Package 'FME'

September 9, 2016

Version 1.3.5

Title A Flexible Modelling Environment for Inverse Modelling, Sensitivity, Identifiability and Monte Carlo Analysis

Author Karline Soetaert <karline.soetaert@nioz.nl>, Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>

Maintainer Karline Soetaert <karline.soetaert@nioz.nl>

Depends R (>= 2.6), deSolve, rootSolve, coda

Imports minpack.lm, MASS, graphics, grDevices, stats, utils, minqa

Suggests diagram

Description Provides functions to help in fitting models to data, to perform Monte Carlo, sensitivity and identifiability analysis. It is intended to work with models be written as a set of differential equations that are solved either by an integration routine from package 'deSolve', or a steady-state solver from package 'rootSolve'. However, the methods can also be used with other types of functions.

License GPL (>= 2)

LazyData yes

Repository CRAN

Repository/R-Forge/Project fme

Repository/R-Forge/Revision 162

Repository/R-Forge/DateTimeStamp 2016-09-08 10:09:45

Date/Publication 2016-09-09 10:13:37

NeedsCompilation yes

R topics documented:

`FME-package` *A Flexible Modelling Environment for Inverse Modelling, Sensitivity, Identifiability, Monte Carlo Analysis.*

Description

R-package FME contains functions to run complex applications of models that produce output as a function of input parameters.

Although it was created to be used with models consisting of ordinary differential equations (ODE), partial differential equations (PDE) or differential algebraic equations (DAE), it can work with other models.

It contains:

- Functions to allow fitting of the model to data.

  Function `modCost` estimates the (weighted) residuals between model output and data, variable and model costs.

  Function `modFit` uses the output of `modCost` to find the best-fit parameters. It provides a wrapper around R's built-in minimisation routines (`optim`, `nlm`, `nlminb`) and `nls.lm` from package `minpack.lm`.

  Package FME also includes an implementation of the pseudo-random search algorithm (function `pseudoOptim`).

- Function `sensFun` estimates the sensitivity functions of selected output variables as a function of model parameters. This is the basis of uni-variate, bi-variate and multi-variate sensitivity analysis.
- Function `collin` uses as input the sensitivity functions and estimates the "collinearity" index for all possible parameter sets. This multivariate sensitivity estimate measures approximate linear dependence and is useful to derive which parameter sets are identifiable given the data set.
- Function `sensRange` produces 'envelopes' around the sensitivity variables, consisting of a time series or a 1-dimensional set, as a function of the sensitivity parameters. It produces "envelopes" around the variables.

- Function `modCRL` calculates the values of single variables as a function of the sensitivity parameters. This function can be used to run simple "what-if" scenarios
- Function `modMCMC` runs a Markov chain Monte Carlo (Bayesian analysis). It implements the delayed rejection - adaptive Metropolis (DRAM) algorithm.
- FME also contains functions to generate multiple parameter values arranged according to a grid (`Grid`) multinormal (`Norm`) or uniform (`Unif`) design, and a latin hypercube sampling (`Latinhyper`) function

Details

Package:	FME
Type:	Package
Version:	1.3.3
Date:	2016-06-20
License:	GNU Public License 2 or above

bug corrections:

changed for version 1.3.

modCost: minlogp was not correctly estimated if more than one observed variable (used the wrong sd).

Author(s)

Karline Soetaert

Thomas Petzoldt

References

Soetaert, K. and Petzoldt, T. 2010. Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME. Journal of Statistical Software 33(3) 1–28. http://www.jstatsoft.org/v33/i03

Examples

```
## Not run:
## show examples (see respective help pages for details)
example(modCost)
example(sensFun)
example(modMCMC)
example(modCRL)


## open the directory with documents
browseURL(paste(system.file(package = "FME"), "/doc", sep = ""))

## open the directory with examples
browseURL(paste(system.file(package = "FME"), "/doc/examples", sep = ""))
```

```
## the vignettes
vignette("FME")
vignette("FMEdyna")
vignette("FMEsteady")
vignette("FMEother")
vignette("FMEmcmc")

edit(vignette("FME"))
edit(vignette("FMEdyna"))
edit(vignette("FMEsteady"))
edit(vignette("FMEother"))
edit(vignette("FMEmcmc"))


## End(Not run)
```

`collin` *Estimates the Collinearity of Parameter Sets*

Description

Based on the sensitivity functions of model variables to a selection of parameters, calculates the "identifiability" of sets of parameter.

The sensitivity functions are a matrix whose (i,j)-th element contains

$$\frac{\partial y_i}{\partial \Theta_j} \cdot \frac{\Delta \Theta_j}{\Delta y_i}$$

and where y_i is an output variable, at a certain (time) instance, i, Δy_i is the scaling of variable y_i, $\Delta \Theta_j$ is the scaling of parameter Θ_j.

Function `collin` estimates the collinearity, or identifiability of all parameter sets or of one parameter set.

As a rule of thumb, a collinearity value less than about 20 is "identifiable".

Usage

```
collin(sensfun, parset = NULL, N = NULL, which = NULL, maxcomb = 5000)

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

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

Arguments

sensfun	model sensitivity functions as estimated by SensFun.
parset	one selected parameter combination, a vector with their names or with the indices to the parameters.
N	the number of parameters in the set; if NULL then all combinations will be tried. Ignored if parset is not NULL.
which	the name or the index to the observed variables that should be used. Default = all observed variables.
maxcomb	the maximal number of combinations that can be tested. If too large, this may produce a huge output. The number of combinations of n parameters out of a total of p parameters is choose(p, n).
x	an object of class collin.
...	additional arguments passed to the methods.

Details

The collinearity is a measure of approximate linear dependence between sets of parameters. The higher its value, the more the parameters are related. With "related" is meant that several paraemter combinations may produce similar values of the output variables.

Value

a data.frame of class collin with one row for each parameter combination (parameters as in sensfun).

Each row contains:

...	for each parameter whether it is present (1) or absent (0) in the set,
N	the number of parameters in the set,
collinearity	the collinearity value.

The data.frame returned by collin has methods for the generic functions print and plot.

Note

It is possible to use collin for selecting parameter sets that can be fine-tuned based on a data set. Thus it is a powerful technique to make model calibration routines more robust, because calibration routines often fail when parameters are strongly related.

In general, when the collinearity index exceeds 20, the linear dependence is assumed to be critical (i.e. it will not be possible or easy to estimate all the parameters in the combination together).

The procedure is explained in Omlin et al. (2001).

1. First the function collin is used to test how far a dataset can be used for estimating certain (combinations of) parameters. After selection of an 'identifiable parameter set' (which has a low "collinearity") they are fine-tuned by calibration.

2. As the sensitivity analysis is a *local* analysis (i.e. its outcome depends on the current values of the model parameters) and the fitting routine is used to estimate the best values of the parameters, this

is an iterative procedure. This means that identifiable parameters are determined, fitted to the data, then a newly identifiable parameter set is determined, fitted, etcetera until convergenc is reached.

See the paper by Omlin et al. (2001) for more information.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References

Brun, R., Reichert, P. and Kunsch, H. R., 2001. Practical Identifiability Analysis of Large Environmental Simulation Models. Water Resour. Res. 37(4): 1015–1030.

Omlin, M., Brun, R. and Reichert, P., 2001. Biogeochemical Model of Lake Zurich: Sensitivity, Identifiability and Uncertainty Analysis. Ecol. Modell. 141: 105–123.

Soetaert, K. and Petzoldt, T., 2010. Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME. Journal of Statistical Software 33(3) 1–28. http://www.jstatsoft.org/v33/i03

Examples

```
## =============================================================================
## Test collinearity values
## =============================================================================

## linearly related set...  => Infinity
collin(cbind(1:5, 2*(1:5)))

## unrelated set            => 1
MM <- matrix(nr = 4, nc = 2, byrow = TRUE,
  data = c(-0.400, -0.374, 0.255, 0.797, 0.690, -0.472, -0.546,  0.049))

collin(MM)

## =============================================================================
## Bacterial model as in Soetaert and Herman, 2009
## =============================================================================

pars <- list(gmax = 0.5,eff = 0.5,
             ks = 0.5, rB = 0.01, dB = 0.01)

solveBact <- function(pars) {
  derivs <- function(t, state, pars) {   # returns rate of change
    with (as.list(c(state, pars)), {
      dBact <-  gmax*eff*Sub/(Sub + ks)*Bact - dB*Bact - rB*Bact
      dSub  <- -gmax    *Sub/(Sub + ks)*Bact + dB*Bact
      return(list(c(dBact, dSub)))
    })
  }
  state   <- c(Bact = 0.1, Sub = 100)
  tout    <- seq(0, 50, by = 0.5)
  ## ode solves the model by integration...
```

```
  return(as.data.frame(ode(y = state, times = tout, func = derivs,
    parms = pars)))
}

out <- solveBact(pars)

## We wish to estimate parameters gmax and eff by fitting the model to
## these data:
Data <- matrix(nc = 2, byrow = TRUE, data =
  c(  2,  0.14,  4,  0.2,    6,  0.38,  8,  0.42,
     10,  0.6,  12,  0.107, 14,  1.3,  16,  2.0,
     18,  3.0,  20,  4.5,   22,  6.15, 24,  11,
     26, 13.8,  28, 20.0,   30,  31 ,  35, 65, 40, 61)
)
colnames(Data) <- c("time","Bact")
head(Data)

Data2 <- matrix(c(2, 100, 20, 93, 30, 55, 50, 0), ncol = 2, byrow = TRUE)
colnames(Data2) <- c("time", "Sub")


## Objective function to minimise
Objective <- function (x) {                  # Model cost
 pars[] <- x
 out   <- solveBact(x)
 Cost  <- modCost(obs = Data2, model = out)  # observed data in 2 data.frames
 return(modCost(obs = Data, model = out, cost = Cost))
}

## 1. Estimate sensitivity functions - all parameters
sF <- sensFun(func = Objective, parms = pars, varscale = 1)

## 2. Estimate the collinearity
Coll <- collin(sF)

## The larger the collinearity, the less identifiable the data set
Coll

plot(Coll, log = "y")

## 20 = magical number above which there are identifiability problems
abline(h = 20, col = "red")

## select "identifiable" sets with 4 parameters
Coll [Coll[,"collinearity"] < 20 & Coll[,"N"]==4,]

## collinearity of one selected parameter set
collin(sF, c(1, 3, 5))
collin(sF, 1:5)

collin(sF, c("gmax", "eff"))
## collinearity of all combinations of 3 parameters
collin(sF, N = 3)
```

```
## The collinearity depends on the value of the parameters:
P      <- pars
P[1:2] <- 1  # was: 0.5
collin(sensFun(Objective, P, varscale = 1))
```

`cross2long` *Convert a dataset in wide (crosstab) format to long (database) format*

Description

Rearranges a data frame in cross tab format by putting all relevant columns below each other, replicating the independent variable and, if necessary, other specified columns. Optionally, an err column is added.

Usage

```
cross2long( data, x, select = NULL, replicate = NULL,
            error = FALSE,  na.rm = FALSE)
```

Arguments

`data`	a data frame (or matrix) with crosstab layout
`x`	name of the independent variable to be replicated
`select`	a vector of column names to be included (see details). All columns are included if not specified.
`replicate`	a vector of names of variables (apart from the independent variable that have to be replicated for every included column (e.g. experimental treatment specification).
`error`	boolean indicating whether the final dataset in long format should contain an extra column for error values (cf. modCost); here filled with 1's.
`na.rm`	whether or not to remove the `NA`s.

Details

The original data frame is converted from a wide (crosstab) layout (one variable per column) to a long (database) layout (all variable value in one column).

As an example of both formats consider the data, called `Dat` consisting of two observed variables, called "Obs1" and "Obs2", both containing two observations, at time 1 and 2:

name	time	val	err
Obs1	1	50	5
Obs1	2	150	15
Obs2	1	1	0.1
Obs2	2	2	0.2

for the long format and

time	Obs1	Obs2
1	50	1
2	150	2

for the crosstab format.

The parameters `x`, `select`, and `replicate` should be disjoint. Although the independent variable always has to be replicated it should not be given by the `replicate` parameter.

Value

A data frame with the following columns:

`name`	Column containing the column names of the original crosstab data frame, `data`
`x`	A replication of the independent variable
`y`	The actual data stacked upon each other in one column
`err`	Optional column, filled with NA values (necessary for some other functions)
`...`	all other columns from the original dataset that had to be replicated (indicated by the parameter `replicate`)

Author(s)

Tom Van Engeland <tom.vanengeland@nioz.nl>

References

Soetaert, K. and Petzoldt, T., 2010. Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME. Journal of Statistical Software 33(3) 1–28. http://www.jstatsoft.org/v33/i03

Examples

