Package ‘nlmixr2est’

May 28, 2024

Type Package

Title Nonlinear Mixed Effects Models in Population PK/PD, Estimation Routines

Version 2.2.2

Maintainer Matthew Fidler <matthew.fidler@gmail.com>

Description Fit and compare nonlinear mixed-effects models in differential equations with flexible dosing information commonly seen in pharmacokinetics and pharmacodynamics (Almquist, Leander, and Jirstrand 2015 <doi:10.1007/s10928-015-9409-1>). Differential equation solving is by compiled C code provided in the 'rxode2' package (Wang, Hallow, and James 2015 <doi:10.1002/psp4.12052>).

License GPL (>= 3)

URL https://github.com/nlmixr2/nlmixr2est,
https://nlmixr2.github.io/nlmixr2est/

Depends nlmixr2data, R (>= 4.0)

Imports backports, checkmate, cli, graphics, knitr, lbfgsb3c, lotri, magrittr, Matrix, methods, minqa, n1qn1 (>= 6.0.1-10), nlme, Rcpp, rex, rxode2 (>= 2.1.3), stats, symengine, utils

Suggests broom.mixed, crayon, data.table, devtools, digest, dplyr (>= 1.1.0), generics, nloptr, qs, sys, testthat, tibble, withr, xgxr, sfsmisc, rxode2parse (>= 2.0.19), rxode2random (>= 2.1.0), minpack.lm

LinkingTo BH, lbfgsb3c, Rcpp, RcppArmadillo (>= 0.11.2.3.1), rxode2 (>= 2.1.3), rxode2parse (>= 2.0.19), rxode2random (>= 2.1.0)

Biarch true

Config/testthat/edition 3

Encoding UTF-8

Language en-US

NeedsCompilation yes

RoxygenNote 7.3.1

LazyData true

Author Matthew Fidler [aut, cre] (<https://orcid.org/0000-0001-8538-6691>),
Yuan Xiong [aut],
Rik Schoemaker [aut] (<https://orcid.org/0000-0002-7538-3005>),
Justin Wilkins [aut] (<https://orcid.org/0000-0002-7099-9396>),
Wenping Wang [aut],
Robert Leary [ctb],
Mason McComb [ctb] (<https://orcid.org/0000-0001-9871-8616>),
Vipul Mann [aut],
Mirjam Trame [ctb],
Mahmoud Abdelwahab [ctb],
Teun Post [ctb],
Richard Hooijmaijers [aut],
Hadley Wickham [ctb],
Dirk Eddelbuettel [cph],
Johannes Pfeifer [ctb],
Robert B. Schnabel [ctb],
Elizabeth Eskow [ctb],
Emmanuelle Comets [ctb],
Audrey Lavenu [ctb],
Marc Lavielle [ctb],
David Ardia [cph],
Katharine Mullen [cph],
Ben Goodrich [ctb]

Repository CRAN

Date/Publication 2024-05-28 19:20:03 UTC

R topics documented:

`addCwres` *Add CWRES*

Description

This returns a new fit object with CWRES attached

Usage

```
addCwres(fit, focei = TRUE, updateObject = TRUE, envir = parent.frame(1))
```

Arguments

`fit`	nlmixr2 fit without WRES/CWRES
`focei`	Boolean indicating if the focei objective function is added. If not the foce objective function is added.
`updateObject`	Boolean indicating if the original fit object should be updated. By default this is true.
`envir`	Environment that should be checked for object to update. By default this is the global environment.

Value

fit with CWRES

Author(s)

Matthew L. Fidler

Examples

```
one.cmt <- function() {
  ini({
    ## You may label each parameter with a comment
    tka <- 0.45 # Log Ka
    tcl <- log(c(0, 2.7, 100)) # Log Cl
    ## This works with interactive models
    ## You may also label the preceding line with label("label text")
    tv <- 3.45; label("log V")
    ## the label("Label name") works with all models
    eta.ka ~ 0.6
    eta.cl ~ 0.3
    eta.v ~ 0.1
    add.sd <- 0.7
  })
  model({
    ka <- exp(tka + eta.ka)
    cl <- exp(tcl + eta.cl)
    v <- exp(tv + eta.v)
    linCmt() ~ add(add.sd)
  })
}

f <- try(nlmixr2(one.cmt, theo_sd, "saem"))
```

```
print(f)

# even though you may have forgotten to add the cwres, you can add it to the data.frame:

if (!inherits(f, "try-error")) {
  f <- try(addCwres(f))
  print(f)
}

# Note this also adds the FOCEi objective function
```

---

addNpde *NPDE calculation for nlmixr2*

Description

NPDE calculation for nlmixr2

Usage

```
addNpde(
  object,
  updateObject = TRUE,
  table = tableControl(),
  ...,
  envir = parent.frame(1)
)
```

Arguments

`object`	nlmixr2 fit object
`updateObject`	Boolean indicating if original object should be updated. By default this is TRUE.
`table`	'tableControl()' list of options
`...`	Other ignored parameters.
`envir`	Environment that should be checked for object to update. By default this is the global environment.

Value

New nlmixr2 fit object

Author(s)

Matthew L. Fidler

Examples

```
one.cmt <- function() {
  ini({
    ## You may label each parameter with a comment
    tka <- 0.45 # Log Ka
    tcl <- log(c(0, 2.7, 100)) # Log Cl
    ## This works with interactive models
    ## You may also label the preceding line with label("label text")
    tv <- 3.45; label("log V")
    ## the label("Label name") works with all models
    eta.ka ~ 0.6
    eta.cl ~ 0.3
    eta.v ~ 0.1
    add.sd <- 0.7
  })
  model({
    ka <- exp(tka + eta.ka)
    cl <- exp(tcl + eta.cl)
    v <- exp(tv + eta.v)
    linCmt() ~ add(add.sd)
  })
}

f <- nlmixr2(one.cmt, theo_sd, "saem")

# even though you may have forgotten to add the NPDE, you can add it to the data.frame:

f <- addNpde(f)
```

---

addTable *Add table information to nlmixr2 fit object without tables*

Description

Add table information to nlmixr2 fit object without tables

Usage

```
addTable(
  object,
  updateObject = FALSE,
  data = object$dataSav,
  thetaEtaParameters = object$foceiThetaEtaParameters,
  table = tableControl(),
```

```
  keep = NULL,
  drop = NULL,
  envir = parent.frame(1)
)
```

Arguments

`object`	nlmixr2 family of objects
`updateObject`	Update the object (default FALSE)
`data`	Saved data from
`thetaEtaParameters`	Internal theta/eta parameters
`table`	a ‘tableControl()‘ list of options
`keep`	Character Vector of items to keep
`drop`	Character Vector of items to drop or NULL
`envir`	Environment to search for updating

Value

Fit with table information attached

Author(s)

Matthew Fidler

Examples

```
one.cmt <- function() {
  ini({
    ## You may label each parameter with a comment
    tka <- 0.45 # Log Ka
    tcl <- log(c(0, 2.7, 100)) # Log Cl
    ## This works with interactive models
    ## You may also label the preceding line with label("label text")
    tv <- 3.45; label("log V")
    ## the label("Label name") works with all models
    eta.ka ~ 0.6
    eta.cl ~ 0.3
    eta.v ~ 0.1
    add.sd <- 0.7
  })
  model({
    ka <- exp(tka + eta.ka)
    cl <- exp(tcl + eta.cl)
    v <- exp(tv + eta.v)
    linCmt() ~ add(add.sd)
```

```
  })
}

# run without tables step
f <- nlmixr2(one.cmt, theo_sd, "saem", control=list(calcTables=FALSE))

print(f)

# Now add the tables

f <- addTable(f)

print(f)
```

assertNlmixrFit *Assert that this is a nlmixr2 fit object*

Description

Will error without nlmixr2 fit object

Usage

```
assertNlmixrFit(fit)
```

Arguments

fit	Fit object

Value

Nothing

Author(s)

Matthew L. Fidler

Examples

```
## Not run:

f <- 4
assertNlmixrFit(f) # throw error


## End(Not run)
```

`assertNlmixrFitData` *Assert that this is a nlmixr2 fit data object*

Description

Will error without nlmixr2 fit data object

Usage

```
assertNlmixrFitData(fit)
```

Arguments

`fit`	Fit object

Value

Nothing

Author(s)

Matthew L. Fidler

Examples

```
## Not run:

f <- 4
assertNlmixrFitData(f) # throw errors


## End(Not run)
```

`bobyqaControl` *Control for bobyqa estimation method in nlmixr2*

Description

Control for bobyqa estimation method in nlmixr2

Usage

```
bobyqaControl(
  npt = NULL,
  rhobeg = NULL,
  rhoend = NULL,
  iprint = 0L,
  maxfun = 100000L,
  returnBobyqa = FALSE,
  stickyRecalcN = 4,
  maxOdeRecalc = 5,
  odeRecalcFactor = 10^(0.5),
  useColor = crayon::has_color(),
  printNcol = floor((getOption("width") - 23)/12),
  print = 1L,
  normType = c("rescale2", "mean", "rescale", "std", "len", "constant"),
  scaleType = c("nlmixr2", "norm", "mult", "multAdd"),
  scaleCmax = 1e+05,
  scaleCmin = 1e-05,
  scaleC = NULL,
  scaleTo = 1,
  rxControl = NULL,
  optExpression = TRUE,
  sumProd = FALSE,
  literalFix = TRUE,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
  compress = TRUE,
  covMethod = c("r", ""),
  adjObf = TRUE,
  ci = 0.95,
  sigdig = 4,
  sigdigTable = NULL,
  ...
)
```

Arguments

`npt` The number of points used to approximate the objective function via a quadratic approximation. The value of npt must be in the interval [n+2,(n+1)(n+2)/2] where n is the number of parameters in ‘par‘. Choices that exceed 2*n+1 are not recommended. If not defined, it will be set to min(n * 2, n+2).

`rhobeg` ‘rhobeg‘ and ‘rhoend‘ must be set to the initial and final values of a trust region radius, so both must be positive with ‘0 < rhoend < rhobeg‘. Typically ‘rhobeg‘ should be about one tenth of the greatest expected change to a variable. If the user does not provide a value, this will be set to ‘min(0.95, 0.2 * max(abs(par)))‘. Note also that smallest difference ‘abs(upper-lower)‘ should be greater than or equal to ‘rhobeg*2‘. If this is not the case then ‘rhobeg‘ will be adjusted.

`rhoend` The smallest value of the trust region radius that is allowed. If not defined, then 1e-6 times the value set for ‘rhobeg‘ will be used.

`iprint` The value of ‘iprint‘ should be set to an integer value in ‘0, 1, 2, 3, ...‘, which controls the amount of printing. Specifically, there is no output if ‘iprint=0‘ and there is output only at the start and the return if ‘iprint=1‘. Otherwise, each new value of ‘rho‘ is printed, with the best vector of variables so far and the corresponding value of the objective function. Further, each new value of the objective function with its variables are output if ‘iprint=3‘. If ‘iprint > 3‘, the objective function value and corresponding variables are output every ‘iprint‘ evaluations. Default value is ‘0‘.

`maxfun` The maximum allowed number of function evaluations. If this is exceeded, the method will terminate.

`returnBobyqa` return the bobyqa output instead of the nlmixr2 fit

`stickyRecalcN` The number of bad ODE solves before reducing the atol/rtol for the rest of the problem.

`maxOdeRecalc` Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.

`odeRecalcFactor`
The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced

`useColor` Boolean indicating if focei can use ASCII color codes

`printNcol` Number of columns to printout before wrapping parameter estimates/gradient

`print` Integer representing when the outer step is printed. When this is 0 or do not print the iterations. 1 is print every function evaluation (default), 5 is print every 5 evaluations.

`normType` This is the type of parameter normalization/scaling used to get the scaled initial values for nlmixr2. These are used with `scaleType` of.

With the exception of `rescale2`, these come from Feature Scaling. The `rescale2` The rescaling is the same type described in the OptdesX software manual.

In general, all all scaling formula can be described by:

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

Where

The other data normalization approaches follow the following formula

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

- `rescale2` This scales all parameters from (-1 to 1). The relative differences between the parameters are preserved with this approach and the constants are:

  $$C_1$$

  = (max(all unscaled values)+min(all unscaled values))/2

  $$C_2$$

  = (max(all unscaled values) - min(all unscaled values))/2
- `rescale` or min-max normalization. This rescales all parameters from (0 to 1). As in the `rescale2` the relative differences are preserved. In this approach:

  $$C_1$$

  = min(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `mean` or mean normalization. This rescales to center the parameters around the mean but the parameters are from 0 to 1. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `std` or standardization. This standardizes by the mean and standard deviation. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = sd(all unscaled values)
- `len` or unit length scaling. This scales the parameters to the unit length. For this approach we use the Euclidean length, that is:

  $$C_1$$

  = 0

  $$C_2$$

  =

  $$\sqrt(v_1^2 + v_2^2 + \cdots + v_n^2)$$

- `constant` which does not perform data normalization. That is

$$C_1$$

= 0

$$C_2$$

= 1

`scaleType` The scaling scheme for nlmixr2. The supported types are:

- `nlmixr2` In this approach the scaling is performed by the following equation:

$$v_{scaled}$$

= (

$$v_{current} - v_{init}$$

)*scaleC[i] + scaleTo

The `scaleTo` parameter is specified by the `normType`, and the scales are specified by `scaleC`.

- `norm` This approach uses the simple scaling provided by the `normType` argument.
- `mult` This approach does not use the data normalization provided by `normType`, but rather uses multiplicative scaling to a constant provided by the `scaleTo` argument.

In this case:

$$v_{scaled}$$

=

$$v_{current}$$

/

$$v_{init}$$

*scaleTo

- `multAdd` This approach changes the scaling based on the parameter being specified. If a parameter is defined in an exponential block (ie exp(theta)), then it is scaled on a linearly, that is:

$$v_{scaled}$$

= (

$$v_{current} - v_{init}$$

) + scaleTo

Otherwise the parameter is scaled multiplicatively.

$$v_{scaled}$$

=

$$v_{current}$$

/

$$v_{init}$$

*scaleTo

scaleCmax
: Maximum value of the scaleC to prevent overflow.

scaleCmin
: Minimum value of the scaleC to prevent underflow.

scaleC
: The scaling constant used with `scaleType=nlmixr2`. When not specified, it is based on the type of parameter that is estimated. The idea is to keep the derivatives similar on a log scale to have similar gradient sizes. Hence parameters like log(exp(theta)) would have a scaling factor of 1 and log(theta) would have a scaling factor of ini_value (to scale by 1/value; ie d/dt(log(ini_value)) = 1/ini_value or scaleC=ini_value)

  - For parameters in an exponential (ie exp(theta)) or parameters specifying powers, boxCox or yeoJohnson transformations , this is 1.
  - For additive, proportional, lognormal error structures, these are given by 0.5*abs(initial_estimate)
  - Factorials are scaled by abs(1/digamma(initial_estimate+1))
  - parameters in a log scale (ie log(theta)) are transformed by log(abs(initial_estimate))*abs(initial_estim

  These parameter scaling coefficients are chose to try to keep similar slopes among parameters. That is they all follow the slopes approximately on a log-scale.

  While these are chosen in a logical manner, they may not always apply. You can specify each parameters scaling factor by this parameter if you wish.

scaleTo
: Scale the initial parameter estimate to this value. By default this is 1. When zero or below, no scaling is performed.

rxControl
: 'rxode2' ODE solving options during fitting, created with 'rxControl()'

optExpression
: Optimize the rxode2 expression to speed up calculation. By default this is turned on.

sumProd
: Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is `FALSE`.

literalFix
: boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is 'TRUE'.

addProp
: specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

  The combined1 error type can be described by the following equation:

  $$y = f + (a + b \times f^c) \times \varepsilon$$

  The combined2 error model can be described by the following equation:

  $$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

  Where:

  - y represents the observed value

- f represents the predicted value
- a is the additive standard deviation
- b is the proportional/power standard deviation
- c is the power exponent (in the proportional case c=1)

`calcTables` This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`

`compress` Should the object have compressed items

`covMethod` Method for calculating covariance. In this discussion, R is the Hessian matrix of the objective function. The S matrix is the sum of individual gradient cross-product (evaluated at the individual empirical Bayes estimates).

- "`r,s`" Uses the sandwich matrix to calculate the covariance, that is: `solve(R) %*% S %*% solve(R)`
- "`r`" Uses the Hessian matrix to calculate the covariance as `2 %*% solve(R)`
- "`s`" Uses the cross-product matrix to calculate the covariance as `4 %*% solve(S)`
- "" Does not calculate the covariance step.

`adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM's default objective function. By default this is `TRUE`

`ci` Confidence level for some tables. By default this is 0.95 or 95% confidence.

`sigdig` Optimization significant digits. This controls:

- The tolerance of the inner and outer optimization is `10^-sigdig`
- The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)
- The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`

`sigdigTable` Significant digits in the final output table. If not specified, then it matches the significant digits in the ‘sigdig‘ optimization algorithm. If ‘sigdig‘ is NULL, use 3.

`...` Ignored parameters

Value

bobqya control structure

Author(s)

Matthew L. Fidler

Examples

```
# A logit regression example with emax model

dsn <- data.frame(i=1:1000)
dsn$time <- exp(rnorm(1000))
```

```
dsn$DV=rbinom(1000,1,exp(-1+dsn$time)/(1+exp(-1+dsn$time)))

mod <- function() {
 ini({
   E0 <- 0.5
   Em <- 0.5
   E50 <- 2
   g <- fix(2)
 })
 model({
   v <- E0+Em*time^g/(E50^g+time^g)
   ll(bin) ~ DV * v - log(1 + exp(v))
 })
}

fit2 <- nlmixr(mod, dsn, est="bobyqa")

print(fit2)

# you can also get the nlm output with

fit2$bobyqa

# The nlm control has been modified slightly to include
# extra components and name the parameters
```

---

boxCox *Cox Box, Yeo Johnson and inverse transformation*

Description

Cox Box, Yeo Johnson and inverse transformation

Usage

```
boxCox(x, lambda = 1)

iBoxCox(x, lambda = 1)

yeoJohnson(x, lambda = 1)

iYeoJohnson(x, lambda = 1)
```

Arguments

x	data to transform
lambda	Cox-box lambda parameter

Value

Cox-Box Transformed Data

Author(s)

Matthew L. Fidler

Examples

```
boxCox(1:3,1) ## Normal
iBoxCox(boxCox(1:3,1))

boxCox(1:3,0) ## Log-Normal
iBoxCox(boxCox(1:3,0),0)

boxCox(1:3,0.5) ## lambda=0.5
iBoxCox(boxCox(1:3,0.5),0.5)

yeoJohnson(seq(-3,3),1) ## Normal
iYeoJohnson(yeoJohnson(seq(-3,3),1))

yeoJohnson(seq(-3,3),0)
iYeoJohnson(yeoJohnson(seq(-3,3),0),0)
```

`cholSE` *Generalized Cholesky Matrix Decomposition*

Description

Performs a (modified) Cholesky factorization of the form

Usage

```
cholSE(matrix, tol = (.Machine$double.eps)^(1/3))
```

Arguments

`matrix`	Matrix to be Factorized.
`tol`	Tolerance; Algorithm suggests (.Machine$double.eps) ^ (1 / 3), default

Details

t(P) %*% A %*% P + E = t(R) %*% R

As detailed in Schnabel/Eskow (1990)

Value

Generalized Cholesky decomposed matrix.

Note

This version does not pivot or return the E matrix

Author(s)

Matthew L. Fidler (translation), Johannes Pfeifer, Robert B. Schnabel and Elizabeth Eskow

References

matlab source: http://www.dynare.org/dynare-matlab-m2html/matlab/chol_SE.html; Slightly different return values

Robert B. Schnabel and Elizabeth Eskow. 1990. "A New Modified Cholesky Factorization," SIAM Journal of Scientific Statistical Computing, 11, 6: 1136-58.

