Package 'episode'

October 29, 2017

Type Package

Title Estimation with Penalisation in Systems of Ordinary Differential Equations

Version 1.0.0

Maintainer Frederik Vissing Mikkelsen <mifretz@gmail.com>

Description

A set of statistical tools for inferring unknown parameters in continuous time processes governed by ordinary differential equations (ODE). Moreover, variance reduction and model selection can be obtained through various implemented penalisation schemes. The package offers two estimation procedures: exact estimation via least squares and a faster approximate estimation via inverse collocation methods. All estimators can handle multiple data sets arising from the same ODE system, but subjected to different interventions.

License GPL-3

LazyData TRUE

Imports Matrix (>= 1.2-9), glmnet (>= 2.0-10), Rcpp (>= 0.12.11), methods (>= 3.4), stats (>= 3.4), nnls (>= 1.4)

Depends R (>= 3.4.0)

LinkingTo Rcpp, RcppArmadillo

SystemRequirements GNU make

RoxygenNote 6.0.1

Suggests testthat, knitr, rmarkdown, devtools

VignetteBuilder knitr

NeedsCompilation yes

Author Frederik Vissing Mikkelsen [aut, cre]

Repository CRAN

Date/Publication 2017-10-29 14:27:39 UTC

R topics documented:

`aim` *Adaptive Integral Matching (AIM)*

Description

Gives approximate parameter estimates using integral matching and optionally adapts weights and scales to these. Feed this to `rodeo` for initialising exact estimation.

Usage

```
aim(o, op, x = NULL, adapts = c("scales", "penalty_factor"), xout = FALSE,
  params = NULL, ...)
```

Arguments

o	An object of class ode (created via mak, plk, etc.).
op	An object of class opt (created via opt). Compatibility check with o is conducted.
x	Matrix of dimension mx(d+1) containing custom time and smoothed values of the processes used for the integral matching, see details. If NULL (default) a linear-interpolation smoother is employed.
adapts	Character vector holding names of what quantities to adapt during algorithm. Possible quantities are: "scales", "weights" and "penalty_factors", see details. Default is "scales" and "penalty_factor".
xout	Logical indicating if a matrix containing the process values at the time points in y in op, linearly interpolated from x should be returned. Default is FALSE.
params	List of vectors of initial parameter values. Default (NULL) translates to the zero-vector. If the ODE system is linear in the parameter vector and the boundary constraints on are not unusual, then params is ignored.
...	Additional arguments passed to aim.

Details

Loss function: Integral matching requires a smoothed process in order to approximate the parameter estimates. More precisely, the loss function

$$RSS/(2 * (n - s)) + lambda * penalty$$

is minimised, where RSS is the sum of the squared 2-norms of

$$x_i - x_{i-1} - \int_{t_{i-1}}^{t_i} f(x(s), context_l * param)ds$$

Here f is the vector field of the ODE-system, x is the smoothed process and param is (internally) scaled with scales in reg.

Custom smoother: The supplied x is a way of customising how x in the loss function is made. Firstly, x must have similiar layout as y in op, i.e., first column is time and the remaining columns contain smoothed values of the process at those time points, see opt-documentation for details.

The number of decreases in time in x must match the number of decreases in time in y in op. The process x in the loss function is simply a linear interpolation of x, hence finer discretisations give more refined integral matching. Each context is handled seperately. Moreover, the compatibility checks in rodeo are also conducted here.

Adaptations: The adapts are ways to adapt quantities in the opt-object and reg-objects in o to the data:

"scales" A standardisation of the columns in the linearisation takes place and carry over to scales in reg-objects in o (generally recommended and default).

"weights" The observational weights (weights in op) are adjusted coordinate-for-coordinate (column-wise) by reciprocal average residual sum of squares across penalty parameters.

"penalty_factor" The penalty factors in reg are adjusted by the reciprocal average magnitude of the estimates (parameters whose average magnitude is 0 join exclude). For extremely large systems, this option can heavily reduce further computations if the returned aim-object is passed to rodeo.

If either "penalty_factor" or "weights" are in adapts a refitting takes place.

Finally, note that lambda and nlambda in returned opt-object may have changed.

Value

An object with S3 class "aim":

o	Original ode-object with adapted quantities.
op	Original opt-object with adapted quantities and default values for lambda_min_ratio, lambda and (if needed) a inserted, if these were originally NULL.
params	A list of matrices (of type dgCMatrix) with the parameter estimates (scaled by scales in reg), one column for each lambda value. One element in the list for each parameter (other than initial state parameter).
x0s	A matrix with the initial state estimates.
rss	A vector of residual sum of squares.
x	Original x, or y in op if x was NULL.
xout	(If xout = TRUE) A matrix containing the values of the process at the time points in y in op, interpolated from x. Use it to check that x works as intended.

See Also

rodeo, rodeo.ode, rodeo.aim, imd

Examples

```
set.seed(123)
# Michaelis-Menten system with two 0-rate reactions
A <- matrix(c(1, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 1, 0,
  0, 0, 1, 1,
  0, 0, 1, 1), ncol = 4, byrow = TRUE)
B <- matrix(c(0, 0, 1, 0,
  1, 1, 0, 0,
  1, 0, 0, 1,
  0, 1, 0, 0,
  1, 0, 0, 1), ncol = 4, byrow = TRUE)
k <- c(0.1, 1.25, 0.5, 0, 0); x0 <- c(E = 5, S = 5, ES = 1.5, P = 1.5)
Time <- seq(0, 10, by = 1)

# Simulate data, in second context the catalytic rate has been inhibited
contexts <- cbind(1, c(1, 1, 0, 1, 1))
m <- mak(A, B, r = reg(contexts = contexts))
y <- numsolve(m, c(Time, Time), cbind(x0, x0 + c(2, -2, 0, 0)), contexts * k)
```

```
y[, -1] <- y[, -1] + matrix(rnorm(prod(dim(y[, -1])), sd = .25), nrow = nrow(y))

# Create optimisation object via data
o <- opt(y, nlambda = 10)

# Fit data using Adaptive Integral Matching on mak-object
a <- aim(m, o)
a$params$rate

# Compare with true parameter on column vector form
matrix(k, ncol = 1)


# Example: Power Law Kinetics
A <- matrix(c(1, 0, 1,
  1, 1, 0), byrow = TRUE, nrow = 2)
p <- plk(A)
x0 <- c(10, 4, 1)
theta <- matrix(c(0, -0.25,
  0.75, 0,
  0, -0.1), byrow = TRUE, nrow = 3)
Time <- seq(0, 1, by = .025)

# Simulate data
y <- numsolve(p, Time, x0, theta)
y[, -1] <- y[, -1] + matrix(rnorm(prod(dim(y[, -1])), sd = .25), nrow = nrow(y))

# Estimation
a <- aim(p, opt(y, nlambda = 10))
a$params$theta

# Compare with true parameter on column vector form
matrix(theta, ncol = 1)


# Example: use custom lowess smoother on data
# Smooth each coordinate of data to get curve
# on extended time grid
ext_time <- seq(0, 1, by = 0.001)
x_smooth <- apply(y[, -1], 2, function(z) {
  # Create loess object
  data <- data.frame(Time = y[, -1], obs = z)
  lo <- loess(obs ~ Time, data)

  # Get smoothed curve on extended time vector
  predict(lo, newdata = data.frame(Time = ext_time))
})

# Bind the extended time
x_smooth <- cbind(ext_time, x_smooth)

# Run aim on the custom smoothed curve
a_custom <- aim(p, opt(y), x = x_smooth)
```

episode — *Estimation with Penalisation In Systems of Ordinary Differential Equations.*

Description

This package provide the tools for approximate and exact parameter estimation in ODE models with regularisation via penalisation.