```
## =============================================================================
## Suppose we have measured sediment oxygen concentration profiles
## =============================================================================

depth  <- 0:7
O2mud  <- c( 6,   1,   0.5, 0.1, 0.05,0,   0,   0)
O2silt <- c( 6,   5,   3,   2,   1.5, 1,   0.5, 0)
O2sand <- c( 6,   6,   5,   4,   3,   2,   1,   0)
zones  <- c("a", "b", "b", "c", "c", "d", "d", "e")
oxygen <- data.frame(depth = depth,
                     zone  = zones,
                     mud   = O2mud,
                     silt  = O2silt,
                     sand  = O2sand
```

```
          )

 cross2long(data = oxygen, x = depth,
            select = c(silt, mud), replicate = zone)

 cross2long(data = oxygen, x = depth,
            select = c(mud, -silt), replicate = zone)

# twice the same column name: replicates
 colnames(oxygen)[4] <- "mud"

 cross2long(data=oxygen, x = depth, select = mud)
```

`gaussianWeights` *A kernel average smoother function to weigh residuals according to a Gaussian density function This function is still experimental... use with care*

Description

A calibration dataset in database format (cf. modCost for the database format) is extended in order to fit model output using a weighted least squares approach. To this end, the observations are replicated for a certain number of times, and weights are assigned to the replicates according to a Gaussian density function. This density has the relevant observation as mean value. The standard deviation, provided as a parameter, determines the number of inserted replicate observations (see Detail).

This weighted regression approach may be interesting when discontinuities exist in the observational data. Under these circumstances small changes in the timing (or more general the position along the axis of the independent variable) of the model output may have a disproportional impact on the overall goodness-of-fit (e.g. timing of nutrient depletion). Additionally, this approach may be used to model uncertainty in the independent variable (e.g. slices of sediment profiles, or the timing of a sampling).

Usage

```
gaussianWeights (obs, x = x, y = y, xmodel, spread, weight = "none",
                 aggregation = x ,ordering)
```

Arguments

`obs`	dataset in long (database) format as is typically used by modCost
`x`	name of the independent variable (typically x, cf. modCost) in `obs`. Defaults to x (not given as character string; cf. subset)
`y`	name of the dependent variable in `obs`. Defaults to y.
`xmodel`	an ordered vector of unique times at which model output is produced. If not given, the independent variable of the observational dataset is used.

spread	standard deviation used to calculate the weights from a normal density function. This value also determines the number of points from the model output that are compared to a specific observa- tion in obs (2 * 3 * spread + 1; containing 99.7% of the Gaussian distribution, centered around the observation of interest).
weight	scaling factor of the modCost function ("sd", "mean", or "none"). The Gaussian weights are multiplied by this factor to account for differences in units.
aggregation	vector of column names from the dataset that are used to aggregate observations while calculating the scaling factor. Defaults to the variable name, "name".
ordering	Optional extra grouping and ordering of observations. Given as a vector of variable names. If none given, ordering will be done by variable name and independent variable. If both aggregation and ordering variables are given, ordering will be done as follows: x within ordering (in reverse order) within aggregation (in reverse order). Aggregation and ordering should be disjoint sets of variable names.

Details

Suppose: spread = 1/24 (days; = 1 hour) x = time in days, 1 per hour

Then: obs_i is replicated 7 times (spread = observational periodicity = 1 hour):

=> obs_i-3 = ... = obs_i-1 = obs_i = obs_i+1 = ... = obs_i+3

The weights (W_i+j, for j = -3 ...3) are calculated as follows: W'_i+j = 1/(spread * sqrt(2pi)) * exp(-1/2 * ((obs_i+j - obs_i)/spread)^2

W_i+j = W'_i+j/sum(W_i-3,...,W_i+3) (such that their sum equals 1)

Value

A modified version of obs is returned with the following extensions:

1. Each observation obs[i] is replicated n times were n represents the number of modelx values within the interval [obs_i - (3 * spread), obs_i + 3 * spread)].

2. These replicate observations get the same x values as their modeled counterparts (xmodel).

3. Weights are given in column, called "err"

The returned data frame has the following columns:

- "name" or another name specified by the first element of aggregation. Usually this column contains the names of the observed variables.
- "x" or another name specified by x
- "y" or another name specified by y
- "err" containing the calculated weights
- The rest of the columns of the data frame given by obs in that order.

Author(s)

Tom Van Engeland <tom.vanengeland@nioz.nl>

Examples

```
## ========================================================================
## A Sediment example
## ========================================================================

## Sediment oxygen concentration is measured every
## centimeter in 3 sediment types
depth <- 0:7
observations <- data.frame(
                    profile = rep(c("mud","silt","sand"), each=8),
                    depth   = depth,
                    O2      = c(c(6,1,0.5,0.1,0.05,0,0,0),
                                c(6,5,3,2,1.5,1,0.5,0),
                                c(6,6,5,4,3,2,1,0)
                              )
                )

## A model generates profiles with a depth resolution of 1 millimeter
modeldepths <- seq(0, 9, by = 0.05)

## All these model outputs are compared with  weighed observations.
gaussianWeights(obs = observations, x = depth, y = O2,
                xmodel = modeldepths,
                spread = 0.1, weight = "none",
                aggregation = profile)



# Weights of one observation in silt at depth 2:
Sub <- subset(observations, subset = (profile == "silt" & depth == 2))
plot(Sub[,-1])
SubWW <- gaussianWeights(obs = Sub, x = depth, y = O2,
                xmodel = modeldepths, spread = 0.5,
                weight="none", aggregation = profile)
SubWW[,-1]
```

`Grid` *Grid Distribution*

Description

Generates parameter sets arranged on a regular grid.

Usage

```
Grid(parRange, num)
```

Arguments

parRange	the range (min, max) of the parameters, a matrix or a data.frame with one row for each parameter, and two columns with the minimum (1st) and maximum (2nd) column.
num	the number of random parameter sets to generate.

Details

The grid design produces the most regular parameter distribution; there is no randomness involved. The number of parameter sets generated with `Grid` will be <= `num`.

Value

a matrix with one row for each generated parameter set, and one column per parameter.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

See Also

`Norm` for (multi)normally distributed random parameter sets.

`Latinhyper` to generates parameter sets using latin hypercube sampling.

`Unif` for uniformly distributed random parameter sets.

`seq` the R-default for generating regular sequences of numbers.

Examples

```
## 4 parameters
parRange <- data.frame(min = c(0, 1, 2, 3), max = c(10, 9, 8, 7))
rownames(parRange) <- c("par1", "par2", "par3", "par4")

## grid
pairs(Grid(parRange, 500), main = "Grid")
```

`Latinhyper` *Latin Hypercube Sampling*

Description

Generates random parameter sets using a latin hypercube sampling algorithm.

Usage

```
Latinhyper(parRange, num)
```

Arguments

parRange	the range (min, max) of the parameters, a matrix or a data.frame with one row for each parameter, and two columns with the minimum (1st) and maximum (2nd) column.
num	the number of random parameter sets to generate.

Details

In the latin hypercube sampling, the space for each parameter is subdivided into num equally-sized segments and one parameter value in each of the segments drawn randomly.

Value

a matrix with one row for each generated parameter set, and one column per parameter.

Note

The latin hypercube distributed parameter sets give better coverage in parameter space than the uniform random design (Unif). It is a reasonable choice in case the number of parameter sets is limited.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References

Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (2007) Numerical Recipes in C. Cambridge University Press.

See Also

Norm for (multi)normally distributed random parameter sets.

Unif for uniformly distributed random parameter sets.

Grid to generate random parameter sets arranged on a regular grid.

Examples

```
## 4 parameters
parRange <- data.frame(min = c(0, 1, 2, 3), max = c(10, 9, 8, 7))
rownames(parRange) <- c("par1", "par2", "par3", "par4")

## Latin hypercube
pairs(Latinhyper(parRange, 100), main = "Latin hypercube")
```

modCost *Calculates the Discrepancy of a Model Solution with Observations*

Description

Given a solution of a model and observed data, estimates the residuals, and the variable and model costs (sum of squared residuals).

Usage

```
modCost(model, obs, x = "time", y = NULL, err = NULL,
        weight = "none", scaleVar = FALSE, cost = NULL,  ...)
```

Arguments

model	model output, as generated by the integration routine or the steady-state solver, a matrix or a data.frame, with one column per dependent and independent variable.
obs	the observed data, either in long (database) format (name, x, y), a data.frame, or in wide (crosstable, or matrix) format - see details.
x	the name of the *independent* variable; it should be a name occurring both in the obs and model data structures.
y	either NULL, the name of the column with the *dependent* variable values,or an index to the dependent variable values; if NULL then the observations are assumed to be in crosstable (matrix) format, and the names of the independent variables are given by the column names of this matrix.
err	either NULL, or the name of the column with the *error* estimates, used to weigh the residuals (see details); if NULL, then the residuals are not weighed.
cost	if not NULL, the output of a previous call to modCost; in this case, the new output will combine both.
weight	only if err=NULL: how to weigh the residuals, one of "none", "std", "mean", see details.
scaleVar	if TRUE, then the residuals of one observed variable are scaled respectively to the number of observations (see details).
...	additional arguments passed to R-function approx.

Details

This function compares model output with observed data.

It computes

1. the weighted *residuals*, one for each data point.
2. the *variable costs*, i.e. the sum of squared weight residuals per variable.
3. the *model cost*, the scaled sum of variable costs .

There are three steps:

1. For any observed data point, i, the *weighted residuals* are estimated as:

$$res_i = \frac{Mod_i - Obs_i}{error_i}$$

with $weight_i = 1/err_i$ and where Mod_i and Obs_i are the modeled, respectively observed value of data point i.

The weights are equal to 1/error, where the latter can be inputted, one for each data point by specifying `err` as an extra column in the observed data.

This can only be done when the data input is in long (database) format.

When `err` is not inputted, then the weights are specified via argument `weight` which is either:

- `"none"`, which sets the weight equal to 1 (the default)
- `"std"`, which sets the weights equal to the reciprocal of the standard deviation of the observed data (can only be used if there is more than 1 data point)
- `"mean"`, which uses 1/mean of the absolute value of the observed data (can only be used if not 0).

2. Then for each observed variable, j, a *variable cost* is estimated as the sum of squared weighted residuals for this variable:

$$Cost_{var_j} = \sum_{i=1}^{n_j} {res_i}^2$$

where n_j is the number of observations for observed variable j.

3. Finally, the *model Cost* is estimated as the scaled sum of variable costs:

$$ModCost = \sum_{j=1}^{n_v} \frac{Cost_{var_j}}{scale_{var_j}}$$

and where $scale_{var_j}$ allows to scale the variable costs relative to the number of observations. This is set by specifying argument `scaleVar`. If `TRUE`, then the variable costs are rescaled. The default is NOT to rescale (i.e. $scale_{var_j}$=1).

The models typically consist of (a system of) differential equations, which are either solved by:

- integration routines, e.g. the routines from package `deSolve`,
- steady-state estimators, as from package `rootSolve`.

The data can be presented in two formats:

- *data table (long) format*; this is a two to four column data.frame that contains the `name` of the observed variable (always the FIRST column), the (optional) `value of the independent variable` (default column name = "time"), the `value of the observation` and the (optional) `value of the error`. For data presented in this format, the names of the column(s) with the independent variable (`x`) and the name of the column that has the value of the dependent variable `y` must be passed to function `modCost`.

- *crosstable (wide) format*; this is a matrix, where each column denotes one dependent (or independent) variable; the column name is the name of the observed variable. When using this format, only the name of the column that contains the dependent variable must be specified (x).

As an example of both formats consider the data, called `Dat` consisting of two observed variables, called "Obs1" and "Obs2", both containing two observations, at time 1 and 2:

name	time	val	err
Obs1	1	50	5
Obs1	2	150	15
Obs2	1	1	0.1
Obs2	2	2	0.2

for the long format and

time	Obs1	Obs2
1	50	1
2	150	2

for the crosstab format. Note, that in the latter case it is not possible to provide separate errors per data point.

By calling modCost several consecutive times (using the `cost` argument), it is possible to combine both types of data files.

Value

a list of type `modCost` containing:

`model` one value, the model cost, which equals the sum of scaled variable costs (see details).

`minlogp` one value, -log(model probablity), where it is assumed that the data are normally distributed, with standard deviation = `error`.

`var` the variable costs, a data.frame with, for each observed variable the following (see details):

- name, the name of the observed variable.
- scale, the scale-factor used to weigh the variable cost, either 1 or 1/(number observations), defaults to 1.
- N, the number of data points per observed variable.
- SSR.unweighted, the sum of squared residuals per observed variable, unweighted.
- SSR, the sum of weighted squared residuals per observed variable(see details).

`residuals` the data residual, a data.frame with several columns:

- name, the name of the observed variable.
- x, the value of the independent variable (if present).
- obs, the observed variable value.
- mod, the corresponding modeled value.
- weight, the factor used to weigh the residuals, 1/error, defaults to 1.
- res, the weighted residuals between model and observations (mod-obs)*weight.
- res.unweighted, the residuals between model and observations (mod-obs).

Note

In the future, it should be possible to have more than one independent variable present. This is not yet implemented, but it should allow e.g. to fit time series of spatially dependent variables.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References

Soetaert, K. and Petzoldt, T., 2010. Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME. Journal of Statistical Software 33(3) 1–28. http://www.jstatsoft.org/v33/i03

Examples

