Package: dMod (via r-universe)

September 11, 2024

Type Package

Title Dynamic Modeling and Parameter Estimation in ODE Models

Version 1.1.0

Date 2020-02-10

Author Daniel Kaschek

Maintainer Daniel Kaschek <daniel.kaschek@gmail.com>

Description The framework provides functions to generate ODEs of reaction networks, parameter transformations, observation functions, residual functions, etc. The framework follows the paradigm that derivative information should be used for optimization whenever possible. Therefore, all major functions produce and can handle expressions for symbolic derivatives.

License GPL (>= 2)

Depends cOde (>= 1.1)

Imports deSolve, rootSolve, ggplot2, parallel, stringr, plyr, dplyr, foreach, doParallel, rjson, stringr, crayon, yaml, rlang, data.table, purrr

Suggests MASS, rPython, pander, knitr, rmarkdown,

RoxygenNote 7.3.2

VignetteBuilder knitr

Repository https://dkaschek.r-universe.dev

RemoteUrl https://github.com/dkaschek/dmod

RemoteRef HEAD

RemoteSha 485b6255cd7421c3964ae35abfd2bcf2d21d6ea7

Contents

`*.fn` *Concatenation of functions*

Description

Used to concatenate observation functions, prediction functions and parameter transformation functions.

Usage

```
## S3 method for class 'fn'
p1 * p2
```

Arguments

p1	function of class obsfn, prdfn, parfn or idfn
p2	function of class obsfn, prdfn, parfn or idfn

Value

Object of the same class as x1 and x2.

Examples

```
# Define a time grid on which to make a prediction by peace-wise linear function.
# Then define a (generic) prediction function based on thid grid.
times <- 0:5
grid <- data.frame(name = "A", time = times, row.names = paste0("p", times))
x <- Xd(grid)

# Define an observable and an observation function
observables <- eqnvec(Aobs = "s*A")
g <- Y(g = observables, f = NULL, states = "A", parameters = "s")

# Collect parameters and define an overarching parameter transformation
# for two "experimental condtions".
dynpars <- attr(x, "parameters")
obspars <- attr(g, "parameters")
innerpars <- c(dynpars, obspars)

trafo <- structure(innerpars, names = innerpars)
trafo_C1 <- replaceSymbols(innerpars, paste(innerpars, "C1", sep = "_"), trafo)
trafo_C2 <- replaceSymbols(innerpars, paste(innerpars, "C2", sep = "_"), trafo)

p <- NULL
p <- p + P(trafo = trafo_C1, condition = "C1")
p <- p + P(trafo = trafo_C2, condition = "C2")

# Collect outer (overarching) parameters and
# initialize with random values
outerpars <- attr(p, "parameters")
pars <- structure(runif(length(outerpars), 0, 1), names = outerpars)

# Predict internal/unobserved states
out1 <- (x*p)(times, pars)
plot(out1)

# Predict observed states in addition to unobserved
out2 <- (g*x*p)(times, pars)
plot(out2)
```

`+.datalist` *Direct sum of datasets*

Description

Used to merge datasets with overlapping conditions.

Usage

```
## S3 method for class 'datalist'
data1 + data2
```

Arguments

`data1`	dataset of class `datalist`
`data2`	dataset of class `datalist`

Details

Each data list contains data frames for a number of conditions. The direct sum of datalist is meant as merging the two data lists and returning the overarching datalist.

Value

Object of class `datalist` for the union of conditions.

Examples

```
# Start with two data frames
mydata1 <- data.frame(
  name = "A",
  time = 0:1,
  value = 1:2,
  sigma = .1,
  compound = c("DEM", "APAP"),
  dose = "0.1"
)

mydata2 <- data.frame(
  name = "A",
  time = 0:1,
  value = 3:4,
  sigma = .1,
  compound = c("APAP", "DCF"),
  dose = "0.1"
)

# Create datalists from dataframes
data1 <- as.datalist(mydata1, split.by = c("compound", "dose"))
```

```
data2 <- as.datalist(mydata2, split.by = c("compound", "dose"))

# Direct sum of datalists
 data <- data1 + data2
 print(data)

# Check the condition.grid (if available)
 condition.grid <- attr(data, "condition.grid")
 print(condition.grid)
```

`+.fn` *Direct sum of functions*

Description

Used to add prediction function, parameter transformation functions or observation functions.

Usage

```
## S3 method for class 'fn'
x1 + x2
```

Arguments

`x1`	function of class `obsfn`, `prdfn` or `parfn`
`x2`	function of class `obsfn`, `prdfn` or `parfn`

Details

Each prediction function is associated to a number of conditions. Adding functions means merging or overwriting the set of conditions.

Value

Object of the same class as `x1` and `x2` which returns results for the union of conditions.

See Also

P, Y, Xs

Examples

```
# Define a time grid on which to make a prediction by peace-wise linear function.
# Then define a (generic) prediction function based on thid grid.
times <- 0:5
grid <- data.frame(name = "A", time = times, row.names = paste0("p", times))
x <- Xd(grid)
```

```
# Define an observable and an observation function
observables <- eqnvec(Aobs = "s*A")
g <- Y(g = observables, f = NULL, states = "A", parameters = "s")

# Collect parameters and define an overarching parameter transformation
# for two "experimental condtions".
dynpars <- attr(x, "parameters")
obspars <- attr(g, "parameters")
innerpars <- c(dynpars, obspars)

trafo <- structure(innerpars, names = innerpars)
trafo_C1 <- replaceSymbols(innerpars, paste(innerpars, "C1", sep = "_"), trafo)
trafo_C2 <- replaceSymbols(innerpars, paste(innerpars, "C2", sep = "_"), trafo)

p <- NULL
p <- p + P(trafo = trafo_C1, condition = "C1")
p <- p + P(trafo = trafo_C2, condition = "C2")

# Collect outer (overarching) parameters and
# initialize with random values
outerpars <- attr(p, "parameters")
pars <- structure(runif(length(outerpars), 0, 1), names = outerpars)

# Predict internal/unobserved states
out1 <- (x*p)(times, pars)
plot(out1)

# Predict observed states in addition to unobserved
out2 <- (g*x*p)(times, pars)
plot(out2)
```

`+.objfn` *Direct sum of objective functions*

Description

Direct sum of objective functions

Usage

```
## S3 method for class 'objfn'
x1 + x2
```

Arguments

`x1`	function of class `objfn`
`x2`	function of class `objfn`

Details

The objective functions are evaluated and their results as added. Sometimes, the evaluation of an objective function depends on results that have been computed internally in a preceding objective function. Therefore, environments are forwarded and all evaluations take place in the same environment. The first objective function in a sum of functions generates a new environment.

Value

Object of class `objfn`.

See Also

normL2, constraintL2, priorL2, datapointL2

Examples

```
## Generate three objective functions
prior <- structure(rep(0, 5), names = letters[1:5])

obj1 <- constraintL2(mu = prior, attr.name = "center")
obj2 <- constraintL2(mu = prior + 1, attr.name = "right")
obj3 <- constraintL2(mu = prior - 1, attr.name = "left")

## Evaluate first objective function on a random vector
pouter <- prior + rnorm(length(prior))
print(obj1(pouter))

## Split into fixed and non-fixed part
fixed <- pouter[4:5]
pouter <- pouter[1:3]
print(obj1(pouter, fixed = fixed))


## Visualize the result by a parameter profile
myfit <- trust(obj1, pouter, rinit = 1, rmax = 10, fixed = fixed)
myprof <- profile(obj1, myfit$argument, "a", fixed = fixed)
plotProfile(myprof)


## Create new objective function by adding the single ones,
## then evalue the random vector again
pouter <- prior + rnorm(length(prior))
obj <- obj1 + obj2 + obj3
print(obj(pouter))
```

+.objlist *Add two lists element by element*

Description

Add two lists element by element

Usage

```
## S3 method for class 'objlist'
out1 + out2
```

Arguments

out1	List of numerics or matrices
out2	List with the same structure as out1 (there will be no warning when mismatching)

Details

If out1 has names, out2 is assumed to share these names. Each element of the list out1 is inspected. If it has a `names` attributed, it is used to do a matching between out1 and out2. The same holds for the attributed `dimnames`. In all other cases, the "+" operator is applied the corresponding elements of out1 and out2 as they are.

Value

List of length of out1.

Examples

```
objlist1 <- dMod:::init_empty_objlist(c(a = 1, b = 2))
objlist1$value  = 1; objlist1$gradient[1:2] <- 1; objlist1$hessian[1:4] <- 1
objlist2 <- dMod:::init_empty_objlist(c(a = 1, d = 2))
objlist2$value  = 1; objlist2$gradient[1:2] <- 1; objlist2$hessian[1:4] <- 1
objlist1 + objlist2
```

addReaction *Add reaction to reaction table*

Description

Add reaction to reaction table

Usage

```
addReaction(eqnlist, from, to, rate, description = names(rate))
```

Arguments

eqnlist	equation list, see eqnlist
from	character with the left hand side of the reaction, e.g. "2*A + B"
to	character with the right hand side of the reaction, e.g. "C + 2*D"
rate	character. The rate associated with the reaction. The name is employed as a description of the reaction.
description	Optional description instead of names(rate).

Value

An object of class eqnlist.

Examples

```
f <- eqnlist()
f <- addReaction(f, "2*A+B", "C + 2*D", "k1*B*A^2")
f <- addReaction(f, "C + A", "B + A", "k2*C*A")



  # Write your example here. You can also add more Start..End blocks if needed.
  # Please mask all output such as print() with the special tag
  #
  # such that the test is not littered. Statements guarded by  are enabled
  # in the example file which is extracted from this test file. To extract the
  # example run
  #    extractExamples()
  # on the R command line.

    ## Generate another equation list
    eq <- eqnlist()
    eq <- addReaction(eq, "A", "pA", "act_A * A * stimulus", "Phosphorylation of A")
    eq <- addReaction(eq, "pA", "A", "deact_A * pA", "Deposphorylation of pA")
   eq <- addReaction(eq, "2*pA", "pA_pA", "form_complex_pA * pA^2", "Complex formation of pA")
    eq <- addReaction(eq, "B", "pB", "act_B * B * pA_pA", "Phosphorylation of B")
    eq <- addReaction(eq, "pB", "B", "deact_B * pB", "Deposphorylation of pB")

    ## Extract data.frame of reactions
    reactions <- getReactions(eq)
     print(reactions)

    ## Get conserved quantities
    cq <- conservedQuantities(eq$smatrix)
     print(cq)

    ## Get fluxes
    fluxes <- getFluxes(eq)
     print(fluxes)

    ## Subsetting of equation list
```

```
subeq1 <- subset(eq, "pB" %in% Product)
 print(subeq1)
subeq2 <- subset(eq, grepl("not_available", Description))
 print(subeq2)

## Time derivatives of observables
observables <- eqnvec(pA_obs = "s1*pA", tA_obs = "s2*(A + pA)")
dobs <- dot(observables, eq)

## Combined equation vector for ODE and observables
f <- c(as.eqnvec(eq), dobs)
 print(f)
```

---

appendObj *Append an objective function to a basic dMod.frame*

Description

Append an objective function to a basic dMod.frame

Usage

```
appendObj(
  dMod.frame,
  prd = list(g * (x * p)),
  obj_data = list(normL2(data, prd, e)),
  obj = list(obj_data),
 pars = list(structure(rnorm(length(getParameters(obj))), names = getParameters(obj))),
 times = list(seq(min(as.data.frame(data)[["time"]]), max(as.data.frame(data)[["time"]])
    * 1.1, length.out = 200)),
  ...,
  keepCalls = F
)
```

Arguments

dMod.frame	A dMod.frame
prd	Expression after which the concatenated prediction function is formed. Has to wrapped in list()
obj_data	Expression after which the objective function which describes the data is formed. Has to wrapped in list()
obj	This object is taken by the standard fitting functions. At typical expression would be `list(obj_data + constr)`. Has to wrapped in list()
pars	A named vector of parameters to run e.g. test simulations of the model. Defaults to random parameters

times	A vector of times to run e.g. test simulations of the model. Defaults to `seq(0, 1*t_max(data), length.out = 200)`
...	Other columns which are mutations of existing ones or new columns.
keepCalls	Store a record of the calls in a new colun? See mutatedMod.frame.

Value

The dMod.frame augmented by standardized columns

`appendParframes` *Make a column "parframes" out of "fits"*

Description

Most plotting functions rely on a column "parframes" to be existent in the dMod.frame

Usage

```
appendParframes(
  dMod.frame,
  parframes = list(as.parframe(fits)),
  keepFits = F,
  ...,
  keepCalls = F
)
```

Arguments

dMod.frame	A dmod.frame, preferably with a column `fits`.
parframes	Expression to turn a column containing a parlist (e.g. fits) into a column of parframes
keepFits	
...	Other columns you want to mutate
keepCalls	Store a record of the calls in a new colun? See mutatedMod.frame.

Value

The dMod.frame containing the column "parframes"

as.data.frame.datalist *Coerce to a Data Frame*

Description

Coerce to a Data Frame

Usage

```
## S3 method for class 'datalist'
as.data.frame(x, ...)

## S3 method for class 'prdlist'
as.data.frame(x, ..., data = NULL, errfn = NULL)
```

Arguments

x	any R object
...	not used right now
data	data list oject
errfn	obsfn object, the error model function to predict sigma

Value

a data frame

as.data.frame.eqnlist *Coerce equation list into a data frame*

Description

Coerce equation list into a data frame

Usage

```
## S3 method for class 'eqnlist'
as.data.frame(x, ...)
```

Arguments

x	object of class eqnlist
...	other arguments

Value

a `data.frame` with columns "Description" (character), "Rate" (character), and one column per ODE state with the state names. The state columns correspond to the stoichiometric matrix.

`as.eqnvec` *Coerce to an equation vector*

Description

An equation list stores an ODE in a list format. The function translates this list into the right-hand sides of the ODE.

Usage

```
as.eqnvec(x, ...)

## S3 method for class 'character'
as.eqnvec(x = NULL, names = NULL, ...)

## S3 method for class 'eqnlist'
as.eqnvec(x, ...)
```

Arguments

`x`	object of class `character` or `eqnlist`
`...`	arguments going to the corresponding methods
`names`	character, the left-hand sides of the equation

Details

If `x` is of class `eqnlist`, getFluxes is called and coerced into a vector of equations.

Value

object of class eqnvec.

as.eventlist *Coerce to eventlist*

Description

Coerce to eventlist

Usage

```
as.eventlist(x, ...)

## S3 method for class 'list'
as.eventlist(x, ...)

## S3 method for class 'data.frame'
as.eventlist(x, ...)
```

Arguments

x	list, data.frame
...	not used

as.objlist *Generate objective list from numeric vector*

Description

Generate objective list from numeric vector

Usage

```
as.objlist(p)
```

Arguments

p	Named numeric vector

Value

list with entries value (`0`), gradient (`rep(0, length(p))`) and hessian (`matrix(0, length(p), length(p))`) of class `obj`.

Examples

```
p <- c(A = 1, B = 2)
as.objlist(p)
```

`as.parframe.parlist` *Coerce object to a parameter frame*

Description

Coerce object to a parameter frame

Usage

```
## S3 method for class 'parlist'
as.parframe(x, sort.by = "value", ...)

as.parframe(x, ...)
```

Arguments

`x`	object to be coerced
`sort.by`	character indicating by which colum the returned parameter frame should be sorted. Defaults to `"value"`.
`...`	other arguments

Value

object of class parframe.

Examples

```
## Generate a prediction function
regfn <- c(y = "sin(a*time)")

g <- Y(regfn, parameters = "a")
x <- Xt(condition = "C1")

## Generate data
data <- datalist(
  C1 = data.frame(
    name = "y",
    time = 1:5,
    value = sin(1:5) + rnorm(5, 0, .1),
    sigma = .1
  )
)

## Initialize parameters and time
pars <- c(a = 1)
times <- seq(0, 5, .1)

plot((g*x)(times, pars), data)
```

```
## Do many fits from random positions and store them into parlist
out <- as.parlist(lapply(1:50, function(i) {
  trust(normL2(data, g*x), pars + rnorm(length(pars), 0, 1), rinit = 1, rmax = 10)
}))

summary(out)

## Reduce parlist to parframe
parframe <- as.parframe(out)
plotValues(parframe)

## Reduce parframe to best fit
bestfit <- as.parvec(parframe)
plot((g*x)(times, bestfit), data)
```

`as.parvec.parframe` *Select a parameter vector from a parameter frame.*

Description

Obtain a parameter vector from a parameter frame.

Usage

```
## S3 method for class 'parframe'
as.parvec(x, index = 1, ...)
```

Arguments

`x`	A parameter frame, e.g., the output of `as.parframe`.
`index`	Integer, the parameter vector with the `index`-th lowest objective value.
`...`	not used right now

Details

With this command, additional information included in the parameter frame as the objective value and the convergence state are removed and a parameter vector is returned. This parameter vector can be used to e.g., evaluate an objective function.

On selection, the parameters in the parameter frame are ordered such, that the parameter vector with the lowest objective value is at '`index`' 1. Thus, the parameter vector with the '`index`'-th lowest objective value is easily obtained.

Value

The parameter vector with the '`index`'-th lowest objective value.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

---

attrs *Select attributes.*

Description

Select or discard attributes from an object.

Usage

```
attrs(x, atr = NULL, keep = TRUE)
```

Arguments

x	The object to work on
atr	An optional list of attributes which are either kept or removed. This parameter defaults to dim, dimnames, names, col.names, and row.names.
keep	For keep = TRUE, atr is a positive list on attributes which are kept, for keep = FALSE, 'atr' are removed.

Value

x with selected attributes.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

Mirjam Fehling-Kaschek, <mirjam.fehling@physik.uni-freiburg.de>

---

blockdiagSymb *Embed two matrices into one blockdiagonal matrix*

Description

Embed two matrices into one blockdiagonal matrix

Usage

```
blockdiagSymb(M, N)
```

Arguments

M	matrix of type character
N	matrix of type character

Value

Matrix of type character containing M and N as upper left and lower right block

Examples

```
M <- matrix(1:9, 3, 3, dimnames = list(letters[1:3], letters[1:3]))
N <- matrix(1:4, 2, 2, dimnames = list(LETTERS[1:2], LETTERS[1:2]))
blockdiagSymb(M, N)
```

---

checkout_hypothesis *Load one row of a dMod.frame into the .GlobalEnv*

Description

Load one row of a dMod.frame into the .GlobalEnv

Usage

```
checkout_hypothesis(dMod.frame, hypothesis, prefix = "", suffix = "")
```

Arguments

dMod.frame	A dMod.frame
hypothesis	character or numeric. specifying the name or the index of the hypothesis
prefix	Prefix appended to the object names in .GlobalEnv
suffix	Suffix appended to the object names in .GlobalEnv

Examples

```
testframe <- dplyr::tibble(hypothesis = c("linear", "quadratic"),
                   plots = list(plot(1:10,1:10), plot(1:10,(1:10)^2)),
                   myfun = list(function(x,a) {a * x}, function(x,a) {a * x^2}),
                   a = c(1:2))


checkout_hypothesis(testframe, "quadratic", prefix = "quad")
quadplots
quadmyfun(1:10, quada)
```

check_and_sanitize_prediction *Run some checks on the prediction in normL2_indiv*

Description

If prediction is NA for observables which are not observed in a condition, they don't matter. In this case, replace NA by 0, such that the error model can be evaluated.

Usage

```
check_and_sanitize_prediction(prediction, data, cn, FLAGNaNInfwarnings)
```

Arguments

prediction	prediction for condition cn
data	datalist
cn	condition name for which the prediction was made
FLAGNaNInfwarnings	print warnings?

Value

the prediction with harmless NA's replaced by 0

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

combine *Combine several data.frames by rowbind*

Description

Combine several data.frames by rowbind

Usage

```
combine(...)
```

Arguments

...	data.frames or matrices with not necessarily overlapping colnames

Details

This function is useful when separating models into independent csv model files, e.g.~a receptor model and several downstream pathways. Then, the models can be recombined into one model by `combine()`.

Value

A `data.frame`

Examples

```
data1 <- data.frame(Description = "reaction 1", Rate = "k1*A", A = -1, B = 1)
data2 <- data.frame(Description = "reaction 2", Rate = "k2*B", B = -1, C = 1)
combine(data1, data2)
```

`compare` *Compare two objects and return differences*

Description

Works eigher on a list or on two arguments. In case of a list, comparison is done with respect to a reference entry. Besides the objects themselves also some of their attributes are compared, i.e. "equations", "parameters" and "events" and "forcings".

Usage

```
compare(vec1, ...)

## S3 method for class 'list'
compare(vec1, vec2 = NULL, reference = 1, ...)

## S3 method for class 'character'
compare(vec1, vec2 = NULL, ...)

## S3 method for class 'eqnvec'
compare(vec1, vec2 = NULL, ...)

## S3 method for class 'data.frame'
compare(vec1, vec2 = NULL, ...)
```

Arguments

`vec1`	object of class eqnvec, `character` or `data.frame`. Alternatively, a list of such objects.
`...`	arguments going to the corresponding methods
`vec2`	same as vec1. Not used if vec1 is a list.
`reference`	numeric of length one, the reference entry.

Value

data.frame or list of data.frames with the differences.

Examples

```
## Compare equation vectors
eq1 <- eqnvec(a = "-k1*a + k2*b", b = "k2*a - k2*b")
eq2 <- eqnvec(a = "-k1*a", b = "k2*a - k2*b", c = "k2*b")
compare(eq1, eq2)

## Compare character vectors
c1 <- c("a", "b")
c2 <- c("b", "c")
compare(c1, c2)

## Compare data.frames
d1 <- data.frame(var = "a", time = 1, value = 1:3, method = "replace")
d2 <- data.frame(var = "a", time = 1, value = 2:4, method = "replace")
compare(d1, d2)

## Compare structures like prediction functions
fn1 <- function(x) x^2
attr(fn1, "equations") <- eq1
attr(fn1, "parameters") <- c1
attr(fn1, "events") <- d1

fn2 <- function(x) x^3
attr(fn2, "equations") <- eq2
attr(fn2, "parameters") <- c2
attr(fn2, "events") <- d2

mylist <- list(f1 = fn1, f2 = fn2)
compare(mylist)
```

---

compile *Compile one or more prdfn, obsfn or parfn objects*

Description

Compile one or more prdfn, obsfn or parfn objects

Usage

```
compile(..., output = NULL, args = NULL, cores = 1, verbose = F)
```

Arguments

...	Objects of class parfn, obsfn or prdfn

output	Optional character of the file to be produced. If several objects were passed, the different C files are all compiled into one shared object file.
args	Additional arguments for the R CMD SHLIB call, e.g. -leinspline.
cores	Number of cores used for compilation when several files are compiled.
verbose	Print compiler output to R command line.

---

confint.parframe *Profile uncertainty extraction*

Description

extract parameter uncertainties from profiles

Usage

```
## S3 method for class 'parframe'
confint(object, parm = NULL, level = 0.95, ..., val.column = "data")
```

Arguments

object	object of class parframe, returned from profile function.
parm	a specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.
level	the confidence level required.
...	not used right now.
val.column	the value column used in the parframe, usually 'data'.

---

conservedQuantities *Determine conserved quantites by finding the kernel of the stoichiometric matrix*

Description

Determine conserved quantites by finding the kernel of the stoichiometric matrix

Usage

```
conservedQuantities(S)
```

Arguments

S	Stoichiometric matrix

Value

Data frame with conserved quantities carrying an attribute with the number of conserved quantities.

Author(s)

Malenke Mader, <Malenka.Mader@fdm.uni-freiburg.de>

Examples

```
# Write your example here. You can also add more Start..End blocks if needed.
# Please mask all output such as print() with the special tag
#
# such that the test is not littered. Statements guarded by  are enabled
# in the example file which is extracted from this test file. To extract the
# example run
#    extractExamples()
# on the R command line.

  ## Generate another equation list
  eq <- eqnlist()
  eq <- addReaction(eq, "A", "pA", "act_A * A * stimulus", "Phosphorylation of A")
  eq <- addReaction(eq, "pA", "A", "deact_A * pA", "Deposphorylation of pA")
 eq <- addReaction(eq, "2*pA", "pA_pA", "form_complex_pA * pA^2", "Complex formation of pA")
  eq <- addReaction(eq, "B", "pB", "act_B * B * pA_pA", "Phosphorylation of B")
  eq <- addReaction(eq, "pB", "B", "deact_B * pB", "Deposphorylation of pB")

  ## Extract data.frame of reactions
  reactions <- getReactions(eq)
   print(reactions)

  ## Get conserved quantities
  cq <- conservedQuantities(eq$smatrix)
   print(cq)

  ## Get fluxes
  fluxes <- getFluxes(eq)
   print(fluxes)

  ## Subsetting of equation list
  subeq1 <- subset(eq, "pB" %in% Product)
   print(subeq1)
  subeq2 <- subset(eq, grepl("not_available", Description))
   print(subeq2)

  ## Time derivatives of observables
  observables <- eqnvec(pA_obs = "s1*pA", tA_obs = "s2*(A + pA)")
  dobs <- dot(observables, eq)

  ## Combined equation vector for ODE and observables
  f <- c(as.eqnvec(eq), dobs)
   print(f)
```

constraintExp2 *Compute a differentiable box prior*

Description

Compute a differentiable box prior

Usage

```
constraintExp2(p, mu, sigma = 1, k = 0.05, fixed = NULL)
```

Arguments

p	Named numeric, the parameter value
mu	Named numeric, the prior values, means of boxes
sigma	Named numeric, half box width
k	Named numeric, shape of box; if 0 a quadratic prior is obtained, the higher k the more box shape, gradient at border of the box (-sigma, sigma) is equal to sigma*k
fixed	Named numeric with fixed parameter values (contribute to the prior value but not to gradient and Hessian)

Value

list with entries: value (numeric, the weighted residual sum of squares), gradient (numeric, gradient) and hessian (matrix of type numeric). Object of class `objlist`.

constraintL2 *Soft L2 constraint on parameters*

Description

Soft L2 constraint on parameters

Usage

```
constraintL2(mu, sigma = 1, attr.name = "prior", condition = NULL)
```

Arguments

mu	named numeric, the prior values
sigma	named numeric of length of mu or numeric of length one or character of length of mu or character of length one
attr.name	character. The constraint value is additionally returned in an attributed with this name
condition	character, the condition for which the constraint should apply. If NULL, applies to any condition.