Elizabeth Eskow and Robert B. Schnabel 1991. "Algorithm 695 - Software for a New Modified Cholesky Factorization," ACM Transactions on Mathematical Software, Vol 17, No 3: 306-312

`foceiControl` *Control Options for FOCEi*

Description

Control Options for FOCEi

Usage

```
foceiControl(
  sigdig = 3,
  ...,
  epsilon = NULL,
  maxInnerIterations = 1000,
  maxOuterIterations = 5000,
  n1qn1nsim = NULL,
  print = 1L,
  printNcol = floor((getOption("width") - 23)/12),
  scaleTo = 1,
  scaleObjective = 0,
  normType = c("rescale2", "mean", "rescale", "std", "len", "constant"),
  scaleType = c("nlmixr2", "norm", "mult", "multAdd"),
  scaleCmax = 1e+05,
  scaleCmin = 1e-05,
  scaleC = NULL,
  scaleC0 = 1e+05,
  derivEps = rep(20 * sqrt(.Machine$double.eps), 2),
  derivMethod = c("switch", "forward", "central"),
  derivSwitchTol = NULL,
  covDerivMethod = c("central", "forward"),
```

```
  covMethod = c("r,s", "r", "s", ""),
  hessEps = (.Machine$double.eps)^(1/3),
  hessEpsLlik = (.Machine$double.eps)^(1/3),
  optimHessType = c("central", "forward"),
  optimHessCovType = c("central", "forward"),
  eventType = c("central", "forward"),
  centralDerivEps = rep(20 * sqrt(.Machine$double.eps), 2),
  lbfgsLmm = 7L,
  lbfgsPgtol = 0,
  lbfgsFactr = NULL,
  eigen = TRUE,
  addPosthoc = TRUE,
  diagXform = c("sqrt", "log", "identity"),
  sumProd = FALSE,
  optExpression = TRUE,
  literalFix = TRUE,
  ci = 0.95,
  useColor = crayon::has_color(),
  boundTol = NULL,
  calcTables = TRUE,
  noAbort = TRUE,
  interaction = TRUE,
  cholSEtol = (.Machine$double.eps)^(1/3),
  cholAccept = 0.001,
  resetEtaP = 0.15,
  resetThetaP = 0.05,
  resetThetaFinalP = 0.15,
  diagOmegaBoundUpper = 5,
  diagOmegaBoundLower = 100,
  cholSEOpt = FALSE,
  cholSECov = FALSE,
  fo = FALSE,
  covTryHarder = FALSE,
 outerOpt = c("nlminb", "bobyqa", "lbfgsb3c", "L-BFGS-B", "mma", "lbfgsbLG", "slsqp",
    "Rvmmin"),
  innerOpt = c("n1qn1", "BFGS"),
  rhobeg = 0.2,
  rhoend = NULL,
  npt = NULL,
  rel.tol = NULL,
  x.tol = NULL,
  eval.max = 4000,
  iter.max = 2000,
  abstol = NULL,
  reltol = NULL,
  resetHessianAndEta = FALSE,
  stateTrim = Inf,
  shi21maxOuter = 0L,
```

```
    shi21maxInner = 20L,
    shi21maxInnerCov = 20L,
    shi21maxFD = 20L,
    gillK = 10L,
    gillStep = 4,
    gillFtol = 0,
    gillRtol = sqrt(.Machine$double.eps),
    gillKcov = 10L,
    gillKcovLlik = 10L,
    gillStepCovLlik = 4.5,
    gillStepCov = 2,
    gillFtolCov = 0,
    gillFtolCovLlik = 0,
    rmatNorm = TRUE,
    rmatNormLlik = TRUE,
    smatNorm = TRUE,
    smatNormLlik = TRUE,
    covGillF = TRUE,
    optGillF = TRUE,
    covSmall = 1e-05,
    adjLik = TRUE,
    gradTrim = Inf,
    maxOdeRecalc = 5,
    odeRecalcFactor = 10^(0.5),
    gradCalcCentralSmall = 1e-04,
    gradCalcCentralLarge = 10000,
    etaNudge = qnorm(1 - 0.05/2)/sqrt(3),
    etaNudge2 = qnorm(1 - 0.05/2) * sqrt(3/5),
    nRetries = 3,
    seed = 42,
    resetThetaCheckPer = 0.1,
    etaMat = NULL,
    repeatGillMax = 1,
    stickyRecalcN = 4,
    gradProgressOfvTime = 10,
    addProp = c("combined2", "combined1"),
    badSolveObjfAdj = 100,
    compress = TRUE,
    rxControl = NULL,
    sigdigTable = NULL,
    fallbackFD = FALSE,
    smatPer = 0.6,
    sdLowerFact = 0.001
  )
```

Arguments

`sigdig` Optimization significant digits. This controls:

- The tolerance of the inner and outer optimization is `10^-sigdig`
- The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)
- The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`

`...` Ignored parameters

`epsilon` Precision of estimate for n1qn1 optimization.

`maxInnerIterations`
Number of iterations for n1qn1 optimization.

`maxOuterIterations`
Maximum number of L-BFGS-B optimization for outer problem.

`n1qn1nsim` Number of function evaluations for n1qn1 optimization.

`print` Integer representing when the outer step is printed. When this is 0 or do not print the iterations. 1 is print every function evaluation (default), 5 is print every 5 evaluations.

`printNcol` Number of columns to printout before wrapping parameter estimates/gradient

`scaleTo` Scale the initial parameter estimate to this value. By default this is 1. When zero or below, no scaling is performed.

`scaleObjective` Scale the initial objective function to this value. By default this is 0 (meaning do not scale)

`normType` This is the type of parameter normalization/scaling used to get the scaled initial values for nlmixr2. These are used with `scaleType` of.

With the exception of `rescale2`, these come from Feature Scaling. The `rescale2` The rescaling is the same type described in the OptdesX software manual.

In general, all all scaling formula can be described by:

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

Where

The other data normalization approaches follow the following formula

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

- `rescale2` This scales all parameters from (-1 to 1). The relative differences between the parameters are preserved with this approach and the constants are:

  $$C_1$$

  = (max(all unscaled values)+min(all unscaled values))/2

  $$C_2$$

  = (max(all unscaled values) - min(all unscaled values))/2
- `rescale` or min-max normalization. This rescales all parameters from (0 to 1). As in the `rescale2` the relative differences are preserved. In this approach:

  $$C_1$$

  = min(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `mean` or mean normalization. This rescales to center the parameters around the mean but the parameters are from 0 to 1. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `std` or standardization. This standardizes by the mean and standard deviation. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = sd(all unscaled values)
- `len` or unit length scaling. This scales the parameters to the unit length. For this approach we use the Euclidean length, that is:

  $$C_1$$

  = 0

  $$C_2$$

  =

  $$\sqrt{(}v_1^2 + v_2^2 + \cdots + v_n^2)$$

- `constant` which does not perform data normalization. That is

$$C_1$$

= 0

$$C_2$$

= 1

`scaleType` The scaling scheme for nlmixr2. The supported types are:

- `nlmixr2` In this approach the scaling is performed by the following equation:

$$v_{scaled}$$

= (

$$v_{current} - v_{init}$$

)*scaleC[i] + scaleTo
The `scaleTo` parameter is specified by the `normType`, and the scales are specified by `scaleC`.

- `norm` This approach uses the simple scaling provided by the `normType` argument.
- `mult` This approach does not use the data normalization provided by `normType`, but rather uses multiplicative scaling to a constant provided by the `scaleTo` argument.
In this case:

$$v_{scaled}$$

=

$$v_{current}$$

/

$$v_{init}$$

*scaleTo

- `multAdd` This approach changes the scaling based on the parameter being specified. If a parameter is defined in an exponential block (ie exp(theta)), then it is scaled on a linearly, that is:

$$v_{scaled}$$

= (

$$v_{current} - v_{init}$$

) + scaleTo
Otherwise the parameter is scaled multiplicatively.

$$v_{scaled}$$

=

$$v_{current}$$

/

v_{init}

*scaleTo

scaleCmax
: Maximum value of the scaleC to prevent overflow.

scaleCmin
: Minimum value of the scaleC to prevent underflow.

scaleC
: The scaling constant used with `scaleType=nlmixr2`. When not specified, it is based on the type of parameter that is estimated. The idea is to keep the derivatives similar on a log scale to have similar gradient sizes. Hence parameters like log(exp(theta)) would have a scaling factor of 1 and log(theta) would have a scaling factor of ini_value (to scale by 1/value; ie d/dt(log(ini_value)) = 1/ini_value or scaleC=ini_value)

  - For parameters in an exponential (ie exp(theta)) or parameters specifying powers, boxCox or yeoJohnson transformations , this is 1.
  - For additive, proportional, lognormal error structures, these are given by 0.5*abs(initial_estimate)
  - Factorials are scaled by abs(1/digamma(initial_estimate+1))
  - parameters in a log scale (ie log(theta)) are transformed by log(abs(initial_estimate))*abs(initial_estin

  These parameter scaling coefficients are chose to try to keep similar slopes among parameters. That is they all follow the slopes approximately on a log-scale.

  While these are chosen in a logical manner, they may not always apply. You can specify each parameters scaling factor by this parameter if you wish.

scaleC0
: Number to adjust the scaling factor by if the initial gradient is zero.

derivEps
: Forward difference tolerances, which is a vector of relative difference and absolute difference. The central/forward difference step size h is calculated as:

  `h = abs(x)*derivEps[1] + derivEps[2]`

derivMethod
: indicates the method for calculating derivatives of the outer problem. Currently supports "switch", "central" and "forward" difference methods. Switch starts with forward differences. This will switch to central differences when abs(delta(OFV)) <= derivSwitchTol and switch back to forward differences when abs(delta(OFV)) > derivSwitchTol.

derivSwitchTol
: The tolerance to switch forward to central differences.

covDerivMethod
: indicates the method for calculating the derivatives while calculating the covariance components (Hessian and S).

covMethod
: Method for calculating covariance. In this discussion, R is the Hessian matrix of the objective function. The S matrix is the sum of individual gradient cross-product (evaluated at the individual empirical Bayes estimates).

  - "`r,s`" Uses the sandwich matrix to calculate the covariance, that is: `solve(R) %*% S %*% solve(R)`
  - "`r`" Uses the Hessian matrix to calculate the covariance as `2 %*% solve(R)`
  - "`s`" Uses the cross-product matrix to calculate the covariance as `4 %*% solve(S)`
  - "" Does not calculate the covariance step.

hessEps
: is a double value representing the epsilon for the Hessian calculation. This is used for the R matrix calculation.

hessEpsLlik
: is a double value representing the epsilon for the Hessian calculation when doing focei generalized log-likelihood estimation. This is used for the R matrix calculation.

optimHessType
: The hessian type for when calculating the individual hessian by numeric differences (in generalized log-likelihood estimation). The options are "central", and "forward". The central differences is what R's 'optimHess()' uses and is the default for this method. (Though the "forward" is faster and still reasonable for most cases). The Shi21 cannot be changed for the Gill83 algorithm with the optimHess in a generalized likelihood problem.

optimHessCovType
: The hessian type for when calculating the individual hessian by numeric differences (in generalized log-likelihood estimation). The options are "central", and "forward". The central differences is what R's 'optimHess()' uses. While this takes longer in optimization, it is more accurate, so for calculating the covariance and final likelihood, the central differences are used. This also uses the modified Shi21 method

eventType
: Event gradient type for dosing events; Can be "central" or "forward"

centralDerivEps
: Central difference tolerances. This is a numeric vector of relative difference and absolute difference. The central/forward difference step size h is calculated as:

  `h = abs(x)*derivEps[1] + derivEps[2]`

lbfgsLmm
: An integer giving the number of BFGS updates retained in the "L-BFGS-B" method, It defaults to 7.

lbfgsPgtol
: is a double precision variable.

  On entry pgtol >= 0 is specified by the user. The iteration will stop when:

  `max(\| proj g_i \| i = 1, ..., n) <= lbfgsPgtol`

  where pg_i is the ith component of the projected gradient.

  On exit pgtol is unchanged. This defaults to zero, when the check is suppressed.

lbfgsFactr
: Controls the convergence of the "L-BFGS-B" method. Convergence occurs when the reduction in the objective is within this factor of the machine tolerance. Default is 1e10, which gives a tolerance of about `2e-6`, approximately 4 sigdigs. You can check your exact tolerance by multiplying this value by `.Machine$double.eps`

eigen
: A boolean indicating if eigenvectors are calculated to include a condition number calculation.

addPosthoc
: Boolean indicating if posthoc parameters are added to the table output.

diagXform
: This is the transformation used on the diagonal of the `chol(solve(omega))`. This matrix and values are the parameters estimated in FOCEi. The possibilities are:

  - `sqrt` Estimates the sqrt of the diagonal elements of `chol(solve(omega))`. This is the default method.
  - `log` Estimates the log of the diagonal elements of `chol(solve(omega))`

- `identity` Estimates the diagonal elements without any transformations

`sumProd`
: Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is `FALSE`.

`optExpression`
: Optimize the rxode2 expression to speed up calculation. By default this is turned on.

`literalFix`
: boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is ‘TRUE‘.

`ci`
: Confidence level for some tables. By default this is 0.95 or 95% confidence.

`useColor`
: Boolean indicating if focei can use ASCII color codes

`boundTol`
: Tolerance for boundary issues.

`calcTables`
: This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`

`noAbort`
: Boolean to indicate if you should abort the FOCEi evaluation if it runs into troubles. (default TRUE)

`interaction`
: Boolean indicate FOCEi should be used (TRUE) instead of FOCE (FALSE)

`cholSEtol`
: tolerance for Generalized Cholesky Decomposition. Defaults to suggested (.Machine$double.eps)^(1/3)

`cholAccept`
: Tolerance to accept a Generalized Cholesky Decomposition for a R or S matrix.

`resetEtaP`
: represents the p-value for reseting the individual ETA to 0 during optimization (instead of the saved value). The two test statistics used in the z-test are either chol(omega^-1) %*% eta or eta/sd(allEtas). A p-value of 0 indicates the ETAs never reset. A p-value of 1 indicates the ETAs always reset.

`resetThetaP`
: represents the p-value for reseting the population mu-referenced THETA parameters based on ETA drift during optimization, and resetting the optimization. A p-value of 0 indicates the THETAs never reset. A p-value of 1 indicates the THETAs always reset and is not allowed. The theta reset is checked at the beginning and when nearing a local minima. The percent change in objective function where a theta reset check is initiated is controlled in `resetThetaCheckPer`.

`resetThetaFinalP`
: represents the p-value for reseting the population mu-referenced THETA parameters based on ETA drift during optimization, and resetting the optimization one final time.

`diagOmegaBoundUpper`
: This represents the upper bound of the diagonal omega matrix. The upper bound is given by diag(omega)*diagOmegaBoundUpper. If `diagOmegaBoundUpper` is 1, there is no upper bound on Omega.

`diagOmegaBoundLower`
: This represents the lower bound of the diagonal omega matrix. The lower bound is given by diag(omega)/diagOmegaBoundUpper. If `diagOmegaBoundLower` is 1, there is no lower bound on Omega.

`cholSEOpt`
: Boolean indicating if the generalized Cholesky should be used while optimizing.

`cholSECov`
: Boolean indicating if the generalized Cholesky should be used while calculating the Covariance Matrix.

`fo` is a boolean indicating if this is a FO approximation routine.

`covTryHarder` If the R matrix is non-positive definite and cannot be corrected to be non-positive definite try estimating the Hessian on the unscaled parameter space.

`outerOpt` optimization method for the outer problem

`innerOpt` optimization method for the inner problem (not implemented yet.)

`rhobeg` Beginning change in parameters for bobyqa algorithm (trust region). By default this is 0.2 or 20 parameters when the parameters are scaled to 1. rhobeg and rhoend must be set to the initial and final values of a trust region radius, so both must be positive with 0 < rhoend < rhobeg. Typically rhobeg should be about one tenth of the greatest expected change to a variable. Note also that smallest difference abs(upper-lower) should be greater than or equal to rhobeg*2. If this is not the case then rhobeg will be adjusted. (bobyqa)

`rhoend` The smallest value of the trust region radius that is allowed. If not defined, then 10^(-sigdig-1) will be used. (bobyqa)

`npt` The number of points used to approximate the objective function via a quadratic approximation for bobyqa. The value of npt must be in the interval [n+2,(n+1)(n+2)/2] where n is the number of parameters in par. Choices that exceed 2*n+1 are not recommended. If not defined, it will be set to 2*n + 1. (bobyqa)

`rel.tol` Relative tolerance before nlminb stops (nlmimb).

`x.tol` X tolerance for nlmixr2 optimizer

`eval.max` Number of maximum evaluations of the objective function (nlmimb)

`iter.max` Maximum number of iterations allowed (nlmimb)

`abstol` Absolute tolerance for nlmixr2 optimizer (BFGS)

`reltol` tolerance for nlmixr2 (BFGS)

`resetHessianAndEta`
is a boolean representing if the individual Hessian is reset when ETAs are reset using the option `resetEtaP`.

`stateTrim` Trim state amounts/concentrations to this value.

`shi21maxOuter` The maximum number of steps for the optimization of the forward-difference step size. When not zero, use this instead of Gill differences.

`shi21maxInner` The maximum number of steps for the optimization of the individual Hessian matrices in the generalized likelihood problem. When 0, un-optimized finite differences are used.

`shi21maxInnerCov`
The maximum number of steps for the optimization of the individual Hessian matrices in the generalized likelihood problem for the covariance step. When 0, un-optimized finite differences are used.

`shi21maxFD` The maximum number of steps for the optimization of the forward difference step size when using dosing events (lag time, modeled duration/rate and bioavailability)

`gillK` The total number of possible steps to determine the optimal forward/central difference step size per parameter (by the Gill 1983 method). If 0, no optimal step size is determined. Otherwise this is the optimal step size determined.

gillStep
: When looking for the optimal forward difference step size, this is This is the step size to increase the initial estimate by. So each iteration the new step size = (prior step size)*gillStep

gillFtol
: The gillFtol is the gradient error tolerance that is acceptable before issuing a warning/error about the gradient estimates.

gillRtol
: The relative tolerance used for Gill 1983 determination of optimal step size.

gillKcov
: The total number of possible steps to determine the optimal forward/central difference step size per parameter (by the Gill 1983 method) during the covariance step. If 0, no optimal step size is determined. Otherwise this is the optimal step size determined.

gillKcovLlik
: The total number of possible steps to determine the optimal forward/central difference step per parameter when using the generalized focei log-likelihood method (by the Gill 1986 method). If 0, no optimal step size is determined. Otherwise this is the optimal step size is determined

gillStepCovLlik
: Same as above but during generalized focei log-likelihood

gillStepCov
: When looking for the optimal forward difference step size, this is This is the step size to increase the initial estimate by. So each iteration during the covariance step is equal to the new step size = (prior step size)*gillStepCov

gillFtolCov
: The gillFtol is the gradient error tolerance that is acceptable before issuing a warning/error about the gradient estimates during the covariance step.

gillFtolCovLlik
: Same as above but applied during generalized log-likelihood estimation.

rmatNorm
: A parameter to normalize gradient step size by the parameter value during the calculation of the R matrix

rmatNormLlik
: A parameter to normalize gradient step size by the parameter value during the calculation of the R matrix if you are using generalized log-likelihood Hessian matrix.

smatNorm
: A parameter to normalize gradient step size by the parameter value during the calculation of the S matrix

smatNormLlik
: A parameter to normalize gradient step size by the parameter value during the calculation of the S matrix if you are using the generalized log-likelihood.

covGillF
: Use the Gill calculated optimal Forward difference step size for the instead of the central difference step size during the central difference gradient calculation.

optGillF
: Use the Gill calculated optimal Forward difference step size for the instead of the central difference step size during the central differences for optimization.

covSmall
: The covSmall is the small number to compare covariance numbers before rejecting an estimate of the covariance as the final estimate (when comparing sandwich vs R/S matrix estimates of the covariance). This number controls how small the variance is before the covariance matrix is rejected.

adjLik
: In nlmixr2, the objective function matches NONMEM's objective function, which removes a 2*pi constant from the likelihood calculation. If this is TRUE, the likelihood function is adjusted by this 2*pi factor. When adjusted this number

more closely matches the likelihood approximations of nlme, and SAS approximations. Regardless of if this is turned on or off the objective function matches NONMEM's objective function.