```
## =============================================================================
## Type 1 input:  name, time, value
## =============================================================================

## Create new data: two observed variables, "a", "b"
Data <- data.frame(name = c(rep("a", 4), rep("b", 4)),
                   time = c(1:4, 2:5), val = c(runif(4), 1:4))

## "a nonsense model"
Mod <- function (t, y, par) {
  da <- 0
  db <- 1
  return(list(c(da, db)))
}

out <- ode(y = c(a = 0.5, b = 0.5), times = 0:6, func = Mod, parms = NULL)

Data   # Show
out

## The cost function
modCost(model = out, obs = Data, y = "val")

## The cost function with a data error added
Dat2 <- cbind(Data, Err = Data$val*0.1)  # error = 10% of value
```

```
modCost(model = out, obs = Dat2, y = "val", err = "Err")


## ========================================================================
## Type 2 input:  Matrix format; column names = variable names
## ========================================================================

## logistic growth model
TT    <- seq(1, 100, 2.5)
N0    <- 0.1
r     <- 0.5
K     <- 100

## analytical solution
Ana <- cbind(time = TT, N = K/(1 + (K/N0 - 1) * exp(-r*TT)))

## numeric solution
logist <- function(t, x, parms) {
  with(as.list(parms), {
    dx <- r * x[1] * (1 - x[1]/K)
    list(dx)
  })
}

time  <- 0:100
parms <- c(r = r, K = K)
x     <- c(N = N0)

## Compare several numerical solutions
Euler <- ode(x, time, logist, parms, hini = 2, method = "euler")
Rk4   <- ode(x, time, logist, parms, hini = 2, method = "rk4")
Lsoda <- ode(x, time, logist, parms) # lsoda is default method
Ana2  <- cbind(time = time, N = K/(1 + (K/N0 - 1) * exp(-r * time)))

## the SSR and residuals with respect to the "data"
cEuler <- modCost(Euler, Ana)$model
cRk4   <- modCost(Rk4  , Ana)$model
cLsoda <- modCost(Lsoda, Ana)$model
cAna   <- modCost(Ana2 , Ana)$model
compare <- data.frame(method = c("euler", "rk4", "lsoda", "Ana"),
                      cost   = c(cEuler, cRk4, cLsoda, cAna))
## Plot Euler, RK and analytic solution
plot(Euler, Rk4, col = c("red", "blue"), obs = Ana,
     main = "logistic growth", xlab = "time", ylab = "N")

legend("bottomright", c("exact", "euler", "rk4"), pch = c(1, NA, NA),
       col = c("black", "red", "blue"), lty = c(NA, 1, 2))
legend("right", ncol = 2, title = "SSR",
       legend = c(as.character(compare[,1]),
                  format(compare[,2], digits = 2)))

compare
```

```
## ========================================================================
## Now suppose we do not know K and r and they are to be fitted...
## The "observations" are the analytical solution
## ========================================================================

## Run the model with initial guess: K = 10, r = 2
parms["K"] <- 10
parms["r"] <-  2

init <-  ode(x, time, logist, parms)

## FITTING algorithm uses modFit
## First define the objective function (model cost) to be minimised

## more general: using modFit
Cost <- function(P) {
  parms["K"] <- P[1]
  parms["r"] <- P[2]
  out <- ode(x, time, logist, parms)
  return(modCost(out, Ana))
}
(Fit<-modFit(p = c(K = 10, r = 2), f = Cost))

summary(Fit)

## run model with the optimized value:
parms[c("K", "r")] <- Fit$par
fitted <-  ode(x, time, logist, parms)

## show results, compared with "observations"
plot(init, fitted, col = c("green", "blue"), lwd = 2, lty = 1,
     obs = Ana, obspar = list(col = "red", pch = 16, cex = 2),
     main = "logistic growth", xlab = "time", ylab = "N")

legend("right", c("initial", "fitted"), col = c("green", "blue"), lwd = 2)

Cost(Fit$par)
```

---

modCRL *Monte Carlo Analysis*

Description

Given a model consisting of differential equations, estimates the global effect of certain (sensitivity) parameters on selected sensitivity variables.

This is done by drawing parameter values according to some predefined distribution, running the model with each of these parameter combinations, and calculating the values of the selected output variables at each output interval.

This function is useful for "what-if" scenarios.

If the output variables consist of a time-series or spatially dependent, use sensRange instead.

Usage

```
modCRL(func, parms = NULL, sensvar = NULL, dist = "unif",
       parInput = NULL, parRange = NULL, parMean = NULL, parCovar = NULL,
       num = 100, ...)

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

## S3 method for class 'modCRL'
plot(x, which = NULL, trace = FALSE, ask = NULL, ...)

## S3 method for class 'modCRL'
pairs(x, which = 1:ncol(x), nsample = NULL, ...)

## S3 method for class 'modCRL'
hist(x, which = 1:ncol(x), ask = NULL, ...)
```

Arguments

func	an R-function that has as first argument parms and that returns a vector with variables whose sensitivity should be estimated.
parms	parameters passed to func; should be either a vector, or a list with named elements. If NULL, then the first element of parInput is taken.
sensvar	the output variables for which the sensitivity needs to be estimated. Either NULL, the default=all output variables, or a vector with output variable names (which should be present in the vector returned by func), or a vector with indices to output variables as present in the output vector returned by func.
dist	the distribution according to which the parameters should be generated, one of "unif" (uniformly random samples), "norm", (normally distributed random samples), "latin" (latin hypercube distribution), "grid" (parameters arranged on a grid). The input parameters for the distribution are specified by parRange (min,max), except for the normally distributed parameters, in which case the distribution is specified by the parameter means parMean and the variance-covariance matrix, parCovar. Note that, if the distribution is "norm" and parRange is given, then a truncated distribution will be generated. (This is useful to prevent for instance that certain parameters become negative). Ignored if parInput is specified.
parRange	the range (min, max) of the sensitivity parameters, a matrix or (preferred) a data.frame with one row for each parameter, and two columns with the minimum (1st) and maximum (2nd) value. The rownames of parRange should be parameter names that are known in argument parms. Ignored if parInput is specified.

parInput	a matrix with dimension (*, npar) with the values of the sensitivity parameters.
parMean	only when dist is "norm": the mean value of each parameter. Ignored if parInput is specified.
parCovar	only when dist is "norm": the parameter's variance-covariance matrix.
num	the number of times the model has to be run. Set large enough. If parInput is specified, then num parameters are selected randomly (from the rows of parInput.
object	an object of class modCRL.
x	an object of class modCRL.
which	the name or the index to the variables and parameters that should be plotted. Default = all variables and parameters.
nsample	the number of xy pairs to be plotted on the upper panel in the pairs plot. When NULL all xy pairs plotted. Set to a lower number in case the graph becomes too dense (and the exported picture too large). This does not affect the histograms on the diagonal plot (which are estimated using all values).
trace	if TRUE, adds smoothed line to the plot.
ask	logical; if TRUE, the user is *ask*ed before each plot, if NULL the user is only asked if more than one page of plots is necessary and the current graphics device is set interactive, see par(ask=.) and dev.interactive.
...	additional arguments passed to function func or to the methods.

Value

a data.frame of type modCRL containing the parameter(s) and the corresponding values of the sensitivity output variables.

The list returned by modCRL has a method for the generic functions summary and plot – see note.

Note

The following *methods* are included:

- summary, estimates summary statistics for the sensitivity variables, a table with as many rows as there are variables (or elements in the vector returned by func) and the following columns: x, the mapping value, Mean, the mean, sd, the standard deviation, Min, the minimal value, Max, the maximal value, q25, q50, q75, the 25th, 50 and 75% quantile.
- plot, produces a plot of the modCRL output, either one plot for each sensitivity variable and with the parameter value on the x-axis. This only works when there is only one parameter!

  OR

  one plot for each parameter value on the x-axis. This only works when there is only one variable!
- hist, produces a histogram of the modCRL output parameters and variables.
- pairs, produces a pairs plot of the modCRL output.

The data.frame of type modCRL has several attributes, which remain hidden, and which are generally not of practical use (they are needed for the S3 methods). There is one exception - see notes in help of sensRange.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>.

References

Soetaert, K. and Petzoldt, T., 2010. Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME. Journal of Statistical Software 33(3) 1–28. http://www.jstatsoft.org/v33/i03

Examples

```
## =============================================================================
## Bacterial growth model as in Soetaert and Herman, 2009
## =============================================================================

pars <- list(gmax = 0.5,eff = 0.5,
             ks = 0.5, rB = 0.01, dB = 0.01)

solveBact <- function(pars) {
  derivs <- function(t,state,pars) {    # returns rate of change
    with (as.list(c(state,pars)), {
      dBact <-  gmax*eff * Sub/(Sub + ks)*Bact - dB*Bact - rB*Bact
      dSub  <- -gmax     * Sub/(Sub + ks)*Bact + dB*Bact
      return(list(c(dBact, dSub)))
    })
  }

 state <- c(Bact = 0.1, Sub = 100)
 tout  <- seq(0, 50, by = 0.5)
 ## ode solves the model by integration...
 return(as.data.frame(ode(y = state, times = tout, func = derivs,
                          parms = pars)))
}

out <- solveBact(pars)

plot(out$time, out$Bact, main = "Bacteria",
  xlab = "time, hour", ylab = "molC/m3", type = "l", lwd = 2)

## Function that returns the last value of the simulation
SF <- function (p) {
  pars["eff"] <- p
  out <- solveBact(pars)
  return(out[nrow(out), 2:3])
}

parRange <- matrix(nr = 1, nc = 2, c(0.2, 0.8),
  dimnames = list("eff", c("min", "max")))
parRange