Details

If sigma is numeric, the function computes the constraint value

$$\left(\frac{p-\mu}{\sigma}\right)^2$$

and its derivatives with respect to p. If sigma is a character, the function computes

$$\left(\frac{p-\mu}{\sigma}\right)^2 + \log(\sigma^2)$$

and its derivatives with respect to p and sigma. Sigma parameters being passed to the function are ALWAYS assumed to be on a log scale, i.e. internally sigma parameters are converted by `exp()`.

Value

object of class `objfn`

See Also

wrss

Examples

```
mu <- c(A = 0, B = 0)
sigma <- c(A = 0.1, B = 1)
myfn <- constraintL2(mu, sigma)
myfn(pars = c(A = 1, B = -1))

# Introduce sigma parameter but fix them (sigma parameters
# are assumed to be passed on log scale)
mu <- c(A = 0, B = 0)
sigma <- paste("sigma", names(mu), sep = "_")
myfn <- constraintL2(mu, sigma)
pars <- c(A = .8, B = -.3, sigma_A = -1, sigma_B = 1)
myfn(pars = pars[c(1, 3)], fixed = pars[c(2, 4)])

# Assume same sigma parameter for both A and B
# sigma is assumed to be passed on log scale
mu <- c(A = 0, B = 0)
```

```
myfn <- constraintL2(mu, sigma = "sigma")
pars <- c(A = .8, B = -.3, sigma = 0)
myfn(pars = pars)
```

---

controls *List, get and set controls for different functions*

Description

Applies to objects of class `objfn`, `parfn`, `prdfn` and `obsfn`. Allows to manipulate different arguments that have been set when creating the objects.

Usage

```
controls(x, ...)

## S3 method for class 'objfn'
controls(x, name = NULL, ...)

## S3 method for class 'fn'
controls(x, condition = NULL, name = NULL, ...)

controls(x, ...) <- value

## S3 replacement method for class 'objfn'
controls(x, name, ...) <- value

## S3 replacement method for class 'fn'
controls(x, condition = NULL, name, ...) <- value
```

Arguments

`x`	function
`...`	arguments going to the appropriate S3 methods
`name`	character, the name of the control
`condition`	character, the condition name
`value`	the new value

Details

If called without further arguments, `controls(x)` lists the available controls within an object. Calling `controls()` with `name` and `condition` returns the control value. The value can be overwritten. If a list or data.frame ist returned, elements of those can be manipulated by the `$`- or `[]`-operator.

Value

Either a print-out or the values of the control.

Examples

```
## parfn with condition
p <- P(eqnvec(x = "-a*x"), method = "implicit", condition = "C1")
controls(p)
controls(p, "C1", "keep.root")
controls(p, "C1", "keep.root") <- FALSE

## obsfn with NULL condition
g <- Y(g = eqnvec(y = "s*x"), f = NULL, states = "x", parameters = "s")
controls(g)
controls(g, NULL, "attach.input")
controls(g, NULL, "attach.input") <- FALSE
```

---

coordTransform *Coordinate transformation for data frames*

Description

Applies a symbolically defined transformation to the `value` column of a data frame. Additionally, if a `sigma` column is present, those values are transformed according to Gaussian error propagation.

Usage

```
coordTransform(data, transformations)
```

Arguments

`data`	data frame with at least columns "name" (character) and "value" (numeric). Can optionally contain a column "sigma" (numeric).
`transformations`	character (the transformation) or named list of characters. In this case, the list names must be a subset of those contained in the "name" column.

Value

The data frame with the transformed values and sigma uncertainties.

Examples

```
mydata1 <- data.frame(name = c("A", "B"), time = 0:5, value = 0:5, sigma = .1)
coordTransform(mydata1, "log(value)")
coordTransform(mydata1, list(A = "exp(value)", B = "sqrt(value)"))
```

covariates.tbl_df *Access the covariates in the data*

Description

Access the covariates in the data

Usage

```
## S3 method for class 'tbl_df'
covariates(x, hypothesis = 1)

covariates(x)

## S3 method for class 'datalist'
covariates(x)

## S3 method for class 'data.frame'
covariates(x)
```

Arguments

x	Either a datalist or a data.frame with mandatory columns c("name", "time", "value", "sigma", "lloq").
hypothesis	The hypothesis in the dMod.frame

Value

The condition.grid of the data

createExample *Open a unit test template.*

Description

Open a unit test template.

Usage

```
createExample(test, testPath = NULL)
```

Arguments

test	Name of the unit test used as file name with "test-" prefixed.
testPath	Unit test folder. Defaults to "inst/tests".

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

`datalist` *Generate a datalist object*

Description

The datalist object stores time-course data in a list of data.frames. The names of the list serve as identifiers, e.g. of an experimental condition, etc.

Usage

```
datalist(...)

as.datalist(x, ...)

## S3 method for class 'data.frame'
as.datalist(x, split.by = NULL, keep.covariates = NULL, ...)

## S3 method for class 'list'
as.datalist(x, names = NULL, ..., condition.grid = attr(x, "condition.grid"))

## S3 replacement method for class 'datalist'
names(x) <- value

is.datalist(x)

## S3 method for class 'datalist'
c(...)
```

Arguments

`...`	data.frame objects to be coerced into a list and additional arguments
`x`	object of class `data.frame` or `list`. Data frames are required to provide "name", "time" and "value" as columns. Columns "sigma" and "lloq" can be provided. If "sigma" and "lloq" are missing, they are imputed with `NA` and `-Inf`, respectively.
`split.by`	vector of columns names which yield a unique identifier (conditions). If NULL, all columns except for the expected standard columns "name", "time", "value", "sigma" and "lloq" will be selected.
`keep.covariates`	vector of additional column names which should be kept in the condition.grid.
`names`	optional names vector, otherwise names are taken from `mylist`
`condition.grid`	Optionally, to manually specify a condition.grid
`value`	The new condition names of the datalist and its condition.grid

Details

Datalists can be plotted, see plotData and merged, see sumdatalist. They are the basic structure when combining model prediction and data via the normL2 objective function.

The standard columns of the datalist data frames are "name" (observable name), "time" (time points), "value" (data value), "sigma" (uncertainty, can be NA), and "lloq" (lower limit of quantification, `-Inf` by default).

Datalists carry the attribute `condition.grid` which contains additional information about different conditions, such as dosing information for the experiment. It can be conveniently accessed by the covariates-function. Reassigning names to a datalist also renames the rows of the `condition.grid`.

Value

Object of class `datalist`.

Object of class datalist

Examples

```
## Generate datalist from scratch
mydata1 <- data.frame(name = "A",
                      time = 0:5,
                      value = 0:5,
                      sigma = .1,
                      lloq = -0.5)

mydata2 <- data.frame(name = "A",
                      time = 0:5,
                      value = sin(0:5),
                      sigma = .1)

data <- datalist(C1 = mydata1, C2 = mydata2)
print(data)
plot(data)

## Generate datalist from singla data.frame
times <- seq(0, 2*pi, length.out = 20)
mydata <- data.frame(name = "A",
                     time = times,
                     value = c(sin(times), 1.5 * sin(times)),
                     sigma = .1,
                     stage = rep(c("upper", "lower"), each = 10),
                     phase = rep(c("first", "second"), each = 20),
                     amplitude = rep(c(1,1.5), each = 20))

data <- as.datalist(mydata, split.by = c("stage", "phase"), keep.covariates = "amplitude")
print(data)
plot(data)

condition.grid <- attr(data, "condition.grid")
print(condition.grid)
```

`datapointL2` *L2 objective function for validation data point*

Description

L2 objective function for validation data point

Usage

```
datapointL2(name, time, value, sigma = 1, attr.name = "validation", condition)
```

Arguments

`name`	character, the name of the prediction, e.g. a state name.
`time`	numeric, the time-point associated to the prediction
`value`	character, the name of the parameter which contains the prediction value.
`sigma`	numeric, the uncertainty of the introduced test data point
`attr.name`	character. The constraint value is additionally returned in an attributed with this name
`condition`	character, the condition for which the prediction is made.

Details

Computes the constraint value

$$\left(\frac{x(t) - \mu}{\sigma}\right)^2$$

and its derivatives with respect to p.

Value

List of class `objlist`, i.e. objective value, gradient and Hessian as list.

See Also

wrss, constraintL2

Examples

```
prediction <- list(a = matrix(c(0, 1), nrow = 1, dimnames = list(NULL, c("time", "A"))))
derivs <- matrix(c(0, 1, 0.1), nrow = 1, dimnames = list(NULL, c("time", "A.A", "A.k1")))
attr(prediction$a, "deriv") <- derivs
p0 <- c(A = 1, k1 = 2)

vali <- datapointL2(name = "A", time = 0, value = "newpoint", sigma = 1, condition = "a")
vali(pars = c(p0, newpoint = 1), env = .GlobalEnv)
```

datapointL2_indiv *DatapointL2 without access to stored predictions*

Description

DatapointL2 without access to stored predictions

Usage

```
datapointL2_indiv(
  name,
  time,
  value,
  sigma = 1,
  attr.name = "validation",
  condition,
  prd_indiv
)
```

Arguments

name	character, the name of the prediction, e.g. a state name.
time	numeric, the time-point associated to the prediction
value	character, the name of the parameter which contains the prediction value.
sigma	numeric, the uncertainty of the introduced test data point
attr.name	character. The constraint value is additionally returned in an attributed with this name
condition	character, the condition for which the prediction is made.
prd_indiv	a prediction function

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

define *Define parameter transformations by* define(), branch() *and* insert()

Description

Define parameter transformations by define(), branch() and insert()

Usage

```
define(trafo, expr, ..., conditionMatch = NULL)

insert(trafo, expr, ..., conditionMatch = NULL)

branch(trafo, table = NULL, conditions = rownames(table))
```

Arguments

trafo	named character vector of parametric expressions or object of class eqnvec
expr	character of the form "lhs ~ rhs" where both lhs and rhs can contain a number of symbols for which vaues are passed by the ... argument
...	used to pass values for symbols as named arguments
conditionMatch	optional character, Use as regular expression to apply the reparameterization only to conditions containing conditionMatch
table	table of covariates as data frame. Rownames are used as unique identifier, usually called "conditions", and columns represent covariates associated with these conditions.
conditions	character vector with condition names. Overwrites the rownames of table.

Value

object of the same class as trafo or list thereof, if branch() has been used.

Examples

```
# Define some parameter names
parameters <- c("A", "B", "k1", "k2")
# Define a covariate table
covtable <- data.frame(dose = c(1, 1, 10),
                       inhibitor = c("no", "inh", "no"),
                       row.names = c("Low_noInh", "Low_Inh", "High_noInh"))

# Start with an empty transformation
trans <- NULL

# Generate the identity transformation for parameters
trans <- define(trans, "x ~ x", x = parameters); print(trans)

# Insert exp(x) wherever you find x
trans <- insert(trans, "x ~ exp(x)", x = parameters); print(trans)

# Some new expressions instead of k1 and k2
trans <- insert(trans, "x ~ y", x = c("k1", "k2"), y = c("q1 + q2", "q1 - q2")); print(trans)

# Define some parameters as 0
trans <- define(trans, "x ~ 0", x = "B"); print(trans)

# The parameter name can also be directly used in the formula
```

```
trans <- insert(trans, "q1 ~ Q"); print(trans)

# Replicate the transformation 3 times with the rownames of covtable as list names
trans <- branch(trans, table = covtable); print(trans)

# Insert the rhs wherever the lhs is found in the transformation
# column names of covtable can be used to perform specific replacements
# for each transformation
trans <- insert(trans, "x ~ x_inh", x = c("Q", "q2"), inh = inhibitor); print(trans)

# Also numbers can be inserted
trans <- define(trans, "A ~ dose", dose = dose); print(trans)


# Turn that into a parameter transformation function
p <- P(trans)
parnames <- getParameters(p)
pars <- rnorm(length(parnames))
names(pars) <- parnames

p(pars)


# Advanced tricks exploiting the quoting-mechanism when capturing "..."

mydataframe <- data.frame(
  name = rep(letters[1:2], each = 3),
  value = 1:6,
  time = rep(1:3, 2),
  sigma = 0.1,
  par1 = rep(0:1, each = 3),
  par2 = rep(9:10, each = 3),
  par3 = rep(1:3, each = 2),
  stringsAsFactors = FALSE
)

parameters <- c("a", "b", "par1", "par2", "par3")
pars_to_insert <- c("par1", "par2")

# this would be the usual way when setting up a model
# pars_to_insert <- intersect(getParameters(g*x), names(data))

trafo <- define(NULL, "x~x", x = parameters)
trafo <- branch(trafo, covariates(as.datalist(mydataframe)))

# Trick 1: Access values from covariates()-Table with get/mget.
    # The names of the parameters which are supplied in the covariates()-table
    # have to be supplied manually.
trafo <- insert(trafo, "name ~ value", value = unlist(mget(pars_to_insert)), name = pars_to_insert)

# Trick 2: Access symbols from current condition-specific trafo with .currentSymbols, access
    # current condition-specific trafo by .currentTrafo
   # The input passed by the dots is "quoted" (substituted) and eval()'ed in the environment
```

```
    # of the lapply(1:length(conditions), function(i) {})
trafo <- insert(trafo, "x~exp(X)", x = .currentSymbols, X = toupper(.currentSymbols))

# Trick 3: Condition specificity. There are two ways to do this
 # 1. Apply reparametrization only for specific conditions using Regular Expressions for the
  # conditionMatch argument. This matches the condition name agains a regex
trafo <- define(NULL, "x~x", x = parameters)
trafo <- branch(trafo, covariates(as.datalist(mydataframe)))

# Conditions starting with 0_9
insert(trafo, "x~x_par3", x = "a", conditionMatch = "^0_9", par3 = par3)
# Conditions NOT starting with 0_9
insert(trafo, "x~0", x = "a", conditionMatch = "^(?!0_9)")
  # 2. Specify conditions by boolean arguments
  #    Conditions which satisfy par1 == 0
insert(trafo, "x~x_par2", par1 == 0, x = parameters, par2 = par2)
 # Special case: Pass two arguments with the same name. This is only possible if one of them
  # is logical and the other is not.
  # Conditions which satisfy par2 == 9
insert(trafo, "x~x_par2", par2 == 9, x = .currentSymbols, par2 = par2)
```

`detectFreeCores` *Detect number of free cores (on UNIX)*

Description

Read `/proc/loadavg` and subtract from the number of cores

Usage

```
detectFreeCores(machine = NULL)
```

Arguments

`machine`	character, e.g. "user@localhost".

distributed_computing *Run an any R function on a remote HPC system with SLURM*

Description

Generates R and bash scrips, transfers them to remote via ssh. 'sshpass' needs to be installed on your local machine to circumvent password entry.

Usage

```
distributed_computing(
  ...,
  jobname,
  partition = "single",
  cores = 16,
  nodes = 1,
  walltime = "01:00:00",
  ssh_passwd = NULL,
  machine = "cluster",
  var_values = NULL,
  no_rep = NULL,
  recover = T,
  purge_local = F,
  compile = F,
  custom_folders = NULL,
  resetSeeds = TRUE,
  returnAll = TRUE
)
```

Arguments

`...`	R code to be remotely executed, parameters to be changed for runs must be named `var_i` see "Details".
`jobname`	Name in characters for the run must be unique, existing will be overwritten. Must not contain the string "Minus".
`partition`	Define the partition used for the SLURM manager. Default is "single". Should only be changed if explicitly wanted.
`cores`	Number of cores per node to be used. If a value is set to 16 < value < 25, the number of possible nodes isseverely limited. 16 or less is possible on all nodes, more on none.
`nodes`	Nodes per task. Default is 1, should not be changed since `distributed_computing()` is set up to be regulated by number of repititions.
`walltime`	Estimated runtime in the format `hh:mm:ss`, default is 1h. Jobs will be canceled after the defined time.
`ssh_passwd`	To be set when the sshpass should be used to automatically authenticate on remote via passphrase. This is an obvious security nightmare...

machine	SSH address in the form user@remote_location.
var_values	List of parameter arrays. The number of arrays (i.e. list entries) must correspond to the number of parameters in the function passed to .... These parameters must be named var_i where the i must replaced by the indes of the corresponding array in var_values. The length of the arrays define the number of nodes used with the j-th node use the j-th entry of the arrays for the corresponding var_i. If no_rep is used, var_valus must be set to NULL.
no_rep	Number of repetitions. Usage of this parameter and var_values are mutual exclusive. When used the function passed to ... is executed no_rep times simultaneously using one node per realization.
recover	Logical parameter, if set to TRUE nothing is calculated, the functions check(), get() and purge() can be used on the results generated previously under the same jobname.
purge_local	Logical, if set to TRUE the purge() function also removes local files.
compile	Logical, if set to TRUE the source files are transferred to remote and compiled there. If set to FALSE, the local shared objects are transferred and used instead.
custom_folders	named vector with exact three entries named 'compiled', 'output' and 'tmp'. The values are strings with relative paths from the current working directory to the respective directory of the compiled files, the temporary folder from which files will be copied to the cluster and the output folder in which the calculated result from the cluster will be saved. The default is NULL, then everything is done from the current working directory. If only a subset of the folders should be changed, all other need to be set to ./.
resetSeeds	logical, if set to TRUE (default) the parameter vector with random seeds .Random.seeds from the transferred work space is deleted on remote. This ensures that each node has uses a different set of (pseudo) random numbers. Set to FALSE at own risk.
returnAll	logical if set to TRUE (default) all results are returned, if set to FALSE only the *result.RData files are returned.

Details

distribute_computing() generates R and bash scrips designed to run on a HPC system managed by the SLURM batch manager. The current workspace together with the scripts are exported and transferred to remote via SSH. If ssh-key authentication is not possible, the ssh-password can be passed and is used by sshpass (which has to be installed on the local machine).

The code to be executed remotely is passed to the ... argument, its final output is stored in cluster_result, which is loaded in the local workspace by the get() function.

It is possible to either run repetitions of the same program realization (by use of the no_rep parameter), or to pass a list of parameter arrays via var_values. The parameters to be changed for every run *must* be named var_i where i corresponds to the i-th array in the var_values parameter.

Value

List of functions check(), get() and purge(). check() checks, if the result is ready. get() copies all files from the remote working directory to local and loads all present results (even if not

all nodes where done) in the current active workspace as the object `cluster_result` which is a list with the results of each node as entries. `purge()` deletes the temporary folder on remote and if `purge_local` is set to `TRUE`.

Examples

```
## Not run:
out_distributed_computing <- distributed_computing(
{
  mstrust(
    objfun=objective_function,
    center=outer_pars,
    studyname = "study",
    rinit = 1,
    rmax = 10,
    fits = 48,
    cores = 16,
    iterlim = 700,
    sd = 4
  )
},
jobname = "my_name",
partition = "single",
cores = 16,
nodes = 1,
walltime = "02:00:00",
ssh_passwd = "password",
machine = "cluster",
var_values = NULL,
no_rep = 20,
recover = F,
compile = F
)
out_distributed_computing$check()
out_distributed_computing$get()
out_distributed_computing$purge()
result <- cluster_result
print(result)


# calculate profiles
var_list <- profile_pars_per_node(best_fit, 4)
profile_jobname <- paste0(fit_filename,"_profiles_opt")
method <- "optimize"
profiles_distributed_computing <- distributed_computing(
  {
    profile(
      obj = obj,
      pars =  best_fit,
      whichPar = (as.numeric(var_1):as.numeric(var_2)),
      limits = c(-5, 5),
      cores = 16,
      method = method,
```

```
      stepControl = list(
        stepsize = 1e-6,
        min = 1e-4,
        max = Inf,
        atol = 1e-2,
        rtol = 1e-2,
        limit = 100
      ),
      optControl = list(iterlim = 20)
    )
  },
  jobname = profile_jobname,
  partition = "single",
  cores = 16,
  nodes = 1,
  walltime = "02:00:00",
  ssh_passwd = "password",
  machine = "cluster",
  var_values = var_list,
  no_rep = NULL,
  recover = F,
  compile = F
)
profiles_distributed_computing$check()
profiles_distributed_computing$get()
profiles_distributed_computing$purge()
profiles  <- NULL
for (i in cluster_result) {
  profiles <- rbind(profiles, i)
}

## End(Not run)
```

`dMod.frame` *Generate a dMod.frame*

Description

Basically, a dMod.frame is a tibble, which is grouped rowwise.

Since the dMod.frame is also designed for interactive use, the class will not be called "dMod.frame" as I initially planned, but will be c("tbl_df"). This way, I don't have to overwrite all the dplyr-verbs.

The dMod.frame object stores all objects that are needed to reproducibly produce a result, such as a plot or profiles, in one row of a tibble. Each row corresponds to a distinct hypothesis, e.g. one would have two distinct rows for fitting with and without a prior.

Usage

```
dMod.frame(hypothesis, g, x, p, data, e = NULL, ...)
```

Arguments

hypothesis	Character. Description of the hypothesis
g	fn
x	fn
p	fn
data	data.frame or datalist, will be coerced to datalist
e	fn
...	other columns, have to be named. They will be placed in a list.

Value

Object of class `tbl_df`, which is grouped rowwise.

Examples

```
## Not run:

# ---- Example
library(dMod)
library(dplyr) # devtools::install_github("dlill/conveniencefunctions")

setwd(tempdir())

## Model definition (text-based, scripting part)
f <- NULL %>%
  addReaction("A", "B", "k1*A", "translation") %>%
  addReaction("B",  "", "k2*B", "degradation") %>%
  as.eqnvec()


events <- eventlist(var = "A", time = 5, value = "A_add", method = "add")

x <- odemodel(f, events = events) %>% Xs

g <- Y(c(Bobs = "s1*B"), x, compile = T, modelname = "obsfn")

conditions <- c("a", "b")

# there is a bug in
# getParameters(g*x)
parameters <- union(getParameters(g), getParameters(x))

trafo <-
  NULL %>%
  define("x~x", x = parameters) %>%
  branch(conditions = conditions) %>%
  insert("x~x_cond", x = "s1", cond = condition) %>%
  insert("x~1", x = "added", conditionMatch = "a") %>%
  insert("x~5", x = "added", conditionMatch = "b") %>%
```

```
  insert("x~exp(x)", x = getSymbols(mytrafo[[i]])) %>%
  {.}

p <- P(trafo)

# Parameter transformation for steady states
pSS <- P(f, method = "implicit", condition = NULL, compile = T)




## Process data

# Data
data <- datalist(
  a = data.frame(time = c(0, 2, 7),
                 name = "Bobs",
                 value = c(.3, .3, .3),
                 sigma = c(.03, .03, .03)),
  b = data.frame(time = c(0, 2, 7),
                 name = "Bobs",
                 value = c(.1, .1, .2),
                 sigma = c(.01, .01, .02))
)


# construct dMod.frame
myframe1 <- dMod.frame("no steady states", g, x, p, data)
print(myframe1)


# Augment by derived objects: prd, obj_data, obj, times, pars
set.seed(4)
myframe2 <- appendObj(myframe1)
print(myframe2)


# Plot the model with random pars
plotCombined(myframe2)

# Fit with prior
myframe3 <- myframe2 %>%
  mutate(constr = list(constraintL2(mu = 0*pars, sigma = 5)),
         obj = list(obj_data + constr))

myframe4 <- myframe3 %>%
 mutate(fits = list(mstrust(obj, pars, studyname = "Fits", fits = 20, cores = 4, blather = T)))


# Inspect Fits
myframe5 <- myframe4 %>%
  appendParframes
```

```
# Visualize
# Little Bug: If you want to use the dots that go to subset(), you need to specify the others
plotCombined(myframe5, 1, 1, str_detect(name, "B"))

plotPars(myframe5)

plotValues(myframe5)

plotValues(myframe5, 1, tol =0.0000000001, value >1)



# Profiles
myframe6 <- myframe5 %>%
  mutate(profiles = list(profile(obj, as.parvec(parframes), whichPar = "k1")))

myframe6$profiles %>% plotProfile()


# Validation

myframe7 <- myframe6 %>%
  mutate(vali = list(datapointL2("A", 2, "mypoint", .1, condition = "a")),
         obj_vali = list(obj_data + constr + vali),
       par_vali = list(c(dMod:::sanitizePars(as.parvec(parframes))$pars, "mypoint" = 0.1 )),
         fits_vali = list(mstrust(obj_vali, par_vali)),
       profile_vali = list(profile(obj_vali, fits_vali %>% as.parframe %>% as.parvec, "mypoint")))

myframe7$profile_vali %>% plotProfile()


# Saving ----

# Save dMod.frame as rds, stage the rds and the .so's it depends on for commit
system("git status")
git_add_dMod.frame(myframe7)
system("git status")

# Alternatetively:
zip_dMod.frame(myframe7)
# Look into you current wd to find the dMod.frame and the .so's zipped into a file.

# # Short version of above ----
#
# myframe <- dMod.frame("no steady states", g, x, p, data) %>%
#   appendObj() %>%
#   mutate(constr = list(constraintL2(mu = 0*pars, sigma = 5)),
#          obj = list(obj_data + constr),
#       fits = list(mstrust(obj, pars, studyname = "Fits", fits = 20, cores = 4, blather = T))) %>%
#   appendParframes() %>%
#   mutate(profiles = list(profile(obj, as.parvec(parframes), whichPar = "k1"))) %>%
#   mutate(vali = list(datapointL2("A", 2, "mypoint", .1, condition = "a")),
#          obj_vali = list(obj_data + constr + vali),
```

```
#       par_vali = list(c(dMod:::sanitizePars(as.parvec(parframes))$pars, "mypoint" = 0.1 )),
#          fits_vali = list(mstrust(obj_vali, par_vali)),
#       profile_vali = list(profile(obj_vali, fits_vali %>% as.parframe %>% as.parvec, "mypoint")))


# Easily test several hypotheses ----

# Fit with various prior strengths
multiframe <- dMod.frame("no steady states", g, x, p, data) %>%
  appendObj() %>%
  rbind(.,.,.,.) %>% # replicate four times
 ungroup() %>% # If you don't ungroup and run a mutate with an "lapply(1:nrow(), function(i) ...", the index i alway
 mutate(constr = map(seq_along(x), function(i) constraintL2(mu = 0*pars[[i]], sigma = 10^(i-3))),
      hypothesis = map_chr(seq_along(x), function(i) paste0(hypothesis[[i]], ", prior sigma = ", 10^(i-3)))) %>%
  rowwise() %>% #regroup by row for convenient interface to mutate()
  mutate(obj = list(obj_data + constr),
      fits = list(mstrust(obj, pars, studyname = "Fits", fits = 10, cores = 4, blather = T))) %>%
  appendParframes()


## Plot
map(1:4, function(i) multiframe %>% plotValues(hypothesis = i))

map(1:4, function(i) multiframe %>% plotCombined(hypothesis = i))

## Profiles
multiframe <- multiframe %>%
  mutate(profiles = list(profile(obj, parframes %>% as.parvec, names(pars), cores = 4)))

## Plot profiles: The "profiles"-column is already a proflist :)
multiframe$profiles %>% plotProfile() +
  coord_cartesian(ylim = c(-0.5, 4), xlim = c(-2,2))



# Quick and dirty analysis of one single hypothesis ----
checkout_hypothesis(multiframe, 4, suffix = "_weak_prior")
parframes_weak_prior



## End(Not run)
```

---

dot *Symbolic time derivative of equation vector given an equation list*

Description

The time evolution of the internal states is defined in the equation list. Time derivatives of observation functions are expressed in terms of the rates of the internal states.