Specify your ODE

The ode system is specified via the `ode`-subclasses: `mak`, `plk`, `rlk` and `ratmak`. In creating these you can also specify numerical solver type via `solver` and regularisation type of the parameter arguments via `reg`. To numerically solve the ODE use `numsolve` and to evaluate the ODE field use `field`. The differentials of both quantities can also be evaluated.

Specify loss function

To specify the loss function use the `reg` in the ODE object to control regularisation and `opt` to control the observations, their weights and the tuning parameter of the regularisation.

Optimise the loss function

Having an `ode` object and an `opt` object, there are two methods for estimating the parameters: approximate estimation via inverse collocation methods, `aim`, or exact estimation via interior point methods, `rodeo`. If desired, call `rodeo` on the results from `aim` to use the approximate estimates for initialising the exact estimation.

field — *Field of Ordinary Differential Equation (ODE) systems.*

Description

Evaluates the vector field of the ODE system specified in the `ode` object (and optionally its differentials with respect to the state and parameter vectors).

Usage

```
field(o, x, param, differentials = FALSE, ...)
```

Arguments

o	An object of class ode (created via mak or plk).
x	A non-negative numeric vector holding the state of the ODE system.
param	A non-negative numeric vector of parameter values for the ODE system (or list of these if multiple arguments are required).
differentials	Logical indicating if the differentials with respect to the state and parameter vector ought to be returned as well.
...	Additional arguments passed to field.

Value

If differentials is FALSE a vector (of length d) holding the evaluated value of the field.

If differentials is TRUE a list is returned with

f	A vector (length d) holding the evaluated value of the field.
f_dx	A sparse matrix (dxd) holding the state differential of the field.
f_dparam	A (list of) sparse matrix (dxp) holding the parameter differential of the field.

See Also

ode, mak, plk, rlk, ratmak

Examples

```
# Example: Michaelis-Menten system
A <- matrix(
c(1, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 1, 0), ncol = 4, byrow = TRUE)
B <- matrix(
c(0, 0, 1, 0,
  1, 1, 0, 0,
  1, 0, 0, 1), ncol = 4, byrow = TRUE)
k <- c(1, 2, 0.5)
x <- c(E = 1, S = 4, ES = 0, P = 0)

m <- mak(A, B)

# Vector field
field(m, x, k)

# ... with differentials
field(m, x, k, TRUE)


# Example: Power law kinetics
A <- matrix(c(1, 0, 1,
              1, 1, 0), byrow = TRUE, nrow = 2)
p <- plk(A)
```

```
x <- c(10, 4, 1)
theta <- matrix(c(0, -0.25,
                  0.75, 0,
                  0, -0.1), byrow = TRUE, nrow = 3)

# Vector field
field(p, x, theta)

# ... with differentials
field(p, x, theta, TRUE)
```

`imd` *Integral Matching Design*

Description

Get the design matrix, formated data and weights used in integral matching.

Usage

```
imd(o, op, x = NULL, params = NULL)
```

Arguments

`o`	An object of class `ode` (created via `mak`, `plk`, etc.).
`op`	An object of class `opt` (created via `opt`). Compatibility check with `o` is conducted.
`x`	Matrix of dimension m-x-(d+1) containing custom time and smoothed values of the processes used for the integral matching, see details. If `NULL` (default) a linear-interpolation smoother is employed.
`params`	List of vectors of initial parameter values. Default (`NULL`) translates to the zero-vector. If the ODE system is linear in the parameter vector and the boundary constraints on are not unusual, then `params` is ignored.

Details

The design matrix is as follows:

$$X = (\int_{t_{i-1}}^{t_i} \frac{df}{dparam}(x(s), context_l * param) * context_l ds)_{i=1}^{n}$$

Here f is the vector field of the ODE-system, x is the smoothed process and params is (internally) scaled with `scales` in `reg` objects in `o`.

Similiarly the observations and weights are concatinateed as follows:

$$Y = (x_{t_i} - x_{t_{i-1}})_{i=1}^{n}$$

$$W = ((w_{t_i} + w_{t_{i-1}})/2)_{i=1}^{n}$$

The number of decreases in time in x must match the number of decreases in time in y in op. The process x is simply a linear interpolation of x, hence finer discretisations give more refined integral matching. Each context is handled seperately. Moreover, the compatibility checks in rodeo are also conducted here.

Value

A list with:

X	List of design matrices, one for each parameter argument.
Y	A vector of observations.
W	A vector of weights.
X0	A matrix of initial states, one for each context. Interpolated from x.
xout	A matrix containing the values of the process at the time points in y, interpolated from x. Use it to check that x works as intended.

See Also

aim, rodeo.aim, numint

Examples

```
set.seed(123)
# Michaelis-Menten system with two 0-rate reactions
A <- matrix(c(1, 1, 0, 0,
              0, 0, 1, 0,
              0, 0, 1, 0,
              0, 1, 0, 0,
              0, 0, 0, 1), ncol = 4, byrow = TRUE)
B <- matrix(c(0, 0, 1, 0,
              1, 1, 0, 0,
              1, 0, 0, 1,
              0, 0, 0, 1,
              0, 0, 1, 0), ncol = 4, byrow = TRUE)
k <- c(2.1, 2.25, 1.5, 0, 0); x0 <- c(E = 8, S = 10, ES = 1.5, P = 1.5)
Time <- seq(0, 10, by = 1)

# Simulate data, in second context the catalytic rate has been doubled
contexts <- cbind(1, c(1, 1, 2, 1, 1))
m <- mak(A, B, r = reg(contexts = contexts))
y <- numsolve(m, c(Time, Time), cbind(x0, x0 + c(2, -2, 0, 0)), contexts * k)
y[, -1] <- y[, -1] + matrix(rnorm(prod(dim(y[, -1])), sd = .5), nrow = nrow(y))

# Get the design matrix used in integral matching
d <- imd(m, opt(y))
head(d$X[[1]])

# Compare with glmnet
lasso <- glmnet::glmnet(x = d$X[[1]], y = d$Y, intercept = FALSE, lower.limits = 0)
```

```
a <- aim(m, opt(y, nlambda = 100), adapts = "scales")
all.equal(lasso$beta, a$params$rate)
```

---

mak *Create 'mak' (Mass Action Kinetics) object*

Description

This function creates an object of class mak (subclass of ode), which holds the basic information of the Mass Action Kinetics system in question.

Usage

```
mak(A, B, s = solver(), r = NULL, rx0 = reg("none", lower = 0, upper =
  Inf, fixed = TRUE))
```

Arguments

A	The reactant stoichiometric matrix (pxd) containing non-negative values. Here p is the number of parameters and d the number of species.
B	The product stoichiometric matrix (pxd) containing non-negative values. Here p is the number of parameters and d the number of species.
s	solver object.
r	An object of class reg giving info about how to regularise and bound the rate parameters. If not provided, the default one is used.
rx0	An object of class reg giving info about how to regularise and bound the initial state parameter. If not provided, the default one is used. This default reg sets fixed = TRUE, which is generally recommended.

Details

Mass Action Kinetics is a class of ODE systems, having the following vector field:

$$\frac{dx}{dt} = (B - A)^T diag(x^A)k$$

with $x^A = (\prod_{i=1}^{d} x_i^{A_{ji}})_{j=1}^{p}$ and k estimatable and non-negative.

Value

An object with S3 class "mak" and "ode".