CRL <- modCRL(func = SF, parRange = parRange)
```

```
plot(CRL)  # plots both variables
plot(CRL, which = c("eff", "Bact"), trace = FALSE) #selects one
```

---

modFit *Constrained Fitting of a Model to Data*

Description

Fitting a model to data, with lower and/or upper bounds

Usage

```
modFit(f, p, ..., lower = -Inf, upper = Inf,
  method = c("Marq", "Port", "Newton",
           "Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN",
           "Pseudo", "bobyqa"), jac = NULL,
  control = list(), hessian = TRUE)

## S3 method for class 'modFit'
summary(object, cov=TRUE,...)

## S3 method for class 'modFit'
deviance(object,  ...)

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

## S3 method for class 'modFit'
residuals(object, ...)

## S3 method for class 'modFit'
df.residual(object, ...)

## S3 method for class 'modFit'
plot(x, ask = NULL, ...)

## S3 method for class 'summary.modFit'
print(x, digits = max(3, getOption("digits") - 3),
              ...)
```

Arguments

f	a function to be minimized, with first argument the vector of parameters over which minimization is to take place. It should return either a vector of *residuals* (of model versus data) or an element of class *modCost* (as returned by a call to modCost.

p	initial values for the parameters to be optimized over.
...	additional arguments passed to function f (modFit) or passed to the methods.
lower	lower bounds on the parameters; if unbounded set equal to -Inf.
upper	upper bounds on the parameters; if unbounded set equal to Inf.
method	The method to be used, one of "Marq", "Port", "Newton", "Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN", "Pseudo", "bobyqa" - see details.
jac	A function that calculates the Jacobian; it should be called as jac(x, ...) and return the matrix with derivatives of the model residuals as a function of the parameters. Supplying the Jacobian can substantially improve performance; see last example.
hessian	TRUE if Hessian is to be estimated. Note that, if set to FALSE, then a summary cannot be estimated.
control	additional control arguments passed to the optimisation routine - see details of nls.lm ("Marq"), nlminb ("Port"), optim ("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN"), nlm ("Newton") or pseudoOptim("Pseudo").
object	an object of class modFit.
x	an object of class modFit.
digits	number of significant digits in printout.
cov	when TRUE also calculates the parameter covariances.
ask	logical; if TRUE, the user is *ask*ed before each plot, if NULL the user is only asked if more than one page of plots is necessary and the current graphics device is set interactive, see par(ask=.) and dev.interactive.

Details

Note that arguments after ... must be matched exactly.

The method to be used is specified by argument method which can be one of the methods from function optim:

- "Nelder-Mead", the default from optim,
- "BFGS", a quasi-Newton method,
- "CG", a conjugate-gradient method,
- "L-BFGS-B", constrained quasi-Newton method,
- "SANN", method of simulated annealing.

Or one of the following:

- "Marq", the Levenberg-Marquardt algorithm (nls.lm from package minpack) - the default. Note that this method is the only least squares method.
- "Newton", a Newton-type algorithm (see nlm),
- "Port", the Port algorithm (see nlminb),
- "Pseudo", a pseudorandom-search algorithm (see pseudoOptim),
- "bobyqa", derivative-free optimization by quadratic approximation from package minqa.

For difficult problems it may be efficient to perform some iterations with `Pseudo`, which will bring the algorithm near the vicinity of a (the) minimum, after which the default algorithm (`Marq`) is used to locate the minimum more precisely.

The implementation for the routines from `optim` differs from `constrOptim` which implements an adaptive barrier algorithm and which allows a more flexible implementation of linear constraints.

For all methods except `L-BFGS-B`, `Port`, `Pseudo`, and `bobyqa` that handle box constraints internally, bounds on parameters are imposed by a transformation of the parameters to be fitted.

In case *both lower and upper bounds* are specified, this is achieved by a tangens and arc tangens transformation.

This is, parameter values, p', generated by the optimisation routine, and which are located in the range [-Inf, Inf] are transformed, before they are passed to `f` as:

$$p = (upper + lower)/2 + (upper - lower) \cdot \arctan(p')/\pi$$

.

which maps them into the interval [lower, upper].

Before the optimisation routine is called, the original parameter values, as given by argument `p` are mapped from [lower,upper] to [-Inf, Inf] by:

$$p' = \tan(\pi/2 \cdot (2p - upper - lower)/(upper - lower))$$

In case *only lower or upper bounds* are specified, this is achieved by a log transformation and a corresponding exponential back transformation.

In case parameters are transformed (all methods) or in case the `method` `Port`, `Pseudo`, `Marq` or `bobyqa` is selected, the *Hessian* is approximated as $2 \cdot J^T \cdot J$, where J is the Jacobian, estimated by finite differences.

This ignores the second derivative terms, but this is reasonable if the method has truly converged to the minimum.

Note that finite differences are not extremely precise.

In case the Levenberg-Marquard method (`Marq`) is used, and parameters are not transformed, 0.5 times the Hessian of the least squares problem is returned by `nls.lm`, the original Marquardt algorithm. To make it compatible, this value is multiplied with 2 and the TRUE Hessian is thus returned by `modFit`.

Value

a list of class *modFit* containing the results as returned from the called optimisation routines.

This includes the following:

`par`	the best set of parameters found.
`ssr`	the sum of squared residuals, evaluated at the best set of parameters.
`Hessian`	A symmetric matrix giving an estimate of the Hessian at the solution found - see note.
`residuals`	the result of the last `f` evaluation; that is, the residuals.

ms	the mean squared residuals, i.e. ssr/length(residuals).
var_ms	the weighted and scaled variable mean squared residuals, one value per observed variable; only when f returns an element of class *modCost*; NA otherwise.
var_ms_unscaled	the weighted, but not scaled variable mean squared residuals
var_ms_unweighted	the raw variable mean squared residuals, unscaled and unweighted.
...	any other arguments returned by the called optimisation routine.

Note: this means that some return arguments of the original optimisation functions are renamed.

More specifically, "objective" and "counts" from routine nlminb (method = "Port") are renamed; "value" and "counts"; "niter" and "minimum" from routine nls.lm (method=Marq) are renamed; "counts" and "value"; "minimum" and "estimate" from routine nlm (method = "Newton") are renamed.

The list returned by modFit has a method for the summary, deviance, coef, residuals, df.residual and print.summary – see note.

Note

The summary *method* is based on an estimate of the parameter covariance matrix. In computing the covariance matrix of the fitted parameters, the problem is treated as if it were a linear least squares problem, linearizing around the parameter values that minimize Chi^2.

The covariance matrix is estimated as $1/(0.5 \cdot Hessian)$.

This computation relies on several things, i.e.:

1. the parameter values are located at the minimum (i.e. the fitting algorithm has converged).
2. the observations y_j are subject to independent errors whose variances are well estimated by $1/(n-p)$ times the residual sum of squares (where n = number of data points, p = number of parameters).
3. the model is not too nonlinear.

This means that the estimated covariance (correlation) matrix and the confidence intervals derived from it may be worthless if the assumptions behind the covariance computation are invalid.

If in doubt about the validity of the summary computations, use Monte Carlo fitting instead, or run a modMCMC.

Other methods included are:

- deviance, which returns the model deviance,
- coef, which extracts the values of the fitted parameters,
- residuals,which extracts the model residuals,
- df.residual which returns the residual degrees of freedom
- print.summary, producing a nice printout of the summary.

Specifying a function to estimate the Jacobian matrix via argument `jac` may increase speed. The Jacobian is used in the methods "Marq", "BFGS", "CG", "L-BFGS", "Port", and is also used at the end, to estimate the Hessian at the optimum.

Specification of the `gradient` in routines "BFGS", "CG", "L-BFGS" from `optim` and "port" from `nlminb` is not allowed here. Within `modFit`, the gradient is rather estimated from the Jacobian `jac` and the function `f`.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>,

Thomas Petzoldt <thomas.petzoldt@tu-dresden.de>

References

Bates, D., Mullen, K. D. Nash, J. C. and Varadhan, R. 2014. minqa: Derivative-free optimization algorithms by quadratic approximation. R package. https://cran.r-project.org/package=minqa

Gay, D. M., 1990. Usage Summary for Selected Optimization Routines. Computing Science Technical Report No. 153. AT&T Bell Laboratories, Murray Hill, NJ 07974.

Powell, M. J. D. (2009). The BOBYQA algorithm for bound constrained optimization without derivatives. Report No. DAMTP 2009/NA06, Centre for Mathematical Sciences, University of Cambridge, UK. http://www.damtp.cam.ac.uk/user/na/NA_papers/NA2009_06.pdf

Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P., 2007. Numerical Recipes in C. Cambridge University Press.

Price, W.L., 1977. A Controlled Random Search Procedure for Global Optimisation. The Computer Journal, 20: 367-370.

Soetaert, K. and Petzoldt, T., 2010. Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME. Journal of Statistical Software 33(3) 1–28. http://www.jstatsoft.org/v33/i03

Please see also additional publications on the help pages of the individual algorithms.

See Also

`constrOptim` for constrained optimization.

Examples

```
## ======================================================================
## logistic growth model
## ======================================================================
TT    <- seq(1, 60, 5)
N0    <- 0.1
r     <- 0.5
K     <- 100

## perturbed analytical solution
Data <- data.frame(
```

```
  time = TT,
     N = K / (1+(K/N0-1) * exp(-r*TT)) * (1 + rnorm(length(TT), sd = 0.01))
)

plot(TT, Data[,"N"], ylim = c(0, 120), pch = 16, col = "red",
     main = "logistic growth", xlab = "time", ylab = "N")


##====================================
## Fitted with analytical solution  #
##====================================

## initial "guess"
parms <- c(r = 2, K = 10, N0 = 5)

## analytical solution
model <- function(parms,time)
  with (as.list(parms), return(K/(1+(K/N0-1)*exp(-r*time))))

## run the model with initial guess and plot results
lines(TT, model(parms, TT), lwd = 2, col = "green")

## FITTING algorithm 1
ModelCost <- function(P) {
 out <- model(P, TT)
 return(Data$N-out)  # residuals
}

(Fita <- modFit(f = ModelCost, p = parms))

times <- 0:60
lines(times, model(Fita$par, times), lwd = 2, col = "blue")
summary(Fita)

##====================================
##  Fitted with numerical solution  #
##====================================

## numeric solution
logist <- function(t, x, parms) {
  with(as.list(parms), {
    dx <- r * x[1] * (1 - x[1]/K)
    list(dx)
  })
}

## model cost,
ModelCost2 <- function(P) {
 out <- ode(y = c(N = P[["N0"]]), func = logist, parms = P, times = c(0, TT))
 return(modCost(out, Data))  # object of class modCost
}

Fit <- modFit(f = ModelCost2, p = parms, lower = rep(0, 3),
```

```
              upper = c(5, 150, 10))

out <- ode(y = c(N = Fit$par[["N0"]]), func = logist, parms = Fit$par,
           times = times)

lines(out, col = "red", lty = 2)
legend("right", c("data", "original", "fitted analytical", "fitted numerical"),
       lty = c(NA, 1, 1, 2), lwd = c(NA, 2, 2, 1),
       col = c("red", "green", "blue", "red"), pch = c(16, NA, NA, NA))
summary(Fit)
plot(residuals(Fit))

## ======================================================================
## the use of the Jacobian
## ======================================================================

## We use modFit to solve a set of linear equations
A <- matrix(nr = 30, nc = 30, runif(900))
X <- runif(30)
B <- A %*% X

## try to find vector "X"; the Jacobian is matrix A

## Function that returns the vector of residuals
FUN <- function(x)
  as.vector(A %*% x - B)

## Function that returns the Jacobian
JAC <- function(x) A

## The port algorithm
print(system.time(
  mf <- modFit(f = FUN, p = runif(30), method = "Port")
))
print(system.time(
  mf <- modFit(f = FUN, p = runif(30), method = "Port", jac = JAC)
))
max(abs(mf$par - X))  # should be very small