Usage

```
dot(observable, eqnlist)
```

Arguments

observable	named character vector or object of type eqnvec
eqnlist	equation list

Details

Observables are translated into an ODE

Value

An object of class eqnvec

Examples

```
# Write your example here. You can also add more Start..End blocks if needed.
# Please mask all output such as print() with the special tag
#
# such that the test is not littered. Statements guarded by  are enabled
# in the example file which is extracted from this test file. To extract the
# example run
#    extractExamples()
# on the R command line.

  ## Generate another equation list
  eq <- eqnlist()
  eq <- addReaction(eq, "A", "pA", "act_A * A * stimulus", "Phosphorylation of A")
  eq <- addReaction(eq, "pA", "A", "deact_A * pA", "Deposphorylation of pA")
 eq <- addReaction(eq, "2*pA", "pA_pA", "form_complex_pA * pA^2", "Complex formation of pA")
  eq <- addReaction(eq, "B", "pB", "act_B * B * pA_pA", "Phosphorylation of B")
  eq <- addReaction(eq, "pB", "B", "deact_B * pB", "Deposphorylation of pB")

  ## Extract data.frame of reactions
  reactions <- getReactions(eq)
   print(reactions)

  ## Get conserved quantities
  cq <- conservedQuantities(eq$smatrix)
   print(cq)

  ## Get fluxes
  fluxes <- getFluxes(eq)
   print(fluxes)

  ## Subsetting of equation list
  subeq1 <- subset(eq, "pB" %in% Product)
   print(subeq1)
  subeq2 <- subset(eq, grepl("not_available", Description))
```

```
 print(subeq2)

## Time derivatives of observables
observables <- eqnvec(pA_obs = "s1*pA", tA_obs = "s2*(A + pA)")
dobs <- dot(observables, eq)

## Combined equation vector for ODE and observables
f <- c(as.eqnvec(eq), dobs)
 print(f)
```

---

eqnlist *Generate eqnlist object*

Description

The eqnlist object stores an ODE as a list of stoichiometric matrix, rate expressions, state names and compartment volumes.

Translates a reaction network, e.g. defined by a data.frame, into an equation list object.

Usage

```
eqnlist(
  smatrix = NULL,
  states = colnames(smatrix),
  rates = NULL,
  volumes = NULL,
  description = NULL
)

as.eqnlist(data, volumes)

## S3 method for class 'data.frame'
as.eqnlist(data, volumes = NULL)

is.eqnlist(x)
```

Arguments

smatrix	Matrix of class numeric. The stoichiometric matrix, one row per reaction/process and one column per state.
states	Character vector. Names of the states.
rates	Character vector. The rate expressions.
volumes	Named character, volume parameters for states. Names must be a subset of the states. Values can be either characters, e.g. "V1", or numeric values for the volume. If volumes is not NULL, missing entries are treated as 1.

description	Character vector. Description of the single processes.
data	data.frame with columns Description, Rate, and one colum for each state reflecting the stoichiometric matrix
x	object of class eqnlist
...	additional arguments to be passed to or from methods.

Details

If data is a data.frame, it must contain columns "Description" (character), "Rate" (character), and one column per ODE state with the state names. The state columns correspond to the stoichiometric matrix.

Value

An object of class eqnlist, basically a list.

Object of class eqnlist

Examples

```
# Generate eqnlist from the constructor
S <- matrix(c(-1, 1, 1, -1),
            nrow = 2, ncol = 2,
            dimnames = list(NULL, c("A", "B")))

rates <- c("k1*A", "k2*B")
description <- c("forward", "backward")

f <- eqnlist(smatrix = S, rates = rates, description = description)
print(f)

# Convert to data.frame
fdata <- as.data.frame(f)
print(fdata)

# Generate eqnlist from data.frame and add volume parameter
f <- as.eqnlist(fdata, volumes = c(A = "Vcyt", B = "Vnuc"))
print(f)
print(as.eqnvec(f))
print(as.eqnvec(f, type = "amount"))
```

---

eqnvec *Generate equation vector object*

Description

The eqnvec object stores explicit algebraic equations, like the right-hand sides of an ODE, observation functions or parameter transformations as named character vectors.

Usage

```
eqnvec(...)

is.eqnvec(x)
```

Arguments

...	mathematical expressions as characters to be coerced, the right-hand sides of the equations
x	obect of any class

Value

object of class eqnvec, basically a named character.

See Also

eqnlist

Examples

```
v <- eqnvec(y = "2*x + offset")
print(v)
is.eqnvec(v)
```

---

eventlist *Eventlist*

Description

An eventlist is a data.frame with the necessary parameters to define an event as columns and specific events as rows. Event time and value can be passed as parameters, which can also be estimated.

Usage

```
eventlist(var = NULL, time = NULL, value = NULL, root = NULL, method = NULL)

addEvent(event, var, time = 0, value = 0, root = NA, method = "replace", ...)
```

Arguments

var	Character, the state to which the event is applied
time	Character or Numeric, the time at which the event happens
value	Character or Numeric, the value of the event
root	Character or NA, condition to trigger the event instead of time

method	Character, options are "replace", "add" or "multiply"
event	object of class eventlist
...	not used

Details

The function addEvent is pipe-friendly

Value

data.frame with class eventlist

Examples

```
eventlist(var = "A", time = "5", value = 1, method = "add")

events <- addEvent(NULL, var = "A", time = "5", value = 1, method = "add")
events <- addEvent(events, var = "A", time = "10", value = 1, method = "add")

# With symbols
events <- eventlist()
# Set "A" to "value_switch" at time "time_Switch"
events <- addEvent(events, var = "A", time = "time_switch", value = "value_switch", method = "replace")
# Set "B" to 2 when "A" reaches "A_target". The time-parameter for internal use will be "time_root".
events <- addEvent(events, var = "B", time = "time_root", value = 2, root = "A - A_target", method = "replace")
```

---

exmpextr *Extract an example from a unit test file.*

Description

Extract an example from a unit test file.

Usage

```
exmpextr(test, testPath = NULL, examplePath = NULL)
```

Arguments

test	File name of the test file, source.
testPath	Path to test files. Defaults to "inst/tests".
examplePath	Path under which the example is stored. Defaults to "inst/examples".

Details

From 'test', an example is extracted and written to a file of the same name, but with removed "test-". The file is saved under 'examplePath'. The special tag #-! can be used to hide code in the test which is enabled again in the example.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

---

expand.grid.alt *Alternative version of expand.grid*

Description

Alternative version of expand.grid

Usage

```
expand.grid.alt(seq1, seq2)
```

Arguments

seq1	Vector, numeric or character
seq2	Vector, numeric or character

Value

Matrix ob combinations of elemens of seq1 and seq2

---

extractExamples *Extract example from unit tests.*

Description

Extract example from unit tests.

Usage

```
extractExamples(testPath = NULL, examplePath = NULL)
```

Arguments

testPath	character, path to test folder
examplePath	character path to example folder

Details

Calls exmpextr for all unit test files.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

`fitErrorModel` *Fit an error model*

Description

Fit an error model to reduced replicate data, see `reduceReplicates`.

Usage

```
fitErrorModel(
  data,
  factors,
  errorModel = "exp(s0)+exp(srel)*x^2",
  par = c(s0 = 1, srel = 0.1),
  plotting = TRUE,
  blather = FALSE,
  ...
)
```

Arguments

`data`	Reduced replicate data, see `reduceReplicates`. Need columns "value", "sigma", "n".
`factors`	'data' is pooled with respect to the columns named here, see Details.
`errorModel`	Character vector defining the error model in terms of the variance. Use `x` to reference the independend variable, see Details.
`par`	Inital values for the parameters of the error model.
`plotting`	If TRUE, a plot of the pooled variance together with the fit of the error model is shown.
`blather`	If TRUE, additional information is returned, such as fit parameters and sigmaLS (original sigma given in input data).
`...`	Parameters handed to the optimizer `optim`.

Details

The variance estimator using $n-1$ data points is chi^2 distributed with $n-1$ degrees of freedom. Given replicates for consecutive time points, the sample variance can be assumed a function of the sample mean. By defining an error model which must hold for all time points, a maximum likelihood estimator for the parameters of the error model can be derived. The parameter '`errorModel`' takes the error model as a character vector, where the mean (independent variable) is refered to as `x`.

It is desireable to estimate the variance from many replicates. The parameter '`data`' must provide one or more columns which define the pooling of data. In case more than one column is announced by '`factors`', all combinations are constructed. If, e.g., '`factors = c("condition", "name")`' is

used, where "condition" is "a", "b", "c" and repeating and "name" is "d", "e" and repeating, the effective conditions used for pooling are "a d", "b e", "c d", "a e", "b d", and "c e".

By default, a plot of the pooled data, sigma and its confidence bound at 68% and 95% is shown.

Value

Returned by default is a data frame with columns as in 'data', but with the sigma values replaced by the derived values, obtained by evaluating the error model with the fit parameters.

If the blather = TRUE option is chosen, fit values of the parameters of the error model are appended, with the column names equal to the parameter names. The error model is appended as the attribute "errorModel". Confidence bounds for sigma at confidence level 68% and 95% are calculated, their values come next in the returned data frame. Finally, the effective conditions are appended to easily check how the pooling was done.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

`forcingsSymb` *Return some useful forcing functions as strings*

Description

Return some useful forcing functions as strings

Usage

```
forcingsSymb(
  type = c("Gauss", "Fermi", "1-Fermi", "MM", "Signal", "Dose"),
  parameters = NULL
)
```

Arguments

`type`	Which function to be returned
`parameters`	Named vector, character or numeric. Replace parameters by the corresponding valus in `parameters`.

Value

String with the function

format.eqnvec *Encode equation vector in format with sufficient spaces*

Description

Encode equation vector in format with sufficient spaces

Usage

```
## S3 method for class 'eqnvec'
format(x, ...)
```

Arguments

x	object of class eqnvec. Alternatively, a named parsable character vector.
...	additional arguments

Value

named character

funC0 *Evaluation of algebraic expressions defined by characters*

Description

Evaluation of algebraic expressions defined by characters

Usage

```
funC0(
  x,
  variables = getSymbols(x, exclude = parameters),
  parameters = NULL,
  compile = FALSE,
  modelname = NULL,
  verbose = FALSE,
  convenient = TRUE,
  warnings = TRUE
)
```

Arguments

x	Object of class eqnvec or a named character vector with the algebraic expressions
variables	character vector, the symbols that should be treated as variables
parameters	character vector, the symbols that should be treated as parameters
compile	Logical. Directly compile the file. If FALSE and modelname is available, the C file is written but not compiled. In this case, compile has to be called separately to compile one or more .c-files into one .so-file. If modelname is not available, an R function is generated and returned.
modelname	file name of the generated C file. See description of parameter compile.
verbose	Print compiler output to R command line.
convenient	logical, if TRUE return a function with argument ... to pass all variables/parameters as named arguments
warnings	logical. Suppress warnings about missing variables/parameters that are automatically replaced by zero values.

Value

Either a prediction function `f(..., attach.input = FALSE)` where the variables/parameters are passed as named arguments or a prediction function `f(M, p, attach.input = FALSE)` where `M` is the matrix of variable values (colums with colnames correspond to different variables) and `p` is the vector of parameter values. The argument `attach.input` determines whether `M` is attached to the output. The function `f` returns a matrix.

Examples

```
library(ggplot2)
myfun <- funC0(c(y = "a*x^4 + b*x^2 + c"))
out <- myfun(a = -1, b = 2, c = 3, x = seq(-2, 2, .1), attach.input = TRUE)
qplot(x = x, y = y, data = as.data.frame(out), geom = "line")
```

---

getCoefficients *Get coefficients from a character*

Description

Get coefficients from a character

Usage

```
getCoefficients(char, symbol)
```

Arguments

char	character, e.g. "2*x + y"
symbol	single character, e.g. "x" or "y"

Value

numeric vector with the coefficients

`getConditions` *Extract the conditions of an object*

Description

Extract the conditions of an object

Usage

```
getConditions(x, ...)

## S3 method for class 'list'
getConditions(x, ...)

## S3 method for class 'fn'
getConditions(x, ...)

## S3 method for class 'tbl_df'
getConditions(x, hypothesis = 1, ...)
```

Arguments

`x`	object from which the conditions should be extracted
`...`	additional arguments (not used right now)
`hypothesis`	The hypothesis in the dMod.frame

Value

The conditions in a format that depends on the class of `x`.

`getDerivs` *Extract the derivatives of an object*

Description

Extract the derivatives of an object

Usage

```
getDerivs(x, ...)

## S3 method for class 'parvec'
getDerivs(x, ...)

## S3 method for class 'prdframe'
getDerivs(x, ...)

## S3 method for class 'prdlist'
getDerivs(x, ...)

## S3 method for class 'list'
getDerivs(x, ...)

## S3 method for class 'objlist'
getDerivs(x, ...)
```

Arguments

x	object from which the derivatives should be extracted
...	additional arguments (not used right now)

Value

The derivatives in a format that depends on the class of `x`. This is `parvec` -> `matrix`, `prdframe` -> `prdframe`, `prdlist` -> `prdlist`, `objlist` -> named `numeric`.

---

getElements *Get Symbols and Numeric constants from a character*

Description

Get Symbols and Numeric constants from a character

Usage

```
getElements(char, exclude = NULL)
```

Arguments

char	Character vector (e.g. equation)
exclude	Character vector, the symbols to be excluded from the return value

Examples

```
getElements(c("A*AB+B^2"))
```

getEquations *Extract the equations of an object*

Description

Extract the equations of an object

Usage

```
getEquations(x, conditions = NULL)

## S3 method for class 'odemodel'
getEquations(x, conditions = NULL)

## S3 method for class 'prdfn'
getEquations(x, conditions = NULL)

## S3 method for class 'fn'
getEquations(x, conditions = NULL)
```

Arguments

x	object from which the equations should be extracted
conditions	character or numeric vector specifying the conditions to which getEquations is restricted. If conditions has length one, the result is not returned as a list.

Value

The equations as list of eqnvec objects.

getEstGridParameterMapping *Get Parameter mappings outer.estgrid - inner.estgrid*

Description

Get Parameter mappings outer.estgrid - inner.estgrid

Usage

```
getEstGridParameterMapping(x)
```

Arguments

x	est.grid

Value

named character

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

Examples

```
est.grid <- data.frame(ID = 1:2,
                       condition = c("A", "B"),
                       k1 = c("k1_A", "k1_B"),
                       k2 = c("k2_A", "k2_B"),
                       k3 = c("k3", NA),
                       k4 = c("k4", "k4"),
                       stringsAsFactors = FALSE)
getEstGridParameterMapping(est.grid)
```

---

getFluxes *Generate list of fluxes from equation list*

Description

Generate list of fluxes from equation list

Usage

```
getFluxes(eqnlist, type = c("conc", "amount"))
```

Arguments

eqnlist	object of class eqnlist.
type	"conc." or "amount" for fluxes in units of concentrations or number of molecules.

Value

list of named characters, the in- and out-fluxes for each state.

Examples

```
# Write your example here. You can also add more Start..End blocks if needed.
# Please mask all output such as print() with the special tag
#
# such that the test is not littered. Statements guarded by  are enabled
# in the example file which is extracted from this test file. To extract the
# example run
#    extractExamples()
# on the R command line.
```

```
## Generate another equation list
eq <- eqnlist()
eq <- addReaction(eq, "A", "pA", "act_A * A * stimulus", "Phosphorylation of A")
eq <- addReaction(eq, "pA", "A", "deact_A * pA", "Deposphorylation of pA")
eq <- addReaction(eq, "2*pA", "pA_pA", "form_complex_pA * pA^2", "Complex formation of pA")
eq <- addReaction(eq, "B", "pB", "act_B * B * pA_pA", "Phosphorylation of B")
eq <- addReaction(eq, "pB", "B", "deact_B * pB", "Deposphorylation of pB")

## Extract data.frame of reactions
reactions <- getReactions(eq)
 print(reactions)

## Get conserved quantities
cq <- conservedQuantities(eq$smatrix)
 print(cq)

## Get fluxes
fluxes <- getFluxes(eq)
 print(fluxes)

## Subsetting of equation list
subeq1 <- subset(eq, "pB" %in% Product)
 print(subeq1)
subeq2 <- subset(eq, grepl("not_available", Description))
 print(subeq2)

## Time derivatives of observables
observables <- eqnvec(pA_obs = "s1*pA", tA_obs = "s2*(A + pA)")
dobs <- dot(observables, eq)

## Combined equation vector for ODE and observables
f <- c(as.eqnvec(eq), dobs)
 print(f)
```

`getLocalDLLs` *Determine loaded DLLs available in working directory*

Description

Determine loaded DLLs available in working directory

Usage

```
getLocalDLLs()
```

Value

Character vector with the names of the loaded DLLs available in the working directory

getObservables *Extract the observables of an object*

Description

Extract the observables of an object

Usage

```
getObservables(x, ...)

## S3 method for class 'tbl_df'
getObservables(x, hypothesis = 1, ...)
```

Arguments

x	object from which the equations should be extracted
...	not used
hypothesis	The hypothesis in the dMod.frame

Value

The equations as a character.

getParameters *Extract the parameters of an object*

Description

Extract the parameters of an object

Usage

```
getParameters(..., conditions = NULL)

## S3 method for class 'odemodel'
getParameters(x, conditions = NULL)

## S3 method for class 'fn'
getParameters(x, conditions = NULL)

## S3 method for class 'parvec'
getParameters(x, conditions = NULL)

## S3 method for class 'prdframe'
```

```
getParameters(x, conditions = NULL)

## S3 method for class 'prdlist'
getParameters(x, conditions = NULL)

## S3 method for class 'eqnlist'
getParameters(x)

## S3 method for class 'eventlist'
getParameters(x)
```

Arguments

...	objects from which the parameters should be extracted
conditions	character vector specifying the conditions to which getParameters is restricted
x	object from which the parameters are extracted

Value

The parameters in a format that depends on the class of x.

`getParameters.data.table` *Title*

Description

Title

Usage

```
## S3 method for class 'data.table'
getParameters(x, ...)
```

Arguments

x

...

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

getParametersToEstimate *Determine which parameters need sensitivities*

Description

Determine which parameters need sensitivities

Usage

```
getParametersToEstimate(est.grid, trafo, reactions)
```

Arguments

est.grid	est.grid = data.table(condition, par1,...,parN)
trafo	symbolic base trafo
reactions	Object of class eqnlist()

Value

character, to go into the estimate-argument of odemodel

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

getParGrids *Get est.grid and fixed.grid*

Description

Get est.grid and fixed.grid

Usage

```
getParGrids(mytrafo, mytrafoL, mycondition.grid, SS_pars = NULL)
```

Arguments

mytrafo	base trafo
mytrafoL	condition specific branched trafo list
mycondition.grid	condition.grid with condition names as rownames e.g. as output from attr(datalist, "condition.grid")
SS_pars	parameters determined by the steady state

Author(s)

Svenja Kemmer

`getReactions` *Generate a table of reactions (data.frame) from an equation list*

Description

Generate a table of reactions (data.frame) from an equation list

Usage

```
getReactions(eqnlist)
```

Arguments

`eqnlist`	object of class eqnlist

Value

`data.frame` with educts, products, rate and description. The first column is a check if the reactions comply with reaction kinetics.

Examples

```
# Write your example here. You can also add more Start..End blocks if needed.
# Please mask all output such as print() with the special tag
#
# such that the test is not littered. Statements guarded by  are enabled
# in the example file which is extracted from this test file. To extract the
# example run
#    extractExamples()
# on the R command line.

  ## Generate another equation list
  eq <- eqnlist()
  eq <- addReaction(eq, "A", "pA", "act_A * A * stimulus", "Phosphorylation of A")
  eq <- addReaction(eq, "pA", "A", "deact_A * pA", "Deposphorylation of pA")
 eq <- addReaction(eq, "2*pA", "pA_pA", "form_complex_pA * pA^2", "Complex formation of pA")
  eq <- addReaction(eq, "B", "pB", "act_B * B * pA_pA", "Phosphorylation of B")
  eq <- addReaction(eq, "pB", "B", "deact_B * pB", "Deposphorylation of pB")

  ## Extract data.frame of reactions
  reactions <- getReactions(eq)
   print(reactions)

  ## Get conserved quantities
  cq <- conservedQuantities(eq$smatrix)
   print(cq)
```

```
## Get fluxes
fluxes <- getFluxes(eq)
 print(fluxes)

## Subsetting of equation list
subeq1 <- subset(eq, "pB" %in% Product)
 print(subeq1)
subeq2 <- subset(eq, grepl("not_available", Description))
 print(subeq2)

## Time derivatives of observables
observables <- eqnvec(pA_obs = "s1*pA", tA_obs = "s2*(A + pA)")
dobs <- dot(observables, eq)

## Combined equation vector for ODE and observables
f <- c(as.eqnvec(eq), dobs)
 print(f)
```

`getStepIndices` *Get the indices of the n largest (not necessarily best) steps of a parframe*

Description

Get the indices of the n largest (not necessarily best) steps of a parframe

Usage

```
getStepIndices(myparframe, nsteps = 5, tol = 1)
```

Arguments

`myparframe`	parframe, result from mstrust
`nsteps`	number of steps
`tol`	tolerance for stepdetection

Value

indices of the largest steps

See Also

parframe

Examples

```
## Generate a prediction function
regfn <- c(y = "sin(a*time)")

g <- Y(regfn, parameters = "a")
x <- Xt(condition = "C1")

## Generate data
data <- datalist(
  C1 = data.frame(
    name = "y",
    time = 1:5,
    value = sin(1:5) + rnorm(5, 0, .1),
    sigma = .1
  )
)

## Initialize parameters and time
pars <- c(a = 1)
times <- seq(0, 5, .1)

## Do many fits from random positions and store them into parlist
out <- as.parlist(lapply(1:50, function(i) {
  trust(normL2(data, g*x), pars + rnorm(length(pars), 0, 1), rinit = 1, rmax = 10)
}))

## Reduce parlist to parframe
parframe <- as.parframe(out)
plotValues(parframe)

## Get steps
getStepIndices(parframe, nsteps = 2, tol = 1)

getSteps(parframe, nsteps = 2)

plotValues(getSteps(parframe, nsteps = 2))
```

---

getSteps *Get the rows of the n largest steps of a parframe*

Description

Get the rows of the n largest steps of a parframe

Usage

```
getSteps(myparframe, nsteps = 5, tol = 1)
```

Arguments

myparframe	parframe, result from mstrust
nsteps	number of steps
tol	tolerance for stepdetection

Value

parframe subsetted to the n largest steps

See Also

parframe

Examples

```
## Generate a prediction function
regfn <- c(y = "sin(a*time)")

g <- Y(regfn, parameters = "a")
x <- Xt(condition = "C1")

## Generate data
data <- datalist(
  C1 = data.frame(
    name = "y",
    time = 1:5,
    value = sin(1:5) + rnorm(5, 0, .1),
    sigma = .1
  )
)

## Initialize parameters and time
pars <- c(a = 1)
times <- seq(0, 5, .1)

## Do many fits from random positions and store them into parlist
out <- as.parlist(lapply(1:50, function(i) {
  trust(normL2(data, g*x), pars + rnorm(length(pars), 0, 1), rinit = 1, rmax = 10)
}))

## Reduce parlist to parframe
parframe <- as.parframe(out)
plotValues(parframe)

## Get steps
getStepIndices(parframe, nsteps = 2, tol = 1)

getSteps(parframe, nsteps = 2)

plotValues(getSteps(parframe, nsteps = 2))
```

ggopen *Open last plot in external pdf viewer*

Description

Convenience function to show last plot in an external viewer.