See Also

ode, numsolve, field

Examples

```
# Michaelis-Menten system
A <- matrix(
c(1, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 1, 0), ncol = 4, byrow = TRUE)
B <- matrix(
c(0, 0, 1, 0,
  1, 1, 0, 0,
  1, 0, 0, 1), ncol = 4, byrow = TRUE)
k <- c(1, 2, 0.5)
x0 <- c(E = 1, S = 4, ES = 0, P = 0)
Time <- seq(0, 1, by = .1)

m <- mak(A, B)

# Solve system
numsolve(m, Time, x0, k)

# Evaluate field
field(m, x0, k)
```

`numint` *Numerical integration of powers and fractions of powers via simpson rule*

Description

Evaluates numerical integrals of powers or fractions of powers of a d-dimensional function x.

Usage

```
numint(time, x, type, A, B)
```

Arguments

`time`	Vector (n) holding time points in between which the integrals are evaluated. Must be one series only (i.e., no decrements).
`x`	Matrix (mx(d+1)) holding discretised function. First column is time, must have no decrements. The remaining columns are function values, which will be interpolated.
`type`	String telling type, must be "power" or "fracpower".
`A`	Matrix (pxd) holding powers in rows.
`B`	Matrix (pxd) holding powers in rows.

Value

A matrix of dimension (m-1)xp holding the row-concatinated blocks of integrals: If `type` is "power" it evaluates the numerical integrals

$$(\int_{t_i}^{t_{i+1}} x(s)^A)_i$$

where t_i are entries in first column of `x` and $x(s)$ is constructed via a linear interpolation of `x`. If `type` is "fracpower" it evaluates the numerical integrals

$$(\int_{t_i}^{t_{i+1}} x(s)^A/(1+x(s)^B))_i$$

where the fraction is evaluated coordinate wise.

The numerical integration uses the simpson rule using the intermediate time points in `time` in between any two consecutive time points in the first column of `x`. To get more accurate integrals include more intermediate time points in `time`.

See Also

imd, aim

Examples

```
# Trajectory of power law kinetics system
A <- matrix(c(1, 0, 1,
              1, 1, 0), byrow = TRUE, nrow = 2)
p <- plk(A)
x0 <- c(10, 4, 1)
theta <- matrix(c(0, -0.25,
                  0.75, 0,
                  0, -0.1), byrow = TRUE, nrow = 3)
Time <- seq(0, 1, by = .025)
traj <- numsolve(p, Time, x0, theta)

# Example: Integrate traj(s)^A between the time points in 'ti'
ti <- seq(0, 1, by = .1)
ss <- numint(time = ti, x = traj, type = "power", A = A, B = A)

# Example: Integrate traj(s)^A / (1 + traj(s)^B) between the time points in 'ti'
B <- matrix(c(0, 2, 1,
              2, 1, 0), byrow = TRUE, nrow = 2)
ss <- numint(time = ti, x = traj, type = "fracpower", A = A, B = B)
```

numsolve *Numerical solver for Ordinary Differential Equation (ODE) systems.*

Description

Calculates numerical solution to the ODE system specified in the ode object and returns the solution path (and optionally its derivatives).

Usage

```
numsolve(o, time, x0, param, approx_sensitivity = NULL, ...)
```

Arguments

o	An object of class ode (created via mak or plk).
time	A numeric vector holding desired time-points at which to evaluate the solution path. By convention of this package, whenever time decreases the parameters and initial conditions are reset, see details.
x0	A non-negative numeric vector or matrix holding the initial conditons for the ODE system.
param	A non-negative numeric vector or matrix holding the parameter values for the ODE system (or a list of these if the ode system has multiple arguments, like ratmak).
approx_sensitivity	Logical or NULL. If NULL (default) the sensitivity equations are not solved and no derivatives are returned. Otherwise if logical, it will solve the sensitivity equations. If TRUE an approximate solution is returned, else a more exact solution (based on the numerical solver) is returned.
...	Additional arguments passed to numeric solver.

Details

As mentioned above, whenever time decreases the system restarts at a new initial condition and with a new parameter vector, called a context/environment. The number of contexts is s, i.e., the number of decreases + 1. To support these s contexts, the new initial conditions and parameter vectors can be supplied through param and x0. For both of these, either a matrix form holding the new initial conditions or parameters in the columns or a concatinated vector will work. In either case param and x0 are coarced to matrices with p (= number of parameters) and d (= dimension of state space) rows respectively (with p and d extracted from o). Hence a check of whether param and x0 have length divisible by p and d respectively is conducted. Both parameter vectors and initial conditions will be recycled if s exceeds the number of these divisors. Therefore, if the same parameter vector and/or initial condition is desired across resets, supply only that vector. A warning will be thrown if p or d is neither a multiple nor a sub-multiple of s.

Value

If approx_sensitivity is NULL a matrix with the first column holding the time vector and all consecutive columns holding the states. A 0/1 convergence code of the solver is given as an attribute.

If approx_sensitivity is logical a list is returned with

trajectory A matrix containing the trajectory of the system (as with approx_sensitivity = NULL).

sensitivity_param A list with arrays of sensitivity solutions of parameters (row = time, colum = state, slice = parameter coordinate).

sensitivity_x0 An array with sensitivity solutions of initial state (row = time, colum = state, slice = initial state coordinate).

For a convergence code of 1, try decreasing tol or increasing step_max in solver object in ode object. Note that the sensitivity equations may have non-finites, even if the convergence code is 0. This is typically due to zero initial states at which the state derivative of the field may be undefined.

See Also

ode, mak, plk, rlk, ratmak

Examples

```
# Example: Michaelis-Menten system
A <- matrix(
c(1, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 1, 0), ncol = 4, byrow = TRUE)
B <- matrix(
c(0, 0, 1, 0,
  1, 1, 0, 0,
  1, 0, 0, 1), ncol = 4, byrow = TRUE)
k <- c(1, 2, 0.5)
x0 <- c(E = 1, S = 4, ES = 0, P = 0)
Time <- seq(0, .5, by = .1)

m <- mak(A, B)

# Solution for one context
numsolve(m, Time, x0, k)

# Solution for two contexts (the latter with faster rate)
numsolve(m, c(Time, Time), x0, cbind(k, k * 1.5))

# Solution for two contexts (the latter with different initial condition)
numsolve(m, c(Time, Time), cbind(x0, x0 + 1.5), k)

# As above, but with sensitivity equations are solved (using approximate solution)
numsolve(m, c(Time, Time), cbind(x0, x0 + 1.5), k, TRUE)


# Example: Power law kinetics
```

```
A <- matrix(c(1, 0, 1,
              1, 1, 0), byrow = TRUE, nrow = 2)
p <- plk(A)
x0 <- c(10, 4, 1)
theta <- matrix(c(0, -0.25,
                  0.75, 0,
                  0, -0.1), byrow = TRUE, nrow = 3)
numsolve(p, Time, x0, theta)
```

---

ode *Abstract 'ode' object*

Description

The abstract class ode, which determines the ordinary differential equation system. Some systems have multiple parameter arguments (so far only ratmak). The bare ode class is abstract, hence in practice you need to use one of

mak Mass Action Kinetics reaction network system.

plk Power Law Kinetics system.

rlk Rational Law Kinetics system.

ratmak Rational Mass Action Kinetics.

to create an actual ode object, which you can use.

Usage

```
ode(...)
```

Arguments

...	Arguments passed to ode.

Value

An object with S3 class "ode".

See Also

mak, plk, rlk, ratmak

opt *Create 'opt' (optimisation) object*

Description

This function creates an object of class opt, which holds data, weights, tuning parameters and control list for optimisation. This is basically the control panel for the loss function optimised in rodeo and aim.