`gradTrim` The parameter to adjust the gradient to if the |gradient| is very large.

`maxOdeRecalc` Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.

`odeRecalcFactor` The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced

`gradCalcCentralSmall` A small number that represents the value where |grad| < gradCalcCentralSmall where forward differences switch to central differences.

`gradCalcCentralLarge` A large number that represents the value where |grad| > gradCalcCentralLarge where forward differences switch to central differences.

`etaNudge` By default initial ETA estimates start at zero; Sometimes this doesn't optimize appropriately. If this value is non-zero, when the `n1qn1` optimization didn't perform appropriately, reset the Hessian, and nudge the ETA up by this value; If the ETA still doesn't move, nudge the ETA down by this value. By default this value is qnorm(1-0.05/2)*1/sqrt(3), the first of the Gauss Quadrature numbers times by the 0.95% normal region. If this is not successful try the second eta nudge number (below). If +-etaNudge2 is not successful, then assign to zero and do not optimize any longer

`etaNudge2` This is the second eta nudge. By default it is qnorm(1-0.05/2)*sqrt(3/5), which is the n=3 quadrature point (excluding zero) times by the 0.95% normal region

`nRetries` If FOCEi doesn't fit with the current parameter estimates, randomly sample new parameter estimates and restart the problem. This is similar to 'PsN' resampling.

`seed` an object specifying if and how the random number generator should be initialized

`resetThetaCheckPer` represents objective function % percentage below which resetThetaP is checked.

`etaMat` Eta matrix for initial estimates or final estimates of the ETAs.

`repeatGillMax` If the tolerances were reduced when calculating the initial Gill differences, the Gill difference is repeated up to a maximum number of times defined by this parameter.

`stickyRecalcN` The number of bad ODE solves before reducing the atol/rtol for the rest of the problem.

`gradProgressOfvTime` This is the time for a single objective function evaluation (in seconds) to start progress bars on gradient evaluations

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

Where:

- y represents the observed value
- f represents the predicted value
- a is the additive standard deviation
- b is the proportional/power standard deviation
- c is the power exponent (in the proportional case c=1)

`badSolveObjfAdj`
: The objective function adjustment when the ODE system cannot be solved. It is based on each individual bad solve.

`compress`
: Should the object have compressed items

`rxControl`
: ‘rxode2‘ ODE solving options during fitting, created with ‘rxControl()‘

`sigdigTable`
: Significant digits in the final output table. If not specified, then it matches the significant digits in the ‘sigdig‘ optimization algorithm. If ‘sigdig‘ is NULL, use 3.

`fallbackFD`
: Fallback to the finite differences if the sensitivity equations do not solve.

`smatPer`
: A percentage representing the number of failed parameter gradients for each individual (which are replaced with the overall gradient for the parameter) out of the total number of gradients parameters (ie ‘ntheta*nsub‘) before the S matrix is considered to be a bad matrix.

`sdLowerFact`
: A factor for multiplying the estimate by when the lower estimate is zero and the error is known to represent a standard deviation of a parameter (like add.sd, prop.sd, pow.sd, lnorm.sd, etc). When zero, no factor is applied. If your initial estimate is 0.15 and your lower bound is zero, then the lower bound would be assumed to be 0.00015.

Details

Note this uses the R’s L-BFGS-B in `optim` for the outer problem and the BFGS `n1qn1` with that allows restoring the prior individual Hessian (for faster optimization speed).

However the inner problem is not scaled. Since most eta estimates start near zero, scaling for these parameters do not make sense.

This process of scaling can fix some ill conditioning for the unscaled problem. The covariance step is performed on the unscaled problem, so the condition number of that matrix may not be reflective of the scaled problem’s condition-number.

Value

The control object that changes the options for the FOCEi family of estimation methods

Author(s)

Matthew L. Fidler

References

Gill, P.E., Murray, W., Saunders, M.A., & Wright, M.H. (1983). Computing Forward-Difference Intervals for Numerical Optimization. Siam Journal on Scientific and Statistical Computing, 4, 310-321.

Shi, H.M., Xie, Y., Xuan, M.Q., & Nocedal, J. (2021). Adaptive Finite-Difference Interval Estimation for Noisy Derivative-Free Optimization.

See Also

`optim`

`n1qn1`

`rxSolve`

Other Estimation control: `nlmixr2NlmeControl()`, `saemControl()`

`getValidNlmixrCtl.bobyqa`

Get valid nlmixr control object

Description

Get valid nlmixr control object

Usage

```
## S3 method for class 'bobyqa'
getValidNlmixrCtl(control)

## S3 method for class 'lbfgsb3c'
getValidNlmixrCtl(control)

## S3 method for class 'n1qn1'
getValidNlmixrCtl(control)

## S3 method for class 'newuoa'
getValidNlmixrCtl(control)

## S3 method for class 'nlm'
getValidNlmixrCtl(control)

## S3 method for class 'nlminb'
getValidNlmixrCtl(control)

## S3 method for class 'nls'
getValidNlmixrCtl(control)
```

```
## S3 method for class 'optim'
getValidNlmixrCtl(control)

getValidNlmixrControl(control, est)

getValidNlmixrCtl(control)

## S3 method for class 'focei'
getValidNlmixrCtl(control)

## S3 method for class 'foce'
getValidNlmixrCtl(control)

## S3 method for class 'fo'
getValidNlmixrCtl(control)

## S3 method for class 'foi'
getValidNlmixrCtl(control)

## S3 method for class 'posthoc'
getValidNlmixrCtl(control)

## S3 method for class 'foce'
getValidNlmixrCtl(control)

## S3 method for class 'nlme'
getValidNlmixrCtl(control)

## S3 method for class 'saem'
getValidNlmixrCtl(control)

## S3 method for class 'rxSolve'
getValidNlmixrCtl(control)

## S3 method for class 'simulate'
getValidNlmixrCtl(control)

## S3 method for class 'simulation'
getValidNlmixrCtl(control)

## S3 method for class 'predict'
getValidNlmixrCtl(control)

## S3 method for class 'tableControl'
getValidNlmixrCtl(control)

## Default S3 method:
getValidNlmixrCtl(control)
```

```
## S3 method for class 'uobyqa'
getValidNlmixrCtl(control)
```

Arguments

control	nlmixr control object
est	Estimation routine

Details

This is based on running the S3 method ‘getValidNlmixrCtl()‘ the ‘control‘ object is put into a list and the class of this new list is ‘c(est, "getValidNlmixrControl")‘

Value

Valid control object based on estimation method run.

---

lbfgsb3cControl *Control for lbfgsb3c estimation method in nlmixr2*

Description

Control for lbfgsb3c estimation method in nlmixr2

Usage

```
lbfgsb3cControl(
  trace = 0,
  factr = 1e+07,
  pgtol = 0,
  abstol = 0,
  reltol = 0,
  lmm = 5L,
  maxit = 10000L,
  returnLbfgsb3c = FALSE,
  stickyRecalcN = 4,
  maxOdeRecalc = 5,
  odeRecalcFactor = 10^(0.5),
  useColor = crayon::has_color(),
  printNcol = floor((getOption("width") - 23)/12),
  print = 1L,
  normType = c("rescale2", "mean", "rescale", "std", "len", "constant"),
  scaleType = c("nlmixr2", "norm", "mult", "multAdd"),
  scaleCmax = 1e+05,
  scaleCmin = 1e-05,
  scaleC = NULL,
```

```
  scaleTo = 1,
  gradTo = 1,
  rxControl = NULL,
  optExpression = TRUE,
  sumProd = FALSE,
  literalFix = TRUE,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
  compress = TRUE,
  covMethod = c("r", ""),
  adjObf = TRUE,
  ci = 0.95,
  sigdig = 4,
  sigdigTable = NULL,
  ...
)
```

Arguments

trace	If positive, tracing information on the progress of the optimization is produced. Higher values may produce more tracing information: for method "L-BFGS-B" there are six levels of tracing. (To understand exactly what these do see the source code: higher levels give more detail.)
factr	controls the convergence of the "L-BFGS-B" method. Convergence occurs when the reduction in the objective is within this factor of the machine tolerance. Default is 1e7, that is a tolerance of about 1e-8.
pgtol	helps control the convergence of the "L-BFGS-B" method. It is a tolerance on the projected gradient in the current search direction. This defaults to zero, when the check is suppressed.
abstol	helps control the convergence of the "L-BFGS-B" method. It is an absolute tolerance difference in x values. This defaults to zero, when the check is suppressed.
reltol	helps control the convergence of the "L-BFGS-B" method. It is an relative tolerance difference in x values. This defaults to zero, when the check is suppressed.
lmm	is an integer giving the number of BFGS updates retained in the "L-BFGS-B" method, It defaults to 5.
maxit	maximum number of iterations.
returnLbfgsb3c	return the lbfgsb3c output instead of the nlmixr2 fit
stickyRecalcN	The number of bad ODE solves before reducing the atol/rtol for the rest of the problem.
maxOdeRecalc	Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.
odeRecalcFactor	The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced
useColor	Boolean indicating if focei can use ASCII color codes

`printNcol` Number of columns to printout before wrapping parameter estimates/gradient

`print` Integer representing when the outer step is printed. When this is 0 or do not print the iterations. 1 is print every function evaluation (default), 5 is print every 5 evaluations.

`normType` This is the type of parameter normalization/scaling used to get the scaled initial values for nlmixr2. These are used with `scaleType` of.

With the exception of `rescale2`, these come from Feature Scaling. The `rescale2` The rescaling is the same type described in the OptdesX software manual.

In general, all all scaling formula can be described by:

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

Where

The other data normalization approaches follow the following formula

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

- `rescale2` This scales all parameters from (-1 to 1). The relative differences between the parameters are preserved with this approach and the constants are:

  $$C_1$$

  = (max(all unscaled values)+min(all unscaled values))/2

  $$C_2$$

  = (max(all unscaled values) - min(all unscaled values))/2
- `rescale` or min-max normalization. This rescales all parameters from (0 to 1). As in the `rescale2` the relative differences are preserved. In this approach:

  $$C_1$$

  = min(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)

- `mean` or mean normalization. This rescales to center the parameters around the mean but the parameters are from 0 to 1. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `std` or standardization. This standardizes by the mean and standard deviation. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = sd(all unscaled values)
- `len` or unit length scaling. This scales the parameters to the unit length. For this approach we use the Euclidean length, that is:

  $$C_1$$

  = 0

  $$C_2$$

  =

  $$\sqrt{(}v_1^2 + v_2^2 + \cdots + v_n^2)$$
- `constant` which does not perform data normalization. That is

  $$C_1$$

  = 0

  $$C_2$$

  = 1

`scaleType` The scaling scheme for nlmixr2. The supported types are:

- `nlmixr2` In this approach the scaling is performed by the following equation:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

)*scaleC[i] + scaleTo

  The `scaleTo` parameter is specified by the `normType`, and the scales are specified by `scaleC`.

- `norm` This approach uses the simple scaling provided by the `normType` argument.
- `mult` This approach does not use the data normalization provided by `normType`, but rather uses multiplicative scaling to a constant provided by the `scaleTo` argument.
  In this case:
  $$v_{scaled}$$
  =
  $$v_{current}$$
  /
  $$v_{init}$$
  *scaleTo
- `multAdd` This approach changes the scaling based on the parameter being specified. If a parameter is defined in an exponential block (ie exp(theta)), then it is scaled on a linearly, that is:
  $$v_{scaled}$$
  = (
  $$v_{current} - v_{init}$$
) + scaleTo
  Otherwise the parameter is scaled multiplicatively.
  $$v_{scaled}$$
  =
  $$v_{current}$$
  /
  $$v_{init}$$
  *scaleTo

`scaleCmax` Maximum value of the scaleC to prevent overflow.

`scaleCmin` Minimum value of the scaleC to prevent underflow.

`scaleC` The scaling constant used with `scaleType=nlmixr2`. When not specified, it is based on the type of parameter that is estimated. The idea is to keep the derivatives similar on a log scale to have similar gradient sizes. Hence parameters like log(exp(theta)) would have a scaling factor of 1 and log(theta) would have a scaling factor of ini_value (to scale by 1/value; ie d/dt(log(ini_value)) = 1/ini_value or scaleC=ini_value)

- For parameters in an exponential (ie exp(theta)) or parameters specifying powers, boxCox or yeoJohnson transformations , this is 1.
- For additive, proportional, lognormal error structures, these are given by 0.5*abs(initial_estimate)
- Factorials are scaled by abs(1/digamma(initial_estimate+1))
- parameters in a log scale (ie log(theta)) are transformed by log(abs(initial_estimate))*abs(initial_estir

	These parameter scaling coefficients are chose to try to keep similar slopes among parameters. That is they all follow the slopes approximately on a log-scale. While these are chosen in a logical manner, they may not always apply. You can specify each parameters scaling factor by this parameter if you wish.
`scaleTo`	Scale the initial parameter estimate to this value. By default this is 1. When zero or below, no scaling is performed.
`gradTo`	this is the factor that the gradient is scaled to before optimizing. This only works with scaleType="nlmixr2".
`rxControl`	‘rxode2‘ ODE solving options during fitting, created with ‘rxControl()‘
`optExpression`	Optimize the rxode2 expression to speed up calculation. By default this is turned on.
`sumProd`	Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is `FALSE`.
`literalFix`	boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is ‘TRUE‘.

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

Where:

- y represents the observed value

- f represents the predicted value

- a is the additive standard deviation

- b is the proportional/power standard deviation

- c is the power exponent (in the proportional case c=1)

`calcTables`	This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`
`compress`	Should the object have compressed items

`covMethod` Method for calculating covariance. In this discussion, R is the Hessian matrix of the objective function. The S matrix is the sum of individual gradient cross-product (evaluated at the individual empirical Bayes estimates).

- "`r,s`" Uses the sandwich matrix to calculate the covariance, that is: `solve(R) %*% S %*% solve(R)`
- "`r`" Uses the Hessian matrix to calculate the covariance as `2 %*% solve(R)`
- "`s`" Uses the cross-product matrix to calculate the covariance as `4 %*% solve(S)`
- "" Does not calculate the covariance step.

`adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM's default objective function. By default this is `TRUE`

`ci` Confidence level for some tables. By default this is 0.95 or 95% confidence.

`sigdig` Optimization significant digits. This controls:

- The tolerance of the inner and outer optimization is `10^-sigdig`
- The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)
- The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`

`sigdigTable` Significant digits in the final output table. If not specified, then it matches the significant digits in the ‘sigdig‘ optimization algorithm. If ‘sigdig‘ is NULL, use 3.

`...` Ignored parameters

Value

bobqya control structure

Author(s)

Matthew L. Fidler

Examples

```
# A logit regression example with emax model

dsn <- data.frame(i=1:1000)
dsn$time <- exp(rnorm(1000))
dsn$DV=rbinom(1000,1,exp(-1+dsn$time)/(1+exp(-1+dsn$time)))

mod <- function() {
 ini({
   E0 <- 0.5
   Em <- 0.5
   E50 <- 2
   g <- fix(2)
 })
 model({
   v <- E0+Em*time^g/(E50^g+time^g)
   ll(bin) ~ DV * v - log(1 + exp(v))
 })
}

fit2 <- nlmixr(mod, dsn, est="lbfgsb3c")

print(fit2)
```

```
# you can also get the nlm output with fit2$lbfgsb3c

fit2$lbfgsb3c

# The nlm control has been modified slightly to include
# extra components and name the parameters
```

n1qn1Control   *Control for n1qn1 estimation method in nlmixr2*

Description

Control for n1qn1 estimation method in nlmixr2

Usage

```
n1qn1Control(
  epsilon = (.Machine$double.eps)^0.25,
  max_iterations = 10000,
  nsim = 10000,
  imp = 0,
  print.functions = FALSE,
  returnN1qn1 = FALSE,
  stickyRecalcN = 4,
  maxOdeRecalc = 5,
  odeRecalcFactor = 10^(0.5),
  useColor = crayon::has_color(),
  printNcol = floor((getOption("width") - 23)/12),
  print = 1L,
  normType = c("rescale2", "mean", "rescale", "std", "len", "constant"),
  scaleType = c("nlmixr2", "norm", "mult", "multAdd"),
  scaleCmax = 1e+05,
  scaleCmin = 1e-05,
  scaleC = NULL,
  scaleTo = 1,
  gradTo = 1,
  rxControl = NULL,
  optExpression = TRUE,
  sumProd = FALSE,
  literalFix = TRUE,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
  compress = TRUE,
  covMethod = c("r", "n1qn1", ""),
  adjObf = TRUE,
  ci = 0.95,
```

```
  sigdig = 4,
  sigdigTable = NULL,
  ...
)
```

Arguments

- `epsilon` Precision of estimate for n1qn1 optimization.
- `max_iterations` Number of iterations
- `nsim` Number of function evaluations
- `imp` Verbosity of messages.
- `print.functions` Boolean to control if the function value and parameter estimates are echoed every time a function is called.
- `returnN1qn1` return the n1qn1 output instead of the nlmixr2 fit
- `stickyRecalcN` The number of bad ODE solves before reducing the atol/rtol for the rest of the problem.
- `maxOdeRecalc` Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.
- `odeRecalcFactor` The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced
- `useColor` Boolean indicating if focei can use ASCII color codes
- `printNcol` Number of columns to printout before wrapping parameter estimates/gradient
- `print` Integer representing when the outer step is printed. When this is 0 or do not print the iterations. 1 is print every function evaluation (default), 5 is print every 5 evaluations.
- `normType` This is the type of parameter normalization/scaling used to get the scaled initial values for nlmixr2. These are used with `scaleType` of.

  With the exception of `rescale2`, these come from Feature Scaling. The `rescale2` The rescaling is the same type described in the OptdesX software manual.

  In general, all all scaling formula can be described by:

  $$v_{scaled}$$

  = (

  $$v_{unscaled} - C_1$$

)/

  $$C_2$$

  Where

  The other data normalization approaches follow the following formula

  $$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

- `rescale2` This scales all parameters from (-1 to 1). The relative differences between the parameters are preserved with this approach and the constants are:

  $$C_1$$

  = (max(all unscaled values)+min(all unscaled values))/2

  $$C_2$$

  = (max(all unscaled values) - min(all unscaled values))/2
- `rescale` or min-max normalization. This rescales all parameters from (0 to 1). As in the `rescale2` the relative differences are preserved. In this approach:

  $$C_1$$

  = min(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `mean` or mean normalization. This rescales to center the parameters around the mean but the parameters are from 0 to 1. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `std` or standardization. This standardizes by the mean and standard deviation. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = sd(all unscaled values)
- `len` or unit length scaling. This scales the parameters to the unit length. For this approach we use the Euclidean length, that is:

  $$C_1$$

  = 0

$$C_2$$

=

$$\sqrt{(}v_1^2 + v_2^2 + \cdots + v_n^2)$$

- `constant` which does not perform data normalization. That is

  $$C_1$$

  = 0

  $$C_2$$

  = 1

`scaleType` The scaling scheme for nlmixr2. The supported types are:

- `nlmixr2` In this approach the scaling is performed by the following equation:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

)*scaleC[i] + scaleTo
  The `scaleTo` parameter is specified by the `normType`, and the scales are specified by `scaleC`.
- `norm` This approach uses the simple scaling provided by the `normType` argument.
- `mult` This approach does not use the data normalization provided by `normType`, but rather uses multiplicative scaling to a constant provided by the `scaleTo` argument.
  In this case:

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo
- `multAdd` This approach changes the scaling based on the parameter being specified. If a parameter is defined in an exponential block (ie exp(theta)), then it is scaled on a linearly, that is:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

) + scaleTo

Otherwise the parameter is scaled multiplicatively.

v_{scaled}

=

$v_{current}$

/

v_{init}

*scaleTo

`scaleCmax` Maximum value of the scaleC to prevent overflow.

`scaleCmin` Minimum value of the scaleC to prevent underflow.