## BFGS
print(system.time(
  mf <- modFit(f = FUN, p = runif(30), method = "BFGS")
))
print(system.time(
  mf <- modFit(f = FUN, p = runif(30), method = "BFGS", jac = JAC)
))
max(abs(mf$par - X))

## Levenberg-Marquardt
print(system.time(
  mf <- modFit(f = FUN, p = runif(30), jac = JAC)
))
max(abs(mf$par - X))
```

modMCMC *Constrained Markov Chain Monte Carlo*

Description

Performs a Markov Chain Monte Carlo simulation, using an adaptive Metropolis (AM) algorithm and including a delayed rejection (DR) procedure.

Usage

```
modMCMC(f, p, ..., jump = NULL,  lower = -Inf, upper = +Inf,
        prior = NULL, var0 = NULL, wvar0 = NULL, n0 = NULL,
        niter = 1000, outputlength = niter, burninlength = 0,
        updatecov = niter, covscale = 2.4^2/length(p),
        ntrydr = 1, drscale = NULL, verbose = TRUE)



## S3 method for class 'modMCMC'
summary(object, remove = NULL, ...)

## S3 method for class 'modMCMC'
pairs(x, Full = FALSE, which = 1:ncol(x$pars),
  remove = NULL, nsample = NULL, ...)

## S3 method for class 'modMCMC'
hist(x, Full = FALSE, which = 1:ncol(x$pars),
  remove = NULL, ask = NULL, ...)

## S3 method for class 'modMCMC'
plot(x, Full = FALSE, which = 1:ncol(x$pars),
  trace = TRUE, remove = NULL, ask = NULL, ...)
```

Arguments

f	the function to be evaluated, with first argument the vector of parameters which should be varied. It should return either the model residuals, an element of class *modCost* (as returned by a call to modCost) or -2*log(likelihood). The latter is equivalent to the sum-of-squares functions when using a Gaussian likelihood and prior.
p	initial values for the parameters to be optimized over.
...	additional arguments passed to function f or to the methods.
jump	jump length, either a *number*, a *vector* with length equal to the total number of parameters, a *covariance matrix*, or a *function* that takes as input the current values of the parameters and produces as output the perturbed parameters. See details.

prior	-2*log(parameter prior probability), either a function that is called as `prior(p)` or `NULL`; in the latter case a non-informative prior is used (i.e. all parameters are equally likely, depending on `lower` and `upper` within min and max bounds).
var0	initial model variance; if `NULL`, it is assumed that the model variance is `1`, and the return element from `f` is -2*log (likelihood). If it has a value, it is assumed that the return element from `f` contain the model residuals or a list of class `modFit`. See details. Good options for `var0` are to use the modelvariance (`modVariance`) as returned by the `summary` method of modFit. When this option is chosen, and the model has several variables, they will all be scaled similarly. See vignette `FMEdyna`. In case the model has several variables with different magnitudes, then it may be better to scale each variable independently. In that case, one can use as `var0`, the mean of the unweighted squared residuals from the model fit as returned from modFit (`var_ms_unweighted`). See vignette `FME`.
wvar0	"weight" for the initial model variance – see details.
n0	parameter used for weighing the initial model variance - if `NULL`, it is estimated as `n0=wvar0*n`, where n = number of observations. See details.
lower	lower bounds on the parameters; for unbounded parameters set equal to `-Inf`.
upper	upper bounds on the parameters; for unbounded parameters set equal to `Inf`.
niter	number of iterations for the MCMC.
outputlength	number of iterations kept in the output; should be smaller or equal to `niter`.
updatecov	number of iterations after which the parameter covariance matrix is (re)evaluated based on the parameters kept thus far, and used to update the MCMC jumps.
covscale	scale factor for the parameter covariance matrix, used to perform the MCMC jumps.
burninlength	number of initial iterations to be removed from output.
ntrydr	maximal number of tries for the delayed rejection procedure. It is generally not a good idea to set this to a too large value.
drscale	for each try during delayed rejection, the cholesky decomposition of the proposal matrix is scaled with this amount; if `NULL`, it is assumed to be `c(0.2,0.25, 0.333, 0.333, ...)`
verbose	if `TRUE`: prints extra output.
object	an object of class `modMCMC`.
x	an object of class `modMCMC`.
Full	If TRUE then not only the parameters will be plotted, but also the function value and (if appropriate) the model variance(s).
which	the name or the index to the parameters that should be plotted. Default = all parameters. If `Full=TRUE`, setting `which =NULL` will plot only the function value and the model variance.
trace	if `TRUE`, adds smoothed line to the plot.
remove	a list with indices of the runs that should be removed (e.g. to remove runs during burnin.
nsample	the number of xy pairs to be plotted on the upper panel in the pairs plot. When `NULL` all xy pairs plotted. Set to a lower number in case the graph becomes too dense (and the exported picture too large). This does not affect the histograms on the diagonal plot (which are estimated using all MCMC draws).

ask — logical; if TRUE, the user is *ask*ed before each plot, if NULL the user is only asked if more than one page of plots is necessary and the current graphics device is set interactive, see par(ask=.) and dev.interactive.

Details

Note that arguments after ... must be matched exactly.

R-function f is called as f(p, ...). It should return either -2 times the log likelihood of the model (one value), the residuals between model and data or an item of class modFit (as created by function modFit.

In the latter two cases, it is assumed that the prior distribution for θ is either non-informative or gaussian. If gaussian, it can be treated as a sum of squares (SS). If the measurement function is defined as:

$$y = f(\theta) + \xi\xi\ N(0, \sigma^2)$$

where ξ is the measurement error, assumed normally distribution, then the posterior for the parameters will be estimated as:

$$p(\theta|y, \sigma^2) \propto exp(-0.5 \cdot (\frac{SS(\theta)}{\sigma^2} + SS_{pri}(\theta))$$

and where σ^2 is the error variance, SS is the sum of squares function $SS(\theta) = \sum(y_i - f(\theta))^2$. If non-informative priors are used, then $SS_{pri}(\theta) = 0$.

The error variance σ^2 is considered a nuisance parameter. A prior distribution of it should be specified and a posterior distribution is estimated.

If wvar0 or n0 is >0, then the variances are sampled as conjugate priors from the inverse gamma distribution with parameters var0 and n0=wvar0*n. Larger values of wvar0 keep these samples closer to var0.

Thus, at each step, 1/ the error variance (σ^{-2}) is sampled from a gamma distribution:

$$p(\sigma^{-2}|y, \theta) \sim \Gamma(\frac{(n_0 + n)}{2}, \frac{(n_0 \cdot var0 + SS(\theta))}{2})$$

where n is the number of data points and where $n0 = n \cdot wvar0$, and where the second argument to the gamma function is the shape parameter.

The prior parameters (var0 and wvar0) are the prior mean for σ^2 and the prior accuracy.

By setting wvar0 equal to 1, equal weight is given to the prior and the current value.

If wvar0 is 0 then the prior is ignored.

If wvar0 is NULL (the default) then the error variances are assumed to be fixed.

var0 estimates the variance of the measured components. In case independent estimates are not available, these variances can be obtained from the mean squares of fitted residuals. (e.g. as reported in modFit). See the examples. (but note that this is not truly independent information)

var0 is either one value, or a value for each observed variable, or a value for each observed data point.

When `var0` is not `NULL`, then `f` is assumed to return the model residuals OR an instance of class `modCost`.

When `var0=NULL`, then `f` should return either -2*log(probability of the model), or an instance of class `modCost`.

`modMCMC` implements the Metropolis-Hastings method. The proposal distribution, which is used to generate new parameter values is the (multidimensional) Gaussian density distribution, with standard deviation given by `jump`.

`jump` can be either one value, a vector of length = number of parameters or a parameter covariance matrix (nrow = ncol = number parameters).

The jump parameter, `jump` thus determines how much the new parameter set will deviate from the old one.

If `jump` is one value, or a vector, then the new parameter values are generated by sampling a normal distribution with standard deviation equal to `jump`. A larger value will lead to larger jumps in the parameter space, but acceptance of new points can get very low. Smaller jump lengths increase the acceptance rate, but the algorithm may move too slowly, and too many runs may be needed to scan the parameter space.

If `jump` is `NULL`, then the jump length is taken as 10% of the parameter value as given in `p`.

`jump` can also be a proposal covariance matrix. In this case, the new parameter values are generated by sampling a multidimensional normal distribution. It can be efficient to initialise `jump` using the parameter covariance as resulting from fitting the model (e.g. using `modFit`) – see examples.

Finally, `jump` can also be an R-function that takes as input the current values of the parameters and returns the new parameter values.

Two methods are implemented to increase the number of accepted runs.

1. In the *"adaptive Metropolis"* method, new parameters are generated with a covariance matrix that is estimated from the parameters generated (and saved) thus far. The idea behind this is that the MCMC method is more efficient if the proposal covariance (to generate new parameter values) is somehow tuned to the shape and size of the target distribution.

   Setting `updatecov` smaller than `niter` will trigger this functionality. In this case, every `updatecov` iterations, the jump covariance matrix will be estimated from the covariance matrix of the saved parameter values. The covariance matrix is scaled with $(2.4^2/npar)$ where npar is the number of parameters, unless `covscale` has been given a different value. Thus, $Jump = (cov(\theta_1, \theta_2, ...\theta_n) \cdot diag(np, +1e^{-16}) \cdot (2.4^2/npar)$ where the small number $1e^{-16}$ is added on the diagonal of the covariance matrix to prevent it from becoming singular.

   Note that a problem of adapting the proposal distribution using the MCMC results so far is that standard convergence results do not apply. One solution is to use adaptation only for the burn-in period and discard the part of the chain where adaptation has been used.

   Thus, when using `updatecov` with a positive value of `burninlength`, the proposal distribution is only updated during burnin. If `burninlength` = 0 though, the updates occur throughout the entire simulation.

   When using the adaptive Metropolis method, it is best to start with a small value of the jump length.

2. In the *"delayed rejection"* method, new parameter values are tried upon rejection. The process of delaying rejection can be iterated for at most `ntrydr` trials. Setting `ntrydr` equal to 1 (the default) toggles off delayed rejection.

During the delayed rejection procedure, new parameters are generated from the last accepted value by scaling the jump covariance matrix with a factor as specified in `drscale`. The acceptance probability of this new set depends on the candidates so far proposed and rejected, in such a way that reversibility of the Markov chain is preserved. See Haario et al. (2005, 2006) for more details.

Convergence of the MCMC chain can be checked via `plot`, which plots for each iteration the values of all parameters, and if `Full` is `TRUE`, of the function value (SS) and (if appropriate) the modeled variance. If converged, there should be no visible drift.

In addition, the methods from package `coda` become available by making the object returned by `modMCMC` of class `mcmc`, as used in the methods of `coda`. For instance, if object `MCMCres` is returned by `modMCMC` then `as.mcmc(MCMCres$pars)` will make an instance of class `mcmc`, usable by `coda`.

The `burninlength` is the number of initial steps that is not included in the output. It can be useful if the initial value of the parameters is far from the optimal value. Starting the MCMC with the best fit parameter set will alleviate the need for using `burninlength`.

Value

a list of class *modMCMC* containing the results as returned from the Markov chain.

This includes the following:

`pars`	an array with dimension (`outputlength`, length(`p`)), containing the parameters of the MCMC at each iteration that is kept.
`SS`	vector with the sum of squares function, one for each row in `pars`.
`naccepted`	the number of accepted runs.
`sig`	the sampled error variance σ^2, a matrix with one row for each row in `pars`.
`bestpar`	the parameter set that gave the highest probability.
`bestfunp`	the function value corresponding to `bestpar`.
`prior`	the parameter prior, one value for each row in `pars`.
`count`	information about the MCMC chain: number of delayed rejection steps (`dr_steps`), the number of alfa steps `Alfasteps`, the number of accepted runs (`num_accepted`) and the number of times the proposal covariance matrix has been updated (`num_covupdate`.)
`settings`	the settings for error covariance calculation, i.e. arguments `var0`, `n0` and `N` the number of data points.

The list returned by `modMCMC` has methods for the generic functions `summary`, `plot`, `pairs` – see note.

Note

The following S3 methods are provided:

- summary, produces summary statistics of the MCMC results
- plot, plots the MCMC results, for all parameters. Use it to check convergence.
- pairs, produces a pairs plot of the MCMC results; overrides the default `gap = 0`, `upper.panel = NA`, and `diag.panel`.

It is also possible to use the methods from the coda package, e.g. densplot.

To do that, first the modMCMC object has to be converted to an mcmc object. See the examples for an application.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

Marko Laine <Marko.Laine@fmi.fi>

References

Laine, M., 2008. Adaptive MCMC Methods With Applications in Environmental and Geophysical Models. Finnish Meteorological Institute contributions 69, ISBN 978-951-697-662-7, Finnish Meteorological Institute, Helsinki.

Haario, H., Saksman, E. and Tamminen, J., 2001. An Adaptive Metropolis Algorithm. Bernoulli 7, pp. 223–242.

Haario, H., Laine, M., Mira, A. and Saksman, E., 2006. DRAM: Efficient Adaptive MCMC. Statistics and Computing, 16(4), 339–354.

Haario, H., Saksman, E. and Tamminen, J., 2005. Componentwise Adaptation for High Dimensional MCMC. Computational Statistics 20(2), 265–274.

Gelman, A. Varlin, J. B., Stern, H. S. and Rubin, D. B., 2004. Bayesian Data Analysis. Second edition. Chapman and Hall, Boca Raton.

Soetaert, K. and Petzoldt, T., 2010. Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME. Journal of Statistical Software 33(3) 1–28. http://www.jstatsoft.org/v33/i03

See Also

modFit for constrained model fitting

Examples