Usage

```
ggopen(plot = last_plot(), command = "xdg-open", ...)
```

Arguments

plot	ggplot2 plot object.
command	character, indicatig which pdf viewer is started.
...	arguments going to ggsave.

gridlist *Collect grids in list*

Description

Ensure all tables are data.tables

Usage

```
gridlist(est.grid, fix.grid)
```

Arguments

est.grid, fix.grid	data.tables, will be coerced to one

Value

list of the grid

Id — *An identity function which vanishes upon concatenation of fns*

Description

An identity function which vanishes upon concatenation of fns

Usage

```
Id()
```

Value

fn of class idfn

Examples

```
x <- Xt()
id <- Id()

(id*x)(1:10, pars = c(a = 1))
(x*id)(1:10, pars = c(a = 1))
str(id*x)
str(x*id)
```

import_sbml — *Import an SMBL model*

Description

Requires AMICI https://github.com/ICB-DCM/AMICI/ and dependencies to be installed on your system Big thanks go to Daniel Weindl!

Usage

```
import_sbml(modelpath, amicipath = NULL)
```

Arguments

modelpath	Path to the sbml file
amicipath	Path to your amici-python-installation, e.g.: AMICIPATH/python

Value

list of eqnlist, parameters and inits

indiv_addGlobalParsToGridlist *Add single-valued parameters to the pargrids*

Description

Add single-valued parameters to the pargrids

Usage

```
indiv_addGlobalParsToGridlist(pars, gridlist, FLAGoverwrite = FALSE)
```

Arguments

pars	character or numeric
gridlist	list(fix.grid, est.grid)
FLAGoverwrite	Overwrite existing parameters in fix.grid or est.grid?

Value

gridlist

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

Examples

```
pars <- c(NewParSymbolic = "NewParSymbolic", NewParFixed = 1)
est.grid <- data.table(ID = 1:2,
                       condition = c("A", "B"),
                       k1 = c("k1_A", "k1_B"),
                       k2 = c("k2_A", "k2_B"),
                       k3 = c("k3", NA),
                       k4 = c("k4", "k4"),
                       stringsAsFactors = FALSE)
fix.grid <- data.table(ID = 1:2,
                       condition = c("A", "B"),
                       k3 = c(NA, 3.5),
                       k5 = c(5.1,5.2),
                       k6 = c(6,6),
                       stringsAsFactors = FALSE)
indiv_addGlobalParsToGridlist(c(NewParSymbolic = "NewParSymbolic", NewParFixed = 1), gridlist(est.grid = est.grid
indiv_addGlobalParsToGridlist(c(k1 = 1), gridlist(est.grid = est.grid, fix.grid = fix.grid), FLAGoverwrite = FALSE
indiv_addGlobalParsToGridlist(c(k1 = 1), gridlist(est.grid = est.grid, fix.grid = fix.grid), FLAGoverwrite = TRUE)
```

`indiv_addLocalParsToGridList` *Add individualized parameters to grids*

Description

Can only add parameters, cannot update existing parameters

Usage

```
indiv_addLocalParsToGridList(pars, gridlist, FLAGoverwrite = FALSE)
```

Arguments

`pars`	data.table(ID, condition, pars...) Only one of ID, condition must be present
`gridlist`	`gridlist()`

Value

modified gridlist

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

Examples

```
pars <- data.table(ID = 1:2, newFix = c(1,2), newEst = c("a", "b"), newMix = c("a", 1))
est.grid <- data.table(ID = 1:2,
                       condition = c("A", "B"),
                       k1 = c("k1_A", "k1_B"),
                       k2 = c("k2_A", "k2_B"),
                       k3 = c("k3", NA),
                       k4 = c("k4", "k4"),
                       stringsAsFactors = FALSE)
fix.grid <- data.table(ID = 1:2,
                       condition = c("A", "B"),
                       k3 = c(NA, 3.5),
                       k5 = c(5.1,5.2),
                       k6 = c(6,6),
                       stringsAsFactors = FALSE)
gl <- gridlist(est.grid = est.grid, fix.grid = fix.grid)
indiv_addLocalParsToGridList(pars, gl)
```

init_empty_objlist *Create an objlist with zeros as entries*

Description

Create an objlist with zeros as entries

Usage

```
init_empty_objlist(pars, deriv = TRUE, FLAGchisquare = FALSE)
```

Arguments

pars	named vector. Only names and length are used
deriv	TRUE or FALSE

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

Examples

```
init_empty_objlist(setNames(rnorm(5), letters[1:5]))
init_empty_objlist(setNames(rnorm(5), letters[1:5]), FLAGchisquare = TRUE)
```

installJuliaForSteadyStates *Install the julia setup*

Description

Installs Julia and the necessary julia packages

Usage

```
installJuliaForSteadyStates(installJulia = FALSE, installJuliaPackages = TRUE)
```

Arguments

installJulia	boolean, default false. If set to true, juliaup and via this then Julia is installed.
installJuliaPackages	boolean, default true. If set to true, the necessary packages are installed.

Value

nothing

jakstat *Time-course data for the JAK-STAT cell signaling pathway*

Description

Phosphorylated Epo receptor (pEpoR), phosphorylated STAT in the cytoplasm (tpSTAT) and total STAT (tSTAT) in the cytoplasmhave been measured at times 0, ..., 60.

lbind *Bind named list of data.frames into one data.frame*

Description

Bind named list of data.frames into one data.frame

Usage

```
lbind(mylist)
```

Arguments

mylist	A named list of data.frame. The data.frames are expected to have the same structure.

Details

Each data.frame ist augented by a "condition" column containing the name attributed of the list entry. Subsequently, the augmented data.frames are bound together by `rbind`.

Value

data.frame with the originial columns augmented by a "condition" column.

load.parlist *Construct fitlist from temporary files.*

Description

An aborted mstrust leaves behind results of already completed fits. This command loads these fits into a fitlist.

Usage

```
load.parlist(folder)
```

Arguments

folder	Path to the folder where the fit has left its results.

Details

The command mstrust saves each completed fit along the multi-start sequence such that the results can be resurected on abortion. This command loads a fitlist from these intermediate results.

Value

An object of class parlist.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

See Also

mstrust

loadDLL *Load shared object for a dMod object*

Description

Usually when restarting the R session, although all objects are saved in the workspace, the dynamic libraries are not linked any more. loadDLL is a wrapper for dyn.load that uses the "modelname" attribute of dMod objects like prediction functions, observation functions, etc. to load the corresponding shared object.

Usage

```
loadDLL(...)
```

Arguments

...	objects of class prdfn, obsfn, parfn, objfn, ...

---

long2wide *Translate long to wide format (inverse of wide2long.matrix)*

Description

Translate long to wide format (inverse of wide2long.matrix)

Usage

```
long2wide(out)
```

Arguments

out	data.frame in long format

Value

data.frame in wide format

---

lsdMod *Print list of dMod objects in .GlobalEnv*

Description

Lists the objects for a set of classes.

Usage

```
lsdMod(
  classlist = c("odemodel", "parfn", "prdfn", "obsfn", "objfn", "datalist"),
  envir = .GlobalEnv
)
```

Arguments

classlist	List of object classes to print.
envir	Alternative environment to search for objects.

Examples

```
## Not run:
lsdMod()
lsdMod(classlist = "prdfn", envir = environment(obj))

## End(Not run)
```

make_pars *Extract pars, fixed and parnames from grids for a given condition*

Description

Extract pars, fixed and parnames from grids for a given condition

Usage

```
make_pars(pars, fixed = NULL, est.grid, fix.grid, ID)
```

Arguments

est.grid	needs condition and ID
fix.grid	
ID	
est.vec	

Examples

```
pars <- c("k1_A" = 1, "k1_B" = 1, "k2_A" = 2, "k3" = 3, "k4" = 4)
fixed <- c("k2_B" = 2.5)

est.grid <- data.frame(ID = 1:2,
                       condition = c("A", "B"),
                       k1 = c("k1_A", "k1_B"),
                       k2 = c("k2_A", "k2_B"),
                       k3 = c("k3", NA),
                       k4 = c("k4", "k4"),
                       stringsAsFactors = FALSE)
fix.grid <- data.frame(ID = 1:2,
                       condition = c("A", "B"),
                       k3 = c(NA, 3.5),
                       k5 = c(5.1,5.2),
                       k6 = c(6,6),
                       stringsAsFactors = FALSE)
make_pars(pars, fixed, est.grid, fix.grid, 1)
make_pars(pars, fixed, est.grid, fix.grid, 2)
```

match.fnargs *dMod match function arguments*

Description

The function is exported for dependency reasons

Usage

```
match.fnargs(arglist, choices)
```

Arguments

arglist	list
choices	character

mname *Get modelname from single object (used internally)*

Description

Get modelname from single object (used internally)

Usage

```
mname(x, conditions = NULL)

## S3 method for class '`NULL`'
mname(x, conditions = NULL)

## S3 method for class 'character'
mname(x, conditions = NULL)

## S3 method for class 'objfn'
mname(x, conditions = NULL)

## S3 method for class 'fn'
mname(x, conditions = NULL)
```

Arguments

x	dMod object
conditions	character vector of conditions

modelname *Get and set modelname*

Description

The modelname attribute refers to the name of a C file associated with a dMod function object like prediction-, parameter transformation- or objective functions.

Usage

```
modelname(..., conditions = NULL)

modelname(x, ...) <- value

## S3 replacement method for class 'fn'
modelname(x, conditions = NULL, ...) <- value

## S3 replacement method for class 'objfn'
modelname(x, conditions = NULL, ...) <- value
```

Arguments

`...`	objects of type `prdfn`, `parfn`, `objfn`
`conditions`	character vector of conditions
`x`	dMod object for which the model name should be set
`value`	character, the new modelname (does not change the C file)

Value

character vector of model names, corresponding to C files in the local directory.

msParframe *Reproducibly construct "random" parframes*

Description

The output of this function can be used for the `center` - argument of `mstrust`

Usage

```
msParframe(
  pars,
  n = 20,
  seed = 12345,
  samplefun = stats::rnorm,
  keepfirst = TRUE,
  ...
)
```

Arguments

pars	Named vector. If samplefun has a "mean"-argument, values of pars will used as mean
n	Integer how many lines should the parframe have
seed	Seed for the random number generator
samplefun	random number generator: rnorm, runif, etc...
keepfirst	boolean, if set to TRUE the first row of the parframe will be the pars
...	arguments going to samplefun

Value

parframe (without metanames)

See Also

mstrust and parframe

Examples

```
msParframe(c(a = 0, b = 100000), 5)

# Parameter specific sigma
msParframe(c(a = 0, b = 100000), 5, samplefun = rnorm, sd = c(100, 0.5))
```

---

mstrust *Non-Linear Optimization, multi start*

Description

Wrapper around trust allowing for multiple fits from randomly chosen initial values.

Usage

```
mstrust(
  objfun,
  center,
  studyname,
  rinit = 0.1,
  rmax = 10,
  fits = 20,
  cores = 1,
  optmethod = "trust",
  samplefun = "rnorm",
  resultPath = ".",
  stats = FALSE,
  output = FALSE,
  cautiousMode = FALSE,
  start1stfromCenter = FALSE,
  ...
)
```

Arguments

objfun	Objective function, see trust.
center	Parameter values around which the initial values for each fit are randomly sampled. The initial values handed to trust are the sum of center and the output of 'samplefun', center + 'samplefun'. See trust, parinit. center Can also be a parframe, then the parameter values are taken from the parframe. In this case, the fits argument is overwritten. To use a reproducible set of initial guesses, generate center with msParframe
studyname	The names of the study or fit. This name is used to determine filenames for interim and final results. See Details.
rinit	Starting trust region radius, see trust.
rmax	Maximum allowed trust region radius, see trust.
fits	Number of fits (jobs).
cores	Number of cores for job parallelization.
samplefun	Function to sample random initial values. It is assumed, that 'samplefun' has a named parameter "n" which defines how many random numbers are to be returned, such as for rnorm or runif. By default rnorm is used. Parameteres for samplefun are simply appended as named parameters to the mstrust call and automatically handed to samplefun by matching parameter names.
resultPath	character indicating the folder where the results should be stored. Defaults to ".".
stats	If true, the same summary statistic as written to the logfile is printed to command line on mstrust completion.
output	logical. If true, writes output to the disc.

cautiousMode	Logical, write every fit to disk in deparsed form (avoids the RDA incompatibility trap) and don't delete intermediate results
...	Additional parameters handed to trust(), samplefun(), or the objective function by matching parameter names. All unmatched parameters are handed to the objective function objfun(). The log file starts with a table telling which parameter was assigend to which function.

Details

By running multiple fits starting at randomly chosen inital parameters, the chisquare landscape can be explored using a deterministic optimizer. Here, trust is used for optimization. The standard procedure to obtain random initial values is to sample random variables from a uniform distribution (rnorm) and adding these to 'center'. It is, however, possible, to employ any other sampling strategy by handing the respective function to mstrust(), 'samplefun'.

In case a special sampling is required, a customized sampling function can be used. If, e.g., inital values leading to a non-physical systems are to be discarded upfront, the objective function can be addapted accordingly.

All started fits either lead to an error or complete converged or unconverged. A statistics about the return status of fits can be shown by setting 'stats' to TRUE.

Fit final and intermediat results are stored under 'studyname'. For each run of mstrust for the same study name, a folder under 'studyname' of the form "trial-x-date" is created. "x" is the number of the trial, date is the current time stamp. In this folder, the intermediate results are stored. These intermediate results can be loaded by load.parlist. These are removed on successfull completion of mstrust. In this case, the final list of fit parameters (parameterList.Rda) and the fit log (mstrust.log) are found instead.

Value

A parlist holding errored and converged fits.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

See Also

1. trust, for the used optimizer, 2. rnorm, runif for two common sampling functions, 3. msParframe for passing a reproducible set of random initial guesses to mstrust, 4. as.parframe for formatting the output to a handy table

mutatedMod.frame *A version of dplyr::mutate*

Description

This is basically dplyr::mutate with two differences 1. It groups the tibble rowwise before mutating 2. It allows to store the calls. This is intended for use when your objects are not standard transformations, such as `prd = g*x*p` but more complicated and you want to keep a record of what you did.

If the result of your ... expressions is not atomic, make sure to wrap your expression in `list()`

Usage

```
mutatedMod.frame(dMod.frame, ..., keepCalls = F)
```

Arguments

`dMod.frame`	A dMod.frame
`...`	Expressions going to mutate()
`keepCalls`	Should the dots ... be recorded?

Examples

```
## Not run:
mytbl <- tibble(a = 1:2, b = letters[1:2]) %>%
mutatedMod.frame(e = paste0(a,b), keepCalls = T) %>%
mutatedMod.frame(f = paste0(e, "=", a, "*", b), keepCalls = T) %>%
mutatedMod.frame(e = paste0(a,"*", b), keepCalls = T)

mytbl$.calls

## End(Not run)
```

nll *Compute the negative log-likelihood*

Description

Gaussian Log-likelihood. Supports NONMEM-like BLOQ handling methods M1, M3 and M4 and estimation of error models

Usage

```
nll(
  nout,
  pars,
  deriv,
  opt.BLOQ = "M3",
 opt.hessian = c(ALOQ_part1 = TRUE, ALOQ_part2 = TRUE, ALOQ_part3 = TRUE, BLOQ_part1 =
    TRUE, BLOQ_part2 = TRUE, BLOQ_part3 = TRUE, PD = TRUE)
)
```

Arguments

nout	data.frame (result of res) or object of class res.
pars	Example named vector of outer parameters to construct the objlist
deriv	TRUE or FALSE
opt.BLOQ	see normIndiv
opt.hessian	see normIndiv

Value

list with entries value (numeric, the weighted residual sum of squares), gradient (numeric, gradient) and hessian (matrix of type numeric).

---

nll_ALOQ *Non-linear log likelihood for the ALOQ part of the data*

Description

Non-linear log likelihood for the ALOQ part of the data

Usage

```
nll_ALOQ(
  nout,
  derivs,
  derivs.err,
  opt.BLOQ = c("M3", "M4NM", "M4BEAL", "M1"),
  opt.hessian = c(ALOQ_part1 = TRUE, ALOQ_part2 = TRUE, ALOQ_part3 = TRUE)
)
```

Arguments

nout	output of res()
derivs, derivs.err	attributes of output of res()
opt.BLOQ	Character denoting the method to deal with BLOQ data
opt.hessian	Named logical vector to include or exclude various non-convex summands of the hessian matrix

---

nll_BLOQ *Non-linear log likelihood for the BLOQ part of the data*

Description

Non-linear log likelihood for the BLOQ part of the data

Usage

```
nll_BLOQ(
  nout.bloq,
  derivs.bloq,
  derivs.err.bloq,
  opt.BLOQ = c("M3", "M4NM", "M4BEAL", "M1"),
  opt.hessian = c(BLOQ_part1 = TRUE, BLOQ_part2 = TRUE, BLOQ_part3 = TRUE)
)
```

Arguments

nout.bloq	The bloq output of res()
derivs.bloq, derivs.err.bloq	attributes of output of res()
opt.BLOQ	Character denoting the method to deal with BLOQ data
opt.hessian	Named logical vector to include or exclude various summands of the hessian matrix

normL2 *L2 norm between data and model prediction*

Description

For parameter estimation and optimization, an objective function is needed. normL2 returns an objective function for the L2 norm of data and model prediction. The resulting objective function can be used for optimization with the trust optimizer, see mstrust.

Usage

```
normL2(data, x, errmodel = NULL, times = NULL, attr.name = "data")
```

Arguments

data	object of class datalist
x	object of class prdfn
errmodel	object of class obsfn. errmodel does not need to be defined for all conditions.
times	numeric vector, additional time points where the prediction function is evaluated. If NULL, time points are extacted from the datalist solely. If the prediction function makes use of events, hand over event times here.
attr.name	character. The constraint value is additionally returned in an attributed with this name
forcings	TO BE FILLED BY DANIEL K
iiv	Example: c("ka", "ETA_EMAX"). Vector with names which are individualized per condition
conditional	Example: data.frame(parname = "GR", covname = "SEX", covvalue = "1", stringsAsFactors = FALSE). * covname can relate to any parameter in the condition.grid of the data. * covvalue is the value of this variable to use for individualization
fixed.grid	data.frame(parname, partask, ids...) Lookup table for fixed parameters
nauxtimes	additional simulation times
cores	to parallelize over conditions not over fits

Details

Objective functions can be combined by the "+" operator, see sumobjfn.

Value

Object of class obsfn, i.e. a function obj(..., fixed, deriv, conditions, env) that returns an objective list, objlist.

Examples

```
## Generate a prediction function

times <- 0:5
grid <- data.frame(name = "A", time = times, row.names = paste0("p", times))
x <- Xd(grid, condition = "C1")

pars <- structure(rep(0, nrow(grid)), names = row.names(grid))

## Simulate data
data.list <- lapply(1:3, function(i) {
  prediction <- x(times, pars + rnorm(length(pars), 0, 1))
  cbind(wide2long(prediction), sigma = 1)
})

data <- as.datalist(do.call(rbind, data.list))

## Generate objective function based on data and model
## Then fit the data and plot the result
obj <- normL2(data, x)
myfit <- trust(obj, pars, rinit = 1, rmax = 10)
plot(x(times, myfit$argument), data)
```

---

normL2_indiv *Fast normL2*

Description

Fast normL2

Usage

```
normL2_indiv(
  data,
  prd0,
  errmodel = NULL,
  est.grid,
  fix.grid,
  times = NULL,
  attr.name = "data"
)
```

Arguments

data

prd0

errmodel

est.grid

`fix.grid`

`times`

`attr.name`

Value

objective function

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

`nullZ` *Find integer-null space of matrix A*

Description

Find integer-null space of matrix A

Usage

```
nullZ(A, tol = sqrt(.Machine$double.eps))
```

Arguments

`A`	matrix for which the null space is searched
`tol`	tolerance to find pivots in rref-function below

Value

null space of A with only integers in it

Author(s)

Malenka Mader, <Malenka.Mader@fdm.uni-freiburg.de>

objframe *Objective frame*

Description

An objective frame is supposed to store the residuals of a model prediction with respect to a data frame.

Usage

```
objframe(mydata, deriv = NULL, deriv.err = NULL)
```

Arguments

mydata	data.frame as being generated by res.
deriv	matrix of the derivatives of the residuals with respect to parameters.
deriv.err	matrix of the derivatives of the error model.

Value

An object of class objframe, i.e. a data frame with attribute "deriv".

objlist *Generate objective list*

Description

An objective list contains an objective value, a gradient, and a Hessian matrix.

Objective lists can contain additional numeric attributes that are preserved or combined with the corresponding attributes of another objective list when both are added by the "+" operator, see sumobjlist.

Objective lists are returned by objective functions as being generated by normL2, constraintL2, priorL2 and datapointL2.

Usage

```
objlist(value, gradient, hessian)
```

Arguments

value	numeric of length 1
gradient	named numeric
hessian	matrix with rownames and colnames according to gradient names

Value

Object of class `objlist`

Examples

```
# objlist(1, c(a = 1, b = 2), matrix(2, nrow = 2, ncol = 2, dimnames = list(c("a", "b"),c("a", "b"))))
```

---

objtimes *Get vector of logarithmically spaced time points*

Description

to help the objective function integrator

Usage

```
objtimes(datatimes, eventtimes = NULL, Nobjtimes = 25)
```

Arguments

`datatimes`	times present in data
`eventtimes`	times present in events (not yet implemented)

Value

vector of times, including datatimes

Examples

```
objtimes(c(0,30,60,90,600))
objtimes(c(30,60,90,600))
objtimes(c(-30,60,90,600))
```

---

obsfn *Observation function*

Description

An observation function is a function is that is concatenated with a prediction function via prodfn to yield a new prediction function, see prdfn. Observation functions are generated by Y. Handling of the conditions is then organized by the `obsfn` object.

Usage

```
obsfn(X2Y, parameters = NULL, condition = NULL)
```

Arguments

X2Y	the low-level observation function generated e.g. by Y.
parameters	character vector with parameter names
condition	character, the condition name

Details

Observation functions can be "added" by the "+" operator, see sumfn. Thereby, observations for different conditions are merged or, overwritten. Observation functions can also be concatenated with other functions, e.g. observation functions (obsfn) or prediction functions (prdfn) by the "*" operator, see prodfn.

Value

Object of class obsfn, i.e. a function x(..., fixed, deriv, conditions, env) which returns a prdlist. The arguments out (prediction) and pars (parameter values) should be passed via the ... argument.

Examples

```
# Define a time grid on which to make a prediction by peace-wise linear function.
# Then define a (generic) prediction function based on thid grid.
times <- 0:5
grid <- data.frame(name = "A", time = times, row.names = paste0("p", times))
x <- Xd(grid)

# Define an observable and an observation function
observables <- eqnvec(Aobs = "s*A")
g <- Y(g = observables, f = NULL, states = "A", parameters = "s")

# Collect parameters and define an overarching parameter transformation
# for two "experimental condtions".
dynpars <- attr(x, "parameters")
obspars <- attr(g, "parameters")
innerpars <- c(dynpars, obspars)

trafo <- structure(innerpars, names = innerpars)
trafo_C1 <- replaceSymbols(innerpars, paste(innerpars, "C1", sep = "_"), trafo)
trafo_C2 <- replaceSymbols(innerpars, paste(innerpars, "C2", sep = "_"), trafo)

p <- NULL
p <- p + P(trafo = trafo_C1, condition = "C1")
p <- p + P(trafo = trafo_C2, condition = "C2")

# Collect outer (overarching) parameters and
# initialize with random values
outerpars <- attr(p, "parameters")
pars <- structure(runif(length(outerpars), 0, 1), names = outerpars)

# Predict internal/unobserved states
out1 <- (x*p)(times, pars)
```

```
plot(out1)

# Predict observed states in addition to unobserved
out2 <- (g*x*p)(times, pars)
plot(out2)
```

---

odemodel *Generate the model objects for use in Xs (models with sensitivities)*

Description

Generate the model objects for use in Xs (models with sensitivities)

Usage

```
odemodel(
  f,
  deriv = TRUE,
  forcings = NULL,
  events = NULL,
  outputs = NULL,
  fixed = NULL,
  estimate = NULL,
  modelname = "odemodel",
  solver = c("deSolve", "Sundials"),
  gridpoints = NULL,
  verbose = FALSE,
  ...
)
```

Arguments

f	Something that can be converted to eqnvec, e.g. a named character vector with the ODE
deriv	logical, generate sensitivities or not
forcings	Character vector with the names of the forcings
events	data.frame of events with columns "var" (character, the name of the state to be affected), "time" (character or numeric, time point), "value" (character or numeric, value), "method" (character, either "replace" or "add"). See events. Events need to be defined here if they contain parameters, like the event time or value. If both, time and value are purely numeric, they can be specified in Xs(), too.
outputs	Named character vector for additional output variables.
fixed	Character vector with the names of parameters (initial values and dynamic) for which no sensitivities are required (will speed up the integration).

estimate	Character vector specifying parameters (initial values and dynamic) for which sensitivities are returned. If estimate is specified, it overwrites 'fixed'.
modelname	Character, the name of the C file being generated.
solver	Solver for which the equations are prepared.
gridpoints	Integer, the minimum number of time points where the ODE is evaluated internally
verbose	Print compiler output to R command line.
...	Further arguments being passed to funC.