`scaleC` The scaling constant used with `scaleType=nlmixr2`. When not specified, it is based on the type of parameter that is estimated. The idea is to keep the derivatives similar on a log scale to have similar gradient sizes. Hence parameters like log(exp(theta)) would have a scaling factor of 1 and log(theta) would have a scaling factor of ini_value (to scale by 1/value; ie d/dt(log(ini_value)) = 1/ini_value or scaleC=ini_value)

- For parameters in an exponential (ie exp(theta)) or parameters specifying powers, boxCox or yeoJohnson transformations , this is 1.
- For additive, proportional, lognormal error structures, these are given by 0.5*abs(initial_estimate)
- Factorials are scaled by abs(1/digamma(initial_estimate+1))
- parameters in a log scale (ie log(theta)) are transformed by log(abs(initial_estimate))*abs(initial_estim

These parameter scaling coefficients are chose to try to keep similar slopes among parameters. That is they all follow the slopes approximately on a log-scale.

While these are chosen in a logical manner, they may not always apply. You can specify each parameters scaling factor by this parameter if you wish.

`scaleTo` Scale the initial parameter estimate to this value. By default this is 1. When zero or below, no scaling is performed.

`gradTo` this is the factor that the gradient is scaled to before optimizing. This only works with scaleType="nlmixr2".

`rxControl` ‘rxode2‘ ODE solving options during fitting, created with ‘rxControl()‘

`optExpression` Optimize the rxode2 expression to speed up calculation. By default this is turned on.

`sumProd` Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is `FALSE`.

`literalFix` boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is ‘TRUE‘.

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

Where:
- y represents the observed value
- f represents the predicted value
- a is the additive standard deviation
- b is the proportional/power standard deviation
- c is the power exponent (in the proportional case c=1)

`calcTables` This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`

`compress` Should the object have compressed items

`covMethod` Method for calculating covariance. In this discussion, R is the Hessian matrix of the objective function. The S matrix is the sum of individual gradient cross-product (evaluated at the individual empirical Bayes estimates).

- `"r,s"` Uses the sandwich matrix to calculate the covariance, that is: `solve(R) %*% S %*% solve(R)`
- `"r"` Uses the Hessian matrix to calculate the covariance as `2 %*% solve(R)`
- `"s"` Uses the cross-product matrix to calculate the covariance as `4 %*% solve(S)`
- `""` Does not calculate the covariance step.

`adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM's default objective function. By default this is `TRUE`

`ci` Confidence level for some tables. By default this is 0.95 or 95% confidence.

`sigdig` Optimization significant digits. This controls:

- The tolerance of the inner and outer optimization is `10^-sigdig`
- The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)
- The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`

`sigdigTable` Significant digits in the final output table. If not specified, then it matches the significant digits in the 'sigdig' optimization algorithm. If 'sigdig' is NULL, use 3.

`...` Ignored parameters

Value

bobqya control structure

Author(s)

Matthew L. Fidler

Examples

```
# A logit regression example with emax model

dsn <- data.frame(i=1:1000)
dsn$time <- exp(rnorm(1000))
dsn$DV=rbinom(1000,1,exp(-1+dsn$time)/(1+exp(-1+dsn$time)))

mod <- function() {
 ini({
   E0 <- 0.5
   Em <- 0.5
   E50 <- 2
   g <- fix(2)
 })
 model({
   v <- E0+Em*time^g/(E50^g+time^g)
   ll(bin) ~ DV * v - log(1 + exp(v))
 })
}

fit2 <- nlmixr(mod, dsn, est="n1qn1")

print(fit2)

# you can also get the nlm output with fit2$n1qn1

fit2$n1qn1

# The nlm control has been modified slightly to include
# extra components and name the parameters
```

`newuoaControl` *Control for newuoa estimation method in nlmixr2*

Description

Control for newuoa estimation method in nlmixr2

Usage

```
newuoaControl(
  npt = NULL,
  rhobeg = NULL,
  rhoend = NULL,
  iprint = 0L,
  maxfun = 100000L,
```

```
  returnNewuoa = FALSE,
  stickyRecalcN = 4,
  maxOdeRecalc = 5,
  odeRecalcFactor = 10^(0.5),
  useColor = crayon::has_color(),
  printNcol = floor((getOption("width") - 23)/12),
  print = 1L,
  normType = c("rescale2", "mean", "rescale", "std", "len", "constant"),
  scaleType = c("nlmixr2", "norm", "mult", "multAdd"),
  scaleCmax = 1e+05,
  scaleCmin = 1e-05,
  scaleC = NULL,
  scaleTo = 1,
  rxControl = NULL,
  optExpression = TRUE,
  sumProd = FALSE,
  literalFix = TRUE,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
  compress = TRUE,
  covMethod = c("r", ""),
  adjObf = TRUE,
  ci = 0.95,
  sigdig = 4,
  sigdigTable = NULL,
  ...
)
```

Arguments

`npt` The number of points used to approximate the objective function via a quadratic approximation for bobyqa. The value of npt must be in the interval [n+2,(n+1)(n+2)/2] where n is the number of parameters in par. Choices that exceed 2*n+1 are not recommended. If not defined, it will be set to 2*n + 1. (bobyqa)

`rhobeg` Beginning change in parameters for bobyqa algorithm (trust region). By default this is 0.2 or 20 parameters when the parameters are scaled to 1. rhobeg and rhoend must be set to the initial and final values of a trust region radius, so both must be positive with 0 < rhoend < rhobeg. Typically rhobeg should be about one tenth of the greatest expected change to a variable. Note also that smallest difference abs(upper-lower) should be greater than or equal to rhobeg*2. If this is not the case then rhobeg will be adjusted. (bobyqa)

`rhoend` The smallest value of the trust region radius that is allowed. If not defined, then 10^(-sigdig-1) will be used. (bobyqa)

`iprint` The value of ‘iprint‘ should be set to an integer value in ‘0, 1, 2, 3, ...‘, which controls the amount of printing. Specifically, there is no output if ‘iprint=0‘ and there is output only at the start and the return if ‘iprint=1‘. Otherwise, each new value of ‘rho‘ is printed, with the best vector of variables so far and the corresponding value of the objective function. Further, each new value of the objective function with its variables are output if 'iprint=3'. If 'iprint > 3', the objective function value and corresponding variables are output every 'iprint' evaluations. Default value is '0'.

`maxfun` The maximum allowed number of function evaluations. If this is exceeded, the method will terminate.

`returnNewuoa` return the newuoa output instead of the nlmixr2 fit

`stickyRecalcN` The number of bad ODE solves before reducing the atol/rtol for the rest of the problem.

`maxOdeRecalc` Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.

`odeRecalcFactor`
The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced

`useColor` Boolean indicating if focei can use ASCII color codes

`printNcol` Number of columns to printout before wrapping parameter estimates/gradient

`print` Integer representing when the outer step is printed. When this is 0 or do not print the iterations. 1 is print every function evaluation (default), 5 is print every 5 evaluations.

`normType` This is the type of parameter normalization/scaling used to get the scaled initial values for nlmixr2. These are used with `scaleType` of.

With the exception of `rescale2`, these come from Feature Scaling. The `rescale2` The rescaling is the same type described in the OptdesX software manual.

In general, all all scaling formula can be described by:

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

Where

The other data normalization approaches follow the following formula

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

- `rescale2` This scales all parameters from (-1 to 1). The relative differences between the parameters are preserved with this approach and the constants are:

$$C_1$$

= (max(all unscaled values)+min(all unscaled values))/2

$$C_2$$

= (max(all unscaled values) - min(all unscaled values))/2

- `rescale` or min-max normalization. This rescales all parameters from (0 to 1). As in the `rescale2` the relative differences are preserved. In this approach:

  $$C_1$$

  = min(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `mean` or mean normalization. This rescales to center the parameters around the mean but the parameters are from 0 to 1. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `std` or standardization. This standardizes by the mean and standard deviation. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = sd(all unscaled values)
- `len` or unit length scaling. This scales the parameters to the unit length. For this approach we use the Euclidean length, that is:

  $$C_1$$

  = 0

  $$C_2$$

  =

  $$\sqrt{(}v_1^2 + v_2^2 + \cdots + v_n^2)$$
- `constant` which does not perform data normalization. That is

  $$C_1$$

  = 0

$$C_2$$

= 1

`scaleType` The scaling scheme for nlmixr2. The supported types are:

- `nlmixr2` In this approach the scaling is performed by the following equation:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

)*scaleC[i] + scaleTo

  The `scaleTo` parameter is specified by the `normType`, and the scales are specified by `scaleC`.
- `norm` This approach uses the simple scaling provided by the `normType` argument.
- `mult` This approach does not use the data normalization provided by `normType`, but rather uses multiplicative scaling to a constant provided by the `scaleTo` argument.

  In this case:

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo
- `multAdd` This approach changes the scaling based on the parameter being specified. If a parameter is defined in an exponential block (ie exp(theta)), then it is scaled on a linearly, that is:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

) + scaleTo

  Otherwise the parameter is scaled multiplicatively.

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo

`scaleCmax` Maximum value of the scaleC to prevent overflow.

`scaleCmin` Minimum value of the scaleC to prevent underflow.

`scaleC` The scaling constant used with `scaleType=nlmixr2`. When not specified, it is based on the type of parameter that is estimated. The idea is to keep the derivatives similar on a log scale to have similar gradient sizes. Hence parameters like log(exp(theta)) would have a scaling factor of 1 and log(theta) would have a scaling factor of ini_value (to scale by 1/value; ie d/dt(log(ini_value)) = 1/ini_value or scaleC=ini_value)

- For parameters in an exponential (ie exp(theta)) or parameters specifying powers, boxCox or yeoJohnson transformations , this is 1.
- For additive, proportional, lognormal error structures, these are given by 0.5*abs(initial_estimate)
- Factorials are scaled by abs(1/digamma(initial_estimate+1))
- parameters in a log scale (ie log(theta)) are transformed by log(abs(initial_estimate))*abs(initial_estir

These parameter scaling coefficients are chose to try to keep similar slopes among parameters. That is they all follow the slopes approximately on a log-scale.

While these are chosen in a logical manner, they may not always apply. You can specify each parameters scaling factor by this parameter if you wish.

`scaleTo` Scale the initial parameter estimate to this value. By default this is 1. When zero or below, no scaling is performed.

`rxControl` ‘rxode2‘ ODE solving options during fitting, created with ‘rxControl()‘

`optExpression` Optimize the rxode2 expression to speed up calculation. By default this is turned on.

`sumProd` Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is `FALSE`.

`literalFix` boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is ‘TRUE‘.

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

Where:

- y represents the observed value

- f represents the predicted value

- a is the additive standard deviation

- b is the proportional/power standard deviation

- c is the power exponent (in the proportional case c=1)

- `calcTables` This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`
- `compress` Should the object have compressed items
- `covMethod` Method for calculating covariance. In this discussion, R is the Hessian matrix of the objective function. The S matrix is the sum of individual gradient cross-product (evaluated at the individual empirical Bayes estimates).
  - "`r,s`" Uses the sandwich matrix to calculate the covariance, that is: `solve(R) %*% S %*% solve(R)`
  - "`r`" Uses the Hessian matrix to calculate the covariance as `2 %*% solve(R)`
  - "`s`" Uses the cross-product matrix to calculate the covariance as `4 %*% solve(S)`
  - "" Does not calculate the covariance step.
- `adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM's default objective function. By default this is `TRUE`
- `ci` Confidence level for some tables. By default this is 0.95 or 95% confidence.
- `sigdig` Optimization significant digits. This controls:
  - The tolerance of the inner and outer optimization is `10^-sigdig`
  - The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)
  - The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`
- `sigdigTable` Significant digits in the final output table. If not specified, then it matches the significant digits in the 'sigdig' optimization algorithm. If 'sigdig' is NULL, use 3.
- `...` Ignored parameters

Value

newuoa control structure

Author(s)

Matthew L. Fidler

Examples

```
# A logit regression example with emax model

dsn <- data.frame(i=1:1000)
dsn$time <- exp(rnorm(1000))
dsn$DV=rbinom(1000,1,exp(-1+dsn$time)/(1+exp(-1+dsn$time)))

mod <- function() {
 ini({
   E0 <- 0.5
```

```
    Em <- 0.5
    E50 <- 2
    g <- fix(2)
  })
  model({
    v <- E0+Em*time^g/(E50^g+time^g)
    ll(bin) ~ DV * v - log(1 + exp(v))
  })
}

fit2 <- nlmixr(mod, dsn, est="newuoa")

print(fit2)

# you can also get the nlm output with

fit2$newuoa

# The nlm control has been modified slightly to include
# extra components and name the parameters
```

`nlmControl` *nlmixr2 defaults controls for nlm*

Description

nlmixr2 defaults controls for nlm

Usage

```
nlmControl(
  typsize = NULL,
  fscale = 1,
  print.level = 0,
  ndigit = NULL,
  gradtol = 1e-06,
  stepmax = NULL,
  steptol = 1e-06,
  iterlim = 10000,
  check.analyticals = FALSE,
  returnNlm = FALSE,
  solveType = c("hessian", "grad", "fun"),
  stickyRecalcN = 4,
  maxOdeRecalc = 5,
  odeRecalcFactor = 10^(0.5),
  eventType = c("central", "forward"),
  shiErr = (.Machine$double.eps)^(1/3),
  shi21maxFD = 20L,
```

```
  optimHessType = c("central", "forward"),
  hessErr = (.Machine$double.eps)^(1/3),
  shi21maxHess = 20L,
  useColor = crayon::has_color(),
  printNcol = floor((getOption("width") - 23)/12),
  print = 1L,
  normType = c("rescale2", "mean", "rescale", "std", "len", "constant"),
  scaleType = c("nlmixr2", "norm", "mult", "multAdd"),
  scaleCmax = 1e+05,
  scaleCmin = 1e-05,
  scaleC = NULL,
  scaleTo = 1,
  gradTo = 1,
  rxControl = NULL,
  optExpression = TRUE,
  sumProd = FALSE,
  literalFix = TRUE,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
  compress = TRUE,
  covMethod = c("r", "nlm", ""),
  adjObf = TRUE,
  ci = 0.95,
  sigdig = 4,
  sigdigTable = NULL,
  ...
)
```

Arguments

`typsize`	an estimate of the size of each parameter at the minimum.
`fscale`	an estimate of the size of `f` at the minimum.
`print.level`	this argument determines the level of printing which is done during the minimization process. The default value of `0` means that no printing occurs, a value of `1` means that initial and final details are printed and a value of 2 means that full tracing information is printed.
`ndigit`	the number of significant digits in the function `f`.
`gradtol`	a positive scalar giving the tolerance at which the scaled gradient is considered close enough to zero to terminate the algorithm. The scaled gradient is a measure of the relative change in `f` in each direction `p[i]` divided by the relative change in `p[i]`.
`stepmax`	a positive scalar which gives the maximum allowable scaled step length. `stepmax` is used to prevent steps which would cause the optimization function to overflow, to prevent the algorithm from leaving the area of interest in parameter space, or to detect divergence in the algorithm. `stepmax` would be chosen small enough to prevent the first two of these occurrences, but should be larger than any anticipated reasonable step.

steptol
: A positive scalar providing the minimum allowable relative step length.

iterlim
: a positive integer specifying the maximum number of iterations to be performed before the program is terminated.

check.analyticals
: a logical scalar specifying whether the analytic gradients and Hessians, if they are supplied, should be checked against numerical derivatives at the initial parameter values. This can help detect incorrectly formulated gradients or Hessians.

returnNlm
: is a logical that allows a return of the ‘nlm‘ object

solveType
: tells if ‘nlm‘ will use nlmixr2’s analytical gradients when available (finite differences will be used for event-related parameters like parameters controlling lag time, duration/rate of infusion, and modeled bioavailability). This can be:

  - ‘"hessian"‘ which will use the analytical gradients to create a Hessian with finite differences.

  - ‘"gradient"‘ which will use the gradient and let ‘nlm‘ calculate the finite difference hessian

  - ‘"fun"‘ where nlm will calculate both the finite difference gradient and the finite difference Hessian

  When using nlmixr2’s finite differences, the "ideal" step size for either central or forward differences are optimized for with the Shi2021 method which may give more accurate derivatives

stickyRecalcN
: The number of bad ODE solves before reducing the atol/rtol for the rest of the problem.

maxOdeRecalc
: Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.

odeRecalcFactor
: The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced

eventType
: Event gradient type for dosing events; Can be "central" or "forward"

shiErr
: This represents the epsilon when optimizing the ideal step size for numeric differentiation using the Shi2021 method

shi21maxFD
: The maximum number of steps for the optimization of the forward difference step size when using dosing events (lag time, modeled duration/rate and bioavailability)

optimHessType
: The hessian type for when calculating the individual hessian by numeric differences (in generalized log-likelihood estimation). The options are "central", and "forward". The central differences is what R’s ‘optimHess()‘ uses and is the default for this method. (Though the "forward" is faster and still reasonable for most cases). The Shi21 cannot be changed for the Gill83 algorithm with the optimHess in a generalized likelihood problem.

hessErr
: This represents the epsilon when optimizing the Hessian step size using the Shi2021 method.

shi21maxHess
: Maximum number of times to optimize the best step size for the hessian calculation

`useColor` Boolean indicating if focei can use ASCII color codes

`printNcol` Number of columns to printout before wrapping parameter estimates/gradient

`print` Integer representing when the outer step is printed. When this is 0 or do not print the iterations. 1 is print every function evaluation (default), 5 is print every 5 evaluations.

`normType` This is the type of parameter normalization/scaling used to get the scaled initial values for nlmixr2. These are used with `scaleType` of.

With the exception of `rescale2`, these come from Feature Scaling. The `rescale2` The rescaling is the same type described in the OptdesX software manual.

In general, all all scaling formula can be described by:

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

Where

The other data normalization approaches follow the following formula

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

- `rescale2` This scales all parameters from (-1 to 1). The relative differences between the parameters are preserved with this approach and the constants are:

  $$C_1$$

  = (max(all unscaled values)+min(all unscaled values))/2

  $$C_2$$

  = (max(all unscaled values) - min(all unscaled values))/2
- `rescale` or min-max normalization. This rescales all parameters from (0 to 1). As in the `rescale2` the relative differences are preserved. In this approach:

  $$C_1$$

  = min(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)

- `mean` or mean normalization. This rescales to center the parameters around the mean but the parameters are from 0 to 1. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `std` or standardization. This standardizes by the mean and standard deviation. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = sd(all unscaled values)
- `len` or unit length scaling. This scales the parameters to the unit length. For this approach we use the Euclidean length, that is:

  $$C_1$$

  = 0

  $$C_2$$

  =

  $$\sqrt{(}v_1^2 + v_2^2 + \cdots + v_n^2)$$
- `constant` which does not perform data normalization. That is

  $$C_1$$

  = 0

  $$C_2$$

  = 1

`scaleType` The scaling scheme for nlmixr2. The supported types are:

- `nlmixr2` In this approach the scaling is performed by the following equation:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

)*scaleC[i] + scaleTo

  The `scaleTo` parameter is specified by the `normType`, and the scales are specified by `scaleC`.

- `norm` This approach uses the simple scaling provided by the `normType` argument.
- `mult` This approach does not use the data normalization provided by `normType`, but rather uses multiplicative scaling to a constant provided by the `scaleTo` argument.

  In this case:

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo
- `multAdd` This approach changes the scaling based on the parameter being specified. If a parameter is defined in an exponential block (ie exp(theta)), then it is scaled on a linearly, that is:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

) + scaleTo

  Otherwise the parameter is scaled multiplicatively.

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo

`scaleCmax` Maximum value of the scaleC to prevent overflow.

`scaleCmin` Minimum value of the scaleC to prevent underflow.