```
## =============================================================================
## Sampling a 3-dimensional normal distribution,
## =============================================================================
# mean = 1:3, sd = 0.1
# f returns -2*log(probability) of the parameter values

NN <- function(p) {
  mu <- c(1,2,3)
  -2*sum(log(dnorm(p, mean = mu, sd = 0.1)))   #-2*log(probability)
}

# simple Metropolis-Hastings
MCMC <- modMCMC(f = NN, p = 0:2, niter = 5000,
                outputlength = 1000, jump = 0.5)

# More accepted values by updating the jump covariance matrix...
```

```
MCMC <- modMCMC(f = NN, p = 0:2, niter = 5000, updatecov = 100,
                outputlength = 1000, jump = 0.5)
summary(MCMC)

plot(MCMC)   # check convergence
pairs(MCMC)

## =============================================================================
## test 2: sampling a 3-D normal distribution, larger standard deviation...
## noninformative priors, lower and upper bounds imposed on parameters
## =============================================================================

NN <- function(p) {
  mu <- c(1,2,2.5)
  -2*sum(log(dnorm(p, mean = mu, sd = 0.5)))   #-2*log(probability)
}

MCMC2 <- modMCMC(f = NN, p = 1:3, niter = 2000, burninlength = 500,
  updatecov = 10, jump = 0.5, lower = c(0, 2, 1), upper = c(1, 3, 3))
plot(MCMC2)
hist(MCMC2, breaks = 20)

## Compare output of p3 with theoretical distribution
hist(MCMC2, which = "p3", breaks = 20)
lines(seq(1, 3, 0.1), dnorm(seq(1, 3, 0.1), mean = 2.5,
  sd = 0.5)/pnorm(3, 2.5, 0.5))
summary(MCMC2)

# functions from package coda...
cumuplot(as.mcmc(MCMC2$pars))
summary(as.mcmc(MCMC2$pars))
raftery.diag(MCMC2$pars)

## =============================================================================
## test 3: sampling a log-normal distribution, log mean=1:4, log sd = 1
## =============================================================================

NL <- function(p) {
  mu <- 1:4
  -2*sum(log(dlnorm(p, mean = mu, sd = 1)))      #-2*log(probability)
}
MCMCl <- modMCMC(f = NL, p = log(1:4), niter = 3000,
                outputlength = 1000, jump = 5)
plot(MCMCl)   # bad convergence
cumuplot(as.mcmc(MCMCl$pars))

MCMCl <- modMCMC (f = NL, p = log(1:4), niter = 3000,
                  outputlength = 1000, jump = 2^(2:5))
plot(MCMCl)   # better convergence but CHECK it!
pairs(MCMCl)
colMeans(log(MCMCl$pars))
apply(log(MCMCl$pars), 2, sd)
```

```
MCMCl <- modMCMC (f = NL, p = rep(1, 4), niter = 3000,
                  outputlength = 1000, jump = 5, updatecov = 100)
plot(MCMCl)
colMeans(log(MCMCl$pars))
apply(log(MCMCl$pars), 2, sd)

## =============================================================================
## Fitting a Monod (Michaelis-Menten) function to data
## =============================================================================

# the observations
#---------------------
Obs <- data.frame(x=c(   28,  55,   83,  110,  138,  225,  375),   # mg COD/l
                  y=c(0.053,0.06,0.112,0.105,0.099,0.122,0.125))   # 1/hour
plot(Obs, pch = 16, cex = 2, xlim = c(0, 400), ylim = c(0, 0.15),
     xlab = "mg COD/l", ylab = "1/hr", main = "Monod")

# the Monod model
#---------------------
Model <- function(p, x) data.frame(x = x, N = p[1]*x/(x+p[2]))

# Fitting the model to the data
#---------------------
# define the residual function
Residuals  <- function(p) (Obs$y - Model(p, Obs$x)$N)

# use modFit to find parameters
P      <- modFit(f = Residuals, p = c(0.1, 1))

# plot best-fit model
x      <-0:375
lines(Model(P$par, x))

# summary of fit
sP    <- summary(P)
sP[]
print(sP)

# Running an MCMC
#---------------------
# estimate parameter covariances
# (to efficiently generate new parameter values)
Covar   <- sP$cov.scaled * 2.4^2/2

# the model variance
s2prior <- sP$modVariance

# set nprior = 0 to avoid updating model variance
MCMC <- modMCMC(f = Residuals, p = P$par,jump = Covar, niter = 1000,
                var0 = s2prior, wvar0 = NULL, updatecov = 100)

plot(MCMC, Full = TRUE)
pairs(MCMC)
```

```
# function from the coda package.
raftery.diag(as.mcmc(MCMC$pars))
cor(MCMC$pars)

cov(MCMC$pars)   # covariances by MCMC
sP$cov.scaled    # covariances by Hessian of fit

x  <- 1:400
SR <- summary(sensRange(parInput = MCMC$pars, func = Model, x = x))
plot(SR, xlab="mg COD/l", ylab = "1/hr", main = "Monod")
points(Obs, pch = 16, cex = 1.5)
```

Norm *Normal Random Distribution*

Description

Generates random parameter sets that are (multi)normally distributed.

Usage

```
Norm(parMean, parCovar, parRange = NULL, num)
```

Arguments

parMean	a vector, with the mean value of each parameter.
parCovar	the parameter variance-covariance matrix.
parRange	the range (min, max) of the parameters, a matrix or a data.frame with one row for each parameter, and two columns with the minimum (1st) and maximum (2nd) column.
num	the number of random parameter sets to generate.

Details

Function Norm, draws parameter sets from a multivariate normal distribution, as specified through the mean value and the variance-covariance matrix of the parameters. In addition, it is possible to impose a minimum and maximum of each parameter, via parRange. This will generate a truncated distribution. Use this for instance if certain parameters cannot become negative.

Value

a matrix with one row for each generated parameter set, and one column per parameter.

Note

For function Norm to work, parCovar must be a valid variance-covariance matrix. (i.e. positive definite). If this is not the case, then the function will fail.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

See Also

`Unif` for uniformly distributed random parameter sets.

`Latinhyper` to generates parameter sets using latin hypercube sampling.

`Grid` to generate random parameter sets arranged on a regular grid

`rnorm` the R-default for generating normally distributed random numbers.

Examples

```
## multinormal parameters: variance-covariance matrix and parameter mean
parCovar <- matrix(data = c(0.5, -0.2, 0.3, 0.4, -0.2, 1.0, 0.1, 0.3,
                   0.3, 0.1, 1.5, -0.7, 1.0, 0.3, -0.7, 4.5), nrow = 4)
parCovar

parMean <- 4:1

## Generated sample
Ndist <- Norm(parCovar = parCovar, parMean = parMean, num = 500)
cov(Ndist)   # check
pairs(Ndist, main = "normal")

## truncated multinormal
Ranges <- data.frame(min = rep(0, 4), max = rep(Inf, 4))

pairs(Norm(parCovar = parCovar, parMean = parMean, parRange = Ranges,
      num = 500), main = "truncated normal")
```

---

obsplot *Plot Method for observed data*

Description

Plot all observed variables in matrix formalt

Usage

```
obsplot(x, ..., which = NULL, xyswap = FALSE, ask = NULL)
```

Arguments

x	a matrix or data.frame, containing the observed data to be plotted. The 'x'-values (first axis) should be the first column. Several other matrices or data.frames can be passed in the ..., after x (unnamed) - see second example. If the first column of x consists of factors, or characters (strings), then it is assumed that the data are presented in long (database) format, where the first three columns contain (name, x, y). See last example.
which	the name(s) or the index to the variables that should be plotted. Default = all variables, except the first column.
ask	logical; if TRUE, the user is *ask*ed before each plot, if NULL the user is only asked if more than one page of plots is necessary and the current graphics device is set interactive, see par(ask=.) and dev.interactive.
xyswap	if TRUE, then x-and y-values are swapped and the y-axis is from top to bottom. Useful for drawing vertical profiles.
...	additional arguments. The graphical arguments are passed to plot.default and points. The dots may contain other matrices and data.frames with observed data to be plotted on the same graphs as x - see second example. The arguments after ... must be matched exactly.

Details

The number of panels per page is automatically determined up to 3 x 3 (par(mfrow = c(3, 3))). This default can be overwritten by specifying user-defined settings for mfrow or mfcol. Set mfrow equal to NULL to avoid the plotting function to change user-defined mfrow or mfcol settings.

Other graphical parameters can be passed as well. Parameters are vectorized, either according to the number of plots (xlab, ylab, main, sub, xlim, ylim, log, asp, ann, axes, frame.plot,panel.first,panel.last, cex.lab,cex.axis,cex.main) or according to the number of lines within one plot (other parameters e.g. col, lty, lwd etc.) so it is possible to assign specific axis labels to individual plots, resp. different plotting style. Plotting parameter ylim, or xlim can also be a list to assign different axis limits to individual plots.

See Also

print.deSolve, ode, deSolve

Examples

```
## 'observed' data
AIRquality <- cbind(DAY = 1:153, airquality[, 1:4])
head(AIRquality)
obsplot(AIRquality, type="l", xlab="Day since May")

## second set of observed data
AIR2 <- cbind( 1:100, Solar.R = 250 * runif(100), Temp = 90-30*cos(2*pi*1:100/365) )
```

```
obsplot(AIRquality, AIR2, type = "l", xlab = "Day since May" , lwd = 1:2)

obsplot(AIRquality, AIR2, type = "l", xlab = "Day since May" ,
         lwd = 1 : 2, which =c("Solar.R", "Temp"),
         xlim = list(c(0, 150), c(0, 100)))

obsplot(AIRquality, AIR2, type = "l", xlab = "Day since May" ,
         lwd = 1 : 2, which =c("Solar.R", "Temp"), log = c("y", ""))

obsplot(AIRquality, AIR2, which = 1:3, xyswap = c(TRUE,FALSE,TRUE))