Usage

```
opt(y, weights = NULL, nlambda = 25, lambda_min_ratio = NULL,
  lambda_decrease = TRUE, lambda = NULL, tol_l = 1e-05)
```

Arguments

y	Numeric matrix (nx(d+1)). First column is time and remaining columns hold observations from different species, see details. Missing values (except in first column) are allowed and are marked with NA or NaN.
weights	Numeric matrix (nxd) of observation weights (optional).
nlambda	Number of lambda values.
lambda_min_ratio	Ratio between smallest and largest value of lambda (latter derived using data). If (n - s) * d > p, the default is 0.0001, else 0.01.
lambda_decrease	Logical indicating if automatically generated lambda sequence should be decreasing (default is TRUE). Consider switching to FALSE if nonzero initialisations are given to rodeo.ode.
lambda	A custom lambda sequence. If not NULL, nlambda, lambda_min_ratio and decrease.lambda are ignored. The optimisation reuses last optimal parameter value for next step, so lambda should preferably be monotonic.
tol_l	Positive numeric tolerance level used for stopping criterion (max-norm of gradients).

Details

Data format: Whenever time (first column of y) decreases, the system restarts. Hence the data is assumed generated from s different contexts, where s - 1 is the number of decreases in the time vector.

Each context has its own initial condition and parameter vector specified through contexts (see reg for details).

Loss function: The loss function optimised in rodeo is:

$$RSS/(2*(n-s)) + \lambda * \sum_{parameter\ argument} \lambda_{factor} * \sum_{j=1}^{p} v_j pen(param_j)$$

where λ belongs to the lambda-sequence and v is penalty_factor. Moreover, the residual sum of squares, RSS, is given as:

$$RSS = \sum_{i=1}^{n} ||(y(t_i) - x(t_i, x_{0l}, context_l * param)) * \sqrt{w(t_i)}||_2^2$$

where param has been (internally) scaled with scales, and $w(t_i)$ and $y(t_i)$ refers to the i'th row of weights and y (with first column removed), respectively. The solution to the ODE system is the x()-function. The subscript 'l' refers to the context, i.e., the columns of contexts in reg-object and x0 in rodeo-functions (x0 is the initial state of the system at the first time point after a decrease in the time-vector). All products are Hadamard products.

Tuning parameter: The lambda sequence can either be fully customised through lambda or automatically generated. In the former case, a monotonic lambda-sequence is highly recommended. Throughout all optimisations, each optimisation step re-uses the old optimum, when sweeping through the lambda-sequence.

If lambda is NULL, an automatically generated sequence is used. A maximal value of lambda (the smallest at which 0 is a optimum in the rate parameter) is calculated and log-equi-distant sequence of length nlambda is generated, with the ratio between the smallest and largest lambda value being lambda_min_ratio. Note: when the opt-object is passed to rodeo.ode, one may want to initialise the optimisation at a non-zero parameter and run the optimisation on the reversed lambda sequence. This is indicated by setting decrease.lambda = FALSE. If, however, the opt-object is passed to aim, glmnet ultimately decides on the lambda sequence, and may cut it short.

Optimisation: A proximal-gradient type of optimisation is employed. The step length is denoted τ. The convergence criteria is $\Delta\eta < 10^3 * \max(|(\Delta param \neq 0)| + |(\Delta x0 \neq 0)|, 1)$ AND $\Delta loss/\Delta\eta < tol_l$, where

$$\Delta\eta = ||\Delta param||/tol_{param} + ||\Delta x0||/tol_{x_0}$$

Value

An object with S3 class "opt".

See Also

aim, rodeo, rodeo.aim, rodeo.ode

Examples

```
# Generate some data
set.seed(123)
A <- matrix(c(1, 0, 1,
              1, 1, 0), byrow = TRUE, nrow = 2)
p <- plk(A)
x0 <- c(10, 4, 1)
theta <- matrix(c(0, -0.25,
                  0.75, 0,
                  0, -0.1), byrow = TRUE, nrow = 3)
Time <- seq(0, 1, by = .025)
```

```
y <- numsolve(p, Time, x0, theta)
y[, -1] <- y[, -1] + matrix(rnorm(prod(dim(y[, -1])), sd = .1), nrow = nrow(y))

# Minimally, you need to supply data:
op <- opt(y)

# More weight on early observations
w <- outer(1 / seq_len(nrow(y)), rep(1, length(x0)))
op <- opt(y, weights = w)

# Less weight on first coordinate
w <- outer(rep(1, nrow(y)), c(1, 2, 2))
op <- opt(y, weights = w)

# Adjust tuning parameter sequence
op <- opt(y, nlambda = 10, lambda_min_ratio = 0.05)
```

---

plk *Create 'plk' (Power Law Kinetics) object*

Description

This function creates an object of class `plk` (subclass of `ode`), which holds the basic information of the Power Law Kinetics system in question.

Usage

```
plk(A, s = solver(), r = NULL, rx0 = reg("none", lower = 0, upper = Inf,
  fixed = TRUE))
```

Arguments

`A`	The matrix of powers (pxd). Here d the number of species.
`s`	`solver` object.
`r`	An object of class `reg` giving info about how to regularise and bound the rate parameters. If not provided, the default one is used.
`rx0`	An object of class `reg` giving info about how to regularise and bound the initial state parameter. If not provided, the default one is used. This default `reg` sets `fixed = TRUE`, which is generally recommended.

Details

Power Law Kinetics is a class of ODE systems, having the following vector field:

$$\frac{dx}{dt} = \theta x^A$$

with $x^A = (\prod_{i=1}^{d} x_i^{A_{ji}})_{j=1}^{p}$ and θ an estimatable parameter matrix of dimension dxp. By convention theta will only be reported as a vector (concatinated column-wise).

Value

An object with S3 class "plk" and "ode".

See Also

ode, numsolve, field

Examples

```
# Power law system
A <- matrix(
c(1, 0, 0, 0,
  0, 1, 2, 0,
  0, 0, 0, 1), ncol = 4, byrow = TRUE)
theta <- matrix(
c(0, 2, -0.5, 0,
  1, 0, 0, 1,
  -1, -1, -1, -1), ncol = 3, byrow = TRUE)
x0 <- c(X = 1, Y = 4, Z = 0.1, W = 0.1)
Time <- seq(0, 1, by = .1)

p <- plk(A)

# Solve system
numsolve(p, Time, x0, theta)

# Evaluate field
field(p, x0, theta)
```

---

print.mak *Print 'mak' object*

Description

Prints objects of class mak, including visualising the reactions.

Usage

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

Arguments

x	Object of class mak.
...	Additional arguments passed to print.

See Also

mak

Examples

```
# Michaelis-Menten system
A <- matrix(
c(1, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 1, 0), ncol = 4, byrow = TRUE)
B <- matrix(
c(0, 0, 1, 0,
  1, 1, 0, 0,
  1, 0, 0, 1), ncol = 4, byrow = TRUE)
m <- mak(A, B)
m
```

---

print.ode *Print 'ode' object*

Description

Prints objects of class ode. Since this class is abstract, it moves on the print method for the derived class.

Usage

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

Arguments

x	Object of class ode.
...	Additional arguments passed to print.

See Also

ode

print.plk *Print 'plk' object*

Description

Prints objects of class plk, including visualising the powers.

Usage

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

Arguments

x	Object of class plk.
...	Additional arguments passed to print.

See Also

plk

Examples

```
# Power law system
A <- matrix(
c(1, 0, 0, 0,
  0, 1, 2, 0,
  0, 0, 0, 0), ncol = 4, byrow = TRUE)
p <- plk(A)
p
```

print.ratmak *Print 'ratmak' object*

Description

Prints objects of class ratmak, and presents the powers.