Value

list with func (ODE object) and extended (ODE+Sensitivities object)

Examples

```
## Not run:

## Generate a compiled ODE model from an equation vector
## The model will not return sensitivities for "switch"
## Files will be generated in your working directory!

f <- eqnvec(A = "-k*A + switch*F")
model <- odemodel(f, forcings = "F", fixed = "switch")
print(model)

## Generate the same model from an equation list
f <- addReaction(NULL, from = "", to = "A", rate = "switch*F", description = "production")
f <- addReaction(f   , from = "A", to = "", rate = "k*A", description = "degradation")
print(f)

model <- odemodel(f, forcings = "F", fixed = "switch")
print(model)


# create forcings
forc1 <- data.frame(name = "F", time = seq(0,5, 0.1), value = sin(seq(0,5,0.1)))
forc2 <- data.frame(name = "F", time = seq(0,5, 0.1), value = exp(-seq(0,5,0.1)))
forc3 <- data.frame(name = "F", time= 0,              value = 0.1)


x <- Xs(model, forc1, condition = "forc1") +
  Xs(model, forc2, condition = "forc2") +
  Xs(model, forc3, condition = "forc3")

g <- Y(c(out1 = "F * A", out2 = "F"), x)

times <-  seq(0,5, 0.001)
pars <- setNames(runif(length(getParameters(x))), getParameters(x))

pred <- (g*x)(times, pars)
```

```
plot(pred)


## End(Not run)
```

---

optionsLSODES *Get default arguments for integrators*

Description

Get default arguments for integrators

Usage

```
optionsLSODES(
  rtol = 1e-06,
  atol = 1e-06,
  jacvec = NULL,
  sparsetype = "sparseint",
  nnz = NULL,
  inz = NULL,
  rootfunc = NULL,
  verbose = FALSE,
  nroot = 0,
  tcrit = NULL,
  hmin = 0,
  hmax = NULL,
  hini = 0,
  ynames = TRUE,
  maxord = NULL,
  maxsteps = 5000,
  lrw = NULL,
  liw = NULL
)

optionsLSODE(
  rtol = 1e-06,
  atol = 1e-06,
  jacfunc = NULL,
  jactype = "fullint",
  mf = NULL,
  rootfunc = NULL,
  verbose = FALSE,
  nroot = 0,
  tcrit = NULL,
  hmin = 0,
```

```
  hmax = NULL,
  hini = 0,
  ynames = TRUE,
  maxord = NULL,
  bandup = NULL,
  banddown = NULL,
  maxsteps = 5000,
  rpar = NULL,
  ipar = NULL,
  nout = 0,
  outnames = NULL
)

optionsLSODA(
  rtol = 1e-06,
  atol = 1e-06,
  jacfunc = NULL,
  jactype = "fullint",
  rootfunc = NULL,
  verbose = FALSE,
  nroot = 0,
  tcrit = NULL,
  hmin = 0,
  hmax = NULL,
  hini = 0,
  ynames = TRUE,
  maxordn = 12,
  maxords = 5,
  bandup = NULL,
  banddown = NULL,
  maxsteps = 5000
)
```

Value

List of arguments

P *Generate a parameter transformation function*

Description

Generate parameter transformation function from a named character vector or object of class eqnvec. This is a wrapper function for Pexpl and Pimpl. See for more details there.

Usage

```
P(
  trafo = NULL,
  parameters = NULL,
  condition = NULL,
  attach.input = FALSE,
  keep.root = TRUE,
  compile = FALSE,
  modelname = NULL,
  method = c("explicit", "implicit"),
  verbose = FALSE
)
```

Arguments

trafo	object of class eqnvec or named character or list thereof. In case, trafo is a list, P() is called on each element and conditions are assumed to be the list names.
parameters	character vector
condition	character, the condition for which the transformation is generated
attach.input	attach those incoming parameters to output which are not overwritten by the parameter transformation.
keep.root	logical, applies for method = "implicit". The root of the last evaluation of the parameter transformation function is saved as guess for the next evaluation.
compile	logical, compile the function (see funC0)
modelname	character, see funC0
method	character, either "explicit" or "implicit"
verbose	Print out information during compilation

Value

An object of class parfn.

---

parfn *Parameter transformation function*

Description

Generate functions that transform one parameter vector into another by means of a transformation, pushing forward the jacobian matrix of the original parameter. Usually, this function is called internally, e.g. by P. However, you can use it to add your own specialized parameter transformations to the general framework.

Usage

```
parfn(p2p, parameters = NULL, condition = NULL)
```

Arguments

p2p	a transformation function for one condition, i.e. a function p2p(p, fixed, deriv) which translates a parameter vector p and a vector of fixed parameter values fixed into a new parameter vector. If deriv = TRUE, the function should return an attribute deriv with the Jacobian matrix of the parameter transformation.
parameters	character vector, the parameters accepted by the function
condition	character, the condition for which the transformation is defined

Value

object of class parfn, i.e. a function p(..., fixed, deriv, conditions, env). The argument pars should be passed via the ... argument.

Contains attributes "mappings", a list of p2p functions, "parameters", the union of parameters acceted by the mappings and "conditions", the total set of conditions.

See Also

sumfn, P

Examples

```
# Define a time grid on which to make a prediction by peace-wise linear function.
# Then define a (generic) prediction function based on thid grid.
times <- 0:5
grid <- data.frame(name = "A", time = times, row.names = paste0("p", times))
x <- Xd(grid)

# Define an observable and an observation function
observables <- eqnvec(Aobs = "s*A")
g <- Y(g = observables, f = NULL, states = "A", parameters = "s")

# Collect parameters and define an overarching parameter transformation
# for two "experimental condtions".
dynpars <- attr(x, "parameters")
obspars <- attr(g, "parameters")
innerpars <- c(dynpars, obspars)

trafo <- structure(innerpars, names = innerpars)
trafo_C1 <- replaceSymbols(innerpars, paste(innerpars, "C1", sep = "_"), trafo)
trafo_C2 <- replaceSymbols(innerpars, paste(innerpars, "C2", sep = "_"), trafo)

p <- NULL
p <- p + P(trafo = trafo_C1, condition = "C1")
p <- p + P(trafo = trafo_C2, condition = "C2")

# Collect outer (overarching) parameters and
# initialize with random values
outerpars <- attr(p, "parameters")
pars <- structure(runif(length(outerpars), 0, 1), names = outerpars)
```

```
# Predict internal/unobserved states
out1 <- (x*p)(times, pars)
plot(out1)

# Predict observed states in addition to unobserved
out2 <- (g*x*p)(times, pars)
plot(out2)
```

---

parframe *Generate a parameter frame*

Description

A parameter frame is a data.frame where the rows correspond to different parameter specifications. The columns are divided into three parts. (1) the meta-information columns (e.g. index, value, constraint, etc.), (2) the attributes of an objective function (e.g. data contribution and prior contribution) and (3) the parameters.

Usage

```
parframe(
  x = NULL,
  parameters = colnames(x),
  metanames = NULL,
  obj.attributes = NULL
)

is.parframe(x)

## S3 method for class 'parframe'
x[i = NULL, j = NULL, drop = FALSE]

## S3 method for class 'parframe'
subset(x, ...)
```

Arguments

x	data.frame.
parameters	character vector, the names of the parameter columns.
metanames	character vector, the names of the meta-information columns.
obj.attributes	character vector, the names of the objective function attributes.
i	row index in any format
j	column index in any format
drop	logical. If TRUE the result is coerced to the lowest possible dimension
...	additional arguments

Details

Parameter frames can be subsetted either by `[ , ]` or by `subset`. If `[ , index]` is used, the names of the removed columns will also be removed from the corresponding attributes, i.e. metanames, obj.attributes and parameters.

Value

An object of class `parframe`, i.e. a data.frame with attributes for the different names. Inherits from data.frame.

See Also

profile, mstrust

Examples

```
## Generate a prediction function
regfn <- c(y = "sin(a*time)")

g <- Y(regfn, parameters = "a")
x <- Xt(condition = "C1")

## Generate data
data <- datalist(
  C1 = data.frame(
    name = "y",
    time = 1:5,
    value = sin(1:5) + rnorm(5, 0, .1),
    sigma = .1
  )
)

## Initialize parameters and time
pars <- c(a = 1)
times <- seq(0, 5, .1)

plot((g*x)(times, pars), data)

## Do many fits from random positions and store them into parlist
out <- as.parlist(lapply(1:50, function(i) {
  trust(normL2(data, g*x), pars + rnorm(length(pars), 0, 1), rinit = 1, rmax = 10)
}))

summary(out)

## Reduce parlist to parframe
parframe <- as.parframe(out)
plotValues(parframe)

## Reduce parframe to best fit
bestfit <- as.parvec(parframe)
plot((g*x)(times, bestfit), data)
```

parlist *Parameter list*

Description

The special use of a parameter list is to save the outcome of multiple optimization runs provided by mstrust, into one list.

Fitlists carry an fit index which must be held unique on merging multiple fitlists.

Usage

```
parlist(...)

as.parlist(x = NULL)

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

## S3 method for class 'parlist'
c(...)
```

Arguments

...	Objects to be coerced to parameter list.
x	list of lists, as returned by trust
object	a parlist

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

See Also

load.parlist, plot.parlist

Examples

```
## Generate a prediction function
regfn <- c(y = "sin(a*time)")

g <- Y(regfn, parameters = "a")
x <- Xt(condition = "C1")

## Generate data
data <- datalist(
```

```
  C1 = data.frame(
    name = "y",
    time = 1:5,
    value = sin(1:5) + rnorm(5, 0, .1),
    sigma = .1
  )
)

## Initialize parameters and time
pars <- c(a = 1)
times <- seq(0, 5, .1)

plot((g*x)(times, pars), data)

## Do many fits from random positions and store them into parlist
out <- as.parlist(lapply(1:50, function(i) {
  trust(normL2(data, g*x), pars + rnorm(length(pars), 0, 1), rinit = 1, rmax = 10)
}))

summary(out)

## Reduce parlist to parframe
parframe <- as.parframe(out)
plotValues(parframe)

## Reduce parframe to best fit
bestfit <- as.parvec(parframe)
plot((g*x)(times, bestfit), data)
```

---

parvec *Parameter vector*

Description

A parameter vector is a named numeric vector (the parameter values) together with a "deriv" attribute (the Jacobian of a parameter transformation by which the parameter vector was generated).

Usage

```
parvec(..., deriv = NULL)

as.parvec(x, ...)

## S3 method for class 'numeric'
as.parvec(x, names = NULL, deriv = NULL, ...)

## S3 method for class 'parvec'
```

```
x[..., drop = FALSE]

## S3 method for class 'parvec'
c(...)
```

Arguments

...	objects to be concatenated
deriv	matrix with rownames (according to names of ...) and colnames according to the names of the parameter by which the parameter vector was generated.
x	numeric or named numeric, the parameter values
names	optional character vector, the parameter names. Otherwise, names are taken from x.
drop	logical, drop empty columns in Jacobian after subsetting. ATTENTION: Be careful with this option. The default behavior is to keep the columns in the Jacobian. This can lead to unintended results when subsetting the parvec and using it e.g. in another parameter transformation.

Value

An object of class parvec, i.e. a named numeric vector with attribute "deriv".

Examples

```
# Generate a parameter vector
v <- parvec(a = 2, b = 3)
print(v)
print(getDerivs(v))

# Parameter vector from a named numeric
M <- matrix(c(1, 1, 0, 1),
     nrow = 2, ncol = 2,
     dimnames = list(c("a", "b"), c("A", "B"))
    )
v <- as.parvec(x = c(a = 2, b = 3), deriv = M)
print(v)
print(getDerivs(v))

# Subsetting of parameter vectors
# Case 1: Dependencies in the Jacobian are maintained
w <- v[1]
print(w)
print(getDerivs(w))

# Case 2: Dependencies are dropped
w <- v[1, drop = TRUE]
print(w)
print(getDerivs(w))

# Concatenating parameter vectors
```

```
w <- parvec(c = 4, d = 5)
print(c(v, w))
print(getDerivs(c(v, w)))
```

`Pexpl` *Parameter transformation*

Description

Parameter transformation

Usage

```
Pexpl(
  trafo,
  parameters = NULL,
  attach.input = FALSE,
  condition = NULL,
  compile = FALSE,
  modelname = NULL,
  verbose = FALSE
)
```

Arguments

`trafo`	Named character vector. Names correspond to the parameters being fed into the model (the inner parameters). The elements of tafo are equations that express the inner parameters in terms of other parameters (the outer parameters)
`parameters`	Character vector. Optional. If given, the generated parameter transformation returns values for each element in `parameters`. If elements of `parameters` are not in `names(trafo)` the identity transformation is assumed.
`attach.input`	attach those incoming parameters to output which are not overwritten by the parameter transformation.
`condition`	character, the condition for which the transformation is generated
`compile`	Logical, compile the function (see funC0)
`modelname`	Character, used if `compile = TRUE`, sets a fixed filename for the C file.
`verbose`	Print compiler output to R command line.

Value

a function `p2p(p, fixed = NULL, deriv = TRUE)` representing the parameter transformation. Here, `p` is a named numeric vector with the values of the outer parameters, `fixed` is a named numeric vector with values of the outer parameters being considered as fixed (no derivatives returned) and `deriv` is a logical determining whether the Jacobian of the parameter transformation is returned as attribute "deriv".

See Also

Pimpl for implicit parameter transformations

Examples

```
logtrafo <- c(k1 = "exp(logk1)", k2 = "exp(logk2)",
              A = "exp(logA)", B = "exp(logB)")
p_log <- P(logtrafo)

pars <- c(logk1 = 1, logk2 = -1, logA = 0, logB = 0)
out <- p_log(pars)
getDerivs(out)
```

Pimpl *Parameter transformation (implicit)*

Description

Parameter transformation (implicit)

Usage

```
Pimpl(
  trafo,
  parameters = NULL,
  condition = NULL,
  keep.root = TRUE,
  positive = TRUE,
  compile = FALSE,
  modelname = NULL,
  verbose = FALSE
)
```

Arguments

trafo	Named character vector defining the equations to be set to zero. Names correspond to dependent variables.
parameters	Character vector, the independent variables.
condition	character, the condition for which the transformation is generated
keep.root	logical, applies for method = "implicit". The root of the last evaluation of the parameter transformation function is saved as guess for the next evaluation.
positive	logical, returns projection to the (semi)positive range. Comes with a warning if the steady state has been found to be negative.
compile	Logical, compile the function (see funC0)
modelname	Character, used if compile = TRUE, sets a fixed filename for the C file.
verbose	Print compiler output to R command line.

Details

Usually, the equations contain the dependent variables, the independent variables and other parameters. The argument p of p2p must provide values for the independent variables and the parameters but ALSO FOR THE DEPENDENT VARIABLES. Those serve as initial guess for the dependent variables. The dependent variables are then numerically computed by multiroot. The Jacobian of the solution with respect to dependent variables and parameters is computed by the implicit function theorem. The function p2p returns all parameters as they are with corresponding 1-entries in the Jacobian.

Value

a function `p2p(p, fixed = NULL, deriv = TRUE)` representing the parameter transformation. Here, `p` is a named numeric vector with the values of the outer parameters, `fixed` is a named numeric vector with values of the outer parameters being considered as fixed (no derivatives returned) and `deriv` is a logical determining whether the Jacobian of the parameter transformation is returned as attribute "deriv".

See Also

Pexpl for explicit parameter transformations

Examples

```
##############################################################################
## Example 1: Steady-state trafo
##############################################################################
f <- c(A = "-k1*A + k2*B",
       B = "k1*A - k2*B")
P.steadyState <- Pimpl(f, "A")

p.outerValues <- c(k1 = 1, k2 = 0.1, A = 10, B = 1)
P.steadyState(p.outerValues)

##############################################################################
## Example 2: Steady-state trafo combined with log-transform
##############################################################################
f <- c(A = "-k1*A + k2*B",
       B = "k1*A - k2*B")
P.steadyState <- Pimpl(f, "A")

logtrafo <- c(k1 = "exp(logk1)", k2 = "exp(logk2)", A = "exp(logA)", B = "exp(logB)")
P.log <- P(logtrafo)

p.outerValue <- c(logk1 = 1, logk2 = -1, logA = 0, logB = 0)
(P.log)(p.outerValue)
(P.steadyState * P.log)(p.outerValue)

##############################################################################
## Example 3: Steady-states with conserved quantitites
##############################################################################
f <- c(A = "-k1*A + k2*B", B = "k1*A - k2*B")
```

```
replacement <- c(B = "A + B - total")
f[names(replacement)] <- replacement

pSS <- Pimpl(f, "total")
pSS(c(k1 = 1, k2 = 2, A = 5, B = 5, total = 3))
```

`plot.datalist` *Plot a list data points*

Description

Plot a list data points

Usage

```
## S3 method for class 'datalist'
plot(x, ..., scales = "free", facet = "wrap")
```

Arguments

`x`	Named list of data.frames as being used in res, i.e. with columns `name`, `time`, `value` and `sigma`.
`...`	Further arguments going to `dplyr::filter`.
`scales`	The scales argument of `facet_wrap` or `facet_grid`, i.e. `"free"`, `"fixed"`, `"free_x"` or `"free_y"`
`facet`	Either `"wrap"` or `"grid"`

Details

The data.frame being plotted has columns `time`, `value`, `sigma`, `name` and `condition`.

Value

A plot object of class `ggplot`.

plot.parlist *Plot a parameter list.*

Description

Plot a parameter list.

Usage

```
## S3 method for class 'parlist'
plot(x, path = FALSE, ...)
```

Arguments

x	fitlist obtained from mstrust
path	print path of parameters from initials to convergence. For this option to be TRUE mstrust must have had the option 'blather'.
...	additional arguments

Details

If path=TRUE:

Author(s)

Malenka Mader, <Malenka.Mader@fdm.uni-freiburg.de>

plotArray *Plot an array of trajectories along the profile of a parameter*

Description

Plot an array of trajectories along the profile of a parameter

Usage

```
plotArray(
  par,
  profs,
  prd,
  times,
  direction = c("up", "down"),
  covtable,
  ...,
  nsimus = 4
)
```

Arguments

par	Character of parameter name for which the array should be generated.
profs	Lists of profiles as being returned by profile.
prd	Named list of matrices or data.frames, usually the output of a prediction function as generated by Xs.
times	Numeric vector of time points for the model prediction.
direction	Character "up" or "down" indicating the direction the value should be traced along the profile starting at the bestfit value.
covtable	Optional covariate table or condition.grid necessary if subsetting is required.
...	Further arguments for subsetting the plot.
nsimus	Number of trajectories/ simulation to be calculated.

Value

A plot object of class ggplot.

Author(s)

Svenja Kemmer, <svenja.kemmer@fdm.uni-freiburg.de>

Examples

```
## Not run:
 plotArray("myparameter", myprofiles, g*x*p, seq(0, 250, 1),
    "up", condition.grid, name == "ProteinA" & condition == "c1")

## End(Not run)
```

---

plotCombined.tbl_df *Plot a list of model predictions and a list of data points in a combined plot*

Description

Plot a list of model predictions and a list of data points in a combined plot

Usage

```
## S3 method for class 'tbl_df'
plotCombined(model, hypothesis = 1, index = 1, ..., plotErrorBands = F)

plotCombined(prediction, ...)

## S3 method for class 'prdlist'
plot(x, data = NULL, ..., scales = "free", facet = "wrap", transform = NULL)
```

```
## S3 method for class 'prdlist'
plotCombined(
  prediction,
  data = NULL,
  ...,
  scales = "free",
  facet = "wrap",
  transform = NULL,
  aesthetics = NULL
)

## S3 method for class 'prdframe'
plot(x, data = NULL, ..., scales = "free", facet = "wrap", transform = NULL)
```

Arguments

...	Further arguments going to dplyr::filter.
plotErrorBands	If the dMod.frame contains an observation function for the error model, use it to show error bands.
prediction	Named list of matrices or data.frames, usually the output of a prediction function as generated by Xs.
x	prediction
data	Named list of data.frames as being used in res, i.e. with columns name, time, value and sigma.
scales	The scales argument of facet_wrap or facet_grid, i.e. "free", "fixed", "free_x" or "free_y"
facet	"wrap" or "grid". Try "wrap_plain" for high amounts of conditions and low amounts of observables.
transform	list of transformation for the states, see coordTransform.
aesthetics	Named list of aesthetic mappings, specified as character, e.g. list(linetype = "name"). Can refer to variables in the condition.grid

Details

The data.frame being plotted has columns time, value, sigma, name and condition.

Value

A plot object of class ggplot.

Examples

```
## Observation function
fn <- eqnvec(
  sine = "1 + sin(6.28*omega*time)",
  cosine = "cos(6.28*omega*time)"
```

```
)
g <- Y(fn, parameters = "omega")

## Prediction function for time
x <- Xt()

## Parameter transformations to split conditions
p <- NULL
for (i in 1:3) {
 p <- p + P(trafo = c(omega = paste0("omega_", i)), condition = paste0("frequency_", i))
}

## Evaluate prediction
times <- seq(0, 1, .01)
pars <- structure(seq(1, 2, length.out = 3), names = attr(p, "parameters"))

prediction <- (g*x*p)(times, pars)

## Plotting prediction
# plot(prediction)
plotPrediction(prediction)
plotPrediction(prediction, scales = "fixed")
plotPrediction(prediction, facet = "grid")
plotPrediction(prediction,
               scales = "fixed",
               transform = list(sine = "x^2", cosine = "x - 1"))

## Simulate data
dataset <- wide2long(prediction)
dataset <- dataset[seq(1, nrow(dataset), 5),]
set.seed(1)
dataset$value <- dataset$value + rnorm(nrow(dataset), 0, .1)
dataset$sigma <- 0.1
data <- as.datalist(dataset, split.by = "condition")

## Plotting data
# plot(data)
plot1 <- plotData(data)
 plot1
## Plotting data and prediction with subsetting
# plot(prediction, data)
plot2 <- plotCombined(prediction, data)
 plot2
plot3 <- plotCombined(prediction, data,
             time <= 0.5 & condition == "frequency_1")
 plot3
plot4 <- plotCombined(prediction, data,
             time <= 0.5 & condition != "frequency_1",
             facet = "grid")
 plot4
plot5 <- plotCombined(prediction, data, aesthetics = list(linetype = "condition"))
 plot5
```

plotData.tbl_df *Plot a list data points*

Description

Plot a list data points

Usage

```
## S3 method for class 'tbl_df'
plotData(dMod.frame, hypothesis = 1, ...)

## S3 method for class 'datalist'
plotData(data, ..., scales = "free", facet = "wrap", transform = NULL)

plotData(data, ...)

## S3 method for class 'data.frame'
plotData(data, ...)
```

Arguments

...	Further arguments going to subset.
data	Named list of data.frames as being used in res, i.e. with columns name, time, value and sigma.
scales	The scales argument of facet_wrap or facet_grid, i.e. "free", "fixed", "free_x" or "free_y"
facet	Either "wrap" or "grid"
transform	list of transformation for the states, see coordTransform.

Details

The data.frame being plotted has columns time, value, sigma, name and condition.

Value

A plot object of class ggplot.

