbor improves the objective, accept it 3. If the neighbor is worse, accept with probability exp(delta / temp) 4. Reduce temperature: temp = temp * cooling_rate

The algorithm is stochastic and may find different solutions on different runs. For best results, use with `with_restarts()` or combine with a local optimizer via `%>>%`.

Value

A solver function with signature (problem, theta0, trace) -> mle_result

See Also

`with_restarts` for multi-start optimization, `gradient_ascent` for local refinement

Examples

```
# Basic simulated annealing
solver <- sim_anneal()

# More exploration (higher initial temp, slower cooling)
solver <- sim_anneal(temp_init = 100, cooling_rate = 0.999)

# Coarse global search, then local refinement
strategy <- sim_anneal(max_iter = 500) %>>% gradient_ascent()
```

---

uniform_sampler *Uniform Sampler Factory*

Description

Creates a sampler function for use with `with_restarts` that generates uniformly distributed starting points.

Usage

```
uniform_sampler(lower, upper)
```

Arguments

`lower`	Lower bounds for each parameter
`upper`	Upper bounds for each parameter

Value

A sampler function

Examples

```
sampler <- uniform_sampler(c(-10, 0.1), c(10, 5))
strategy <- with_restarts(gradient_ascent(), n = 20, sampler = sampler)
```

unless_converged *Conditional Refinement*

Description

Applies a refinement solver only if the first solver did not converge. If refinement is applied, trace data from both solvers is merged.

Usage

```
unless_converged(solver, refinement)
```

Arguments

`solver`	Primary solver function
`refinement`	Solver to use if primary doesn't converge

Value

A new solver function with conditional refinement

Examples

```
# Use Newton-Raphson to refine if gradient ascent doesn't converge
strategy <- unless_converged(gradient_ascent(max_iter = 50), newton_raphson())
```

update.mle_problem *Update an mle_problem*

Description

Create a new problem with some fields updated.

Usage

```
## S3 method for class 'mle_problem'
update(object, ...)
```

Arguments

`object`	An mle_problem
`...`	Named arguments to update

Value

New mle_problem

with_penalty *Add penalty term to log-likelihood*

Description

Transforms a log-likelihood by subtracting a penalty term. Useful for regularized estimation (e.g., LASSO, Ridge regression).

Usage

```
with_penalty(loglike, penalty, lambda = 1)
```

Arguments

loglike	Base log-likelihood function
penalty	Penalty function taking theta and returning numeric
lambda	Penalty weight (non-negative numeric, default: 1.0)

Value

Transformed log-likelihood function

Examples

```
# Regression with L2 penalty (Ridge)
loglike <- function(theta) -sum((theta - c(1, 2))^2)

# Add L2 penalty
loglike_penalized <- with_penalty(
  loglike,
  penalty = penalty_l2(),
  lambda = 0.1
)
loglike_penalized(c(1, 2))  # Evaluate penalized likelihood
```

with_restarts *Multiple Random Restarts*

Description

Runs a solver from multiple starting points and returns the best result. Essential for problems with multiple local optima.

Usage

```
with_restarts(solver, n, sampler, max_reject = 100L)
```

Arguments

solver	A solver function
n	Number of restarts (including the provided theta0)
sampler	Function that generates random starting points. Called with no arguments, should return a parameter vector. Samples are automatically constrained using problem$constraint.
max_reject	Maximum rejection attempts per sample before projection

Details

The sampler generates candidate starting points, which are automatically filtered/projected using the problem's constraint. This means samplers can be simple distributions without constraint awareness.

Value

A new solver function with restart capability

Examples

```
# 20 random restarts - constraint applied automatically from problem
sampler <- uniform_sampler(c(-10, 0), c(10, 5))
strategy <- with_restarts(gradient_ascent(), n = 20, sampler = sampler)

# Can also compose with other operators
strategy <- with_restarts(gradient_ascent(), n = 10, sampler = sampler) %>>%
  newton_raphson()
```

---

with_subsampling *Create stochastic log-likelihood with subsampling*

Description

Transforms a log-likelihood function to use only a random subsample of observations. Useful for stochastic gradient ascent on large datasets.

Usage

```
with_subsampling(loglike, data, subsample_size, replace = FALSE)
```

Arguments

loglike	Base log-likelihood function. Should accept theta and data.
data	Observations (vector, matrix, or data.frame)
subsample_size	Number of observations to sample per evaluation
replace	Sample with replacement (logical, default: FALSE)

Value

Transformed log-likelihood function

Examples

```
# Original likelihood uses all data
data <- rnorm(10000, mean = 5, sd = 2)

loglike <- function(theta, obs = data) {
  sum(dnorm(obs, mean = theta[1], sd = theta[2], log = TRUE))
}

# Stochastic version uses random subsample
loglike_stoch <- with_subsampling(
  loglike,
  data = data,
  subsample_size = 100
)

# Each call uses different random subsample
loglike_stoch(c(5, 2))
loglike_stoch(c(5, 2))  # Different value
```

%>>% *Sequential Solver Composition*

Description

Chains two solvers sequentially. The result of the first solver becomes the starting point for the second. This enables coarse-to-fine strategies.

Usage

```
s1 %>>% s2
```

Arguments

s1	First solver function
s2	Second solver function

Details

Trace data from all solvers in the chain is merged into a single trace with stage boundaries preserved.

Value

A new solver function that runs s1 then s2

Examples

```
# Coarse-to-fine: grid search to find good region, then gradient ascent
strategy <- grid_search(lower = c(-10, 0.1), upper = c(10, 5), n = 5) %>>%
  gradient_ascent()

# Three-stage refinement
strategy <- grid_search(lower = c(-10, 0.1), upper = c(10, 5), n = 3) %>>%
  gradient_ascent() %>>%
  newton_raphson()
```

Index

%>>%, *10*, *22*, 36

bfgs, 3, *10*

chain, 4, *6*
clear_cache, 5, *18*
compose_transforms, 5
coordinate_ascent, 6

fisher_scoring, 7, *22*

get_fisher, 8
get_score, 9
gradient_ascent, *7*, 10, *22*, *31*
grid_search, 11

is_converged, 12
is_mle_constraint, 13
is_mle_problem, 13
is_solver_result, 14
is_tracing, 14

lbfgsb, 15

mle_constraint, 16, *18*
mle_problem, 17
mle_trace, 19

nelder_mead, *7*, 20
newton_raphson, *8*, *10*, 21
normal_sampler, 22
num_iterations, 23

optimization_path, 23

penalty_elastic_net, 24
penalty_l1, 25
penalty_l2, 26
plot.mle_trace_data, 26
plot.solver_result, 27
print.mle_problem *(*mle_problem*)*, 17
print.mle_trace *(*mle_trace*)*, 19
print.mle_trace_data, 28

race, 28, *29*
race_operator, 29
random_search, 30

sim_anneal, 31

uniform_sampler, 32
unless_converged, 33
update.mle_problem, 33

with_penalty, 34
with_restarts, *31*, 34
with_subsampling, 35