Usage

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

Arguments

x	Object of class ratmak.
...	Additional arguments passed to print.

See Also

ratmak

Examples

```
# Rational mass action kinetics
A <- matrix(
c(1, 0, 0, 0,
  0, 1, 2, 0,
  1, 0, 0, 1), ncol = 4, byrow = TRUE)
x0 <- c(X = 1, Y = 4, Z = 0.1, W = 0.1)
time <- seq(0, 1, by = .1)

rmak <- ratmak(A, diag(4))
rmak
```

`print.reg` *Print 'reg' object*

Description

Prints objects of class `reg`.

Usage

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

Arguments

`x`	Object of class `reg`.
`...`	Additional arguments passed to `print`.

See Also

reg

Examples

```
# Example: default regularisation
reg()
```

print.rlk *Print 'rlk' object*

Description

Prints objects of class `rlk` and presents the fractions.

Usage

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

Arguments

`x`	Object of class `rlk`.
`...`	Additional arguments passed to `print`.

See Also

rlk

Examples

```
# Rational law kinetics
A <- matrix(
c(1, 0, 0, 0,
  0, 1, 2, 0,
  0, 0, 0, 1), ncol = 4, byrow = TRUE)
r <- rlk(A, A[c(2, 1, 3), ])
r
```

print.solver *Print 'solver' object*

Description

Prints objects of class `solver`.

Usage

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

Arguments

`x`	Object of class `solver`.
`...`	Additional arguments passed to `print`.

See Also

solver

Examples

```
# Dormand-Prince 4/5 scheme
solver("dp45")

# Runge-Kutta-Fehlberg 4/5 scheme (default)
solver()
```

`ratmak` *Create 'ratmak' (Rational Mass Action Kinetics) object*

Description

This function creates an object of class `ratmak` (subclass of `ode`), which holds the basic information of the Rational Mass Action Kinetics system in question.

Usage

```
ratmak(A, C, s = solver(), r1 = NULL, r2 = NULL, rx0 = reg("none", lower
  = 0, upper = Inf, fixed = TRUE))
```

Arguments

`A`	The matrix of powers (bxd). Here d the number of species.
`C`	The stoichiometric coefficient matrix (rxd). Here r is the number of reactions.
`s`	`solver` object.
`r1`	An object of class `reg` giving info about how to regularise and bound the first parameter. If not provided, the default one is used.
`r2`	An object of class `reg` giving info about how to regularise and bound the second parameters. If not provided, the default one is used.
`rx0`	An object of class `reg` giving info about how to regularise and bound the initial state parameter. If not provided, the default one is used. This default `reg` sets `fixed = TRUE`, which is generally recommended.

Details

Rational Mass Action Kinetics is a class of ODE systems, having the following vector field:

$$\frac{dx}{dt} = C^T * (\theta_1 * x^A)/(1 + \theta_2 * x^A)$$

with $x^A = (\prod_{i=1}^{d} x_i^{A_{ji}})_{j=1}^{b}$, θ_1 and θ_2 estimatable parameter matrices of dimension rxb. θ_2 is restricted to non-negative values. The fraction is entry-wise. By convention the theta's will only be reported as vectors (concatinated column-wise).

Value

An object with S3 class "ratmak" and "ode".

See Also

ode, numsolve, field

Examples

```
# Rational mass action kinetics
A <- matrix(
c(1, 0, 0, 0,
  0, 1, 2, 0,
  1, 0, 0, 1), ncol = 4, byrow = TRUE)
x0 <- c(X = 1, Y = 4, Z = 0.1, W = 0.1)
time <- seq(0, 1, by = .1)

rmak <- ratmak(A, diag(4))

# Solve system
numsolve(o = rmak, time = time, x0 = x0,
  param = list(theta1 = t(A * 1:3),
      theta2 = t((A + 1) * 3:1)))

# Evaluate field
field(rmak, x0,
  param = list(theta1 = t(A * 1:3),
      theta2 = t((A + 1) * 3:1)))
```

---

reg *Create 'reg' (regularisation) object*

Description

This function creates an object of class `reg`, which holds regularisation type, tuning parameter scales, penalty factors and control list for optimisation. This is basically the control panel for the optimisation in `rodeo` and `aim`.

Usage

```
reg(reg_type = "l1", a = NULL, lower = -Inf, upper = Inf,
  lambda_factor = 1, exclude = NULL, penalty_factor = NULL,
  contexts = NULL, scales = NULL, fixed = FALSE, screen = NULL,
  exact_gradient = NULL, step_max = 200,
  step_screen = ceiling(step_max/10), step_cycle = step_screen,
  backtrack_max = 50, tol = 1e-05, tau_init = 0.1, tau_min = 1e-10,
  tau_scale = 0.5)
```

Arguments

reg_type	Character string determining the regularisation. Must be one of: "l1" (default), "l2", "elnet", "scad", "mcp" or "none". See details.
a	Numeric value of tuning parameter in elnet (must be between 0 and 1, default is 0.5), scad (must be larger than 2, default = 3.7) or mcp (must be larger than 1, default = 3).
lower	Either numeric vector of length 1 or p, of lower limit(s) for parameters, must be smaller or equal to upper.
upper	Either numeric vector of length 1 or p, of upper limit(s) for parameters, must be larger or equal to lower.
lambda_factor	Non-negative numeric value which will be multiplied with the tuning parameter in opt.
exclude	A vector indices to exclude from model (this is how to specify an infinite penalty factor). Default is none.
penalty_factor	A non-negative vector (p) of individual penalty weights. Defaults to 1 for each parameter. Will always be rescaled so its mean is 1.
contexts	A non-negative matrix (pxs) specifying design on context-level, see details. Defaults to a matrix of ones.
scales	A positive vector (p) of scales, see details. Defaults to a vector of ones. Will be rescaled to mean to 1.
fixed	Logical indicating if this parameter is fixed or should be optimised.
screen	Logical indicating if a faster optimisation relying on screening should be adapted. If NULL, it is set to TRUE iff p > 20.
exact_gradient	Logical indicating if exact gradient should be used. If NULL, it is set to TRUE iff p > 20.
step_max	Positive integer giving the maximal number of steps in optimisation procedure, per lambda.
step_screen	Positive integer giving the number of steps between screenings (defaults to ceiling(step_max / 50)).
step_cycle	Positive integer giving the number of steps per optimisation cycle, defaults to step_screen.
backtrack_max	Positive integer giving the maximal number of backtracking steps taken in each optimisation step.
tol	Positive numeric tolerance level used for parameter space.
tau_init	Positive initial step length for backtracking.
tau_min	Positive numeric giving minimal value of step length in backtracking.
tau_scale	Scaling parameter of step length, must be in (0,1).

Details

Data format: The data (y in opt) is assumed generated from s different contexts. A new context begins whenever the time (the first column of y in opt) decreases. Hence s - 1 is the number of decreases in the time points.

Each context has its own initial condition and parameter vector specified through contexts in reg. More precisely, the effective parameter in context l is the element-wise product of a baseline parameter (scaled by scales in reg) and the l'th column of contexts. This enables the user to pre-specify different scales for each parameter, as well as different scales for the same parameter across contexts.

The default setup sets contexts to a matrix of ones. The following are examples of case-specific modifications for the mak ODE class: If reaction j does not take place in context l then set $contexts_{j,l} = 0$. If reaction j has a 50% increase in rate in context l then set $contexts_{j,l} = 1.5$. If reaction j has independent rates in contexts l and l', then write that reaction twice in mak-object (with j' denoting its second appearance) and set $contexts_{j,l'} = 0$ and $contexts_{j',l} = 0$.