`scaleC` The scaling constant used with `scaleType=nlmixr2`. When not specified, it is based on the type of parameter that is estimated. The idea is to keep the derivatives similar on a log scale to have similar gradient sizes. Hence parameters like log(exp(theta)) would have a scaling factor of 1 and log(theta) would have a scaling factor of ini_value (to scale by 1/value; ie d/dt(log(ini_value)) = 1/ini_value or scaleC=ini_value)

- For parameters in an exponential (ie exp(theta)) or parameters specifying powers, boxCox or yeoJohnson transformations , this is 1.
- For additive, proportional, lognormal error structures, these are given by 0.5*abs(initial_estimate)
- Factorials are scaled by abs(1/digamma(initial_estimate+1))
- parameters in a log scale (ie log(theta)) are transformed by log(abs(initial_estimate))*abs(initial_estir

These parameter scaling coefficients are chose to try to keep similar slopes among parameters. That is they all follow the slopes approximately on a log-scale.

While these are chosen in a logical manner, they may not always apply. You can specify each parameters scaling factor by this parameter if you wish.

`scaleTo` Scale the initial parameter estimate to this value. By default this is 1. When zero or below, no scaling is performed.

`gradTo` this is the factor that the gradient is scaled to before optimizing. This only works with scaleType="nlmixr2".

`rxControl` ‘rxode2‘ ODE solving options during fitting, created with ‘rxControl()‘

`optExpression` Optimize the rxode2 expression to speed up calculation. By default this is turned on.

`sumProd` Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is `FALSE`.

`literalFix` boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is ‘TRUE‘.

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

Where:

- y represents the observed value

- f represents the predicted value

- a is the additive standard deviation

- b is the proportional/power standard deviation

- c is the power exponent (in the proportional case c=1)

`calcTables` This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`

`compress` Should the object have compressed items

`covMethod` allows selection of "r", which uses nlmixr2’s ‘nlmixr2Hess()‘ for the hessian calculation or "nlm" which uses the hessian from ‘stats::nlm(.., hessian=TRUE)‘. When using ‘nlmixr2’s‘ hessian for optimization or ‘nlmixr2’s‘ gradient for solving this defaults to "nlm" since ‘stats::optimHess()‘ assumes an accurate gradient and is faster than ‘nlmixr2Hess‘

`adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM’s default objective function. By default this is `TRUE`

`ci` Confidence level for some tables. By default this is 0.95 or 95% confidence.

`sigdig` Optimization significant digits. This controls:

- The tolerance of the inner and outer optimization is `10^-sigdig`
- The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)
- The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`

`sigdigTable` Significant digits in the final output table. If not specified, then it matches the significant digits in the ‘sigdig‘ optimization algorithm. If ‘sigdig‘ is NULL, use 3.

`...` additional arguments to be passed to `f`.

Value

nlm control object

Author(s)

Matthew L. Fidler

Examples

```
# A logit regression example with emax model

dsn <- data.frame(i=1:1000)
dsn$time <- exp(rnorm(1000))
dsn$DV=rbinom(1000,1,exp(-1+dsn$time)/(1+exp(-1+dsn$time)))

mod <- function() {
 ini({
   E0 <- 0.5
   Em <- 0.5
   E50 <- 2
   g <- fix(2)
 })
 model({
   v <- E0+Em*time^g/(E50^g+time^g)
   ll(bin) ~ DV * v - log(1 + exp(v))
 })
}

fit2 <- nlmixr(mod, dsn, est="nlm")

print(fit2)

# you can also get the nlm output with fit2$nlm

fit2$nlm
```

```
# The nlm control has been modified slightly to include
# extra components and name the parameters
```

---

nlminbControl *nlmixr2 nlminb defaults*

Description

nlmixr2 nlminb defaults

Usage

```
nlminbControl(
  eval.max = 200,
  iter.max = 150,
  trace = 0,
  abs.tol = 0,
  rel.tol = 1e-10,
  x.tol = 1.5e-08,
  xf.tol = 2.2e-14,
  step.min = 1,
  step.max = 1,
  sing.tol = rel.tol,
  scale = 1,
  scale.init = NULL,
  diff.g = NULL,
  rxControl = NULL,
  optExpression = TRUE,
  sumProd = FALSE,
  literalFix = TRUE,
  returnNlminb = FALSE,
  solveType = c("hessian", "grad", "fun"),
  stickyRecalcN = 4,
  maxOdeRecalc = 5,
  odeRecalcFactor = 10^(0.5),
  eventType = c("central", "forward"),
  shiErr = (.Machine$double.eps)^(1/3),
  shi21maxFD = 20L,
  optimHessType = c("central", "forward"),
  hessErr = (.Machine$double.eps)^(1/3),
  shi21maxHess = 20L,
  useColor = crayon::has_color(),
  printNcol = floor((getOption("width") - 23)/12),
  print = 1L,
  normType = c("rescale2", "mean", "rescale", "std", "len", "constant"),
  scaleType = c("nlmixr2", "norm", "mult", "multAdd"),
```

```
  scaleCmax = 1e+05,
  scaleCmin = 1e-05,
  scaleC = NULL,
  scaleTo = 1,
  gradTo = 1,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
  compress = TRUE,
  covMethod = c("r", "nlminb", ""),
  adjObf = TRUE,
  ci = 0.95,
  sigdig = 4,
  sigdigTable = NULL,
  ...
)
```

Arguments

eval.max	Maximum number of evaluations of the objective function allowed. Defaults to 200.
iter.max	Maximum number of iterations allowed. Defaults to 150.
trace	The value of the objective function and the parameters is printed every trace'th iteration. When 0 no trace information is to be printed
abs.tol	Absolute tolerance. Defaults to 0 so the absolute convergence test is not used. If the objective function is known to be non-negative, the previous default of '1e-20' would be more appropriate
rel.tol	Relative tolerance. Defaults to '1e-10'.
x.tol	X tolerance. Defaults to '1.5e-8'.
xf.tol	false convergence tolerance. Defaults to '2.2e-14'.
step.min	Minimum step size. Default to '1.'.
step.max	Maximum step size. Default to '1.'.
sing.tol	singular convergence tolerance; defaults to 'rel.tol;.
scale	See PORT documentation (or leave alone).
scale.init	... probably need to check PORT documentation
diff.g	an estimated bound on the relative error in the objective function value
rxControl	'rxode2' ODE solving options during fitting, created with 'rxControl()'
optExpression	Optimize the rxode2 expression to speed up calculation. By default this is turned on.
sumProd	Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is FALSE.
literalFix	boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is 'TRUE'.

`returnNlminb` logical; when TRUE this will return the nlminb result instead of the nlmixr2 fit object

`solveType` tells if ‘nlm‘ will use nlmixr2’s analytical gradients when available (finite differences will be used for event-related parameters like parameters controlling lag time, duration/rate of infusion, and modeled bioavailability). This can be:

- ‘"hessian"‘ which will use the analytical gradients to create a Hessian with finite differences.

- ‘"gradient"‘ which will use the gradient and let ‘nlm‘ calculate the finite difference hessian

- ‘"fun"‘ where nlm will calculate both the finite difference gradient and the finite difference Hessian

When using nlmixr2’s finite differences, the "ideal" step size for either central or forward differences are optimized for with the Shi2021 method which may give more accurate derivatives

`stickyRecalcN` The number of bad ODE solves before reducing the atol/rtol for the rest of the problem.

`maxOdeRecalc` Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.

`odeRecalcFactor` The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced

`eventType` Event gradient type for dosing events; Can be "central" or "forward"

`shiErr` This represents the epsilon when optimizing the ideal step size for numeric differentiation using the Shi2021 method

`shi21maxFD` The maximum number of steps for the optimization of the forward difference step size when using dosing events (lag time, modeled duration/rate and bioavailability)

`optimHessType` The hessian type for when calculating the individual hessian by numeric differences (in generalized log-likelihood estimation). The options are "central", and "forward". The central differences is what R’s ‘optimHess()‘ uses and is the default for this method. (Though the "forward" is faster and still reasonable for most cases). The Shi21 cannot be changed for the Gill83 algorithm with the optimHess in a generalized likelihood problem.

`hessErr` This represents the epsilon when optimizing the Hessian step size using the Shi2021 method.

`shi21maxHess` Maximum number of times to optimize the best step size for the hessian calculation

`useColor` Boolean indicating if focei can use ASCII color codes

`printNcol` Number of columns to printout before wrapping parameter estimates/gradient

`print` Integer representing when the outer step is printed. When this is 0 or do not print the iterations. 1 is print every function evaluation (default), 5 is print every 5 evaluations.

`normType` This is the type of parameter normalization/scaling used to get the scaled initial values for nlmixr2. These are used with `scaleType` of.

With the exception of `rescale2`, these come from Feature Scaling. The `rescale2` The rescaling is the same type described in the OptdesX software manual.

In general, all all scaling formula can be described by:

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

Where

The other data normalization approaches follow the following formula

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

- `rescale2` This scales all parameters from (-1 to 1). The relative differences between the parameters are preserved with this approach and the constants are:

  $$C_1$$

  = (max(all unscaled values)+min(all unscaled values))/2

  $$C_2$$

  = (max(all unscaled values) - min(all unscaled values))/2
- `rescale` or min-max normalization. This rescales all parameters from (0 to 1). As in the `rescale2` the relative differences are preserved. In this approach:

  $$C_1$$

  = min(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `mean` or mean normalization. This rescales to center the parameters around the mean but the parameters are from 0 to 1. In this approach:

  $$C_1$$

  = mean(all unscaled values)

$$C_2$$

= max(all unscaled values) - min(all unscaled values)

- `std` or standardization. This standardizes by the mean and standard deviation. In this approach:

$$C_1$$

= mean(all unscaled values)

$$C_2$$

= sd(all unscaled values)

- `len` or unit length scaling. This scales the parameters to the unit length. For this approach we use the Euclidean length, that is:

$$C_1$$

= 0

$$C_2$$

=

$$\sqrt(v_1^2 + v_2^2 + \cdots + v_n^2)$$

- `constant` which does not perform data normalization. That is

$$C_1$$

= 0

$$C_2$$

= 1

`scaleType` The scaling scheme for nlmixr2. The supported types are:

- `nlmixr2` In this approach the scaling is performed by the following equation:

$$v_{scaled}$$

= (

$$v_{current} - v_{init}$$

)*scaleC[i] + scaleTo
The `scaleTo` parameter is specified by the `normType`, and the scales are specified by `scaleC`.

- `norm` This approach uses the simple scaling provided by the `normType` argument.

- `mult` This approach does not use the data normalization provided by `normType`, but rather uses multiplicative scaling to a constant provided by the `scaleTo` argument.
  In this case:

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo
- `multAdd` This approach changes the scaling based on the parameter being specified. If a parameter is defined in an exponential block (ie exp(theta)), then it is scaled on a linearly, that is:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

) + scaleTo
  Otherwise the parameter is scaled multiplicatively.

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo

`scaleCmax` Maximum value of the scaleC to prevent overflow.

`scaleCmin` Minimum value of the scaleC to prevent underflow.

`scaleC` The scaling constant used with `scaleType=nlmixr2`. When not specified, it is based on the type of parameter that is estimated. The idea is to keep the derivatives similar on a log scale to have similar gradient sizes. Hence parameters like log(exp(theta)) would have a scaling factor of 1 and log(theta) would have a scaling factor of ini_value (to scale by 1/value; ie d/dt(log(ini_value)) = 1/ini_value or scaleC=ini_value)

- For parameters in an exponential (ie exp(theta)) or parameters specifying powers, boxCox or yeoJohnson transformations , this is 1.
- For additive, proportional, lognormal error structures, these are given by 0.5*abs(initial_estimate)
- Factorials are scaled by abs(1/digamma(initial_estimate+1))
- parameters in a log scale (ie log(theta)) are transformed by log(abs(initial_estimate))*abs(initial_estir

These parameter scaling coefficients are chose to try to keep similar slopes among parameters. That is they all follow the slopes approximately on a log-scale.

While these are chosen in a logical manner, they may not always apply. You can specify each parameters scaling factor by this parameter if you wish.

`scaleTo` Scale the initial parameter estimate to this value. By default this is 1. When zero or below, no scaling is performed.

`gradTo` this is the factor that the gradient is scaled to before optimizing. This only works with scaleType="nlmixr2".

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

Where:

- y represents the observed value

- f represents the predicted value

- a is the additive standard deviation

- b is the proportional/power standard deviation

- c is the power exponent (in the proportional case c=1)

`calcTables` This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`

`compress` Should the object have compressed items

`covMethod` Method for calculating covariance. In this discussion, R is the Hessian matrix of the objective function. The S matrix is the sum of individual gradient cross-product (evaluated at the individual empirical Bayes estimates).

- "`r,s`" Uses the sandwich matrix to calculate the covariance, that is: `solve(R) %*% S %*% solve(R)`
- "`r`" Uses the Hessian matrix to calculate the covariance as `2 %*% solve(R)`
- "`s`" Uses the cross-product matrix to calculate the covariance as `4 %*% solve(S)`
- "" Does not calculate the covariance step.

`adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM's default objective function. By default this is `TRUE`

`ci` Confidence level for some tables. By default this is 0.95 or 95% confidence.

`sigdig` Optimization significant digits. This controls:

- The tolerance of the inner and outer optimization is `10^-sigdig`
- The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)

- The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`

`sigdigTable` Significant digits in the final output table. If not specified, then it matches the significant digits in the ‘sigdig‘ optimization algorithm. If ‘sigdig‘ is NULL, use 3.

`...` Further arguments to be supplied to `objective`.

Author(s)

Matthew L. Fidler

Examples

```
# A logit regression example with emax model

dsn <- data.frame(i=1:1000)
dsn$time <- exp(rnorm(1000))
dsn$DV=rbinom(1000,1,exp(-1+dsn$time)/(1+exp(-1+dsn$time)))

mod <- function() {
 ini({
   E0 <- 0.5
   Em <- 0.5
   E50 <- 2
   g <- fix(2)
 })
 model({
   v <- E0+Em*time^g/(E50^g+time^g)
   ll(bin) ~ DV * v - log(1 + exp(v))
 })
}

fit2 <- nlmixr(mod, dsn, est="nlminb")

print(fit2)

# you can also get the nlm output with fit2$nlminb

fit2$nlminb
```

`nlmixr2` *nlmixr2 fits population PK and PKPD non-linear mixed effects models.*

Description

nlmixr2 is an R package for fitting population pharmacokinetic (PK) and pharmacokinetic-pharmacodynamic (PKPD) models.

Usage

```
nlmixr2(
  object,
  data,
  est = NULL,
  control = list(),
  table = tableControl(),
  ...,
  save = NULL,
  envir = parent.frame()
)

nlmixr(
  object,
  data,
  est = NULL,
  control = list(),
  table = tableControl(),
  ...,
  save = NULL,
  envir = parent.frame()
)

## S3 method for class '`function`'
nlmixr2(
  object,
  data = NULL,
  est = NULL,
  control = NULL,
  table = tableControl(),
  ...,
  save = NULL,
  envir = parent.frame()
)

## S3 method for class 'rxUi'
nlmixr2(
  object,
  data = NULL,
  est = NULL,
  control = NULL,
  table = tableControl(),
  ...,
  save = NULL,
  envir = parent.frame()
)

## S3 method for class 'nlmixr2FitCore'
```

```
nlmixr2(
  object,
  data = NULL,
  est = NULL,
  control = NULL,
  table = tableControl(),
  ...,
  save = NULL,
  envir = parent.frame()
)

## S3 method for class 'nlmixr2FitData'
nlmixr2(
  object,
  data = NULL,
  est = NULL,
  control = NULL,
  table = tableControl(),
  ...,
  save = NULL,
  envir = parent.frame()
)
```

Arguments

`object`	Fitted object or function specifying the model.
`data`	nlmixr data
`est`	estimation method (all methods are shown by ‘nlmixr2AllEst()‘). Methods can be added for other tools
`control`	The estimation control object. These are expected to be different for each type of estimation method
`table`	The output table control object (like ‘tableControl()‘)
`...`	Other parameters
`save`	Boolean to save a nlmixr2 object in a rds file in the working directory. If `NULL`, uses option "nlmixr2.save"
`envir`	Environment where the nlmixr object/function is evaluated before running the estimation routine.

Details

The nlmixr2 generalized function allows common access to the nlmixr2 estimation routines.

The nlmixr object has the following fields:

Field	Description
conditionNumber	Condition number, that is the highest divided by the lowest eigenvalue in the population covariance matrix
cor	Correlation matrix
phiR	correlation matrix of each individual’s eta (if present)

objDF	Data frame containing objective function information (AIC, BIC, etc.)
time	Duration of different parts of the analysis (e.g. setup, optimization, calculation of covariance, etc.)
theta	Estimates for eta for each individual
etaObf	Estimates for eta for each individual, This also includes the objective function for each individual
fixef	Estimates of fixed effects
foceiControl	Estimation options if focei was used
ui	Final estimates for the model
dataMergeFull	Full data merge with the fit output and the original dataset; Also includes nlmixrLlikObs which includes t
censInfo	Gives the censorng information abot the fit (the type of censoring that was seend and handled in the datas
dataLloq	Gives the lloq from the dataset (average) when cesoring has occured; Requires the fit to have a table step
dataUloq	Gives the uloq from the dataset (average) when censoring has occured; requires the fit to have a table step
eta	IIV values for each indiviudal
dataMergeInner	Inner data merge with the fit output and the original dataset; Also includes nlmixrLlikObs which includes
rxControl	Integration options used to control rxode2
dataMergeLeft	Left data merge with the fit output and the original dataset; Also includes nlmixrLlikObs which includes t
omega	Matrix containing the estimates of the multivarte normal covariance matrix for between subject varaibiliti
covMethod	Method used to calculate covariance of the fixed effects
modelName	Name of the R object containing the model
origData	Original dataset
phiRSE	Relative standard error of each individuals eta
dataMergeRight	Right data merge with the fit output and the original dataset; Also includes nlmixrLlikObs which includes
ipredModel	rxode2 estimation model for fit (internal will likely be removed from visibility
phiSE	Standard error of each individuals eta
parFixed	Table of parameter estimates (rounded and pretty looking)
parFixedDF	Table of parameter estimates as a data frame
omegaR	The correlation matirx of omega with standard deviations for the diagonal pieces
iniUi	The initial model used to start the estimation
finalUi	The model with the estimates replaced as values
scaleInfo	The scaling factors used for nlmixr2 estimation in focei; The can be changed by foceiControl(scaleC=...)
table	These are the table options that were used when generating the table output (were CWRES included, etc
shrink	This is a table of shrinkages for all the individual ETAs as well as the variance shrinkage as well as summ
env	This is the environment where all the information for the fit is stored outside of the data-frame. It is an R
seed	This is the initial seed used for saem
simInfo	This returns a list of all the fit information used for a traditional rxode2 simulation, which you can tweak
runInfo	This returns a list of all the warnings or fit information
parHistStacked	Value of objective function and parameters at each iteration (tall format)
parHist	Value of objective function and parameters at each iteration (wide format)
cov	Variance-covariance matrix

Value

Either a nlmixr2 model or a nlmixr2 fit object

nlmixr modeling mini-language

Rationale

nlmixr estimation routines each have their own way of specifying models. Often the models are specified in ways that are most intuitive for one estimation routine, but do not make sense for another estimation routine. Sometimes, legacy estimation routines like `nlme` have their own syntax that is outside of the control of the nlmixr package.

The unique syntax of each routine makes the routines themselves easier to maintain and expand, and allows interfacing with existing packages that are outside of nlmixr (like `nlme`). However, a model definition language that is common between estimation methods, and an output object that is uniform, will make it easier to switch between estimation routines and will facilitate interfacing output with external packages like Xpose.

The nlmixr mini-modeling language, attempts to address this issue by incorporating a common language. This language is inspired by both R and NONMEM, since these languages are familiar to many pharmacometricians.

Initial Estimates and boundaries for population parameters

nlmixr models are contained in a R function with two blocks: `ini` and `model`. This R function can be named anything, but is not meant to be called directly from R. In fact if you try you will likely get an error such as `Error: could not find function "ini"`.

The `ini` model block is meant to hold the initial estimates for the model, and the boundaries of the parameters for estimation routines that support boundaries (note nlmixr's `saem` and `nlme` do not currently support parameter boundaries).