Examples

```
## Observation function
fn <- eqnvec(
  sine = "1 + sin(6.28*omega*time)",
  cosine = "cos(6.28*omega*time)"
)
g <- Y(fn, parameters = "omega")
```

```
## Prediction function for time
x <- Xt()

## Parameter transformations to split conditions
p <- NULL
for (i in 1:3) {
 p <- p + P(trafo = c(omega = paste0("omega_", i)), condition = paste0("frequency_", i))
}

## Evaluate prediction
times <- seq(0, 1, .01)
pars <- structure(seq(1, 2, length.out = 3), names = attr(p, "parameters"))

prediction <- (g*x*p)(times, pars)

## Plotting prediction
# plot(prediction)
plotPrediction(prediction)
plotPrediction(prediction, scales = "fixed")
plotPrediction(prediction, facet = "grid")
plotPrediction(prediction,
               scales = "fixed",
               transform = list(sine = "x^2", cosine = "x - 1"))

## Simulate data
dataset <- wide2long(prediction)
dataset <- dataset[seq(1, nrow(dataset), 5),]
set.seed(1)
dataset$value <- dataset$value + rnorm(nrow(dataset), 0, .1)
dataset$sigma <- 0.1
data <- as.datalist(dataset, split.by = "condition")

## Plotting data
# plot(data)
plot1 <- plotData(data)
 plot1
## Plotting data and prediction with subsetting
# plot(prediction, data)
plot2 <- plotCombined(prediction, data)
 plot2
plot3 <- plotCombined(prediction, data,
             time <= 0.5 & condition == "frequency_1")
 plot3
plot4 <- plotCombined(prediction, data,
             time <= 0.5 & condition != "frequency_1",
             facet = "grid")
 plot4
plot5 <- plotCombined(prediction, data, aesthetics = list(linetype = "condition"))
 plot5
```

plotFluxes *Plot Fluxes given a list of flux Equations*

Description

Plot Fluxes given a list of flux Equations

Usage

```
plotFluxes(pouter, x, times, fluxEquations, nameFlux = "Fluxes:", ...)
```

Arguments

pouter	parameters
x	The model prediction function x(times, pouter, fixed, ...)
times	Numeric vector of time points for the model prediction
fluxEquations	list of chars containing expressions for the fluxes, if names are given, they are shown in the legend. Easy to obtain via subset.eqnlist, see Examples.
nameFlux	character, name of the legend.
...	Further arguments going to x, such as fixed or conditions

Value

A plot object of class ggplot.

Examples

```
## Not run:

plotFluxes(bestfit, x, times, subset(f, "B"%in%Product)$rates, nameFlux = "B production")

## End(Not run)
```

plotPars.tbl_df *Plot parameter values for a fitlist*

Description

Plot parameter values for a fitlist

Usage

```
## S3 method for class 'tbl_df'
plotPars(dMod.frame, hypothesis = 1, ..., nsteps = 3, tol = 1)

## S3 method for class 'parframe'
plotPars(x, tol = 1, ...)

plotPars(x, ...)
```

Arguments

...	arguments for subsetting of x
nsteps	number of steps from the waterfall plot
tol	maximal allowed difference between neighboring objective values to be recognized as one.
x	parameter frame as obtained by as.parframe(mstrust)

---

plotPaths.tbl_df *Profile likelihood: plot of the parameter paths.*

Description

Profile likelihood: plot of the parameter paths.

Usage

```
plotPaths.tbl_df(dMod.frame, hypothesis = 1, ...)

plotPaths(
  profs,
  ...,
  whichPar = NULL,
  sort = FALSE,
  relative = TRUE,
  scales = "fixed"
)
```

Arguments

hypothesis	numeric, can be longer than 1
...	arguments going to subset
profs	profile or list of profiles as being returned by profile
whichPar	Character or index vector, indicating the parameters that are taken as possible reference (x-axis)

sort	Logical. If paths from different parameter profiles are plotted together, possible combinations are either sorted or all combinations are taken as they are.
relative	logical indicating whether the origin should be shifted.
scales	character, either "free" or "fixed".

Details

See profile for examples.

Value

A plot object of class ggplot.

`plotPathsMulti` *Profile likelihood: plot all parameter paths belonging to one profile in one plot*

Description

Profile likelihood: plot all parameter paths belonging to one profile in one plot

Usage

```
plotPathsMulti(profs, whichpars, npars = 5)
```

Arguments

profs	Lists of profiles as being returned by profile.
whichpars	Character vector of parameter names for which the profile paths should be generated.
npars	Numeric vector of number of colored and named parameter paths.

Value

A plot object of class ggplot for length(whichpars) = 1 and otherwise an object of class cowplot.

Author(s)

Svenja Kemmer, <svenja.kemmer@fdm.uni-freiburg.de>

Examples

```
## Not run:
 plotPathsMulti(myprofiles, c("mypar1", "mypar2"), npars = 5)

## End(Not run)
```

`plotPrediction.tbl_df` *Plot a list of model predictions*

Description

Plot a list of model predictions

Usage

```
## S3 method for class 'tbl_df'
plotPrediction(dMod.frame, hypothesis = 1, index = 1, ...)

plotPrediction(prediction, ...)

## S3 method for class 'prdlist'
plotPrediction(
  prediction,
  ...,
  errfn = NULL,
  scales = "free",
  facet = "wrap",
  transform = NULL
)
```

Arguments

`...`	Further arguments going to `dplyr::filter`.
`prediction`	Named list of matrices or data.frames, usually the output of a prediction function as generated by Xs.
`errfn`	error model function
`scales`	The scales argument of `facet_wrap` or `facet_grid`, i.e. `"free"`, `"fixed"`, `"free_x"` or `"free_y"`
`facet`	Either `"wrap"` or `"grid"`
`transform`	list of transformation for the states, see coordTransform.

Details

The data.frame being plotted has columns `time`, `value`, `name` and `condition`.

Value

A plot object of class `ggplot`.

Examples

```
## Observation function
fn <- eqnvec(
  sine = "1 + sin(6.28*omega*time)",
  cosine = "cos(6.28*omega*time)"
)
g <- Y(fn, parameters = "omega")

## Prediction function for time
x <- Xt()

## Parameter transformations to split conditions
p <- NULL
for (i in 1:3) {
 p <- p + P(trafo = c(omega = paste0("omega_", i)), condition = paste0("frequency_", i))
}

## Evaluate prediction
times <- seq(0, 1, .01)
pars <- structure(seq(1, 2, length.out = 3), names = attr(p, "parameters"))

prediction <- (g*x*p)(times, pars)

## Plotting prediction
# plot(prediction)
plotPrediction(prediction)
plotPrediction(prediction, scales = "fixed")
plotPrediction(prediction, facet = "grid")
plotPrediction(prediction,
               scales = "fixed",
               transform = list(sine = "x^2", cosine = "x - 1"))

## Simulate data
dataset <- wide2long(prediction)
dataset <- dataset[seq(1, nrow(dataset), 5),]
set.seed(1)
dataset$value <- dataset$value + rnorm(nrow(dataset), 0, .1)
dataset$sigma <- 0.1
data <- as.datalist(dataset, split.by = "condition")

## Plotting data
# plot(data)
plot1 <- plotData(data)
 plot1
## Plotting data and prediction with subsetting
# plot(prediction, data)
plot2 <- plotCombined(prediction, data)
 plot2
plot3 <- plotCombined(prediction, data,
             time <= 0.5 & condition == "frequency_1")
 plot3
plot4 <- plotCombined(prediction, data,
```

```
                time <= 0.5 & condition != "frequency_1",
                facet = "grid")
 plot4
plot5 <- plotCombined(prediction, data, aesthetics = list(linetype = "condition"))
 plot5
```

`plotProfile.tbl_df` *Profile likelihood plot*

Description

Profile likelihood plot

Usage

```
## S3 method for class 'tbl_df'
plotProfile(dMod.frame, hypothesis = 1, ...)

## S3 method for class 'parframe'
plotProfile(profs, ..., maxvalue = 5, parlist = NULL)

## S3 method for class 'list'
plotProfile(profs, ..., maxvalue = 5, parlist = NULL)

plotProfile(profs, ...)
```

Arguments

`hypothesis`	numeric, can be longer than 1
`...`	logical going to subset before plotting.
`profs`	Lists of profiles as being returned by profile.
`maxvalue`	Numeric, the value where profiles are cut off.
`parlist`	Matrix or data.frame with columns for the parameters to be added to the plot as points. If a "value" column is contained, deltas are calculated with respect to lowest chisquare of profiles.

Details

See profile for examples.

Value

A plot object of class `ggplot`.

plotProfilesAndPaths *Profile likelihood: plot profiles along with their parameter paths*

Description

Profile likelihood: plot profiles along with their parameter paths

Usage

```
plotProfilesAndPaths(profs, whichpars, npars = 5)
```

Arguments

profs	Lists of profiles as being returned by profile.
whichpars	Character vector of parameter names for which the profile paths should be generated.
npars	Numeric vector of colored and named parameter paths.

Value

A plot object of class ggplot.

Author(s)

Svenja Kemmer, <svenja.kemmer@fdm.uni-freiburg.de>

Examples

```
## Not run:
 plotProfilesAndPaths(myprofiles, c("mypar1", "mypar2"), npars = 5)

## End(Not run)
```

plotResiduals *Plot residuals for a fitlist*

Description

Plot residuals for a fitlist

Usage

```
plotResiduals(parframe, x, data, split = "condition", errmodel = NULL, ...)
```

Arguments

parframe	Object of class parframe, e.g. returned by mstrust
x	Prediction function returning named list of data.frames with names as data.
data	Named list of data.frames, i.e. with columns name, time, value and sigma.
split	List of characters specifying how to summarise the residuals by sqrt(res_i^2), split[1] used for x-axis, split[2] for grouping (color), and any additional for facet_wrap()
errmodel	object of type prdfn, the error model function.
...	Additional arguments for x

Value

A plot object of class ggplot with data.frame as attribute attr(P,"out").

Examples

```
## Not run:
 # time axis:
 plotResiduals(myfitlist, g*x*p, data,
    c("time","index","condition","name"),
    conditions = myconditions[1:4])
 # condition axis (residuals summed over time for each observable and condition):
 plotResiduals(myfitlist, g*x*p, data,  c("condition","name","index"))

## End(Not run)
```

---

plotValues.tbl_df *Plotting objective values of a collection of fits*

Description

Plotting objective values of a collection of fits

Usage

```
## S3 method for class 'tbl_df'
plotValues(dMod.frame, hypothesis = 1, ..., tol = 1)

## S3 method for class 'parframe'
plotValues(x, tol = 1, ...)

plotValues(x, ...)
```

Arguments

...	arguments for subsetting of x
tol	maximal allowed difference between neighboring objective values to be recognized as one.
x	data.frame with columns "value", "converged" and "iterations", e.g. a parframe.

---

prdfn *Prediction function*

Description

A prediction function is a function x(..., fixed, deriv, conditions). Prediction functions are generated by Xs, Xf or Xd. For an example see the last one.

Usage

```
prdfn(P2X, parameters = NULL, condition = NULL)
```

Arguments

P2X	transformation function as being produced by Xs.
parameters	character vector with parameter names
condition	character, the condition name

Details

Prediction functions can be "added" by the "+" operator, see sumfn. Thereby, predictions for different conditions are merged or overwritten. Prediction functions can also be concatenated with other functions, e.g. observation functions (obsfn) or parameter transformation functions (parfn) by the "*" operator, see prodfn.

Value

Object of class prdfn, i.e. a function x(..., fixed, deriv, conditions, env) which returns a prdlist. The arguments times and pars (parameter values) should be passed via the ... argument, in this order.

Examples

```
# Define a time grid on which to make a prediction by peace-wise linear function.
# Then define a (generic) prediction function based on thid grid.
times <- 0:5
grid <- data.frame(name = "A", time = times, row.names = paste0("p", times))
x <- Xd(grid)

# Define an observable and an observation function
observables <- eqnvec(Aobs = "s*A")
```

```
g <- Y(g = observables, f = NULL, states = "A", parameters = "s")

# Collect parameters and define an overarching parameter transformation
# for two "experimental condtions".
dynpars <- attr(x, "parameters")
obspars <- attr(g, "parameters")
innerpars <- c(dynpars, obspars)

trafo <- structure(innerpars, names = innerpars)
trafo_C1 <- replaceSymbols(innerpars, paste(innerpars, "C1", sep = "_"), trafo)
trafo_C2 <- replaceSymbols(innerpars, paste(innerpars, "C2", sep = "_"), trafo)

p <- NULL
p <- p + P(trafo = trafo_C1, condition = "C1")
p <- p + P(trafo = trafo_C2, condition = "C2")

# Collect outer (overarching) parameters and
# initialize with random values
outerpars <- attr(p, "parameters")
pars <- structure(runif(length(outerpars), 0, 1), names = outerpars)

# Predict internal/unobserved states
out1 <- (x*p)(times, pars)
plot(out1)

# Predict observed states in addition to unobserved
out2 <- (g*x*p)(times, pars)
plot(out2)
```

`prdframe` *Prediction frame*

Description

A prediction frame is used to store a model prediction in a matrix. The columns of the matrix are "time" and one column per state. The prediction frame has attributes "deriv", the matrix of sensitivities with respect to "outer parameters" (see P), an attribute "sensitivities", the matrix of sensitivities with respect to the "inner parameters" (the model parameters, left-hand-side of the parameter transformation) and an attributes "parameters", the parameter vector of inner parameters to produce the prediction frame.

Prediction frames are usually the constituents of prediction lists (prdlist). They are produced by Xs, Xd or Xf. When you define your own prediction functions, see `P2X` in prdfn, the result should be returned as a prediction frame.

Usage

```
prdframe(
  prediction = NULL,
  deriv = NULL,
```

```
  sensitivities = NULL,
  parameters = NULL
)
```

Arguments

prediction	matrix of model prediction
deriv	matrix of sensitivities wrt outer parameters
sensitivities	matrix of sensitivitie wrt inner parameters
parameters	names of the outer paramters

Value

Object of class prdframe, i.e. a matrix with other matrices and vectors as attributes.

---

prdlist *Prediction list*

Description

A prediction list is used to store a list of model predictions from different prediction functions or the same prediction function with different parameter specifications. Each entry of the list is a prdframe.

Usage

```
prdlist(...)

as.prdlist(x, ...)

## S3 method for class 'list'
as.prdlist(x = NULL, names = NULL, ...)
```

Arguments

...	objects of class prdframe conditions.
x	list of prediction frames
names	character vector, the list names, e.g. the names of the experimental

PRD_indiv *Title*

Description

Title

Usage

```
PRD_indiv(prd0, est.grid, fix.grid)
```

Arguments

`prd0`

`est.grid`

`fix.grid`

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

predict.prdfn *Model Predictions*

Description

Make a model prediction for times and a parameter frame. The function is a generalization of the standard prediction by a prediction function object in that it allows to pass a parameter frame instead of a single parameter vector.

Usage

```
## S3 method for class 'prdfn'
predict(object, ..., times, pars, data = NULL)
```

Arguments

`object`	prediction function
`...`	Further arguments goint to the prediction function
`times`	numeric vector of time points
`pars`	parameter frame, e.g. output from mstrust or profile
`data`	data list object. If data is passed, its condition.grid attribute is used to augment the output dataframe by additional columns. `"data"` itself is returned as an attribute.

Value

A data frame

`predtimes` *Get a vector of linearly spaced time points for prediction*

Description

for nice plots

Usage

```
predtimes(datatimes, eventtimes = NULL, Nobjtimes = 100)
```

Arguments

`datatimes`	
`eventtimes`	
`Nobjtimes`	number of time points in total

Value

vector of time points

Examples

```
predtimes(c(30,60,60,90))
predtimes(c(-30,60,60,90))
```

`print.eqnlist` *Print or pander equation list*

Description

Print or pander equation list

Usage

```
## S3 method for class 'eqnlist'
print(x, pander = FALSE, ...)
```

Arguments

`x`	object of class eqnlist
`pander`	logical, use pander for output (used with R markdown)
`...`	additional arguments

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>
Daniel Kaschek, <daniel.kaschek@physik.uni-freiburg.de>

---

print.eqnvec *Print equation vector*

Description

Print equation vector

Usage

```
## S3 method for class 'eqnvec'
print(x, width = 140, pander = FALSE, ...)
```

Arguments

x	object of class eqnvec.
width	numeric, width of the print-out
pander	logical, use pander for output (used with R markdown)
...	not used right now

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

---

print.parfn *Pretty printing parameter transformations*

Description

Pretty printing parameter transformations

Usage

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

Arguments

x	prediction function
...	additional arguments

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

print.parvec *Pretty printing for a parameter vector*

Description

Pretty printing for a parameter vector

Usage

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

Arguments

x	object of class parvec
...	not used yet.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

print0 *Print object and its "default" attributes only.*

Description

Print object and its "default" attributes only.

Usage

```
print0(x, list_attributes = TRUE)
```

Arguments

x	Object to be printed
list_attributes	Prints the names of all attribute of x, defaults to TRUE

Details

Before the 'x' is printed by print.default, all its arguments not in the default list of attrs are removed.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

Mirjam Fehling-Kaschek, <mirjam.fehling@physik.uni-freiburg.de>

priorL2 *L2 objective function for prior value*

Description

As a prior function, it returns derivatives with respect to the penalty parameter in addition to parameter derivatives.

Usage

```
priorL2(mu, lambda = "lambda", attr.name = "prior", condition = NULL)
```

Arguments

mu	Named numeric, the prior values
lambda	Character of length one. The name of the penalty paramter in p.
attr.name	character. The constraint value is additionally returned in an attributed with this name
condition	character, the condition for which the constraint should apply. If NULL, applies to any condition.

Details

Computes the constraint value

$$e^{\lambda}\|p-\mu\|^2$$

and its derivatives with respect to p and lambda.

Value

List of class objlist, i.e. objective value, gradient and Hessian as list.

See Also

wrss

Examples

```
p <- c(A = 1, B = 2, C = 3, lambda = 0)
mu <- c(A = 0, B = 0)
obj <- priorL2(mu = mu, lambda = "lambda")
obj(pars = p + rnorm(length(p), 0, .1))
```

`profile` *Profile-likelihood (PL) computation*

Description

Profile-likelihood (PL) computation

Usage

```
profile(
  obj,
  pars,
  whichPar,
  alpha = 0.05,
  limits = c(lower = -Inf, upper = Inf),
  method = c("integrate", "optimize"),
  stepControl = NULL,
  algoControl = NULL,
  optControl = NULL,
  verbose = FALSE,
  cores = 1,
  cautiousMode = FALSE,
  ...
)
```

Arguments

`obj`	Objective function `obj(pars, fixed, ...)` returning a list with "value", "gradient" and "hessian". If attribute "valueData" and/or "valuePrior are returned they are attached to the return value.
`pars`	Parameter vector corresponding to the log-liklihood optimum.
`whichPar`	Numeric or character vector. The parameters for which the profile is computed.
`alpha`	Numeric, the significance level based on the chisquare distribution with df=1
`limits`	Numeric vector of length 2, the lower and upper deviance from the original value of `pars[whichPar]`
`method`	Character, either `"integrate"` or `"optimize"`. This is a short-cut for setting stepControl, algoControl and optControl by hand.
`stepControl`	List of arguments controlling the step adaption. Defaults to integration set-up, i.e. `list(stepsize = 1e-4, min = 1e-4, max = Inf, atol = 1e-2, rtol = 1e-2, limit = 100)`
`algoControl`	List of arguments controlling the fast PL algorithm. defaults to `list(gamma = 1, W = "hessian", reoptimize = FALSE, correction = 1, reg = .Machine$double.eps)`
`optControl`	List of arguments controlling the `trust()` optimizer. Defaults to `list(rinit = .1, rmax = 10, iterlim = 10, fterm = sqrt(.Machine$double.eps), mterm = sqrt(.Machine$double.eps))`. See trust for more details.

verbose	Logical, print verbose messages.
cores	number of cores used when computing profiles for several parameters.
cautiousMode	Logical, write every step to disk and don't delete intermediate results
...	Arguments going to obj()

Details

Computation of the profile likelihood is based on the method of Lagrangian multipliers and Euler integration of the corresponding differential equation of the profile likelihood paths.

`algoControl`: Since the Hessian which is needed for the differential equation is frequently misspecified, the error in integration needs to be compensated by a correction factor `gamma`. Instead of the Hessian, an identity matrix can be used. To guarantee that the profile likelihood path stays on the true path, each point proposed by the differential equation can be used as starting point for an optimization run when `reoptimize = TRUE`. The correction factor `gamma` is adapted based on the amount of actual correction. If this exceeds the value `correction`, `gamma` is reduced. In some cases, the Hessian becomes singular. This leads to problems when inverting the Hessian. To avoid this problem, the pseudoinverse is computed by removing all singular values lower than `reg`.

`stepControl`: The Euler integration starts with `stepsize`. In each step the predicted change of the objective function is compared with the actual change. If this is larger than `atol`, the stepsize is reduced. For small deviations, either compared the abolute tolerance `atol` or the relative tolerance `rtol`, the stepsize may be increased. `max` and `min` are upper and lower bounds for `stepsize`. `limit` is the maximum number of steps that are take for the profile computation. `stop` is a character, usually "value" or "data", for which the significance level `alpha` is evaluated.

Value

Named list of length one. The name is the parameter name. The list enty is a matrix with columns "value" (the objective value), "constraint" (deviation of the profiled paramter from the original value), "stepsize" (the stepsize take for the iteration), "gamma" (the gamma value employed for the iteration), "valueData" and "valuePrior" (if specified in obj), one column per parameter (the profile paths).

Examples

```
## Not run:

## Parameter transformation
trafo <- eqnvec(a = "exp(loga)",
                b = "exp(logb)",
                c = "exp(loga)*exp(logb)*exp(logc)")
p <- P(trafo)

## Objective function
obj1 <- constraintL2(mu = c(a = .1, b = 1, c = 10), sigma = .6)
obj2 <- constraintL2(mu = c(loga = 0, logb = 0), sigma = 10)
obj <- obj1*p + obj2

## Initialize parameters and obtain fit
pars <- c(loga = 1, logb = 1, logc = 1)
```

```
myfit <- trust(obj, pars, rinit = 1, rmax = 10)
myfit.fixed <- trust(obj, pars[-1], rinit = 1, rmax = 10, fixed = pars[1])

## Compute profiles by integration method
profiles.approx <- do.call(
  rbind,
  lapply(1:3, function(i) {
    profile(obj, myfit$argument, whichPar = i, limits = c(-3, 3),
            method = "integrate")
  })
)

## Compute profiles by repeated optimization
profiles.exact <- do.call(
  rbind,
  lapply(1:3, function(i) {
    profile(obj, myfit$argument, whichPar = i, limits = c(-3, 3),
            method = "optimize")
  })
)

## Compute profiles for fit with fixed element by integration method
profiles.approx.fixed <- do.call(
  rbind,
  lapply(1:2, function(i) {
    profile(obj, myfit.fixed$argument, whichPar = i, limits = c(-3, 3),
            method = "integrate",
            fixed = pars[1])
  })
)

## Plotting
plotProfile(profiles.approx)
plotProfile(list(profiles.approx, profiles.exact))
plotProfile(list(profiles.approx, profiles.approx.fixed))

plotPaths(profiles.approx, sort = TRUE)
plotPaths(profiles.approx, whichPar = "logc")
plotPaths(list(profiles.approx, profiles.approx.fixed), whichPar = "logc")

## Confidence Intervals
confint(profiles.approx, val.column = "value")


## End(Not run)
```

`profile_pars_per_node` *Generate parameter list for distributed profile calculation*

Description

Generates list of `WhichPar` entries to facillitate distribute profile calculation.

Usage

```
profile_pars_per_node(parameters, fits_per_node)
```

Arguments

parameters	list of parameters
fits_per_node	numerical, number of parameters that will be send to each node.

Details

Lists to split the parameters for which the profiles are calculated on the different nodes.

Value

List with two arrays: from contains the number of the starting parameter, while to stores the respective upper end of the parameter list per node.

Examples

```
## Not run:
parameter_list <- setNames(1:10, letters[1:10])
var_list <- profile_pars_per_node(parameter_list, 4)

## End(Not run)
```

---

progressBar *Progress bar*

Description

Progress bar

Usage

```
progressBar(percentage, size = 50, number = TRUE)
```

Arguments

percentage	Numeric between 0 and 100
size	Integer, the size of the bar print-out
number	Logical, Indicates whether the percentage should be printed out.

python_version_request

Check if rPython comes with the correct Python version

Description

rPython is liked against a certain Python version found on the system. If Python code called from R requires a specific Python version, the rPython package needs to be reinstalled. This functions helps to do this in one line.

Usage

```
python_version_request(version)
```

Arguments

version	character indicating the requested Python version

Value

TRUE if rPython is linked against the requested version. Otherwise, the user is asked if rPython should be reinstalled with the correctly linked Python version.

python_version_rpython

Get the Python version to which rPython is linked

Description

Get the Python version to which rPython is linked

Usage

```
python_version_rpython()
```

Value

The Python version and additional information

python_version_sys *Check which Python versions are installed on the system*

Description

Check which Python versions are installed on the system

Usage

```
python_version_sys(version = NULL)
```

Arguments

version	NULL or character. Check for specific version

Value

Character vector with the python versions and where they are located.

P_indiv *Title*

Description

Title

Usage

```
P_indiv(p0, est.grid, fix.grid)
```

Arguments

p0

est.grid

fix.grid

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

`reduceReplicates` *Reduce replicated measurements to mean and standard deviation*

Description

Obtain the mean and standard deviation from replicates per condition.

Usage

```
reduceReplicates(file, select = "condition", datatrans = NULL)
```

Arguments

`file`	Data file of csv. See Format for details.
`select`	Names of the columns in the data file used to define conditions, see Details.
`datatrans`	Character vector describing a function to transform data. Use `x` to refere to data.

Format

The following columns are mandatory for the data file.

name Name of the observed species.

time Measurement time point.

value Measurement value.

condition The condition under which the observation was made.

In addition to these columns, any number of columns can follow to allow a fine grained definition of conditions. The values of all columns named in '`select`' are then merged to get the set of conditions.