Loss function: The loss function optimised in rodeo is:

$$RSS/(2 * (n - s)) + \lambda * \sum_{parameter\ argument} \lambda_{factor} * \sum_{j=1}^{p} v_j pen(param_j)$$

where λ belongs to the lambda-sequence in opt-object and v is penalty_factor in reg. Moreover, the residual sum of squares, RSS, is given as:

$$RSS = \sum_{i=1}^{n} ||(y(t_i) - x(t_i, x_{0l}, context_l * param)) * \sqrt{w(t_i)}||_2^2$$

where param has been (internally) scaled with scales, and $w(t_i)$ and $y(t_i)$ refers to the i'th row of weights and y in opt (y with first column removed), respectively. The solution to the ODE system is the x()-function. The subscript 'l' refers to the context, i.e., the columns of contexts and x0 in rodeo-functions (x0 is the initial state of the system at the first time point after a decrease in the time-vector). All products are Hadamard products.

Regularisation: The type of regularisation is chosen via reg_type in reg:

l1: Least Absolute Shrinkage and Selection Operator (Lasso) penalty. The penalty is the absolute value: $pen(param_j) = |param_j|$

l2: Ridge penalty. The penalty is the squaring: $pen(param_j) = param_j^2/2$

elnet: Elastic Net. A convex combination of l1 and l2 penalties: $pen(param_j) = (1 - a)param_j^2/2 + a|param_j|$, 0<=a<=1. Note if a = 0 or a = 1, then the penalty is automatically reduced to "l2" and "l1" respectively.

scad: Smoothly Clipped Absolute Deviation penalty:

$$pen(param_j) = \int_0^{param_j} min(max((a\lambda - |param|)/(\lambda(a-1)), 0), 1) dparam, a > 2.$$

mcp: Maximum Concave Penalty penalty:

$$pen(param_j) = \int_0^{param_j} max(1 - |param|/(\lambda a), 0) dparam, a > 1.$$

none: No penalty. Not recommended for large systems. This overwrites both user-supplied and automatically generated lambda-sequences.

Optimisation: A proximal-gradient type of optimisation is employed. The step length is denoted τ.

The flag screen (in reg) is passed onto the optimisation, it simply tells the optimisation to focus entirely on optimising a subset of the parameters, which where selected due to large gradients. At most step_screen (in reg) steps are taken before a re-evaluation of the screened subset takes place.

The convergence criteria is $\Delta\eta < 10^3 * \max(|(\Delta param \neq 0)| + |(\Delta x0 \neq 0)|, 1)$ AND $\Delta loss/\Delta\eta < tol_l$, where

$$\Delta\eta = ||\Delta param||/tol_{param} + ||\Delta x0||/tol_{x_0}$$

Tuning parameter: The lambda sequence can either be fully customised through lambda in opt or automatically generated. In the former case, a monotonic lambda-sequence is highly recommended. Throughout all optimisations, each optimisation step re-uses the old optimum, when sweeping through the lambda-sequence.

If lambda is NULL, an automatically generated sequence is used. A maximal value of lambda (the smallest at which 0 is a optimum in the rate parameter) is calculated and log-equidistant sequence of length nlambda is generated, with the ratio between the smallest and largest lambda value being lambda.min.ratio. Note: when the opt-object is passed to rodeo.ode, one may want to initialise the optimisation at a non-zero parameter and run the optimisation on the reversed lambda sequence. This is indicated by setting decrease.lambda = FALSE. If, however, the opt-object is passed to aim, glmnet ultimately decides on the lambda sequence, and may cut it short.

Gradient evaluations: Two gradient evaluations are implemented: exact and approximate. The computational time of exact gradient is generally longer than that of approximate gradient, but its relative magnitude depends on the solver in the ode-object passed to rodeo.

Value

An object with S3 class "reg".

See Also

aim, rodeo, rodeo.aim, rodeo.ode

Examples

```
# Use 'reg' object to specify regularisation when creating 'ode' objects

# Example: power law kinetics with SCAD penalty on rate parameter
A <- matrix(
c(1, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 1, 0), ncol = 4, byrow = TRUE)
p <- plk(A, r = reg("scad"))

# Example: power law kinetics as above, with lower bound of -1
```

```
p <- plk(A, r = reg("scad", lower = -1))

# Example: rational mass action kinetics with
# MCP penalty on first parameter argument and l2 on second
B <- matrix(
c(0, 0, 1, 0,
  1, 1, 0, 0,
  1, 0, 0, 1), ncol = 4, byrow = TRUE)
rmak <- ratmak(A, B, r1 = reg("mcp"), r2 = reg("l2"))

# Example: mass action kinetics with
# no penalty on rate parameter and l2 on initial state
m <- mak(A, B, r = reg("none"), rx0 = reg("l2"))
```

`rlk` *Create 'rlk' (Rational Law Kinetics) object*

Description

This function creates an object of class `rlk` (subclass of ode), which holds the basic information of the Rational Law Kinetics system in question.

Usage

```
rlk(A, B, s = solver(), r = NULL, rx0 = reg("none", lower = 0, upper =
  Inf, fixed = TRUE))
```

Arguments

`A`	The matrix of powers (pxd). Here d the number of species.
`B`	The matrix of powers (pxd). Here d the number of species.
`s`	`solver` object.
`r`	An object of class `reg` giving info about how to regularise and bound the rate parameters. If not provided, the default one is used.
`rx0`	An object of class `reg` giving info about how to regularise and bound the initial state parameter. If not provided, the default one is used. This default `reg` sets `fixed = TRUE`, which is generally recommended.

Details

Rational Law Kinetics is a class of ODE systems, having the following vector field:

$$\frac{dx}{dt} = \theta(x^A/(1+x^B))$$

with $x^A = (\prod_{i=1}^{d} x_i^{A_{ji}})_{j=1}^{p}$ and θ an estimatable parameter matrix of dimension dxp. By convention theta will only be reported as a vector (concatinated column-wise).

Value

An object with S3 class "rlk" and "ode".

See Also

ode, numsolve, field

Examples

```
# Rational law kinetics
A <- matrix(
c(1, 0, 0, 0,
  0, 1, 2, 0,
  0, 0, 0, 1), ncol = 4, byrow = TRUE)
theta <- matrix(
c(0, 2, -0.5, 0,
  1, 0, 0, 1,
  -1, -1, -1, -1), ncol = 3, byrow = TRUE)
x0 <- c(X = 1, Y = 4, Z = 0.1, W = 0.1)
time <- seq(0, 1, by = .1)
r <- rlk(A, A[c(2, 1, 3), ])

# Solve system
numsolve(o = r, time = time, x0 = x0, param = theta)

# Evaluate field
field(r, x0, theta)
```

---

rodeo *Regularised Ordinary Differential Equation Optimisation (RODEO) generic*

Description

Fit the parameters (and optionally initial states) for an Ordinary Differential Equation model with data sampled across contexts. Two implementations exist:

`rodeo.ode` The raw form of rodeo, based on `ode` (created via `mak`, `plk`, etc.) and `opt`-objects. If needed, everything can be user-specified through these two objects. However, parameter initialisations are always required and no action towards adjusting weights or scales of the parameters are taken.

`rodeo.aim` The more automatic form of rodeo, where automatic parameter initialisations and (optional) adjustments of scales and weights are available through `aim`.

Usage

```
rodeo(x, ...)
```

Arguments

x	Object that specifies ode system.
...	Additional arguments passed to rodeo.