To explain how these initial estimates are specified we will start with an annotated example:

```
f <- function(){ ## Note the arguments to the function are currently
                 ## ignored by nlmixr
    ini({
        ## Initial conditions for population parameters (sometimes
        ## called theta parameters) are defined by either `<-` or '='
        lCl <- 1.6      #log Cl (L/hr)
        ## Note that simple expressions that evaluate to a number are
        ## OK for defining initial conditions (like in R)
        lVc = log(90)  #log V (L)
        ## Also a comment on a parameter is captured as a parameter label
        lKa <- 1 #log Ka (1/hr)
        ## Bounds may be specified by c(lower, est, upper), like NONMEM:
        ## Residuals errors are assumed to be population parameters
        prop.err <- c(0, 0.2, 1)
    })
    ## The model block will be discussed later
    model({})
}
```

As shown in the above examples:

- Simple parameter values are specified as a R-compatible assignment
- Boundaries my be specified by `c(lower, est, upper)`.
- Like NONMEM, `c(lower,est)` is equivalent to `c(lower,est,Inf)`
- Also like NONMEM, `c(est)` does not specify a lower bound, and is equivalent to specifying the parameter without R's 'c' function.

- The initial estimates are specified on the variance scale, and in analogy with NONMEM, the square roots of the diagonal elements correspond to coefficients of variation when used in the exponential IIV implementation

These parameters can be named almost any R compatible name. Please note that:

- Residual error estimates should be coded as population estimates (i.e. using an '=' or '<-' statement, not a '~').
- Naming variables that start with "_" are not supported. Note that R does not allow variable starting with "_" to be assigned without quoting them.
- Naming variables that start with "rx_" or "nlmixr_" is not supported since rxode2 and nlmixr2 use these prefixes internally for certain estimation routines and calculating residuals.
- Variable names are case sensitive, just like they are in R. "CL" is not the same as "Cl".

Initial Estimates for between subject error distribution (NONMEM's $OMEGA)

In mixture models, multivariate normal individual deviations from the population parameters are estimated (in NONMEM these are called eta parameters). Additionally the variance/covariance matrix of these deviations is also estimated (in NONMEM this is the OMEGA matrix). These also have initial estimates. In nlmixr these are specified by the '~' operator that is typically used in R for "modeled by", and was chosen to distinguish these estimates from the population and residual error parameters.

Continuing the prior example, we can annotate the estimates for the between subject error distribution

```
f <- function(){
    ini({
        lCl <- 1.6      #log Cl (L/hr)
        lVc = log(90)  #log V (L)
        lKa <- 1 #log Ka (1/hr)
        prop.err <- c(0, 0.2, 1)
        ## Initial estimate for ka IIV variance
        ## Labels work for single parameters
        eta.ka ~ 0.1 # BSV Ka

        ## For correlated parameters, you specify the names of each
        ## correlated parameter separated by a addition operator `+`
        ## and the left handed side specifies the lower triangular
        ## matrix initial of the covariance matrix.
        eta.cl + eta.vc ~ c(0.1,
                            0.005, 0.1)
        ## Note that labels do not currently work for correlated
        ## parameters.  Also do not put comments inside the lower
        ## triangular matrix as this will currently break the model.
    })
    ## The model block will be discussed later
    model({})
}
```

As shown in the above examples:

- Simple variances are specified by the variable name and the estimate separated by '~'.
- Correlated parameters are specified by the sum of the variable labels and then the lower triangular matrix of the covariance is specified on the left handed side of the equation. This is also separated by '~'.

Currently the model syntax does not allow comments inside the lower triangular matrix.

Model Syntax for ODE based models (NONMEM's $PK, $PRED, $DES and $ERROR)

Once the initialization block has been defined, you can define a model in terms of the defined variables in the `ini` block. You can also mix in RxODE blocks into the model.

The current method of defining a nlmixr model is to specify the parameters, and then possibly the RxODE lines:

Continuing describing the syntax with an annotated example:

```
f <- function(){
    ini({
        lCl <- 1.6      #log Cl (L/hr)
        lVc <- log(90)   #log Vc (L)
        lKA <- 0.1      #log Ka (1/hr)
        prop.err <- c(0, 0.2, 1)
        eta.Cl ~ 0.1 ## BSV Cl
        eta.Vc ~ 0.1 ## BSV Vc
        eta.KA ~ 0.1 ## BSV Ka
    })
    model({
        ## First parameters are defined in terms of the initial estimates
        ## parameter names.
        Cl <- exp(lCl + eta.Cl)
        Vc = exp(lVc + eta.Vc)
        KA <- exp(lKA + eta.KA)
        ## After the differential equations are defined
        kel <- Cl / Vc;
        d/dt(depot)    = -KA*depot;
        d/dt(centr)  =  KA*depot-kel*centr;
        ## And the concentration is then calculated
        cp = centr / Vc;
        ## Last, nlmixr is told that the plasma concentration follows
        ## a proportional error (estimated by the parameter prop.err)
        cp ~ prop(prop.err)
    })
}
```

A few points to note:

- Parameters are often defined before the differential equations.
- The differential equations, parameters and error terms are in a single block, instead of multiple sections.

- State names, calculated variables cannot start with either "rx_" or "nlmixr_" since these are used internally in some estimation routines.
- Errors are specified using the '~'. Currently you can use either add(parameter) for additive error, prop(parameter) for proportional error or add(parameter1) + prop(parameter2) for additive plus proportional error. You can also specify norm(parameter) for the additive error, since it follows a normal distribution.
- Some routines, like saem require parameters in terms of Pop.Parameter + Individual.Deviation.Parameter + Covariate*Covariate.Parameter. The order of these parameters do not matter. This is similar to NONMEM's mu-referencing, though not quite so restrictive.
- The type of parameter in the model is determined by the initial block; Covariates used in the model are missing in the ini block. These variables need to be present in the modeling dataset for the model to run.

Model Syntax for solved PK systems

Solved PK systems are also currently supported by nlmixr with the 'linCmt()' pseudo-function. An annotated example of a solved system is below:

##'

```
f <- function(){
    ini({
        lCl <- 1.6      #log Cl (L/hr)
        lVc <- log(90)   #log Vc (L)
        lKA <- 0.1      #log Ka (1/hr)
        prop.err <- c(0, 0.2, 1)
        eta.Cl ~ 0.1 ## BSV Cl
        eta.Vc ~ 0.1 ## BSV Vc
        eta.KA ~ 0.1 ## BSV Ka
    })
    model({
        Cl <- exp(lCl + eta.Cl)
        Vc = exp(lVc + eta.Vc)
        KA <- exp(lKA + eta.KA)
        ## Instead of specifying the ODEs, you can use
        ## the linCmt() function to use the solved system.
        ##
        ## This function determines the type of PK solved system
        ## to use by the parameters that are defined.  In this case
        ## it knows that this is a one-compartment model with first-order
        ## absorption.
        linCmt() ~ prop(prop.err)
    })
}
```

A few things to keep in mind:

- While RxODE allows mixing of solved systems and ODEs, this has not been implemented in nlmixr yet.

- The solved systems implemented are the one, two and three compartment models with or without first-order absorption. Each of the models support a lag time with a tlag parameter.
- In general the linear compartment model figures out the model by the parameter names. nlmixr currently knows about numbered volumes, Vc/Vp, Clearances in terms of both Cl and Q/CLD. Additionally nlmixr knows about elimination micro-constants (ie K12). Mixing of these parameters for these models is currently not supported.

Checking model syntax

After specifying the model syntax you can check that nlmixr is interpreting it correctly by using the `nlmixr` function on it.

Using the above function we can get:

```
> nlmixr(f)
## 1-compartment model with first-order absorption in terms of Cl
## Initialization:
################################################################################
Fixed Effects ($theta):
    lCl     lVc     lKA
1.60000 4.49981 0.10000

Omega ($omega):
     [,1] [,2] [,3]
[1,]  0.1  0.0  0.0
[2,]  0.0  0.1  0.0
[3,]  0.0  0.0  0.1

## Model:
################################################################################
Cl <- exp(lCl + eta.Cl)
Vc = exp(lVc + eta.Vc)
KA <- exp(lKA + eta.KA)
## Instead of specifying the ODEs, you can use
## the linCmt() function to use the solved system.
##
## This function determines the type of PK solved system
## to use by the parameters that are defined.  In this case
## it knows that this is a one-compartment model with first-order
## absorption.
linCmt() ~ prop(prop.err)
```

In general this gives you information about the model (what type of solved system/RxODE), initial estimates as well as the code for the model block.

Using the model syntax for estimating a model

Once the model function has been created, you can use it and a dataset to estimate the parameters for a model given a dataset.

This dataset has to have RxODE compatible events IDs. Both Monolix and NONMEM use a a very similar standard to what nlmixr can support.

Once the data has been converted to the appropriate format, you can use the `nlmixr` function to run the appropriate code.

The method to estimate the model is:

```
fit <- nlmixr(model.function, dataset, est="est", control=estControl(options))
```

Currently `nlme` and `saem` are implemented. For example, to run the above model with `saem`, we could have the following:

```
> f <- function(){
    ini({
        lCl <- 1.6      #log Cl (L/hr)
        lVc <- log(90)   #log Vc (L)
        lKA <- 0.1      #log Ka (1/hr)
        prop.err <- c(0, 0.2, 1)
        eta.Cl ~ 0.1 ## BSV Cl
        eta.Vc ~ 0.1 ## BSV Vc
        eta.KA ~ 0.1 ## BSV Ka
    })
    model({
        ## First parameters are defined in terms of the initial estimates
        ## parameter names.
        Cl <- exp(lCl + eta.Cl)
        Vc = exp(lVc + eta.Vc)
        KA <- exp(lKA + eta.KA)
        ## After the differential equations are defined
        kel <- Cl / Vc;
        d/dt(depot)    = -KA*depot;
        d/dt(centr)  =  KA*depot-kel*centr;
        ## And the concentration is then calculated
        cp = centr / Vc;
        ## Last, nlmixr is told that the plasma concentration follows
        ## a proportional error (estimated by the parameter prop.err)
        cp ~ prop(prop.err)
    })
}
> fit.s <- nlmixr(f,d,est="saem",control=saemControl(n.burn=50,n.em=100,print=50));
Compiling RxODE differential equations...done.
c:/Rtools/mingw_64/bin/g++  -I"c:/R/R-34~1.1/include" -DNDEBUG    -I"d:/Compiler/gcc-4.9.3/local330/i
In file included from c:/R/R-34~1.1/library/RCPPAR~1/include/armadillo:52:0,
           from c:/R/R-34~1.1/library/RCPPAR~1/include/RcppArmadilloForward.h:46,
                from c:/R/R-34~1.1/library/RCPPAR~1/include/RcppArmadillo.h:31,
                 from saem3090757b4bd1x64.cpp:1:
c:/R/R-34~1.1/library/RCPPAR~1/include/armadillo_bits/compiler_setup.hpp:474:96: note: #pragma messa
  #pragma message ("WARNING: use of OpenMP disabled; this compiler doesn't support OpenMP 3.0+")
                                                                                                ^
c:/Rtools/mingw_64/bin/g++ -shared -s -static-libgcc -o saem3090757b4bd1x64.dll tmp.def saem3090757b4b
done.
```

```
1:    1.8174   4.6328   0.0553   0.0950   0.0950   0.0950   0.6357
50:    1.3900   4.2039   0.0001   0.0679   0.0784   0.1082   0.1992
100:    1.3894   4.2054   0.0107   0.0686   0.0777   0.1111   0.1981
150:    1.3885   4.2041   0.0089   0.0683   0.0778   0.1117   0.1980
Using sympy via SnakeCharmR
## Calculate ETA-based prediction and error derivatives:
Calculate Jacobian...................done.
Calculate sensitivities.......
done.
## Calculate d(f)/d(eta)
## ...
## done
## ...
## done
The model-based sensitivities have been calculated
Calculating Table Variables...
done
```

The options for `saem` are controlled by `saemControl`. You may wish to make sure the minimization is complete in the case of `saem`. You can do that with `traceplot` which shows the iteration history with the divided by burn-in and EM phases. In this case, the burn in seems reasonable; you may wish to increase the number of iterations in the EM phase of the estimation. Overall it is probably a semi-reasonable solution.

nlmixr output objects

In addition to unifying the modeling language sent to each of the estimation routines, the outputs currently have a unified structure.

You can see the fit object by typing the object name:

```
> fit.s
 -- nlmixr SAEM fit (ODE); OBJF calculated from FOCEi approximation -------------
      OBJF      AIC      BIC Log-likelihood Condition Number
  62337.09 62351.09 62399.01      -31168.55          82.6086

 -- Time (sec; fit.s$time): -----------------------------------------------------
           saem setup Likelihood Calculation covariance table
 elapsed 430.25 31.64                   1.19          0  3.44

 -- Parameters (fit.s$par.fixed): -----------------------------------------------
              Parameter Estimate     SE
 lCl      log Cl (L/hr)     1.39 0.0240  1.73       4.01 (3.83, 4.20)    26.6
 lVc         log Vc (L)     4.20 0.0256 0.608       67.0 (63.7, 70.4)    28.5
 lKA      log Ka (1/hr)  0.00924 0.0323  349.      1.01 (0.947, 1.08)    34.3
 prop.err      prop.err    0.198                             19.8
         Shrink(SD)
 lCl         0.248
 lVc          1.09
 lKA          4.19
 prop.err     1.81
```

```
  No correlations in between subject variability (BSV) matrix
 Full BSV covariance (fit.s$omega) or correlation (fit.s$omega.R; diagonals=SDs)
  Distribution stats (mean/skewness/kurtosis/p-value) available in fit.s$shrink

-- Fit Data (object fit.s is a modified data.frame): ---------------------------
# A tibble: 6,947 x 22
  ID     TIME    DV  PRED    RES    WRES IPRED  IRES  IWRES CPRED   CRES
* <fct> <dbl> <dbl> <dbl>  <dbl>   <dbl> <dbl> <dbl>  <dbl> <dbl>  <dbl>
1 1      0.25  205.  198.   6.60  0.0741  189.  16.2  0.434  198.   6.78
2 1      0.5   311.  349. -38.7  -0.261   330. -19.0 -0.291  349. -38.3
3 1      0.75  389.  464. -74.5  -0.398   434. -45.2 -0.526  463. -73.9
# ... with 6,944 more rows, and 11 more variables: CWRES <dbl>, eta.Cl <dbl>,
#   eta.Vc <dbl>, eta.KA <dbl>, depot <dbl>, centr <dbl>, Cl <dbl>, Vc <dbl>,
#   KA <dbl>, kel <dbl>, cp <dbl>
```

This example shows what is typical printout of a nlmixr fit object. The elements of the fit are:

- The type of fit (`nlme`, `saem`, etc)
- Metrics of goodness of fit (`AIC`, `BIC`, and `logLik`).
  - To align the comparison between methods, the FOCEi likelihood objective is calculated regardless of the method used and used for goodness of fit metrics.
  - This FOCEi likelihood has been compared to NONMEM's objective function and gives the same values (based on the data in Wang 2007)
  - Also note that `saem` does not calculate an objective function, and the FOCEi is used as the only objective function for the fit.
  - Even though the objective functions are calculated in the same manner, caution should be used when comparing fits from various estimation routines.
- The next item is the timing of each of the steps of the fit.
  - These can be also accessed by (`fit.s$time`).
  - As a mnemonic, the access for this item is shown in the printout. This is true for almost all of the other items in the printout.
- After the timing of the fit, the parameter estimates are displayed (can be accessed by `fit.s$par.fixed`)
  - While the items are rounded for R printing, each estimate without rounding is still accessible by the '$' syntax. For example, the '$Untransformed' gives the untransformed parameter values.
  - The Untransformed parameter takes log-space parameters and back-transforms them to normal parameters. Not the CIs are listed on the back-transformed parameter space.
  - Proportional Errors are converted to
- Omega block (accessed by `fit.s$omega`)
- The table of fit data. Please note:
  - A nlmixr fit object is actually a data frame. Saving it as a Rdata object and then loading it without nlmixr will just show the data by itself. Don't worry; the fit information has not vanished, you can bring it back by simply loading nlmixr, and then accessing the data.
  - Special access to fit information (like the `$omega`) needs nlmixr to extract the information.

- If you use the $ to access information, the order of precedence is:
  * Fit data from the overall data.frame
  * Information about the parsed nlmixr model (via `$uif`)
  * Parameter history if available (via `$par.hist` and `$par.hist.stacked`)
  * Fixed effects table (via `$par.fixed`)
  * Individual differences from the typical population parameters (via `$eta`)
  * Fit information from the list of information generated during the post-hoc residual calculation.
  * Fit information from the environment where the post-hoc residual were calculated
  * Fit information about how the data and options interacted with the specified model (such as estimation options or if the solved system is for an infusion or an IV bolus).
- While the printout may displays the data as a `data.table` object or `tbl` object, the data is NOT any of these objects, but rather a derived data frame.
- Since the object *is* a data.frame, you can treat it like one.

In addition to the above properties of the fit object, there are a few additional that may be helpful for the modeler:

- `$theta` gives the fixed effects parameter estimates (in NONMEM the `thetas`). This can also be accessed in `fixed.effects` function. Note that the residual variability is treated as a fixed effect parameter and is included in this list.
- `$eta` gives the random effects parameter estimates, or in NONMEM the `etas`. This can also be accessed in using the `random.effects` function.

Author(s)

Matthew L. Fidler

Examples

```
one.cmt <- function() {
 ini({
   ## You may label each parameter with a comment
   tka <- 0.45 # Ka
   tcl <- log(c(0, 2.7, 100)) # Log Cl
   ## This works with interactive models
   ## You may also label the preceding line with label("label text")
   tv <- 3.45; label("log V")
   ## the label("Label name") works with all models
   eta.ka ~ 0.6
   eta.cl ~ 0.3
   eta.v ~ 0.1
   add.sd <- 0.7
   prop.sd <- 0.01
 })
 model({
```

```
    ka <- exp(tka + eta.ka)
    cl <- exp(tcl + eta.cl)
    v <- exp(tv + eta.v)
    linCmt() ~ add(add.sd) + prop(prop.sd)
  })
}

# fitF <- nlmixr(one.cmt, theo_sd, "focei")

fitS <- nlmixr(one.cmt, theo_sd, "saem")
```

`nlmixr2AllEst` *Show all the current estimation methods*

Description

Show all the current estimation methods

Usage

```
nlmixr2AllEst()
```

Value

List of supported nlmixr2 estimation options (est=...)

Examples

```
nlmixr2AllEst()
```

`nlmixr2AugPredSolve` *Augmented Prediction for nlmixr2 fit*

Description

Augmented Prediction for nlmixr2 fit

Usage

```
nlmixr2AugPredSolve(
  fit,
  covsInterpolation = c("locf", "nocb", "linear", "midpoint"),
  minimum = NULL,
  maximum = NULL,
  length.out = 51L,
  ...
)

## S3 method for class 'nlmixr2FitData'
augPred(
  object,
  primary = NULL,
  minimum = NULL,
  maximum = NULL,
  length.out = 51,
  ...
)
```

Arguments

`fit` Nlmixr2 fit object

`covsInterpolation`
: specifies the interpolation method for time-varying covariates. When solving ODEs it often samples times outside the sampling time specified in `events`. When this happens, the time varying covariates are interpolated. Currently this can be:

- `"linear"` interpolation, which interpolates the covariate by solving the line between the observed covariates and extrapolating the new covariate value.
- `"constant"` – Last observation carried forward (the default).
- `"NOCB"` – Next Observation Carried Backward. This is the same method that NONMEM uses.
- `"midpoint"` Last observation carried forward to midpoint; Next observation carried backward to midpoint.

`minimum` an optional lower limit for the primary covariate. Defaults to `min(primary)`.

`maximum` an optional upper limit for the primary covariate. Defaults to `max(primary)`.

`length.out` an optional integer with the number of primary covariate values at which to evaluate the predictions. Defaults to 51.

`...` some methods for the generic may require additional arguments.

`object` a fitted model object from which predictions can be extracted, using a `predict` method.