Details

Experiments are usually repeated multiple times possibly under different conditions leading to replicted measurements. The column "Condition" in the data allows to group the data by their condition. However, sometimes, a more fine grained grouping is desirable. In this case, any number of additional columns can be append to the data. These columns are referred to as "condition identifier". Which of the condition identifiers are used to do the grouping is user defined by anouncing the to '`select`'. The mandatory column "Condition" is always used. The total set of different conditions is thus defined by all combinations of values occuring in the selected condition identifiers. The replicates of each condition are then reduced to mean and variance.New conditions names are derived by merging all conditions which were used in mean and std.

Value

A data frame of the following variables

time Measurement time point.

name Name of the observed species.

value Mean of replicates.

sigma Standard error of the mean, NA for single measurements.

n The number of replicates reduced.

condition The condition for which the value and sigma were calculated. If more than one column were used to define the condition, this variable holds the effecive condition which is the combination of all applied single conditions.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

`renameDerivPars` *Update attr(prediction, "deriv") to correct par.grid.outer names*

Description

Update attr(prediction, "deriv") to correct par.grid.outer names

Usage

```
renameDerivPars(pred0, pars, est.grid, cn)
```

Arguments

`pred0`	prd0(times,pars)[1]
`pars`	pars
`est.grid`	est.grid
`cn`	name of condition

Value

pred0 with updated deriv argument

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

renameDerivParsInObjlist *Rename the gradient and hessian of an objlist*

Description

Rename the gradient and hessian of an objlist

Usage

```
renameDerivParsInObjlist(objlist, parnames)
```

Arguments

objlist	an objlist with grid.inner parnames
parnames	vector(grid.inner = grid.outer), e.g. as the parnames element from the result of make_pars()

Value

objlist with new names

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

Examples

```
ol <- dMod:::init_empty_objlist(c(a = 1, b = 2))
parnames <- c(a = "c", b = "d")
renameDerivParsInObjlist(ol, parnames)
```

repar *Reparameterization*

Description

Reparameterization

Usage

```
repar(expr, trafo = NULL, ..., reset = FALSE)
```

Arguments

expr	character of the form "lhs ~ rhs" where rhs reparameterizes lhs. Both lhs and rhs can contain a number of symbols whose values need to be passed by the ... argument.
trafo	character or equation vector or list thereof. The object where the replacement takes place in
...	pass symbols as named arguments
reset	logical. If true, the trafo element corresponding to lhs is reset according to rhs. If false, lhs wherever it occurs in the rhs of trafo is replaced by rhs of the formula.

Details

Left and right-hand side of expr are searched for symbols. If separated by "_", symbols are recognized as such, e.g. in Delta_x where the symbols are "Delta" and "x". Each symbol for which values (character or numbers) are passed by the ... argument is replaced.

Value

an equation vector with the reparameterization.

Examples

```
innerpars <- letters[1:3]
constraints <- c(a = "b + c")
mycondition <- "cond1"

trafo <- repar("x ~ x", x = innerpars)
trafo <- repar("x ~ y", trafo, x = names(constraints), y = constraints)
trafo <- repar("x ~ exp(x)", trafo, x = innerpars)
trafo <- repar("x ~ x + Delta_x_condition", trafo, x = innerpars, condition = mycondition)
```

---

repWithNames *Named repititions*

Description

Wrapper on rep() to input names instead of length.

Usage

```
repWithNames(x, names)
```

Arguments

x	Value to be repeated.
names	List of names.

res *Compare data and model prediction by computing residuals*

Description

When sigma is given in the data, it always has priority over the sigma in the error model

Usage

```
res(data, out, err = NULL)
```

Arguments

data	data.frame with name (factor), time (numeric), value (numeric) and sigma (numeric)
out	output of ode(), optionally augmented with attributes "deriv" (output of ode() for the sensitivity equations) and "parameters" (character vector of parameter names, a subsest of those contained in the sensitivity equations). If "deriv" is given, also "parameters" needs to be given.
err	output of the error model function

Value

data.frame with the original data augmented by columns "prediction" (numeric, the model prediction), "residual" (numeric, difference between prediction and data value), "weighted.residual" (numeric, residual devided by sigma). If "deriv" was given, the returned data.frame has an attribute "deriv" (data.frame with the derivatives of the residuals with respect to the parameters).

resolveRecurrence *Place top elements into bottom elemens*

Description

Place top elements into bottom elemens

Usage

```
resolveRecurrence(variables)
```

Arguments

variables	named character vector

Details

If the names of top vector elements occur in the bottom of the vector, they are replaced by the character of the top entry. Useful for steady state conditions.

Value

named character vector of the same length as `variables`

Examples

```
resolveRecurrence(c(A = "k1*B/k2", C = "A*k3+k4", D="A*C*k5"))
```

---

rref *Transform matrix A into reduced row echelon form this function is written along the lines of the rref-matlab function.*

Description

Transform matrix A into reduced row echelon form this function is written along the lines of the rref-matlab function.

Usage

```
rref(A, tol = sqrt(.Machine$double.eps), verbose = FALSE, fractions = FALSE)
```

Arguments

`A`	matrix for which the reduced row echelon form is searched
`tol`	tolerance to find pivots
`verbose`	logical, print verbose information
`fractions`	logical, not used right now.

Value

a list of two entries is returned; ret[[1]] is the reduced row echelon form of A, ret[[2]] is the index of columns in which a pivot was found

Author(s)

Malenka Mader, <Malenka.Mader@fdm.uni-freiburg.de>

runbg *Run an R expression in the background (only on UNIX)*

Description

Generate an R code of the expression that is copied via scp to any machine (ssh-key needed). Then collect the results.

Usage

```
runbg(
  ...,
  machine = "localhost",
  filename = NULL,
  input = ls(.GlobalEnv),
  compile = FALSE,
  wait = FALSE,
  recover = F
)
```

Arguments

...	Some R code
machine	Character vector, e.g. "localhost" or "knecht1.fdm.uni-freiburg.de" or c(localhost, localhost).
filename	Character, defining the filename of the temporary file. Random file name ist chosen if NULL.
input	Character vector, the objects in the workspace that are stored into an R data file and copied to the remove machine.
compile	Logical. If TRUE, C files are copied and compiled on the remote machine. Otherwise, the .so files are copied.
wait	Logical. Wait until executed. If TRUE, the code checks if the result file is already present in which case it is loaded. If not present, runbg() starts, produces the result and loads it as .runbgOutput directly into the workspace. If wait = FALSE, runbg() starts in the background and the result is only loaded into the workspace when the get() function is called, see Value section.
recover	Logical. This option is useful to recover the three functions check(), get() and purge(), e.g. when a session has crashed. Then, the three functions are recreated without restarting the job. They can then be used to get the results of a job wihtout having to do it manually. Requires the correct filename, so if the previous runbg was run with filename = NULL, you have to specify the tmp_filename manually.

Details

runbg() generates a workspace from the input argument and copies the workspace and all C files or .so files to the remote machines via scp. This will only work if *an ssh-key had been generated and added to the authorized keys on the remote machine*. The code snippet, i.e. the ... argument, can include several intermediate results but only the last call which is not redirected into a variable is returned via the variable .runbgOutput, see example below.

Value

List of functions check, get() and purge(). check() checks, if the result is ready. get() copies the result file to the working directory and loads it into the workspace as an object called .runbgOutput. This object is a list named according to the machines that contains the results returned by each machine. purge() deletes the temporary folder from the working directory and the remote machines.

Examples

```
## Not run:
out_job1 <- runbg({
         M <- matrix(rnorm(1e2), 10, 10)
         solve(M)
         }, machine = c("localhost", "localhost"), filename = "job1")
out_job1$check()
out_job1$get()
result <- .runbgOutput
print(result)
out_job1$purge()

## End(Not run)
## Not run:
#' Recover a runbg job with the option "recover"
out_job1 <- runbg({
         M <- matrix(rnorm(1e2), 10, 10)
         solve(M)
         }, machine = c("localhost", "localhost"), filename = "job1")
Sys.sleep(1)
remove(out_job1)
try(out_job1$check())
out_job1 <- runbg({
  "This code is not run"
}, machine = c("localhost", "localhost"), filename = "job1", recover = T)
out_job1$get()
result <- .runbgOutput
print(result)
out_job1$purge()

## End(Not run)
```

runbgInstall *Remote install dMod to a ssh-reachable host*

Description

Install your local dMod version to a remote host via ssh.

Usage

```
runbgInstall(sshtarget, source = NULL, type = "local")
```

Arguments

sshtarget	The ssh host url.
source	If type = local, source must point to the source directory of your dMod version. This is most probably you local dMod git repository.
type	Which dMod to install. At the moment, only your local version is supported.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

runbg_bwfor *Run an R expression on the bwForCluster*

Description

Generate an R code of the expression that is copied via scp to the bwForCluster (ssh-key needed). Then collect the results.

Usage

```
runbg_bwfor(
  ...,
  machine,
  filename = NULL,
  nodes = 1,
  cores = 1,
  walltime = "01:00:00",
  input = ls(.GlobalEnv),
  compile = TRUE,
  recover = F
)
```

Arguments

...	Some R code
machine	e.g. fr_dk846@bwfor.cluster.uni-mannheim.de
filename	Character, defining the filename of the temporary file. Random file name ist chosen if NULL.
nodes	Number of nodes, e.g. 10
cores	Number of cores, e.g. 16
walltime	estimated runtime in the format hh:mm:ss, e.g. 01:30:00. Jobs with a walltime up to 30 min are sent to a quick queue. When the walltime is exceeded, all jobs are automatically killed by the queue.
input	Character vector, the objects in the workspace that are stored into an R data file and copied to the remove machine.
compile	Logical. If TRUE, C files are copied and compiled on the remote machine. Otherwise, the .so files are copied.
recover	Logical, If TRUE, the scripts will not be started again. Can be used to get back the check and get functions for an already started process, e.g. after local session has aborted.

Details

runbg() generates a workspace from the input argument and copies the workspace and all C files or .so files to the remote machines via scp. This will only work if *an ssh-key had been generated and added to the authorized keys on the remote machine*. The code snippet, i.e. the ... argument, can include several intermediate results but only the last call which is not redirected into a variable is returned via the variable .runbgOutput, see example below.

Value

List of functions check(), get() and purge(). check() checks, if the result is ready. get() copies the result file to the working directory and loads it into the workspace as an object called .runbgOutput. This object is a list named according to the machines that contains the results returned by each machine. purge() deletes the temporary folder from the working directory and the remote machines.

Examples

```
## Not run:
out_job1 <- runbg({
   mstrust(obj, center, fits = 10, cores = 2)
 },
 machine = "bwfor", nodes = 2, cores = "2:best",
 walltime = "00:01:00",
 filename = "job1")
out_job1$check()
out_job1$get()
out_job1$purge()
result <- .runbgOutput
```

```
print(result)

## End(Not run)
```

`runbg_bwfor_slurm` *Run an R expression on the bwForCluster via sshpass and slurm*

Description

Generate an R code of the expression that is copied via `scp` to the bwForCluster. Then collect the results. ssh-key not needed. Password can be provided via an additional argument. sshpass needs to be installed on your local machine.

Usage

```
runbg_bwfor_slurm(
  ...,
  machine,
  filename = NULL,
  nodes = 1,
  cores = 1,
  partition = "single",
  walltime = "01:00:00",
  input = ls(.GlobalEnv),
  compile = TRUE,
  recover = F,
  password = "'begin__end'"
)
```

Arguments

`...`	Some R code
`machine`	e.g. `fr_dk846@bwfor.cluster.uni-mannheim.de`
`filename`	Character, defining the filename of the temporary file. Random file name ist chosen if NULL. Must not contain the string "Minus".
`nodes`	Number of nodes, e.g. 10
`cores`	Number of cores, e.g. 16
`partition`	character, the partition where to start the job
`walltime`	estimated runtime in the format `hh:mm:ss`, e.g. `01:30:00`. Jobs with a walltime up to 30 min are sent to a quick queue. When the walltime is exceeded, all jobs are automatically killed by the queue.
`input`	Character vector, the objects in the workspace that are stored into an R data file and copied to the remove machine.

compile	Logical. If TRUE, C files are copied and compiled on the remote machine. Otherwise, the .so files are copied.
recover	Logical, If TRUE, the scripts will not be started again. Can be used to get back the check and get functions for an already started process, e.g. after local session has aborted.
password	Your ssh password in plain text (yes, no joke unfortunately), the password is handed over to sshpass for automatic login on the cluster. If NULL, the standard ssh/scp is used and you will be asked for your password multiple times while uploading the scripts.

Details

runbg() generates a workspace from the input argument and copies the workspace and all C files or .so files to the remote machines via scp. This will only work if *an ssh-key had been generated and added to the authorized keys on the remote machine*. The code snippet, i.e. the ... argument, can include several intermediate results but only the last call which is not redirected into a variable is returned via the variable .runbgOutput, see example below.

Value

List of functions check(), get() and purge(). check() checks, if the result is ready. get() copies the result file to the working directory and loads it into the workspace as an object called .runbgOutput. This object is a list named according to the machines that contains the results returned by each machine. purge() deletes the temporary folder from the working directory and the remote machines.

Examples

```
## Not run:
out_job1 <- runbg({
   mstrust(obj, center, fits = 10, cores = 2)
 },
 machine = "bwfor", nodes = 2, cores = "2:best",
 walltime = "00:01:00",
 filename = "job1")
out_job1$check()
out_job1$get()
out_job1$purge()
result <- .runbgOutput
print(result)

## End(Not run)
```

runbg_bwfor_sshpass *Run an R expression on the bwForCluster via sshpass*

Description

Generate an R code of the expression that is copied via scp to the bwForCluster. Then collect the results. ssh-key not needed. Password can be provided via an additional argument. sshpass needs to be installed on your local machine.

Usage

```
runbg_bwfor_sshpass(
  ...,
  machine,
  filename = NULL,
  nodes = 1,
  cores = 1,
  walltime = "01:00:00",
  input = ls(.GlobalEnv),
  compile = TRUE,
  recover = F,
  password = "'begin__end'"
)
```

Arguments

...	Some R code
machine	e.g. fr_dk846@bwfor.cluster.uni-mannheim.de
filename	Character, defining the filename of the temporary file. Random file name ist chosen if NULL.
nodes	Number of nodes, e.g. 10
cores	Number of cores, e.g. 16
walltime	estimated runtime in the format hh:mm:ss, e.g. 01:30:00. Jobs with a walltime up to 30 min are sent to a quick queue. When the walltime is exceeded, all jobs are automatically killed by the queue.
input	Character vector, the objects in the workspace that are stored into an R data file and copied to the remove machine.
compile	Logical. If TRUE, C files are copied and compiled on the remote machine. Otherwise, the .so files are copied.
recover	Logical, If TRUE, the scripts will not be started again. Can be used to get back the check and get functions for an already started process, e.g. after local session has aborted.
password	Your ssh password in plain text (yes, no joke unfortunately), the password is handed over to sshpass for automatic login on the cluster.

Details

runbg() generates a workspace from the input argument and copies the workspace and all C files or .so files to the remote machines via scp. This will only work if *an ssh-key had been generated and added to the authorized keys on the remote machine*. The code snippet, i.e. the ... argument, can include several intermediate results but only the last call which is not redirected into a variable is returned via the variable .runbgOutput, see example below.

Value

List of functions check(), get() and purge(). check() checks, if the result is ready. get() copies the result file to the working directory and loads it into the workspace as an object called .runbgOutput. This object is a list named according to the machines that contains the results returned by each machine. purge() deletes the temporary folder from the working directory and the remote machines.

Examples

```
## Not run:
out_job1 <- runbg({
   mstrust(obj, center, fits = 10, cores = 2)
 },
 machine = "bwfor", nodes = 2, cores = "2:best",
 walltime = "00:01:00",
 filename = "job1")
out_job1$check()
out_job1$get()
out_job1$purge()
result <- .runbgOutput
print(result)

## End(Not run)
```

---

saveShiny *Save the necessary objects for dModtoShiny with the correct names*

Description

Save the necessary objects for dModtoShiny with the correct names

Usage

```
saveShiny(
  reactions = myeqnlist,
  x,
  parameters,
  fixed = NULL,
  data,
```

```
  profiles,
  pubref = "none",
  errmodel = NULL
)
```

Arguments

reactions	eqnlist-object
x	prediction funciton
parameters	Parframe, result of mstrust()
fixed	Character? fixed parameters?
data	datalist
profiles	Parframe, result of profile(). Is a proflist also possible?
pubref	Link to publication
errmodel	obsfn

---

saveShiny_dMod.frame *SaveShiny for dMod.frames*

Description

SaveShiny for dMod.frames

Usage

```
saveShiny_dMod.frame(
  dMod.frame,
  hypothesis = 1,
  reactions = dMod.frame$reactions[[hypothesis]],
  pubref = "none",
  fixed = dMod.frame$fixed[[hypothesis]],
  projectname,
  appfolder = "/home/mfehling/ShinyApps/dModtoShiny/"
)
```

Arguments

dMod.frame	dmod.frame
hypothesis	hypothesis
reactions	eqnlist which are the basis for prd
pubref	character. pubmed link or similar
fixed	
projectname	Name of the folder being created on fermi

Examples

```
## Not run:
library(dMod)
# library(dplyr) # devtools::install_github("dlill/conveniencefunctions")
library(conveniencefunctions)

setwd(tempdir())

## Model definition (text-based, scripting part)
reactions <- NULL %>%
  addReaction("A", "B", "k1*A", "translation") %>%
  addReaction("B",  "", "k2*B", "degradation")

f <- reactions %>%
  as.eqnvec()


events <- eventlist(var = "A", time = 5, value = "A_add", method = "add")

x <- odemodel(f, events = events) %>% Xs

g <- Y(c(Bobs = "s1*B"), x, compile = T, modelname = "obsfn")

conditions <- c("a", "b")

# there is a bug in
# getParameters(g*x)
parameters <- union(getParameters(g), getParameters(x))

trafo <-
  NULL %>%
  define("x~x", x = parameters) %>%
  branch(conditions = conditions) %>%
  insert("x~x_cond", x = "s1", cond = condition) %>%
  insert("x~1", x = "added", conditionMatch = "a") %>%
  insert("x~5", x = "added", conditionMatch = "b") %>%
  insert("x~exp(x)", x = getSymbols(mytrafo[[i]])) %>%
  {.}

p <- P(trafo)

# Parameter transformation for steady states
pSS <- P(f, method = "implicit", condition = NULL, compile = T)




## Process data

# Data
data <- datalist(
  a = data.frame(time = c(0, 2, 7),
```

```
                  name = "Bobs",
                  value = c(.3, .3, .3),
                  sigma = c(.03, .03, .03)),
  b = data.frame(time = c(0, 2, 7),
                  name = "Bobs",
                  value = c(.1, .1, .2),
                  sigma = c(.01, .01, .02))
)


myframe <- dMod.frame("no steady states", g, x, p, data) %>%
  appendObj() %>%
  mutate(constr = list(constraintL2(mu = 0*pars, sigma = 5)),
         obj = list(obj_data + constr),
       fits = list(mstrust(obj, pars, studyname = "Fits", fits = 20, cores = 4, blather = T))) %>%
  appendParframes() %>%
  mutate(profiles = list(profile(obj, as.parvec(parframes), whichPar = "k1"))) %>%
  mutate(vali = list(datapointL2("A", 2, "mypoint", .1, condition = "a")),
         obj_vali = list(obj_data + constr + vali),
       par_vali = list(c(dMod:::sanitizePars(as.parvec(parframes))$pars, "mypoint" = 0.1 )),
         fits_vali = list(mstrust(obj_vali, par_vali)),
       profile_vali = list(profile(obj_vali, fits_vali %>% as.parframe %>% as.parvec, "mypoint"))) %>%
  {.}

myframe <- myframe %>% tibble::add_column(reactions = list(reactions), fixed = list(NULL))


saveShiny_dMod.frame(myframe, projectname = "dModFrameTest")


## End(Not run)
```

`scaledependent` *Scale dependent flux sensitivity Analysis*

Description

This function performs the scale dependent flux sensitivity analysis for a simulated system. It prints a pdf-file to the working directory for each condition of the system.

Usage

```
scaledependent(
  frame,
  x,
  obstates,
  bestfit,
  time,
  p,
```

```
  bound = 0.001,
  figurename = "scaledependent",
  timeunit = "h",
  width = 16,
  height = 9,
  linewidth = 2
)
```

Arguments

frame	Reactionlist of the model
x	Xs function of the model
obstates	The states under observation whose changes are analysed
bestfit	The parameter values which are used for the simulation
time	Array of observed timepoints
p	P function of the system
bound	Absolute value which a flux sensitive component has to reach at any point in time to be shown in the outputplots
figurename	Name of the saved pdf
timeunit	Unit of the time at the x-axis of the plots
linewidth	Linewidth of the coloured fluxes in the plots
time, width	Dimension of the plots

Examples

```
#Scale dependent flux sensitivity Analysis for the SIR-Model with two different values for beta

frame <- NULL
frame <-addReaction(frame, from = "S", to = "I", rate = "beta*S*I")
frame <-addReaction(frame, from = "I", to = "R", rate = "gamma*I")
x <-Xs(odemodel(frame))

obstates = c("S", "I", "R")

par <- getParameters(x)
p <- NULL
for (i in c(0.0026, 0.0027)){
  trafo <- repar("x~x", x=par)
  trafo <- repar("x~y", x = "beta", y= i, trafo)
  parameter <- structure(trafo, names = par)
  p <- p+P(parameter, condition= paste0(i))}


time <- seq(0, 14, length.out=1000)
bestfit <- c(S = 7.637603e+02, I = 6.184001e-01, R = 8.251977e-16, gamma = 4.582934e-01)

scaledependent(frame, x, obstates, bestfit, time, p)
```

`scaleinvariant` *Scale invariant flux sensitivity Analysis*

Description

This function performs the scale invariant flux sensitivity analysis for a simulated system. It prints a pdf-file to the working directory for each condition of the system.

Usage

```
scaleinvariant(
  frame,
  x,
  obstates,
  bestfit,
  time,
  p,
  bound = 0.001,
  figurename = "scaleinvariant",
  timeunit = "h",
  width = 16,
  height = 9,
  linewidth = 2
)
```

Arguments

`frame`	Reactionlist of the model
`x`	Xs function of the Model
`obstates`	The states under observation whose changes are analysed
`bestfit`	The parameter values which are used for the simulation
`time`	Array of observed timepoints
`p`	P function of the system
`bound`	Absolute value which a flux sensitive component has to reach at any point in time to be shown in the outputplots
`figurename`	Name of the saved pdf
`timeunit`	Unit of the time at the x-axis of the plots
`linewidth`	Linewidth of the coloured fluxes in the plots
`time, width`	Dimensions of the plots

Examples

```
#Scale invariant flux sensitivity Analysis for the SIR-Model with two different values for beta

frame <- NULL
frame <-addReaction(frame, from = "S", to = "I", rate = "beta*S*I")
frame <-addReaction(frame, from = "I", to = "R", rate = "gamma*I")
x <-Xs(odemodel(frame))

obstates = c("S", "I", "R")

par <- getParameters(x)
p <- NULL
for (i in c(0.0026, 0.0027)){
  trafo <- repar("x~x", x=par)
  trafo <- repar("x~y", x = "beta", y= i, trafo)
  parameter <- structure(trafo, names = par)
  p <- p+P(parameter, condition= paste0(i))}


time <- seq(0, 14, length.out=1000)
bestfit <- c(S = 7.637603e+02, I = 6.184001e-01, R = 8.251977e-16, gamma = 4.582934e-01)

scaleinvariant(frame, x, obstates, bestfit, time, p)
```

---

scale_color_dMod *Standard dMod color palette*

Description

Standard dMod color palette

Usage

```
scale_color_dMod(...)
```

Arguments

...	arguments goint to codescale_color_manual()

Examples

```
library(ggplot2)
times <- seq(0, 2*pi, 0.1)
values <- sin(times)
data <- data.frame(
   time = times,
   value = c(values, 1.2*values, 1.4*values, 1.6*values),
   group = rep(c("C1", "C2", "C3", "C4"), each = length(times))
)
qplot(time, value, data = data, color = group, geom = "line") +
   theme_dMod() + scale_color_dMod()
```

scale_fill_dMod *Standard dMod color scheme*

Description

Standard dMod color scheme

Usage

```
scale_fill_dMod(...)
```

Arguments

...	arguments goint to codescale_color_manual()

stat.parlist *Gather statistics of a fitlist*

Description

Gather statistics of a fitlist

Usage

```
stat.parlist(x)
```

Arguments

x	The fitlist

steadyStates *Calculate analytical steady states.*

Description

This function follows the method published in [1]. Find the latest version of the tool and some examples under [2]. The determined steady-state solution is tailored to parameter estimation. Please note that kinetic parameters might be fixed for solution of steady-state equations. Note that additional parameters might be introduced to ensure positivity of the solution.

The function calls a python script via the reticulate package. Use python3.x

Usage

```
steadyStates(
  model,
  file = NULL,
  rates = NULL,
  forcings = NULL,
  givenCQs = NULL,
  neglect = NULL,
  sparsifyLevel = 2,
  outputFormat = "R",
  testSteady = "T"
)
```

Arguments

model	Either name of the csv-file or the eqnlist of the model.
file	Name of the file to which the steady-state equations are saved.
rates	Character vector, flux vector of the system
forcings	Character vector with the names of the forcings
givenCQs	(Unnamed) Character vector with conserved quantities. Use the format c("A + pA = totA", "B + pB = totB"). The format c("A + pA", "B + pB") works also. If NULL, conserved quantities are automatically calculated.
neglect	Character vector with names of states and parameters that must not be used for solving the steady-state equations
sparsifyLevel	numeric, Upper bound for length of linear combinations used for simplifying the stoichiometric matrix
outputFormat	Define the output format. By default "R" generating dMod compatible output. To obtain an output appropriate for d2d [2] "M" must be selected.
testSteady	Boolean, if "T" the correctness of the obtained steady states is numerically checked (this can be very time intensive). If "F" this is skipped.