Details

For details on the loss function, optimisation, etc. See documentation of opt.

Value

An object with S3 class "rodeo":

o	Original ode-object.
op	Original opt-object with default values for lambda_min_ratio, lambda and (if needed) a inserted, if these were originally NULL.
params	Parameter estimates, stored as list of sparse column format matrices, "dgCMatrix" (or a list of those if multiple initialisations). Rows represent coordinates and columns represent the lambda value.
x0s	Initial state estimates stored in a matrix (or array). Rows represent coordinates, columns represent the lambda value and (if multiple initialisations) slices represent initialisations.
dfs	A matrix (or array, if multiple initialisations) of degrees of freedom. Row represents a parameter (the first is always the initial state parameter), columns represent lambda, slices represent initialisation, if multiple are provided.
codes	A matrix (or array) of convergence codes organised as dfs. **0:** The convergence criteria is met (see details in opt). Current estimate is probably a local minimum. **1:** Backtracking in the last iteration yields no numerical improvement, but no unsual behavior observed. Current estimate is probably a local minimum. However, if exact_gradient = FALSE in the reg-object in the ode-object, changing this may improve the code. Alternatively one can adjust backtracking via backtrack_max and tau_min in reg objects in ode object. **2:** The optimisation procedure exceeded maximal number of steps (step_max in reg objects). **3:** The last gradient was unsually large. Either the tolerances in reg objects are off or the ODE systems is very sensitive and runs over long time spans. In the latter case, initialisation(s) may have inappropriate zeros (change initialisation and/or make sure they start at smaller lambda value). **4:** The numeric ODE solver exceeded maximal number of steps. Check if supplied initial states were out of bounds, if not increase step_max (or tol) in reg-objects in ode-object.
steps	A matrix (or array) holding number of steps used in optimisation procedure. Organised as dfs.
losses	A vector (or matrix) of unpenalised losses at optimum for each lambda value (stored row-wise if multiple are provided).

penalties	A matrix (or array) of penalties for each parameter, organised as dfs.
jerr	A matrix (or array) of summary codes (for internal debugging), organised as dfs.

See Also

rodeo.aim, rodeo.ode

Examples

```
set.seed(123)
# Michaelis-Menten system with two 0-rate reactions
A <- matrix(c(1, 1, 0, 0,
 0, 0, 1, 0,
 0, 0, 1, 0,
 0, 0, 0, 1,
 1, 0, 0, 0), ncol = 4, byrow = TRUE)
B <- matrix(c(0, 0, 1, 0,
 1, 1, 0, 0,
 1, 0, 0, 1,
 0, 0, 1, 0,
 1, 0, 1, 0), ncol = 4, byrow = TRUE)
k <- c(1.1, 2.5, 0.25, 0, 0); x0 <- c(E = 2, S = 2, ES = 8.5, P = 2.5)
Time <- seq(0, 10, by = 1)

# Simulate data, in second context the catalytic rate has been doubled
contexts <- cbind(1, c(1, 1, 2, 1, 1))
m <- mak(A, B, r = reg(contexts = contexts))
y <- numsolve(m, c(Time, Time), cbind(x0, x0 + c(2, -2, 0, 0)), contexts * k)
y[, -1] <- y[, -1] + matrix(rnorm(prod(dim(y[, -1])), sd = .5), nrow = nrow(y))

# Example: fit data using rodeo on mak-object
op <- opt(y, nlambda = 10)
fit <- rodeo(m, op, x0 = NULL, params = NULL)
fit$params$rate

# Example: fit dat using rodeo on aim-object
a <- aim(m, op)
a$params$rate
fit <- rodeo(a)
fit$params$rate
```

---

rodeo.aim *Regularised Ordinary Differential Equation Optimisation (RODEO) initialised via Adaptive Integral Matching*

Description

Fit the parameters for an ODE model with data sampled across different contexts.

Usage

```
## S3 method for class 'aim'
rodeo(x, adjusts = c("lambda"), trace = FALSE, ...)
```

Arguments

x	aim-object created via aim-function.
adjusts	Character vector holding names of what quantities to adjust during algorithm. Possible quantities are: "lambda", "scales" and "weights".
trace	Logical indicating if status messages should be printed during rodeo.
...	Additional arguments passed to rodeo.

Details

The adapted quantities (scales, weights, penalty_factor) of x (returned by aim) are fed to the exact estimator rodeo. This estimator then traverses the lambda sequence in reverse order initialised in the last estimates from aim.

If desired, the quantities lambda, scales and weights are adjusted as in aim.

Value

An object with S3 class "rodeo":

o	Original ode-object.
op	Original opt-object with default values for lambda_min_ratio, lambda and (if needed) a inserted, if these were originally NULL.
params	Parameter estimates, stored as list of sparse column format matrices, "dgCMatrix" (or a list of those if multiple initialisations). Rows represent coordinates and columns represent the lambda value.
x0s	Initial state estimates stored in a matrix (or array). Rows represent coordinates, columns represent the lambda value and (if multiple initialisations) slices represent initialisations.
dfs	A matrix (or array, if multiple initialisations) of degrees of freedom. Row represents a parameter (the first is always the initial state parameter), columns represent lambda, slices represent initialisation, if multiple are provided.
codes	A matrix (or array) of convergence codes organised as dfs.

0: The convergence criteria is met (see details in opt). Current estimate is probably a local minimum.

1: Backtracking in the last iteration yields no numerical improvement, but no unsual behavior observed. Current estimate is probably a local minimum. However, if exact_gradient = FALSE in the reg-object in the ode-object, changing this may improve the code. Alternatively one can adjust backtracking via backtrack_max and tau_min in reg objects in ode object.

2: The optimisation procedure exceeded maximal number of steps (step_max in reg objects).

3: The last gradient was unsually large. Either the tolerances in `reg` objects are off or the ODE systems is very sensitive and runs over long time spans. In the latter case, initialisation(s) may have inappropriate zeros (change initialisation and/or make sure they start at smaller lambda value).

4: The numeric ODE solver exceeded maximal number of steps. Check if supplied initial states were out of bounds, if not increase `step_max` (or `tol`) in `reg`-objects in ode-object.

`steps` A matrix (or array) holding number of steps used in optimisation procedure. Organised as `dfs`.

`losses` A vector (or matrix) of unpenalised losses at optimum for each lambda value (stored row-wise if multiple are provided).

`penalties` A matrix (or array) of penalties for each parameter, organised as `dfs`.

`jerr` A matrix (or array) of summary codes (for internal debugging), organised as `dfs`.

See Also

rodeo, aim, rodeo.ode

Examples

```
set.seed(123)
# Example: Power Law Kinetics
A <- matrix(c(1, 0, 1,
  1, 1, 0), byrow = TRUE, nrow = 2)
p <- plk(A)
x0 <- c(10, 4, 1)
theta <- matrix(c(0, -0.25,
  0.75, 0,
  0, -0.1), byrow = TRUE, nrow = 3)
Time <- seq(0, 1, by = .025)

# Simulate data
y <- numsolve(p, Time, x0, theta)
y[, -1] <- y[, -1] + matrix(rnorm(prod(dim(y[, -1])), sd = .25), nrow = nrow(y))

# Estimation via aim
a <- aim(p, opt(y, nlambda = 10))
a$params$theta

# Supply to rodeo
rod <- rodeo(a)
rod$params$theta

# Compare with true parameter on column vector form
matrix(theta, ncol = 1)


# Example: include data from an intervened system
# where the first complex in A is inhibited
```

```
contexts <- cbind(1, c(0, 0, 0, 1, 1, 1))
y2 <- numsolve(p, Time, x0 + 1, theta * contexts[, 2])
y2[, -1] <- y2[, -1] + matrix(rnorm(prod(dim(y2[, -1])), sd = .25), nrow = nrow(y2))

# Estimation via aim
a <- aim(plk(A, r = reg(contexts = contexts)), opt(rbind(y, y2), nlambda = 10))
a$params$theta

# Supply to rodeo
rod <- rodeo(a)
rod$params$theta
```

`rodeo.ode` *Regularised Ordinary Differential Equation Optimisation (RODEO)*

Description

Fit the parameters (and optionally initial states) for a ordinary differential equation model with data sampled across different contexts.