`primary` an optional one-sided formula specifying the primary covariate to be used to generate the augmented predictions. By default, if a covariate can be extracted from the data used to generate `object` (using `getCovariate`), it will be used as `primary`.

Value

Stacked data.frame with observations, individual/population predictions.

Author(s)

Matthew L. Fidler

`nlmixr2CreateOutputFromUi`
Create nlmixr output from the UI

Description

Create nlmixr output from the UI

Usage

```
nlmixr2CreateOutputFromUi(
  ui,
  data = NULL,
  control = NULL,
  table = NULL,
  env = NULL,
  est = "none"
)
```

Arguments

`ui`	This is the UI that will be used for the translation
`data`	This has the data
`control`	focei control for data creation
`table`	Table options
`env`	Environment setup which needs the following: - '$table' for table options - '$origData' – Original Data - '$dataSav' – Processed data from .foceiPreProcessData - '$idLvl' – Level information for ID factor added - '$covLvl' – Level information for items to convert to factor - '$ui' for ui object - '$fullTheta' Full theta information - '$etaObf' data frame with ID, etas and OBJI - '$cov' For covariance - '$covMethod' for the method of calculating the covariance - '$adjObf' Should the objective function value be adjusted - '$objective' objective function value - '$extra' Extra print information - '$method' Estimation method (for printing) - '$omega' Omega matrix - '$theta' Is a theta data frame - '$model' a list of model information for table generation. Needs a 'predOnly' model - '$message' Message for display - '$est' estimation method - '$ofvType' (optional) tells the type of ofv is currently being use There are some more details that need to be described here
`est`	Estimation method

Value

nlmixr fit object

Author(s)

Matthew L. Fidler

---

nlmixr2Est.bobyqa *Generic for nlmixr2 estimation methods*

Description

Generic for nlmixr2 estimation methods

Usage

```
## S3 method for class 'bobyqa'
nlmixr2Est(env, ...)

## S3 method for class 'focei'
nlmixr2Est(env, ...)

## S3 method for class 'foce'
nlmixr2Est(env, ...)

## S3 method for class 'posthoc'
nlmixr2Est(env, ...)

## S3 method for class 'foi'
nlmixr2Est(env, ...)

## S3 method for class 'fo'
nlmixr2Est(env, ...)

## S3 method for class 'output'
nlmixr2Est(env, ...)

## S3 method for class 'lbfgsb3c'
nlmixr2Est(env, ...)

## S3 method for class 'n1qn1'
nlmixr2Est(env, ...)

## S3 method for class 'newuoa'
nlmixr2Est(env, ...)

## S3 method for class 'nlm'
```

```
nlmixr2Est(env, ...)

## S3 method for class 'nlme'
nlmixr2Est(env, ...)

## S3 method for class 'nlminb'
nlmixr2Est(env, ...)

nlmixr2Est(env, ...)

## Default S3 method:
nlmixr2Est(env, ...)

## S3 method for class 'nls'
nlmixr2Est(env, ...)

## S3 method for class 'optim'
nlmixr2Est(env, ...)

## S3 method for class 'rxSolve'
nlmixr2Est(env, ...)

## S3 method for class 'simulate'
nlmixr2Est(env, ...)

## S3 method for class 'simulation'
nlmixr2Est(env, ...)

## S3 method for class 'predict'
nlmixr2Est(env, ...)

## S3 method for class 'saem'
nlmixr2Est(env, ...)

## S3 method for class 'uobyqa'
nlmixr2Est(env, ...)
```

Arguments

`env`	Environment for the nlmixr2 estimation routines. This needs to have: - rxode2 ui object in '$ui' - data to fit in the estimation routine in '$data' - control for the estimation routine's control options in '$ui'
`...`	Other arguments provided to 'nlmixr2Est()' provided for flexibility but not currently used inside nlmixr

Details

This is a S3 generic that allows others to use the nlmixr2 environment to do their own estimation routines

Value

nlmixr2 fit object

Author(s)

Matthew Fidler

`nlmixr2Gill83` *Get the optimal forward difference interval by Gill83 method*

Description

Get the optimal forward difference interval by Gill83 method

Usage

```
nlmixr2Gill83(
  what,
  args,
  envir = parent.frame(),
  which,
  gillRtol = sqrt(.Machine$double.eps),
  gillK = 10L,
  gillStep = 2,
  gillFtol = 0
)
```

Arguments

`what`	either a function or a non-empty character string naming the function to be called.
`args`	a *list* of arguments to the function call. The `names` attribute of `args` gives the argument names.
`envir`	an environment within which to evaluate the call. This will be most useful if `what` is a character string and the arguments are symbols or quoted expressions.
`which`	Which parameters to calculate the forward difference and optimal forward difference interval
`gillRtol`	The relative tolerance used for Gill 1983 determination of optimal step size.
`gillK`	The total number of possible steps to determine the optimal forward/central difference step size per parameter (by the Gill 1983 method). If 0, no optimal step size is determined. Otherwise this is the optimal step size determined.

`gillStep` When looking for the optimal forward difference step size, this is This is the step size to increase the initial estimate by. So each iteration the new step size = (prior step size)*gillStep

`gillFtol` The gillFtol is the gradient error tolerance that is acceptable before issuing a warning/error about the gradient estimates.

Value

A data frame with the following columns:

- info Gradient evaluation/forward difference information
- hf Forward difference final estimate
- df Derivative estimate
- df2 2nd Derivative Estimate
- err Error of the final estimate derivative
- aEps Absolute difference for forward numerical differences
- rEps Relative Difference for backward numerical differences
- aEpsC Absolute difference for central numerical differences
- rEpsC Relative difference for central numerical differences

The `info` returns one of the following:

- "Not Assessed" Gradient wasn't assessed
- "Good Success" in Estimating optimal forward difference interval
- "High Grad Error" Large error; Derivative estimate error `fTol` or more of the derivative
- "Constant Grad" Function constant or nearly constant for this parameter
- "Odd/Linear Grad" Function odd or nearly linear, df = K, df2 ~ 0
- "Grad changes quickly" df2 increases rapidly as h decreases

Author(s)

Matthew Fidler

Examples

```
## These are taken from the numDeriv::grad examples to show how
## simple gradients are assessed with nlmixr2Gill83

nlmixr2Gill83(sin, pi)

nlmixr2Gill83(sin, (0:10)*2*pi/10)

func0 <- function(x){ sum(sin(x))  }
nlmixr2Gill83(func0 , (0:10)*2*pi/10)

func1 <- function(x){ sin(10*x) - exp(-x) }
curve(func1,from=0,to=5)
```

```
x <- 2.04
numd1 <- nlmixr2Gill83(func1, x)
exact <- 10*cos(10*x) + exp(-x)
c(numd1$df, exact, (numd1$df - exact)/exact)

x <- c(1:10)
numd1 <- nlmixr2Gill83(func1, x)
exact <- 10*cos(10*x) + exp(-x)
cbind(numd1=numd1$df, exact, err=(numd1$df - exact)/exact)

sc2.f <- function(x){
  n <- length(x)
   sum((1:n) * (exp(x) - x)) / n
}

sc2.g <- function(x){
  n <- length(x)
  (1:n) * (exp(x) - 1) / n
}

x0 <- rnorm(100)
exact <- sc2.g(x0)

g <- nlmixr2Gill83(sc2.f, x0)

max(abs(exact - g$df)/(1 + abs(exact)))
```

`nlmixr2Hess` *Calculate Hessian*

Description

Unlike 'stats::optimHess' which assumes the gradient is accurate, nlmixr2Hess does not make as strong an assumption that the gradient is accurate but takes more function evaluations to calculate the Hessian. In addition, this procedures optimizes the forward difference interval by `nlmixr2Gill83`

Usage

```
nlmixr2Hess(par, fn, ..., envir = parent.frame())
```

Arguments

`par`	Initial values for the parameters to be optimized over.
`fn`	A function to be minimized (or maximized), with first argument the vector of parameters over which minimization is to take place. It should return a scalar result.
`...`	Extra arguments sent to `nlmixr2Gill83`

envir — an environment within which to evaluate the call. This will be most useful if `what` is a character string and the arguments are symbols or quoted expressions.

Details

If you have an analytical gradient function, you should use ‘stats::optimHess‘

Value

Hessian matrix based on Gill83

Author(s)

Matthew Fidler

See Also

nlmixr2Gill83, optimHess

Examples

```
 func0 <- function(x){ sum(sin(x))  }
 x <- (0:10)*2*pi/10
 nlmixr2Hess(x, func0)

fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
    x1 <- x[1]
    x2 <- x[2]
    c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
       200 *      (x2 - x1 * x1))
}

h1 <- optimHess(c(1.2,1.2), fr, grr)

h2 <- optimHess(c(1.2,1.2), fr)

## in this case h3 is closer to h1 where the gradient is known

h3 <- nlmixr2Hess(c(1.2,1.2), fr)
```

`nlmixr2Keywords` *A list and description of the fields in the nlmxir2 object*

Description

A list and description of the fields in the nlmxir2 object

Usage

```
nlmixr2Keywords
```

Format

A data frame with 2 columns and 40 or more rows

Field Name of the field in the nlmixr2 object

Description Description of the information in the field

`nlmixr2Logo` *Messages the nlmixr2 logo...*

Description

Messages the nlmixr2 logo...

Usage

```
nlmixr2Logo(str = "", version = sessionInfo()$otherPkgs$nlmixr2$Version)
```

Arguments

`str`	String to print
`version`	Version information (by default use package version)

Value

nothing; Called to display version information

Author(s)

Matthew L. Fidler

`nlmixr2NlmeControl` *Control Values for nlme Fit with extra options for nlmixr*

Description

The values supplied in the function call replace the defaults and a list with all possible arguments is returned. The returned list is used as the 'control' argument to the 'nlme' function.

Usage

```
nlmixr2NlmeControl(
  maxIter = 100,
  pnlsMaxIter = 100,
  msMaxIter = 100,
  minScale = 0.001,
  tolerance = 1e-05,
  niterEM = 25,
  pnlsTol = 0.001,
  msTol = 1e-06,
  returnObject = FALSE,
  msVerbose = FALSE,
  msWarnNoConv = TRUE,
  gradHess = TRUE,
  apVar = TRUE,
  .relStep = .Machine$double.eps^(1/3),
  minAbsParApVar = 0.05,
  opt = c("nlminb", "nlm"),
  natural = TRUE,
  sigma = NULL,
  optExpression = TRUE,
  literalFix = TRUE,
  sumProd = FALSE,
  rxControl = NULL,
  method = c("ML", "REML"),
  random = NULL,
  fixed = NULL,
  weights = NULL,
  verbose = TRUE,
  returnNlme = FALSE,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
  compress = TRUE,
  adjObf = TRUE,
  ci = 0.95,
  sigdig = 4,
  sigdigTable = NULL,
  muRefCovAlg = TRUE,
```

```
  ...
)

nlmeControl(
  maxIter = 100,
  pnlsMaxIter = 100,
  msMaxIter = 100,
  minScale = 0.001,
  tolerance = 1e-05,
  niterEM = 25,
  pnlsTol = 0.001,
  msTol = 1e-06,
  returnObject = FALSE,
  msVerbose = FALSE,
  msWarnNoConv = TRUE,
  gradHess = TRUE,
  apVar = TRUE,
  .relStep = .Machine$double.eps^(1/3),
  minAbsParApVar = 0.05,
  opt = c("nlminb", "nlm"),
  natural = TRUE,
  sigma = NULL,
  optExpression = TRUE,
  literalFix = TRUE,
  sumProd = FALSE,
  rxControl = NULL,
  method = c("ML", "REML"),
  random = NULL,
  fixed = NULL,
  weights = NULL,
  verbose = TRUE,
  returnNlme = FALSE,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
  compress = TRUE,
  adjObf = TRUE,
  ci = 0.95,
  sigdig = 4,
  sigdigTable = NULL,
  muRefCovAlg = TRUE,
  ...
)
```

Arguments

maxIter	maximum number of iterations for the nlme optimization algorithm. Default is 50.
pnlsMaxIter	maximum number of iterations for the PNLS optimization step inside the nlme

	optimization. Default is 7.
msMaxIter	maximum number of iterations for nlminb (iter.max) or the nlm (iterlim, from the 10-th step) optimization step inside the nlme optimization. Default is 50 (which may be too small for e.g. for overparametrized cases).
minScale	minimum factor by which to shrink the default step size in an attempt to decrease the sum of squares in the PNLS step. Default 0.001.
tolerance	tolerance for the convergence criterion in the nlme algorithm. Default is 1e-6.
niterEM	number of iterations for the EM algorithm used to refine the initial estimates of the random effects variance-covariance coefficients. Default is 25.
pnlsTol	tolerance for the convergence criterion in PNLS step. Default is 1e-3.
msTol	tolerance for the convergence criterion in nlm, passed as the gradtol argument to the function (see documentation on nlm). Default is 1e-7.
returnObject	a logical value indicating whether the fitted object should be returned when the maximum number of iterations is reached without convergence of the algorithm. Default is FALSE.
msVerbose	a logical value passed as the trace to nlminb(.., control= list(trace = *, ..)) or as argument print.level to nlm(). Default is FALSE.
msWarnNoConv	logical indicating if a warning should be signalled whenever the minimization (by opt) in the LME step does not converge; defaults to TRUE.
gradHess	a logical value indicating whether numerical gradient vectors and Hessian matrices of the log-likelihood function should be used in the nlm optimization. This option is only available when the correlation structure (corStruct) and the variance function structure (varFunc) have no "varying" parameters and the pdMat classes used in the random effects structure are pdSymm (general positive-definite), pdDiag (diagonal), pdIdent (multiple of the identity), or pdCompSymm (compound symmetry). Default is TRUE.
apVar	a logical value indicating whether the approximate covariance matrix of the variance-covariance parameters should be calculated. Default is TRUE.
.relStep	relative step for numerical derivatives calculations. Default is .Machine$double.eps^(1/3).
minAbsParApVar	numeric value - minimum absolute parameter value in the approximate variance calculation. The default is 0.05.
opt	the optimizer to be used, either "nlminb" (the default) or "nlm".
natural	a logical value indicating whether the pdNatural parametrization should be used for general positive-definite matrices (pdSymm) in reStruct, when the approximate covariance matrix of the estimators is calculated. Default is TRUE.
sigma	optionally a positive number to fix the residual error at. If NULL, as by default, or 0, sigma is estimated.
optExpression	Optimize the rxode2 expression to speed up calculation. By default this is turned on.
literalFix	boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is ‘TRUE‘.
sumProd	Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is FALSE.

`rxControl` ‘rxode2‘ ODE solving options during fitting, created with ‘rxControl()‘

`method` a character string. If `"REML"` the model is fit by maximizing the restricted log-likelihood. If `"ML"` the log-likelihood is maximized. Defaults to `"ML"`.

`random` optionally, any of the following: (i) a two-sided formula of the form `r1+...+rn~x1+...+xm | g1/.../gQ`, with `r1,...,rn` naming parameters included on the right hand side of `model`, `x1+...+xm` specifying the random-effects model for these parameters and `g1/.../gQ` the grouping structure (`Q` may be equal to 1, in which case no `/` is required). The random effects formula will be repeated for all levels of grouping, in the case of multiple levels of grouping; (ii) a two-sided formula of the form `r1+...+rn~x1+..+xm`, a list of two-sided formulas of the form `r1~x1+...+xm`, with possibly different random-effects models for different parameters, a `pdMat` object with a two-sided formula, or list of two-sided formulas (i.e. a non-`NULL` value for `formula(random)`), or a list of pdMat objects with two-sided formulas, or lists of two-sided formulas. In this case, the grouping structure formula will be given in `groups`, or derived from the data used to fit the nonlinear mixed-effects model, which should inherit from class `groupedData`,; (iii) a named list of formulas, lists of formulas, or `pdMat` objects as in (ii), with the grouping factors as names. The order of nesting will be assumed the same as the order of the order of the elements in the list; (iv) an `reStruct` object. See the documentation on `pdClasses` for a description of the available `pdMat` classes. Defaults to `fixed`, resulting in all fixed effects having also random effects.

`fixed` a two-sided linear formula of the form `f1+...+fn~x1+...+xm`, or a list of two-sided formulas of the form `f1~x1+...+xm`, with possibly different models for different parameters. The `f1,...,fn` are the names of parameters included on the right hand side of `model` and the `x1+...+xm` expressions define linear models for these parameters (when the left hand side of the formula contains several parameters, they all are assumed to follow the same linear model, described by the right hand side expression). A `1` on the right hand side of the formula(s) indicates a single fixed effects for the corresponding parameter(s).

`weights` an optional `varFunc` object or one-sided formula describing the within-group heteroscedasticity structure. If given as a formula, it is used as the argument to `varFixed`, corresponding to fixed variance weights. See the documentation on `varClasses` for a description of the available `varFunc` classes. Defaults to `NULL`, corresponding to homoscedastic within-group errors.

`verbose` an optional logical value. If `TRUE` information on the evolution of the iterative algorithm is printed. Default is `FALSE`.

`returnNlme` Returns the nlme object instead of the nlmixr object (by default FALSE). If any of the nlme specific options of ‘random‘, ‘fixed‘, ‘sens‘, the nlme object is returned

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

Where:
- y represents the observed value
- f represents the predicted value
- a is the additive standard deviation
- b is the proportional/power standard deviation
- c is the power exponent (in the proportional case c=1)

`calcTables` This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`

`compress` Should the object have compressed items

`adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM's default objective function. By default this is `TRUE`

`ci` Confidence level for some tables. By default this is 0.95 or 95% confidence.

`sigdig` Optimization significant digits. This controls:

- The tolerance of the inner and outer optimization is `10^-sigdig`
- The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)
- The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`

`sigdigTable` Significant digits in the final output table. If not specified, then it matches the significant digits in the ‘sigdig‘ optimization algorithm. If ‘sigdig‘ is NULL, use 3.

`muRefCovAlg` This controls if algebraic expressions that can be mu-referenced are treated as mu-referenced covariates by:

1. Creating a internal data-variable ‘nlmixrMuDerCov#‘ for each algebraic mu-referenced expression

2. Change the algebraic expression to ‘nlmixrMuDerCov# * mu_cov_theta‘

3. Use the internal mu-referenced covariate for saem

4. After optimization is completed, replace ‘model()‘ with old ‘model()‘ expression

5. Remove ‘nlmixrMuDerCov#‘ from nlmix2 output

In general, these covariates should be more accurate since it changes the system to a linear compartment model. Therefore, by default this is ‘TRUE‘.

`...` Further, named control arguments to be passed to `nlminb` (apart from `trace` and `iter.max` mentioned above), where used (`eval.max` and those from `abs.tol` down).

Value

a nlmixr-nlme list

See Also

Other Estimation control: `foceiControl()`, `saemControl()`

Examples

```
nlmeControl()
nlmixr2NlmeControl()
```

`nlmixr2Validate` *Validate nlmixr2*

Description

This allows easy validation/qualification of nlmixr2 by running the testing suite on your system.

Usage

```
nlmixr2Validate(type = NULL, skipOnCran = TRUE)

nmTest(type = NULL, skipOnCran = TRUE)
```

Arguments

`type`	of test to be run
`skipOnCran`	when ‘TRUE‘ skip the test on CRAN.