## ' a data.frame, with 'treatments', presented in long database format
Data <- ToothGrowth[,c(2,3,1)]
head   (Data)
obsplot(Data, ylab = "len", xlab = "dose")

# same, plotted as two observed data sets
obsplot(subset(ToothGrowth, supp == "VC", select = c(dose, len)),
        subset(ToothGrowth, supp == "OJ", select = c(dose, len)))
```

`pseudoOptim` *Pseudo-random Search Optimisation Algorithm of Price (1977)*

Description

Fits a model to data, using the pseudo-random search algorithm of Price (1977), a random-based fitting technique.

Usage

```
pseudoOptim(f, p,..., lower, upper, control = list())
```

Arguments

`f`	function to be minimised, its first argument should be the vector of parameters over which minimization is to take place. It should return a scalar result, the model cost, e.g the sum of squared residuals.
`p`	initial values of the parameters to be optimised.
`...`	arguments passed to funtion `f`.
`lower`	minimal values of the parameters to be optimised; these must be specified; they cannot be -Inf.
`upper`	maximal values of the parameters to be optimised; these must be specified; they cannot be +Inf.
`control`	a list of control parameters - see details.

Details

The `control` argument is a list that can supply any of the following components:

- npop, number of elements in the population. Defaults to max(5*length(p),50).
- numiter, maximal number of iterations to be performed. Defaults to 10000. The algorithm either stops when `numiter` iterations has been performed or when the remaining variation is less than `varleft`.
- centroid, number of elements from which to estimate a new parameter vector, defaults to 3.
- varleft, relative variation remaining; if below this value the algorithm stops; defaults to 1e-8.
- verbose, if TRUE, more verbose output will contain the parameters in the final population, their respective population costs and the cost at each succesful interation. Defaults to `FALSE`.

see the book of Soetaert and Herman (2009) for a description of the algorithm AND for a line to line explanation of the function code.

Value

a list containing:

`par`	the optimised parameter values.
`cost`	the model cost, or function evaluation associated to the optimised parameter values, i.e. the minimal cost.
`iterations`	the number of iterations performed.

and if control\$verbose is TRUE:

`poppar`	all parameter vectors remaining in the population, matrix of dimension (npop,length(par)).
`popcost`	model costs associated with all population parameter vectors, vector of length npop.
`rsstrace`	a 2-columned matrix with the iteration number and the model cost at each succesful iteration.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References

Soetaert, K. and Herman, P. M. J., 2009. A Practical Guide to Ecological Modelling. Using R as a Simulation Platform. Springer, 372 pp.

Price, W.L., 1977. A Controlled Random Search Procedure for Global Optimisation. The Computer Journal, 20: 367-370.

Examples

```
amp    <- 6
period <- 5
phase  <- 0.5

x <- runif(20)*13
y <- amp*sin(2*pi*x/period+phase) + rnorm(20, mean = 0, sd = 0.05)
plot(x, y, pch = 16)


cost <- function(par)
    sum((par[1] * sin(2*pi*x/par[2]+par[3])-y)^2)

p1 <- optim(par = c(amplitude = 1, phase = 1, period = 1), fn = cost)
p2 <- optim(par = c(amplitude = 1, phase = 1, period = 1), fn = cost,
            method = "SANN")
p3 <- pseudoOptim(p = c(amplitude = 1, phase = 1, period = 1),
            lower = c(0, 1e-8, 0), upper = c(100, 2*pi, 100),
            f = cost, control = c(numiter = 3000, verbose = TRUE))

curve(p1$par[1]*sin(2*pi*x/p1$par[2]+p1$par[3]), lty = 2, add = TRUE)
curve(p2$par[1]*sin(2*pi*x/p2$par[2]+p2$par[3]), lty = 3, add = TRUE)
curve(p3$par[1]*sin(2*pi*x/p3$par[2]+p3$par[3]), lty = 1, add = TRUE)
legend ("bottomright", lty = c(1, 2, 3),
         c("Price", "Mathematical", "Simulated annealing"))
```

`sensFun` *Local Sensitivity Analysis*

Description

Given a model consisting of differential equations, estimates the local effect of certain parameters on selected sensitivity variables by calculating a matrix of so-called sensitivity functions. In this matrix the (i,j)-th element contains

$$\frac{\partial y_i}{\partial \Theta_j} \cdot \frac{\Delta \Theta_j}{\Delta y_i}$$

and where y_i is an output variable (at a certain time instance), Θ_j is a parameter, and Δy_i is the scaling of variable y_i, $\Delta \Theta_j$ is the scaling of parameter Θ_j.

Usage

```
sensFun(func, parms, sensvar = NULL, senspar = names(parms),
        varscale = NULL, parscale = NULL, tiny = 1e-8, map = 1, ...)

## S3 method for class 'sensFun'
summary(object, vars = FALSE, ...)
```

```
## S3 method for class 'sensFun'
pairs(x, which = NULL, ...)

## S3 method for class 'sensFun'
plot(x, which = NULL, legpos="topleft", ask = NULL, ...)

## S3 method for class 'summary.sensFun'
plot(x, which = 1:nrow(x), ...)
```

Arguments

func	an R-function that has as first argument parms and that returns a matrix or data.frame with the values of the output variables (columns) at certain output intervals (rows), and – optionally – a mapping variable (by default the first column).
parms	parameters passed to func; should be either a vector, or a list with named elements. If NULL, then the first element of parInput is taken.
sensvar	the output variables for which the sensitivity needs to be estimated. Either NULL, the default, which selects all variables, or a vector with variable names (which should be present in the matrix returned by func), or a vector with indices to variables as present in the output matrix (note that the column of this matrix with the mapping variable should not be selected).
senspar	the parameters whose sensitivity needs to be estimated, the default=all parameters. Either a vector with parameter *names*, or a vector with *indices* to positions of parameters in parms.
varscale	the scaling (weighing) factor for sensitivity variables, NULL indicates that the variable value is used.
parscale	the scaling (weighing) factor for sensitivity parameters, NULL indicates that the parameter value is used.
tiny	the perturbation, or numerical difference, factor, see details.
map	the column number with the (independent) mapping variable in the output matrix returned by func. For dynamic models solved by integration, this will be the (first) column with time. For 1-D spatial output, this column will be some distance variable. Set to NULL if there is no mapping variable. Mapping variables should not be selected for estimating sensitivity functions; they are used for plotting.
...	additional arguments passed to func or to the methods.
object	an object of class sensFun.
x	an object of class sensFun.
vars	if FALSE: summaries per parameter are returned; if TRUE, summaries per parameter and per variable are returned.
which	the name or the index to the variables that should be plotted. Default = all variables.

legpos	position of the legend; set to NULL to avoid plotting a legend.
ask	logical; if TRUE, the user is *ask*ed before each plot, if NULL the user is only asked if more than one page of plots is necessary and the current graphics device is set interactive, see par(ask = ...) and dev.interactive.

Details

There are essentially two ways in which to use function sensFun.

- When func returns a matrix or data frame with output values, sensFun can be used for sensitivity analysis, estimating the *impact* of parameters on output variables.
- When func returns an instance of class modCost (as returned by a call to function modCost), then sensFun can be used for *parameter identifiability*. In this case the results from sensFun are used as input to function collin. See the help file for collin.

For each sensitivity parameter, the number of sensitivity functions estimated is: length(sensvar) * length(mapping variable), i.e. one for each element returned by func (except the mapping variable).

The sensitivity functions are estimated numerically. This means that each parameter value Θ_j is perturbed as $\max(tiny, \Theta_j \cdot (1 + tiny))$

Value

a data.frame of class sensFun containing the sensitivity functions this is one row for each sensitivity variable at each independent (time or position) value and the following columns:

x, the value of the independent (mapping) variable, usually time (solver= "ode.."), or distance (solver= "steady.1D")

var, the name of the observed variable,

..., a number of columns, one for each sensitivity parameter

The data.frame returned by sensFun has methods for the generic functions summary, plot, pairs – see note.

Note

Sensitivity functions are generated by perturbing one by one the parameters with a very small amount, and quantifying the differences in the output.

It is important that the output is generated with high precision, else it is possible, that the sensitivity functions are just noise. For instance, when used with a dynamic model (using solver from deSolve) set the tolerances atol and rtol to a lower value, to see if the sensitivity results make sense.

The following methods are provided:

- *summary*. Produces summary statistics of the sensitivity functions, a data.frame with: one row for each parameter and the following columns:
  - L1: the L1-norm $\frac{1}{n} \cdot \sum |S_{ij}|$,
  - L2: the L2-norm $\cdot \sqrt{\frac{1}{n} \sum S_{ij} \cdot S_{ij}}$,
  - Mean: the mean of the sensitivity functions,

  - Min: the minimal value of the sensitivity functions,
  - Max: the maximal value of the sensitivity functions.

- *var* the summary of the variables sensitivity functions, a data.frame with the same columns as `model` and one row for each parameter + variable combination. This is only outputted if the variable names are effectively known
- *plot* plots the sensitivity functions for each parameter; each parameter has its own color.

  By default, the sensitivity functions for all variables are plotted in one figure, unless `which` gives a selection of variables; in that case, each variable will be plotted in a separate figure, and the figures aligned in a rectangular grid, unless par `mfrow` is passed as an argument.
- *pairs* produces a pairs plot of the sensitivity results; per parameter.

  By default, the sensitivity functions for all variables are plotted in one figure, unless `which` gives a selection of variables.

  Overrides the default `gap = 0`, `upper.panel = NA`, and `diag.panel`.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References

Soetaert, K. and Herman, P. M. J., 2009. A Practical Guide to Ecological Modelling – Using R as a Simulation Platform. Springer, 390 pp.

Brun, R., Reichert, P. and Kunsch, H.R., 2001. Practical Identifiability Analysis of Large Environmental Simulation Models. Water Resour. Res. 37(4): 1015–1030.

Soetaert, K. and Petzoldt, T., 2010. Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME. Journal of Statistical Software 33(3) 1–28. http://www.jstatsoft.org/v33/i03

Examples

```
## =============================================================================
## Bacterial growth model as in Soetaert and Herman, 2009
## =============================================================================
pars <- list(gmax = 0.5, eff = 0.5,
             ks = 0.5, rB = 0.01, dB = 0.01)

solveBact <- function(pars) {
  derivs <- function(t, state, pars) { # returns rate of change
    with (as.list(c(state, pars)), {
      dBact <-  gmax * eff * Sub/(Sub + ks) * Bact - dB * Bact - rB * Bact
      dSub  <- -gmax       * Sub/(Sub + ks) * Bact + dB * Bact
      return(list(c(dBact, dSub)))
    })
  }
  state   <- c(Bact = 0.1, Sub = 100)
  tout    <- seq(0, 50, by = 0.5)
  ## ode solves the model by integration ...
  return(as.data.frame(ode(y = state, times = tout, func = derivs,
```

```
    parms = pars)))
}

out <- solveBact(pars)

plot(out$time, out$Bact, ylim = range(c(out$Bact, out$Sub)),
     xlab = "time, hour", ylab = "molC/m3", type = "l", lwd = 2)
lines(out$time, out$Sub, lty = 2, lwd = 2)
lines(out$time, out$Sub + out$Bact)

legend("topright", c("Bacteria", "Glucose", "TOC"),
       lty = c(1, 2, 1), lwd = c(2, 2, 1))

## sensitivity functions
SnsBact <- sensFun(func = solveBact, parms = pars,
                   sensvar = "Bact", varscale = 1)
head(SnsBact)
plot(SnsBact)
plot(SnsBact, type = "b", pch = 15:19, col = 2:6,
     main = "Sensitivity all vars")

summary(SnsBact)
plot(summary(SnsBact))

SF <- sensFun(func = solveBact, parms = pars,
              sensvar = c("Bact", "Sub"), varscale = 1)
head(SF)
tail(SF)

summary(SF, var = TRUE)

plot(SF)
plot(SF, which = c("Sub","Bact"))
pm <- par(mfrow = c(1,3))
plot(SF, which = c("Sub", "Bact"), mfrow = NULL)
plot(SF, mfrow = NULL)
par(mfrow = pm)

## Bivariate sensitivity
pairs(SF)  # same color
pairs(SF, which = "Bact", col = "green", pch = 15)
pairs(SF, which = c("Bact", "Sub"), col = c("green", "blue"))
mtext(outer = TRUE, side = 3, line = -2,
      "Sensitivity functions", cex = 1.5)

## pairwise correlation
cor(SnsBact[,-(1:2)])
```

---

sensRange *Sensitivity Ranges of a Timeseries or 1-D Variables*

Description

Given a model consisting of differential equations, estimates the global effect of certain (sensitivity) parameters on a time series or on 1-D spatial series of selected sensitivity variables.

This is done by drawing parameter values according to some predefined distribution, running the model with each of these parameter combinations, and calculating the values of the selected output variables at each output interval.

This function thus produces 'envelopes' around the sensitivity variables.

Usage

```
sensRange(func, parms = NULL, sensvar = NULL, dist = "unif",
          parInput = NULL, parRange = NULL, parMean = NULL,
          parCovar = NULL, map = 1, num = 100,  ...)