Usage

```
## S3 method for class 'ode'
rodeo(x, op, x0, params, trace = FALSE, ...)
```

Arguments

`x`	ode-object created via `mak`, `plk` etc.
`op`	opt-object created via `opt`-function (compatibility check with `x` is conducted).
`x0`	A vector (or matrix) of non-negative initialisation(s) for the initial state in the optimisation problem, see details.
`params`	A list of initialisations for the parameter arguments in the optimisation problem, see details. A list of matrices if multiple initialisations are desired. Sparse matrix "dgCMatrix" is allowed.
`trace`	Logical indicating if status messages should be printed during `rodeo`.
`...`	Additional arguments passed to `rodeo`.

Details

For running a single initialisation of the optimisation procedure, supply `x0` as vector (where the initial states for the contexts are concatinated) and `params` as a list with a vector entry for each parameter. The resulting estimates are stored in matrices, with each column representing a lambda-value. The convergence codes, steps and losses are stored as vectors, one entry for each value of lambda.

For running multiple initialisations, supply `x0` as matrix with the individual initialisations as columns (each column vector as described above) and `params` as list of matrices with initialisations stored column-wise.

The initial state estimates (for the different lambda-values) are returned as a matrix (array) (row = coordinate, column = lambda-value, (slice = initialisation)) and the parameter estimates are returned as a list of (lists with) sparse matrices (row = coordinate, column = lambda-value). The convergence codes, steps and losses are stored as matrices (row = lambda, column = initialisation).

For details on the loss function, optimisation, etc. See documentation of `opt`.

If explicitly setting `x0` to `NULL`, the system uses the first observation of each context (throws error if non finites).

If explicitly setting `params` to `NULL`, 0 initialisations are used.

Value

An object with S3 class "rodeo":

`o` Original ode-object.

`op` Original opt-object with default values for `lambda_min_ratio`, `lambda` and (if needed) `a` inserted, if these were originally `NULL`.

`params` Parameter estimates, stored as list of sparse column format matrices, "dgCMatrix" (or a list of those if multiple initialisations). Rows represent coordinates and columns represent the `lambda` value.

`x0s` Initial state estimates stored in a matrix (or array). Rows represent coordinates, columns represent the `lambda` value and (if multiple initialisations) slices represent initialisations.

`dfs` A matrix (or array, if multiple initialisations) of degrees of freedom. Row represents a parameter (the first is always the initial state parameter), columns represent lambda, slices represent initialisation, if multiple are provided.

`codes` A matrix (or array) of convergence codes organised as `dfs`.

- **0:** The convergence criteria is met (see details in `opt`). Current estimate is probably a local minimum.
- **1:** Backtracking in the last iteration yields no numerical improvement, but no unsual behavior observed. Current estimate is probably a local minimum. However, if `exact_gradient` = FALSE in the `reg`-object in the `ode`-object, changing this may improve the code. Alternatively one can adjust backtracking via `backtrack_max` and `tau_min` in `reg` objects in `ode` object.
- **2:** The optimisation procedure exceeded maximal number of steps (`step_max` in `reg` objects).
- **3:** The last gradient was unsually large. Either the tolerances in `reg` objects are off or the ODE systems is very sensitive and runs over long time spans. In the latter case, initialisation(s) may have inappropriate zeros (change initialisation and/or make sure they start at smaller lambda value).
- **4:** The numeric ODE solver exceeded maximal number of steps. Check if supplied initial states were out of bounds, if not increase `step_max` (or `tol`) in `reg`-objects in `ode`-object.

steps	A matrix (or array) holding number of steps used in optimisation procedure. Organised as dfs.
losses	A vector (or matrix) of unpenalised losses at optimum for each lambda value (stored row-wise if multiple are provided).
penalties	A matrix (or array) of penalties for each parameter, organised as dfs.
jerr	A matrix (or array) of summary codes (for internal debugging), organised as dfs.

See Also

rodeo, rodeo.aim

Examples

```
set.seed(123)
# Example: Michaelis-Menten system with two 0-rate reactions
A <- matrix(c(1, 1, 0, 0,
              0, 0, 1, 0,
              0, 0, 1, 0,
              0, 1, 0, 0,
              0, 0, 0, 1), ncol = 4, byrow = TRUE)
B <- matrix(c(0, 0, 1, 0,
              1, 1, 0, 0,
              1, 0, 0, 1,
              0, 0, 0, 1,
              0, 0, 1, 0), ncol = 4, byrow = TRUE)
k <- c(1, 2, 0.5, 0, 0); x0 <- c(E = 2, S = 8, ES = 0.5, P = 0.5)
Time <- seq(0, 10, by = 1)

# Simulate data, in second context the catalytic rate has been doubled
m <- mak(A, B)
contexts <- cbind(1, c(1, 1, 2, 1, 1))
y <- numsolve(m, c(Time, Time), cbind(x0, x0 + c(2, -2, 0, 0)), contexts * k)
y[, -1] <- y[, -1] + matrix(rnorm(prod(dim(y[, -1])), sd = .1), nrow = nrow(y))

# Fit data using rodeo on mak-object
op <- opt(y)
fit <- rodeo(m, op, x0 = NULL, params = NULL)

# Example: fit data knowing doubled catalytic rate
m_w_doubled <- mak(A, B, r = reg(contexts = contexts))
fit <- rodeo(m_w_doubled, op, x0 = NULL, params = NULL)
```

---

solver *Create 'solver' object*

Description

This function creates an object of class `solver`, which holds the basic information of numeric solver applied to the ode-systems.

Usage

```
solver(name = "rkf45", step_max = 100, tol = 1e-06, h_init = 1e-04, ...)
```

Arguments

`name`	Character string naming the ODE-solver. Must be one of: "rk23" (Runge-Kutta order 2/3), "bs23" (Bogacki-Shampine order 2/3), "dp45" (Dormand-Prince order 4/5) or "rkf45" (Runge-Kutta-Fehlberg order 4/5, default).
`step_max`	Positive integer giving the maximal number of steps the solver may take between two consecutive time points.
`tol`	Positive numeric tolerance level used for embedded pair solver.
`h_init`	Positive numeric giving initial discretisation of time interval for solver.
`...`	Additional arguments passed to `solver`.

Value

An object with S3 class "solver".

See Also

ode

Examples

```
# Use 'solver' object to specify numerical solver when creating 'ode' objects

# Example: power law kinetics with Dormand-Prince order 4/5 solver
A <- matrix(
c(1, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 1, 0), ncol = 4, byrow = TRUE)
p <- plk(A, s = solver("dp45"))

# Example: ... and with more steps
p <- plk(A, s = solver("dp45", step_max = 1e3))

# Example: rational mass action kinetics with Runge-Kutta order 2/3 solver
B <- matrix(
c(0, 0, 1, 0,
  1, 1, 0, 0,
  1, 0, 0, 1), ncol = 4, byrow = TRUE)
rmak <- ratmak(A, B, s = solver("rk23"))
```

Index