Value

Character vector of steady-state equations.

Author(s)

Marcus Rosenblatt, <marcus.rosenblatt@fdm.uni-freiburg.de>

References

[1] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4863410/

[2] https://github.com/marcusrosenblatt/AlyssaPetit

[3] https://github.com/Data2Dynamics/d2d

Examples

```
## Not run:
  library(dMod)
  setwd(tempdir())

  reactions <- eqnlist()
  reactions <-   addReaction(reactions, "Tca_buffer", "Tca_cyto",
                             "import_Tca*Tca_buffer", "Basolateral uptake")
  reactions <-   addReaction(reactions, "Tca_cyto", "Tca_buffer",
                             "export_Tca_baso*Tca_cyto", "Basolateral efflux")
  reactions <-   addReaction(reactions, "Tca_cyto", "Tca_canalicular",
                             "export_Tca_cana*Tca_cyto", "Canalicular efflux")
  reactions <-   addReaction(reactions, "Tca_canalicular", "Tca_buffer",
                             "transport_Tca*Tca_canalicular", "Transport bile")

  mysteadies <- steadyStates(reactions)
  print(mysteadies)

  x <- Xs(odemodel(reactions))

  parameters <- getParameters(x)
  trafo <- `names<-`(parameters, parameters)
 trafo <- repar("inner~steadyEqn", trafo, inner = names(mysteadies), steadyEqn = mysteadies)

  pSS <- P(trafo, condition = "steady")

  set.seed(2)
  pars <- structure(runif( length(getParameters(pSS)), 0,1), names = getParameters(pSS))

  prediction <- (x*pSS)(seq(0,10, 0.1), pars, deriv = F)
  plot(prediction)

## End(Not run)
```

`steadyStateToolJulia` *Calculated the steady states of a given model*

Description

Uses julia to calculate the steady state transformations

Usage

```
steadyStateToolJulia(el, forcings = NULL, neglect = NULL, verboseLevel = 1)
```

Arguments

`el`	the equation list

forcings	vector of strings, default c("","). The names of the forcings which will be set to zero.
neglect	vector of strings, default c("","). The names of the variables which will be neglected as fluxParameters and therefore will not be solved for.
verboseLevel	integer, default 1. The level of verbosity of the output, right now only 1 (all) and 0 (no) is implementes.

Value

named vector with the steady state transformations. The names are the inner, the values are the outer parameters

---

strelide *Elide character vector*

Description

Elide character vector

Usage

```
strelide(string, width, where = "right", force = FALSE)
```

Arguments

string	String subject to eliding
width	Width including eliding ... of return string
where	Eliding can happen at 'left', 'middel', or 'right'. Defaults to 'right'.
force	Elide, even is <string> is shorter than <width>. Default to 'FALSE'.

Details

Elide a string to <width>. Eliding can happen at 'left', 'middle', or 'right'. #' If forcing = FALSE, which is the default, strings shorten than <width> are returend unaltered; forcing = TRUE inserts eliding symbols (...) in any case.

Value

Elided string of length <width>.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

strpad *Pad string to desired width*

Description

Pad string to desired width

Usage

```
strpad(string, width, where = "right", padding = " ", autoelide = FALSE)
```

Arguments

string	String to pad
width	Desired width of padded string
where	Padding can be inserted to the right or left of <string>. Default to 'right'.
padding	A single character with with the padding space is filled. Defaults to blank ' ' yielding invisible padding.
autoelide	If TRUE, <string> is elided if it is wider than <width>. The position of eliding follows <where>. Defaults to FALSE.

Value

Padded string of length <width>.

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

submatrix *Submatrix of a matrix returning ALWAYS a matrix*

Description

Submatrix of a matrix returning ALWAYS a matrix

Usage

```
submatrix(M, rows = 1:nrow(M), cols = 1:ncol(M))
```

Arguments

M	matrix
rows	Index vector
cols	Index vector

Value

The matrix `M[rows, cols]`, keeping/adjusting attributes like ncol nrow and dimnames.

`subset.eqnlist` *subset of an equation list*

Description

subset of an equation list

Usage

```
## S3 method for class 'eqnlist'
subset(x, ...)
```

Arguments

`x`	the equation list
`...`	logical expression for subsetting

Details

The argument `...` can contain "Educt", "Product", "Rate" and "Description". The "

Value

An object of class eqnlist

Examples

```
reactions <- data.frame(Description = c("Activation", "Deactivation"),
                        Rate = c("act*A", "deact*pA"), A=c(-1,1), pA=c(1, -1) )
f <- as.eqnlist(reactions)
subset(f, "A" %in% Educt)
subset(f, "pA" %in% Product)
subset(f, grepl("act", Rate))
```

`sumDuplicatedParsInDeriv` *Sum redundant columns*

Description

Remove redundancies (happens when a parameter is duplicated and mapped to the same outer parameter). For example in est.grid = data.table(ID = 1, init_A = "Atot", Atot= "Atot", ...)

Usage

```
sumDuplicatedParsInDeriv(der)
```

Arguments

`der`	deriv of a single condition, as used in [renameDerivPars]

Value

der with redundant columns summed and duplicates removed

`sumDuplicatedParsInObjlist` *Remove redundant outer names*

Description

Remove redundancies (happens when a parameter is duplicated and mapped to the same outer parameter). For example in est.grid = data.table(ID = 1, init_A = "Atot", Atot= "Atot", ...)

Usage

```
sumDuplicatedParsInObjlist(ol)
```

Arguments

`objlist`	objlist with potentially duplicated names

Value

objlist with duplicated gradient and hessian elements summed and redundancies removed

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

Examples

```
ol <- dMod:::init_empty_objlist(c("S2" = 2, "S3" = 3, S2 = 3))
ol$gradient <- ol$gradient + 1:3
ol$hessian <- ol$hessian + 1:9
sumDuplicatedParsInObjlist(ol)
```

`summary.eqnvec` *Summary of an equation vector*

Description

Summary of an equation vector

Usage

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

Arguments

`object`	of class eqnvec.
`...`	additional arguments

Author(s)

Wolfgang Mader, <Wolfgang.Mader@fdm.uni-freiburg.de>

`symmetryDetection` *Search for symmetries in the loaded model*

Description

This function follows the method published in [1].

The function calls a python script via rPython. Usage problems might occur when different python versions are used. The script was written and tested for python 2.7.12, sympy 0.7.6.

Recently, users went into problems with RJSONIO when rPython was used. Unless a sound solution is available, please try to reinstall RJSONIO in these cases.

Usage

```
symmetryDetection(
  f,
  obsvect = NULL,
  prediction = NULL,
  initial = NULL,
  ansatz = "uni",
  pMax = 2,
  inputs = NULL,
  fixed = NULL,
  cores = 1,
  allTrafos = FALSE
)
```

Arguments

f	object containing the ODE for which as.eqnvec() is defined
obsvect	vector of observation functions
prediction	vector containing prediction to be tested
initial	vector containing initial values
ansatz	type of infinitesimal ansatz used for the analysis (uni, par, multi)
pMax	maximal degree of infinitesimal ansatz
inputs	specify the input variables
fixed	variables to concider fixed
cores	maximal number of cores used for the analysis
allTrafos	do not remove transformations with a common parameter factor

References

[1] https://journals.aps.org/pre/abstract/10.1103/PhysRevE.92.012920

Examples

```
## Not run:
eq <- NULL
eq <- addReaction(eq, "A", "B", "k1*A")
eq <- addReaction(eq, "B", "A", "k2*B")

observables <- eqnvec(Aobs = "alpha * A")

symmetryDetection(eq, observables)


## End(Not run)
```

theme_dMod *Standard plotting theme of dMod*

Description

Standard plotting theme of dMod

Usage

```
theme_dMod(base_size = 12, base_family = "")
```

Arguments

base_size	numeric, font-size
base_family	character, font-name

timepointL2_indiv *DatapointL2 without env access*

Description

DatapointL2 without env access

Usage

```
timepointL2_indiv(
  name,
  time,
  value,
  sigma = 1,
  attr.name = "timepointL2",
  condition,
  prd_indiv
)
```

Arguments

name

time

value

sigma

attr.name

condition

prd_indiv

Author(s)

Daniel Lill (daniel.lill@physik.uni-freiburg.de)

`trust` *Non-Linear Optimization*

Description

This function carries out a minimization or maximization of a function using a trust region algorithm. See the references for details.

Usage

```
trust(
  objfun,
  parinit,
  rinit,
  rmax,
  parscale,
  iterlim = 100,
  fterm = sqrt(.Machine$double.eps),
  mterm = sqrt(.Machine$double.eps),
  minimize = TRUE,
  blather = FALSE,
  parupper = Inf,
  parlower = -Inf,
  printIter = FALSE,
  traceFile = NULL,
  ...
)

trustL1(
  objfun,
  parinit,
  mu = 0 * parinit,
  one.sided = FALSE,
  lambda = 1,
  rinit,
  rmax,
  parscale,
  iterlim = 100,
  fterm = sqrt(.Machine$double.eps),
  mterm = sqrt(.Machine$double.eps),
  minimize = TRUE,
  blather = FALSE,
  blather2 = FALSE,
```

```
  parupper = Inf,
  parlower = -Inf,
  printIter = FALSE,
  ...
)
```

Arguments

objfun	an R function that computes value, gradient, and Hessian of the function to be minimized or maximized and returns them as a list with components value, gradient, and hessian. Its first argument should be a vector of the length of parinit followed by any other arguments specified by the ... argument.
parinit	starting parameter values for the optimization. Must be feasible (in the domain).
rinit	starting trust region radius. The trust region radius (see details below) is adjusted as the algorithm proceeds. A bad initial value wastes a few steps while the radius is adjusted, but does not keep the algorithm from working properly.
rmax	maximum allowed trust region radius. This may be set very large. If set small, the algorithm traces a steepest descent path (steepest ascent, when minimize = FALSE).
parscale	an estimate of the size of each parameter at the minimum. The algorithm operates as if optimizing function(x, ...) objfun(x / parscale, ...). May be missing in which case no rescaling is done. See also the details section below.
iterlim	a positive integer specifying the maximum number of iterations to be performed before the program is terminated.
fterm	a positive scalar giving the tolerance at which the difference in objective function values in a step is considered close enough to zero to terminate the algorithm.
mterm	a positive scalar giving the tolerance at which the two-term Taylor-series approximation to the difference in objective function values in a step is considered close enough to zero to terminate the algorithm.
minimize	If TRUE minimize. If FALSE maximize.
blather	If TRUE return extra info.
parupper	named numeric vector of upper bounds. If not named, first value will be used for all parameters.
parlower	named numeric vector of lower bounds. If not named, first value will be used for all parameters.
printIter	print iteration information to R console
...	additional argument to objfun
mu	named numeric value. The reference value for L1 penalized parameters.
one.sided	logical. One-sided penalization.
lambda	strength of the L1 penalty
blather2	even more information

Details

See Fletcher (1987, Section 5.1) or Nocedal and Wright (1999, Section 4.2) for detailed expositions.

Value

A list containing the following components:

- value: the value returned by objfun at the final iterate.
- gradient: the gradient returned by objfun at the final iterate.
- hessian: the Hessian returned by objfun at the final iterate.
- argument: the final iterate
- converged: if TRUE the final iterate was deemed optimal by the specified termination criteria.
- iterations: number of trust region subproblems done (including those whose solutions are not accepted).
- argpath: (if blather == TRUE) the sequence of iterates, not including the final iterate.
- argtry: (if blather == TRUE) the sequence of solutions of the trust region subproblem.
- steptype: (if blather == TRUE) the sequence of cases that arise in solutions of the trust region subproblem. "Newton" means the Newton step solves the subproblem (lies within the trust region). Other values mean the subproblem solution is constrained. "easy-easy" means the eigenvectors corresponding to the minimal eigenvalue of the rescaled Hessian are not all orthogonal to the gradient. The other cases are rarely seen. "hard-hard" means the Lagrange multiplier for the trust region constraint is minus the minimal eigenvalue of the rescaled Hessian; "hard-easy" means it isn't.
- accept: (if blather == TRUE) indicates which of the sequence of solutions of the trust region subproblem were accepted as the next iterate. (When not accepted the trust region radius is reduced, and the previous iterate is kept.)
- r: (if blather == TRUE) the sequence of trust region radii.
- rho: (if blather == TRUE) the sequence of ratios of actual over predicted decrease in the objective function in the trust region subproblem, where predicted means the predicted decrease in the two-term Taylor series model used in the subproblem.
- valpath: (if blather == TRUE) the sequence of objective function values at the iterates.
- valtry: (if blather == TRUE) the sequence of objective function values at the solutions of the trust region subproblem.
- preddiff: (if blather == TRUE) the sequence of predicted differences using the two-term Taylor-series model between the function values at the current iterate and at the solution of the trust region subproblem.
- stepnorm: (if blather == TRUE) the sequence of norms of steps, that is distance between current iterate and proposed new iterate found in the trust region subproblem.

`unique.parframe` *Extract those lines of a parameter frame with unique elements in the value column*

Description

Extract those lines of a parameter frame with unique elements in the value column

Usage

```
## S3 method for class 'parframe'
unique(x, incomparables = FALSE, tol = 1, ...)
```

Arguments

x	parameter frame
incomparables	not used. Argument exists for compatibility with S3 generic.
tol	tolerance to decide when values are assumed to be equal, see plotValues().
...	additional arguments being passed to plotValues(), e.g. for subsetting.

Value

A subset of the parameter frame x.

---

vcov *Get variance-covariance matrix from trust result*

Description

Get variance-covariance matrix from trust result

Usage

```
vcov(fit, parupper = NULL, parlower = NULL)
```

Arguments

fit	return from trust or selected fit from parlist
parupper	upper limit for parameter values. Parameters are considered fixed when exceeding these limits.
parlower	lower limit for parameter values. Parameters are considered fixed when exceeding these limits.

`wide2long` *Translate wide output format (e.g. from ode) into long format*

Description

Translate wide output format (e.g. from ode) into long format

Usage

```
wide2long(out, keep = 1, na.rm = FALSE)
```

Arguments

`out`	data.frame or matrix or list of matrices in wide format
`keep`	Index vector, the columns to keep
`na.rm`	Logical, if `TRUE`, missing values are removed in the long format.

Details

The function assumes that out[,1] represents a time-like vector whereas out[,-1] represents the values. Useful for plotting with ggplot. If a list is supplied, the names of the list are added as extra column names "condition"

Value

data.frame in long format, i.e. columns "time" (out[,1]), "name" (colnames(out[,-1])), "value" (out[,-1]) and, if out was a list, "condition" (names(out))

`wide2long.data.frame` *Translate wide output format (e.g. from ode) into long format*

Description

Translate wide output format (e.g. from ode) into long format

Usage

```
## S3 method for class 'data.frame'
wide2long(out, keep = 1, na.rm = FALSE)
```

Arguments

`out`	data.frame or matrix or list of matrices in wide format
`keep`	Index vector, the columns to keep
`na.rm`	Logical, if `TRUE`, missing values are removed in the long format.

Details

The function assumes that out[,1] represents a time-like vector whereas out[,-1] represents the values. Useful for plotting with ggplot. If a list is supplied, the names of the list are added as extra column names "condition"

Value

data.frame in long format, i.e. columns "time" (out[,1]), "name" (colnames(out[,-1])), "value" (out[,-1]) and, if out was a list, "condition" (names(out))

`wide2long.list` *Translate wide output format (e.g. from ode) into long format*

Description

Translate wide output format (e.g. from ode) into long format

Usage

```
## S3 method for class 'list'
wide2long(out, keep = 1, na.rm = FALSE)
```

Arguments

`out`	list of matrices in wide format
`keep`	Index vector, the columns to keep
`na.rm`	Logical, if `TRUE`, missing values are removed in the long format.

Details

The function assumes that out[,1] represents a time-like vector whereas out[,-1] represents the values. Useful for plotting with ggplot. If a list is supplied, the names of the list are added as extra column names "condition"

Value

data.frame in long format, i.e. columns "time" (out[,1]), "name" (colnames(out[,-1])), "value" (out[,-1]) and, if out was a list, "condition" (names(out))

wide2long.matrix *Translate wide output format (e.g. from ode) into long format*

Description

Translate wide output format (e.g. from ode) into long format

Usage

```
## S3 method for class 'matrix'
wide2long(out, keep = 1, na.rm = FALSE)
```

Arguments

out	data.frame or matrix or list of matrices in wide format
keep	Index vector, the columns to keep
na.rm	Logical, if TRUE, missing values are removed in the long format.

Details

The function assumes that out[,1] represents a time-like vector whereas out[,-1] represents the values. Useful for plotting with ggplot. If a list is supplied, the names of the list are added as extra column names "condition"

Value

data.frame in long format, i.e. columns "time" (out[,1]), "name" (colnames(out[,-1])), "value" (out[,-1]) and, if out was a list, "condition" (names(out))

write.eqnlist *Write equation list into a csv file*

Description

Write equation list into a csv file

Usage

```
write.eqnlist(eqnlist, ...)
```

Arguments

eqnlist	object of class eqnlist
...	Arguments going to write.table

Xd *Model prediction function from data.frame*

Description

Model prediction function from data.frame

Usage

```
Xd(data, condition = NULL)
```

Arguments

data	data.frame with columns "name", "time", and row names that are taken as parameter names. The data frame can contain a column "value" to initialize the parameters.
condition	either NULL (generic prediction for any condition) or a character, denoting the condition for which the function makes a prediction.

Value

Object of class prdfn, i.e. a function `x(times pars, deriv = TRUE, conditions = NULL)`, see also Xs. Attributes are "parameters", the parameter names (row names of the data frame), and possibly "pouter", a named numeric vector which is generated from `data$value`.

Examples

```
# Generate a data.frame and corresponding prediction function
timesD <- seq(0, 2*pi, 0.5)
mydata <- data.frame(name = "A", time = timesD, value = sin(timesD),
                     row.names = paste0("par", 1:length(timesD)))
x <- Xd(mydata)

# Evaluate the prediction function at different time points
times <- seq(0, 2*pi, 0.01)
pouter <- structure(mydata$value, names = rownames(mydata))
prediction <- x(times, pouter)
plot(prediction)
```

Xf *Model prediction function for ODE models without sensitivities.*

Description

Interface to get an ODE into a model function `x(times, pars, forcings, events)` returning ODE output. It is a reduced version of Xs, missing the sensitivities.

Usage

```
Xf(
  odemodel,
  forcings = NULL,
  events = NULL,
  condition = NULL,
  optionsOde = list(method = "lsoda"),
  fcontrol = NULL
)
```

Arguments

`odemodel`	Object of class odemodel.
`forcings`	see Xs
`events`	see Xs
`condition`	either NULL (generic prediction for any condition) or a character, denoting the condition for which the function makes a prediction.
`optionsOde`	list with arguments to be passed to odeC() for the ODE integration.
`fcontrol`	list with additional fine-tuning arguments for the forcing interpolation. See approxfun for possible arguments.

Details

Can be used to integrate additional quantities, e.g. fluxes, by adding them to `f`. All quantities that are not initialised by pars in `x(..., forcings, events)` are initialized with 0. For more details and the return value see Xs.

Xs *Model prediction function for ODE models.*

Description

Interface to combine an ODE and its sensitivity equations into one model function `x(times, pars, deriv = TRUE)` returning ODE output and sensitivities.

Usage

```
Xs(
  odemodel,
  forcings = NULL,
  events = NULL,
  names = NULL,
  condition = NULL,
  optionsOde = list(method = "lsoda"),
  optionsSens = list(method = "lsodes"),
  fcontrol = NULL
)
```

Arguments

odemodel	object of class odemodel
forcings	data.frame with columns name (factor), time (numeric) and value (numeric). The ODE forcings.
events	data.frame of events with columns "var" (character, the name of the state to be affected), "time" (numeric, time point), "value" (numeric, value), "method" (character, either "replace", "add" or "multiply"). See events. ATTENTION: Sensitivities for event states will only be correctly computed if defined within odemodel(). Specify events within Xs() only for forward simulation.
names	character vector with the states to be returned. If NULL, all states are returned.
condition	either NULL (generic prediction for any condition) or a character, denoting the condition for which the function makes a prediction.
optionsOde	list with arguments to be passed to odeC() for the ODE integration.
optionsSens	list with arguments to be passed to odeC() for integration of the extended system
fcontrol	list with additional fine-tuning arguments for the forcing interpolation. See approxfun for possible arguments.

Value

Object of class prdfn. If the function is called with parameters that result from a parameter transformation (see P), the Jacobian of the parameter transformation and the sensitivities of the ODE are multiplied according to the chain rule for differentiation. The result is saved in the attributed "deriv", i.e. in this case the attibutes "deriv" and "sensitivities" do not coincide.

Xt *Generate a prediction function that returns times*

Description

Function to deal with non-ODE models within the framework of dMod. See example.

Usage

```
Xt(condition = NULL)
```

Arguments

condition	either NULL (generic prediction for any condition) or a character, denoting the condition for which the function makes a prediction.

Value

Object of class prdfn.

Examples

```
x <- Xt()
g <- Y(c(y = "a*time^2+b"), f = NULL, parameters = c("a", "b"))

times <- seq(-1, 1, by = .05)
pars <- c(a = .1, b = 1)

plot((g*x)(times, pars))
```

Y *Observation functions.*

Description

Creates an object of type obsfn that evaluates an observation function and its derivatives based on the output of a model prediction function, see prdfn, as e.g. produced by Xs.

Usage

```
Y(
  g,
  f = NULL,
  states = NULL,
  parameters = NULL,
  condition = NULL,
  attach.input = TRUE,
  deriv = TRUE,
  compile = FALSE,
  modelname = NULL,
  verbose = FALSE
)
```

Arguments

g	Named character vector or equation vector defining the observation function
f	Named character of equations or object that can be converted to eqnvec or object of class fn. If f is provided, states and parameters are guessed from f.
states	character vector, alternative definition of "states", usually the names of f. If both, f and states are provided, the states argument overwrites the states derived from f.
parameters	character vector, alternative definition of the "parameters", usually the symbols contained in "g" and "f" except for states and the code word time. If both, f and parameters are provided, the parameters argument overwrites the parameters derived from f and g.
condition	either NULL (generic prediction for any condition) or a character, denoting the condition for which the function makes a prediction.
attach.input	logical, indiating whether the original input should be returned with the output.
deriv	logical, generate function to evaluate derivatives of observables. Necessary for parameter estimation.
compile	Logical, compile the function (see funC0)
modelname	Character, used if compile = TRUE, sets a fixed filename for the C file.
verbose	Print compiler output to R command line.

Details

For odemodels with forcings, it is best, to pass the prediction function x to the "f"-argument instead of the equations themselves. If an eqnvec is passed to "f" in this case, the forcings and states have to be specified manually via the "states"-argument.

Value

Object of class obsfn, i.e. a function y(..., deriv = TRUE, conditions = NULL) representing the evaluation of the observation function. Arguments out (model prediction) and pars (parameter values) shoudl be passed by the ... argument. If out has the attribute "sensitivities", the result of y(out, pars), will have an attributed "deriv" which reflecs the sensitivities of the observation with respect to the parameters. If pars is the result of a parameter transformation p(pars) (see P), the Jacobian of the parameter transformation and the sensitivities of the observation function are multiplied according to the chain rule for differentiation.

Examples

```
# Define a time grid on which to make a prediction by peace-wise linear function.
# Then define a (generic) prediction function based on thid grid.
times <- 0:5
grid <- data.frame(name = "A", time = times, row.names = paste0("p", times))
x <- Xd(grid)

# Define an observable and an observation function
observables <- eqnvec(Aobs = "s*A")
g <- Y(g = observables, f = NULL, states = "A", parameters = "s")
```

```
# Collect parameters and define an overarching parameter transformation
# for two "experimental condtions".
dynpars <- attr(x, "parameters")
obspars <- attr(g, "parameters")
innerpars <- c(dynpars, obspars)

trafo <- structure(innerpars, names = innerpars)
trafo_C1 <- replaceSymbols(innerpars, paste(innerpars, "C1", sep = "_"), trafo)
trafo_C2 <- replaceSymbols(innerpars, paste(innerpars, "C2", sep = "_"), trafo)

p <- NULL
p <- p + P(trafo = trafo_C1, condition = "C1")
p <- p + P(trafo = trafo_C2, condition = "C2")

# Collect outer (overarching) parameters and
# initialize with random values
outerpars <- attr(p, "parameters")
pars <- structure(runif(length(outerpars), 0, 1), names = outerpars)

# Predict internal/unobserved states
out1 <- (x*p)(times, pars)
plot(out1)

# Predict observed states in addition to unobserved
out2 <- (g*x*p)(times, pars)
plot(out2)
```

`%.*%` *Multiplication of objective functions with scalars*

Description

The `%.*%` operator allows to multiply objects of class objlist or objfn with a scalar.

Usage

```
x1 %.*% x2
```

Arguments

`x1`	object of class objfn or objlist.
`x2`	numeric of length one.

Value

An objective function or objlist object.

Index