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

## S3 method for class 'summary.sensRange'
plot(x, xyswap = FALSE,
              which = NULL, legpos = "topleft",
              col = c(grey(0.8), grey(0.7)),
              quant = FALSE, ask = NULL, obs = NULL,
              obspar = list(), ...)

## S3 method for class 'sensRange'
plot(x, xyswap = FALSE,
              which = NULL, ask = NULL, ...)
```

Arguments

func	an R-function that has as first argument parms and that returns a matrix or data.frame with the values of the output variables (columns) at certain output intervals (rows), and – optionally – a mapping variable (by default the first column).
parms	parameters passed to func; should be either a vector, or a list with named elements. If NULL, then the first element of parInput is taken.
sensvar	the output variables for which the sensitivity needs to be estimated. Either NULL, the default, which selects all variables, or a vector with variable names (which should be present in the matrix returned by func), or a vector with indices to variables as present in the output matrix (note that the column of this matrix with the mapping variable should not be selected).
dist	the distribution according to which the parameters should be generated, one of "unif" (uniformly random samples), "norm", (normally distributed random samples), "latin" (latin hypercube distribution), "grid" (parameters arranged on a grid). The input parameters for the distribution are specified by parRange (min,max), except for the normally distributed parameters, in which case the

	distribution is specified by the parameter means parMean and the variance-covariance matrix, parCovar. Note that, if the distribution is "norm" and parRange is given, then a truncated distribution will be generated. (This is useful to prevent for instance that certain parameters become negative). Ignored if parInput is specified.
parRange	the range (min, max) of the sensitivity parameters, a matrix or (preferred) a data.frame with one row for each parameter, and two columns with the minimum (1st) and maximum (2nd) value. The rownames of parRange should be parameter names that are known in argument parms. Ignored if parInput is specified.
parInput	a matrix with dimension (*, npar) with the values of the sensitivity parameters.
parMean	only when dist is "norm": the mean value of each parameter. Ignored if parInput is specified.
parCovar	only when dist is "norm": the parameter's variance-covariance matrix.
num	the number of times the model has to be run. Set large enough. If parInput is specified, then num parameters are selected randomly (from the rows of parInput.
map	the column number with the (independent) mapping variable in the output matrix returned by func. For dynamic models solved by integration, this will be the (first) column with time. For 1-D spatial output, this column will be some distance variable. Set to NULL if there is no mapping variable. Mapping variables should not be selected for estimating sensitivity ranges; they are used for plotting.
object	an object of class sensRange.
x	an object of class sensRange.
legpos	position of the legend; set to NULL to avoid plotting a legend.
xyswap	if TRUE, then x-and y-values are swapped and the y-axis is from top to bottom. Useful for drawing vertical profiles.
which	the name or the index to the variables that should be plotted. Default = all variables.
col	the two colors of the polygons that should be plotted.
quant	if TRUE, then the median surrounded by the quantiles q25-q75 and q95-q95 are plotted, else the min-max and mean +- sd are plotted.
ask	logical; if TRUE, the user is *ask*ed before each plot, if NULL the user is only asked if more than one page of plots is necessary and the current graphics device is set interactive, see par(ask=...) and dev.interactive.
obs	a data.frame or matrix with "observed data" that will be added as points to the plots. obs can also be a list with multiple data.frames and/or matrices containing observed data. The first column of obs should contain the time or space-variable. If obs is not NULL and which is NULL, then the variables, common to both obs and x will be plotted.
obspar	additional graphics arguments passed to points, for plotting the observed data. If obs is a list containing multiple observed data sets, then the graphics arguments can be a vector or a list (e.g. for xlim, ylim), specifying each data set separately.
...	additional arguments passed to func or to the methods.

Details

Models solved by integration (i.e. by using one of 'ode', 'ode.1D', 'ode.band', 'ode.2D'), have the output already in a form usable by sensRange.

Value

a data.frame of type sensRange containing the parameter set and the corresponding values of the sensitivity output variables.

The list returned by sensRange has a method for the generic functions summary,plot and plot.summary – see note.

Note

The following *methods* are included:

- summary, estimates summary statistics for the sensitivity variables, a data.frame with as many rows as there are mapping variables (or rows in the matrix returned by func) and the following columns: x, the mapping value, Mean, the mean, sd, the standard deviation, Min, the minimal value, Max, the maximal value, q25, q50, q75, the 25th, 50 and 75% quantile
- plot, produces a "matplot" of the sensRange output, one plot for each sensitivity variable and with the mapping variable on the x-axis.

  Each variable will be plotted in a separate figure, and the figures aligned in a rectangular grid, unless par mfrow is passed as an argument.
- summary.plot, produces a plot of the summary of the sensRange output, one plot for each sensitivity variable and with the ranges and mean +- standard deviation or the quantiles as coloured polygons.

  Each variable will be plotted in a separate figure, and the figures aligned in a rectangular grid, unless par mfrow is passed as an argument.

The output for models solved by a steady-state solver (i.e. one of 'steady', 'steady.1D', 'steady.band', 'steady.2D' needs to be rearranged – see examples.

For plot.summary.sensRange and plot.sensRange, the number of panels per page is automatically determined up to 3 x 3 (par(mfrow = c(3, 3))). This default can be overwritten by specifying user-defined settings for mfrow or mfcol. Set mfrow equal to NULL to avoid the plotting function to change user-defined mfrow or mfcol settings.

Other graphical parameters can be passed as well. Parameters are vectorized, either according to the number of plots (xlab, ylab, main, sub, xlim, ylim, log, asp, ann, axes, frame.plot,panel.first,panel.last, cex.lab,cex.axis,cex.main) or according to the number of lines within one plot (other parameters e.g. col, lty, lwd etc.) so it is possible to assign specific axis labels to individual plots, resp. different plotting style. Plotting parameter ylim, or xlim can also be a list to assign different axis limits to individual plots.

Similarly, the graphical parameters for observed data, as passed by obspar can be vectorized, according to the number of observed data sets (when obs is a list).

The data.frame of type sensRange has several attributes, which remain hidden, and which are generally not of practical use (they are needed for the S3 methods).

There is one exception, i.e. if parameter values are imposed via argument parInput, and these parameters are generated by a Markov chain (modMCMC). If the number of draws, num, is less than

the number of rows in parInput, then num random draws will be taken. Attribute, "pset" then contains the index to the parameters that have been selected.

The sensRange method only represents the distribution of the model response variables as a function of the parameter values. But an additional source of noise is due to the *model error*, as represented by the sampled values of sigma in the Markov chain. In order to represent also this source of error, gaussian noise should be added to each sensitivity output variables, with a standard deviation that corresponds to the original parameter draw – see vignette "FMEother".

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

References

Soetaert, K. and Petzoldt, T., 2010. Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME. Journal of Statistical Software 33(3) 1–28. http://www.jstatsoft.org/v33/i03

Examples

```
## =============================================================================
## Bacterial growth model from Soetaert and Herman, 2009
## =============================================================================

pars <- list(gmax = 0.5,eff = 0.5,
             ks = 0.5, rB = 0.01, dB = 0.01)

solveBact <- function(pars) {
  derivs <- function(t,state,pars) {    # returns rate of change
    with (as.list(c(state,pars)), {
      dBact <- gmax*eff * Sub/(Sub + ks)*Bact - dB*Bact - rB*Bact
      dSub  <- -gmax    * Sub/(Sub + ks)*Bact + dB*Bact
      return(list(c(dBact,dSub)))
    })
  }

  state <- c(Bact = 0.1,Sub = 100)
  tout  <- seq(0, 50, by = 0.5)
  ## ode solves the model by integration ...
  return(as.data.frame(ode(y = state, times = tout, func = derivs,
    parms = pars)))
}

out <- solveBact(pars)

mf  <-par(mfrow = c(2,2))

plot(out$time, out$Bact, main = "Bacteria",
     xlab = "time, hour", ylab = "molC/m3", type = "l", lwd = 2)
```

```
## the sensitivity parameters
parRanges <- data.frame(min = c(0.4, 0.4, 0.0), max = c(0.6, 0.6, 0.02))
rownames(parRanges)<- c("gmax", "eff", "rB")
parRanges

tout <- 0:50
## sensitivity to rB; equally-spaced parameters ("grid")
SensR <- sensRange(func = solveBact, parms = pars, dist = "grid",
                   sensvar = "Bact", parRange = parRanges[3,], num = 50)

Sens  <-summary(SensR)
plot(Sens, legpos = "topleft", xlab = "time, hour", ylab = "molC/m3",
     main = "Sensitivity to rB", mfrow = NULL)

## sensitivity to all; latin hypercube
Sens2 <- summary(sensRange(func = solveBact, parms = pars, dist = "latin",
           sensvar = c("Bact", "Sub"), parRange = parRanges, num = 50))

## Plot all variables; plot mean +- sd, min max
plot(Sens2, xlab = "time, hour", ylab = "molC/m3",
     main = "Sensitivity to gmax,eff,rB", mfrow = NULL)

par(mfrow = mf)

## Select one variable for plotting; plot the quantiles
plot(Sens2, xlab = "time, hour", ylab = "molC/m3", which = "Bact", quant = TRUE)

## Add data
data <- cbind(time = c(0,10,20,30), Bact = c(0,1,10,45))
plot(Sens2, xlab = "time, hour", ylab = "molC/m3", quant = TRUE,
  obs = data, obspar = list(col = "darkblue", pch = 16, cex = 2))
```

`Unif` *Uniform Random Distribution*

Description

Generates uniformly distributed random parameter sets.

Usage

```
Unif(parRange, num)
```

Arguments

`parRange`	the range (min, max) of the parameters, a matrix or a data.frame with one row for each parameter, and two columns with the minimum (1st) and maximum (2nd) value.

num the number of random parameter sets to generate.

Details

In the uniform sampling, each parameter is uniformly random distributed over its range.

Value

a matrix with one row for each generated parameter set, and one column per parameter.

Note

For small sample sizes, the latin hypercube distributed parameter sets (Latinhyper) may give better coverage in parameter space than the uniform random design.

Author(s)

Karline Soetaert <karline.soetaert@nioz.nl>

See Also

Norm for (multi)normally distributed random parameter sets.

Latinhyper to generates parameter sets using latin hypercube sampling.

Grid to generate random parameter sets arranged on a regular grid

runif the R-default for generating uniformally distributed random numbers.

Examples

```
## 4 parameters
parRange <- data.frame(min = c(0, 1, 2, 3), max = c(10, 9, 8, 7))
rownames(parRange) <- c("par1", "par2", "par3", "par4")

## uniform
pairs(Unif(parRange, 100), main = "Uniformly random")
```

Index