Value

Nothing, called for its side effects

Author(s)

Matthew L. Fidler

`nlmixr2Version` *Display nlmixr2’s version*

Description

Display nlmixr2’s version

Usage

```
nlmixr2Version()
```

Value

Nothing, called for its side effects

Author(s)

Matthew L. Fidler

nlmixrAddObjectiveFunctionDataFrame

Add objective function data frame to the current objective function

Description

Add objective function data frame to the current objective function

Usage

```
nlmixrAddObjectiveFunctionDataFrame(fit, objDf, type, etaObf = NULL)
```

Arguments

`fit`	nlmixr fit object
`objDf`	nlmixr objective function data frame which has column names "OBJF", "AIC", "BIC", "Log-likelihood" and "Condition#(Cov)" "Condition#(Cor)"
`type`	Objective Function Type
`etaObf`	Eta objective function table to add (with focei) to give focei objective function

Value

Nothing, called for side effects

Author(s)

Matthew L. Fidler

nlmixrAddTiming

Manually add time to a nlmixr2 object

Description

Manually add time to a nlmixr2 object

Usage

```
nlmixrAddTiming(object, name, time)
```

Arguments

`object`	nlmixr2 object
`name`	string of the timing name
`time`	time (in seconds)

Value

Nothing, called for side effects

Author(s)

Matthew L. Fidler

Examples

```
one.cmt <- function() {
 ini({
   ## You may label each parameter with a comment
   tka <- 0.45 # Ka
   tcl <- log(c(0, 2.7, 100)) # Log Cl
   ## This works with interactive models
   ## You may also label the preceding line with label("label text")
   tv <- 3.45; label("log V")
   ## the label("Label name") works with all models
   eta.ka ~ 0.6
   eta.cl ~ 0.3
   eta.v ~ 0.1
   add.sd <- 0.7
 })
 model({
   ka <- exp(tka + eta.ka)
   cl <- exp(tcl + eta.cl)
   v <- exp(tv + eta.v)
   linCmt() ~ add(add.sd)
 })
}

fit <- nlmixr(one.cmt, theo_sd, est="saem")

# will add to the current setup
nlmixrAddTiming(fit, "setup", 3)

# Add a new item to the timing dataframe
nlmixrAddTiming(fit, "new", 3)
```

---

nlmixrCbind *nlmixrCbind*

Description

'cbind' for 'nlmixr' objects that preserve the fit information

Usage

```
nlmixrCbind(fit, extra)
```

Arguments

`fit`	nlmixr fit
`extra`	data to cbind to nlmixr fit

Value

fit expanded with extra values, without disturbing the fit information

Author(s)

Matthew L. Fidler

`nlmixrClone` *Clone nlmixr environment*

Description

Clone nlmixr environment

Usage

```
nlmixrClone(x)
```

Arguments

`x`	nlmixr fit

Value

cloned nlmixr environment

Author(s)

Matthew L. Fidler

Examples

```
## Not run:

one.cmt <- function() {
  ini({
    ## You may label each parameter with a comment
    tka <- 0.45 # Log Ka
    tcl <- log(c(0, 2.7, 100)) # Log Cl
    ## This works with interactive models
    ## You may also label the preceding line with label("label text")
    tv <- 3.45; label("log V")
    ## the label("Label name") works with all models
    eta.ka ~ 0.6
    eta.cl ~ 0.3
    eta.v ~ 0.1
    add.sd <- 0.7
  })
  model({
    ka <- exp(tka + eta.ka)
    cl <- exp(tcl + eta.cl)
    v <- exp(tv + eta.v)
    linCmt() ~ add(add.sd)
  })
}

f <- nlmixr2(one.cmt, theo_sd, "saem")

nlmixrClone(f)


## End(Not run)
```

---

nlmixrWithTiming *Time a part of a nlmixr operation and add to nlmixr object*

Description

Time a part of a nlmixr operation and add to nlmixr object

Usage

```
nlmixrWithTiming(name, code, envir = NULL)
```

Arguments

name	Name of the timing to be integrated
code	Code to be evaluated and timed
envir	can be either the nlmixr2 fit data, the nlmixr2 fit environment or NULL, which implies it is going to be added to the nlmixr fit when it is finalized. If the function is being called after a fit is created, please supply this environmental variable

Value

Result of code

Author(s)

Matthew L. Fidler

Examples

```
one.cmt <- function() {
 ini({
   ## You may label each parameter with a comment
   tka <- 0.45 # Ka
   tcl <- log(c(0, 2.7, 100)) # Log Cl
   ## This works with interactive models
   ## You may also label the preceding line with label("label text")
   tv <- 3.45; label("log V")
   ## the label("Label name") works with all models
   eta.ka ~ 0.6
   eta.cl ~ 0.3
   eta.v ~ 0.1
   add.sd <- 0.7
 })
 model({
   ka <- exp(tka + eta.ka)
   cl <- exp(tcl + eta.cl)
   v <- exp(tv + eta.v)
   linCmt() ~ add(add.sd)
 })
}
fit <- nlmixr(one.cmt, theo_sd, est="saem")

nlmixrWithTiming("time1", {
   Sys.sleep(1)
   # note this can be nested, time1 will exclude the timing from time2
   nlmixrWithTiming("time2", {
      Sys.sleep(1)
   }, envir=fit)
}, envir=fit)

print(fit)
```

`nlsControl` *nlmixr2 defaults controls for nls*

Description

nlmixr2 defaults controls for nls

Usage

```
nlsControl(
  maxiter = 10000,
  tol = 1e-05,
  minFactor = 1/1024,
  printEval = FALSE,
  warnOnly = FALSE,
  scaleOffset = 0,
  nDcentral = FALSE,
  algorithm = c("LM", "default", "plinear", "port"),
  ftol = sqrt(.Machine$double.eps),
  ptol = sqrt(.Machine$double.eps),
  gtol = 0,
  diag = list(),
  epsfcn = 0,
  factor = 100,
  maxfev = integer(),
  nprint = 0,
  solveType = c("grad", "fun"),
  stickyRecalcN = 4,
  maxOdeRecalc = 5,
  odeRecalcFactor = 10^(0.5),
  eventType = c("central", "forward"),
  shiErr = (.Machine$double.eps)^(1/3),
  shi21maxFD = 20L,
  useColor = crayon::has_color(),
  printNcol = floor((getOption("width") - 23)/12),
  print = 1L,
  normType = c("rescale2", "mean", "rescale", "std", "len", "constant"),
  scaleType = c("nlmixr2", "norm", "mult", "multAdd"),
  scaleCmax = 1e+05,
  scaleCmin = 1e-05,
  scaleC = NULL,
  scaleTo = 1,
  gradTo = 1,
  trace = FALSE,
  rxControl = NULL,
  optExpression = TRUE,
  sumProd = FALSE,
```

```
  literalFix = TRUE,
  returnNls = FALSE,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
  compress = TRUE,
  adjObf = TRUE,
  ci = 0.95,
  sigdig = 4,
  sigdigTable = NULL,
  ...
)
```

Arguments

`maxiter`	A positive integer specifying the maximum number of iterations allowed.
`tol`	A positive numeric value specifying the tolerance level for the relative offset convergence criterion.
`minFactor`	A positive numeric value specifying the minimum step-size factor allowed on any step in the iteration. The increment is calculated with a Gauss-Newton algorithm and successively halved until the residual sum of squares has been decreased or until the step-size factor has been reduced below this limit.
`printEval`	a logical specifying whether the number of evaluations (steps in the gradient direction taken each iteration) is printed.
`warnOnly`	a logical specifying whether `nls()` should return instead of signalling an error in the case of termination before convergence. Termination before convergence happens upon completion of `maxiter` iterations, in the case of a singular gradient, and in the case that the step-size factor is reduced below `minFactor`.
`scaleOffset`	a constant to be added to the denominator of the relative offset convergence criterion calculation to avoid a zero divide in the case where the fit of a model to data is very close. The default value of `0` keeps the legacy behaviour of `nls()`. A value such as `1` seems to work for problems of reasonable scale with very small residuals.
`nDcentral`	only when *numerical* derivatives are used: `logical` indicating if *central* differences should be employed, i.e., `numericDeriv(*, central=TRUE)` be used.
`algorithm`	character string specifying the algorithm to use. The default algorithm is a Gauss-Newton algorithm. Other possible values are `"plinear"` for the Golub-Pereyra algorithm for partially linear least-squares models and `"port"` for the 'nl2sol' algorithm from the Port library – see the references. Can be abbreviated.
`ftol`	non-negative numeric. Termination occurs when both the actual and predicted relative reductions in the sum of squares are at most `ftol`. Therefore, `ftol` measures the relative error desired in the sum of squares.
`ptol`	non-negative numeric. Termination occurs when the relative error between two consecutive iterates is at most `ptol`. Therefore, `ptol` measures the relative error desired in the approximate solution.
`gtol`	non-negative numeric. Termination occurs when the cosine of the angle between result of `fn` evaluation $fvec$ and any column of the Jacobian is at most `gtol` in

absolute value. Therefore, `gtol` measures the orthogonality desired between the function vector and the columns of the Jacobian.

`diag` a list or numeric vector containing positive entries that serve as multiplicative scale factors for the parameters. Length of `diag` should be equal to that of `par`. If not, user-provided `diag` is ignored and `diag` is internally set.

`epsfcn` (used if `jac` is not provided) is a numeric used in determining a suitable step for the forward-difference approximation. This approximation assumes that the relative errors in the functions are of the order of `epsfcn`. If `epsfcn` is less than the machine precision, it is assumed that the relative errors in the functions are of the order of the machine precision.

`factor` positive numeric, used in determining the initial step bound. This bound is set to the product of `factor` and the $|\texttt{diag} * \texttt{par}|$ if nonzero, or else to `factor` itself. In most cases `factor` should lie in the interval (0.1,100). 100 is a generally recommended value.

`maxfev` integer; termination occurs when the number of calls to `fn` has reached `maxfev`. Note that `nls.lm` sets the value of `maxfev` to `100*(length(par) + 1)` if `maxfev = integer()`, where `par` is the list or vector of parameters to be optimized.

`nprint` is an integer; set `nprint` to be positive to enable printing of iterates

`solveType` tells if ‘nlm‘ will use nlmixr2’s analytical gradients when available (finite differences will be used for event-related parameters like parameters controlling lag time, duration/rate of infusion, and modeled bioavailability). This can be:

- ‘"hessian"‘ which will use the analytical gradients to create a Hessian with finite differences.

- ‘"gradient"‘ which will use the gradient and let ‘nlm‘ calculate the finite difference hessian

- ‘"fun"‘ where nlm will calculate both the finite difference gradient and the finite difference Hessian

When using nlmixr2’s finite differences, the "ideal" step size for either central or forward differences are optimized for with the Shi2021 method which may give more accurate derivatives

`stickyRecalcN` The number of bad ODE solves before reducing the atol/rtol for the rest of the problem.

`maxOdeRecalc` Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.

`odeRecalcFactor`

The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced

`eventType` Event gradient type for dosing events; Can be "central" or "forward"

`shiErr` This represents the epsilon when optimizing the ideal step size for numeric differentiation using the Shi2021 method

`shi21maxFD` The maximum number of steps for the optimization of the forward difference step size when using dosing events (lag time, modeled duration/rate and bioavailability)

`useColor` Boolean indicating if focei can use ASCII color codes `printNcol` Number of columns to printout before wrapping parameter estimates/gradient

`print` Integer representing when the outer step is printed. When this is 0 or do not print the iterations. 1 is print every function evaluation (default), 5 is print every 5 evaluations.

`normType` This is the type of parameter normalization/scaling used to get the scaled initial values for nlmixr2. These are used with `scaleType` of.

With the exception of `rescale2`, these come from Feature Scaling. The `rescale2` The rescaling is the same type described in the OptdesX software manual.

In general, all all scaling formula can be described by:

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

Where

The other data normalization approaches follow the following formula

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

- `rescale2` This scales all parameters from (-1 to 1). The relative differences between the parameters are preserved with this approach and the constants are:

  $$C_1$$

  = (max(all unscaled values)+min(all unscaled values))/2

  $$C_2$$

  = (max(all unscaled values) - min(all unscaled values))/2
- `rescale` or min-max normalization. This rescales all parameters from (0 to 1). As in the `rescale2` the relative differences are preserved. In this approach:

  $$C_1$$

  = min(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)

- `mean` or mean normalization. This rescales to center the parameters around the mean but the parameters are from 0 to 1. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `std` or standardization. This standardizes by the mean and standard deviation. In this approach:

  $$C_1$$

  = mean(all unscaled values)

  $$C_2$$

  = sd(all unscaled values)
- `len` or unit length scaling. This scales the parameters to the unit length. For this approach we use the Euclidean length, that is:

  $$C_1$$

  = 0

  $$C_2$$

  =

  $$\sqrt{(}v_1^2 + v_2^2 + \cdots + v_n^2)$$
- `constant` which does not perform data normalization. That is

  $$C_1$$

  = 0

  $$C_2$$

  = 1

`scaleType` The scaling scheme for nlmixr2. The supported types are:

- `nlmixr2` In this approach the scaling is performed by the following equation:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

)*scaleC[i] + scaleTo

  The `scaleTo` parameter is specified by the `normType`, and the scales are specified by `scaleC`.

- `norm` This approach uses the simple scaling provided by the `normType` argument.
- `mult` This approach does not use the data normalization provided by `normType`, but rather uses multiplicative scaling to a constant provided by the `scaleTo` argument.
  In this case:
  $$v_{scaled}$$
  =
  $$v_{current}$$
  /
  $$v_{init}$$
  *scaleTo
- `multAdd` This approach changes the scaling based on the parameter being specified. If a parameter is defined in an exponential block (ie exp(theta)), then it is scaled on a linearly, that is:
  $$v_{scaled}$$
  = (
  $$v_{current} - v_{init}$$
) + scaleTo
  Otherwise the parameter is scaled multiplicatively.
  $$v_{scaled}$$
  =
  $$v_{current}$$
  /
  $$v_{init}$$
  *scaleTo

`scaleCmax` Maximum value of the scaleC to prevent overflow.

`scaleCmin` Minimum value of the scaleC to prevent underflow.

`scaleC` The scaling constant used with `scaleType=nlmixr2`. When not specified, it is based on the type of parameter that is estimated. The idea is to keep the derivatives similar on a log scale to have similar gradient sizes. Hence parameters like log(exp(theta)) would have a scaling factor of 1 and log(theta) would have a scaling factor of ini_value (to scale by 1/value; ie d/dt(log(ini_value)) = 1/ini_value or scaleC=ini_value)

- For parameters in an exponential (ie exp(theta)) or parameters specifying powers, boxCox or yeoJohnson transformations , this is 1.
- For additive, proportional, lognormal error structures, these are given by 0.5*abs(initial_estimate)
- Factorials are scaled by abs(1/digamma(initial_estimate+1))
- parameters in a log scale (ie log(theta)) are transformed by log(abs(initial_estimate))*abs(initial_estir

These parameter scaling coefficients are chose to try to keep similar slopes among parameters. That is they all follow the slopes approximately on a log-scale.

While these are chosen in a logical manner, they may not always apply. You can specify each parameters scaling factor by this parameter if you wish.

`scaleTo` Scale the initial parameter estimate to this value. By default this is 1. When zero or below, no scaling is performed.

`gradTo` this is the factor that the gradient is scaled to before optimizing. This only works with scaleType="nlmixr2".

`trace` logical value indicating if a trace of the iteration progress should be printed. Default is `FALSE`. If `TRUE` the residual (weighted) sum-of-squares, the convergence criterion and the parameter values are printed at the conclusion of each iteration. Note that `format()` is used, so these mostly depend on `getOption("digits")`. When the `"plinear"` algorithm is used, the conditional estimates of the linear parameters are printed after the nonlinear parameters. When the `"port"` algorithm is used the objective function value printed is half the residual (weighted) sum-of-squares.

`rxControl` ‘rxode2‘ ODE solving options during fitting, created with ‘rxControl()‘

`optExpression` Optimize the rxode2 expression to speed up calculation. By default this is turned on.

`sumProd` Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is `FALSE`.

`literalFix` boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is ‘TRUE‘.

`returnNls` logical; when TRUE, will return the nls object instead of the nlmixr object

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

Where:

- y represents the observed value
- f represents the predicted value
- a is the additive standard deviation
- b is the proportional/power standard deviation
- c is the power exponent (in the proportional case c=1)

`calcTables` This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`

`compress` Should the object have compressed items `adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM's default objective function. By default this is `TRUE`

`ci` Confidence level for some tables. By default this is 0.95 or 95% confidence.

`sigdig` Optimization significant digits. This controls:

- The tolerance of the inner and outer optimization is `10^-sigdig`
- The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)
- The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`

`sigdigTable` Significant digits in the final output table. If not specified, then it matches the significant digits in the 'sigdig' optimization algorithm. If 'sigdig' is NULL, use 3.

`...` Additional optional arguments. None are used at present.

Value

nls control object

Author(s)

Matthew L. Fidler

Examples

```
if (rxode2parse::.linCmtSens()) {

one.cmt <- function() {
  ini({
   tka <- 0.45
   tcl <- log(c(0, 2.7, 100))
   tv <- 3.45
   add.sd <- 0.7
 })
 model({
   ka <- exp(tka)
   cl <- exp(tcl)
   v <- exp(tv)
   linCmt() ~ add(add.sd)
 })
}

# Uses nlsLM from minpack.lm if available

fit1 <- nlmixr(one.cmt, nlmixr2data::theo_sd, est="nls", nlsControl(algorithm="LM"))

# Uses port and respect parameter boundaries
fit2 <- nlmixr(one.cmt, nlmixr2data::theo_sd, est="nls", nlsControl(algorithm="port"))
```

```
# You can access the underlying nls object with `$nls`
fit2$nls
}
```

---

nmNearPD *C++ implementation of Matrix's nearPD*

Description

With 'ensureSymmetry' it makes sure it is symmetric by applying 0.5*(t(x) + x) before using nmNearPD

Usage

```
nmNearPD(
  x,
  keepDiag = FALSE,
  do2eigen = TRUE,
  doDykstra = TRUE,
  only.values = FALSE,
  ensureSymmetry = !isSymmetric(x),
  eig.tol = 1e-06,
  conv.tol = 1e-07,
  posd.tol = 1e-08,
  maxit = 100L,
  trace = FALSE
)
```

Arguments

`x`	numeric $n \times n$ approximately positive definite matrix, typically an approximation to a correlation or covariance matrix. If `x` is not symmetric (and `ensureSymmetry` is not false), `symmpart(x)` is used.
`keepDiag`	logical, generalizing `corr`: if `TRUE`, the resulting matrix should have the same diagonal (`diag(x)`) as the input matrix.
`do2eigen`	logical indicating if a `posdefify()` eigen step should be applied to the result of the Higham algorithm.
`doDykstra`	logical indicating if Dykstra's correction should be used; true by default. If false, the algorithm is basically the direct fixpoint iteration $Y_k = P_U(P_S(Y_{k-1}))$.
`only.values`	logical; if `TRUE`, the result is just the vector of eigenvalues of the approximating matrix.
`ensureSymmetry`	logical; by default, `symmpart(x)` is used whenever `isSymmetric(x)` is not true. The user can explicitly set this to `TRUE` or `FALSE`, saving the symmetry test. *Beware* however that setting it `FALSE` for an **a**symmetric input `x`, is typically nonsense!

`eig.tol` defines relative positiveness of eigenvalues compared to largest one, λ_1. Eigenvalues λ_k are treated as if zero when $\lambda_k/\lambda_1 \leq eig.tol$.

`conv.tol` convergence tolerance for Higham algorithm.

`posd.tol` tolerance for enforcing positive definiteness (in the final `posdefify` step when `do2eigen` is `TRUE`).

`maxit` maximum number of iterations allowed.

`trace` logical or integer specifying if convergence monitoring should be traced.

Details

This implements the algorithm of Higham (2002), and then (if `do2eigen` is true) forces positive definiteness using code from `posdefify`. The algorithm of Knol and ten Berge (1989) (not implemented here) is more general in that it allows constraints to (1) fix some rows (and columns) of the matrix and (2) force the smallest eigenvalue to have a certain value.

Note that setting `corr = TRUE` just sets `diag(.) <- 1` within the algorithm.

Higham (2002) uses Dykstra's correction, but the version by Jens Oehlschlägel did not use it (accidentally), and still gave reasonable results; this simplification, now only used if `doDykstra = FALSE`, was active in `nearPD()` up to Matrix version 0.999375-40.

Value

If `only.values = TRUE`, a numeric vector of eigenvalues of the approximating matrix; Otherwise, as by default, an S3 object of `class` `"nearPD"`, basically a list with components

`mat` a matrix of class `dpoMatrix`, the computed positive-definite matrix.

`eigenvalues` numeric vector of eigenvalues of `mat`.

`corr` logical, just the argument `corr`.

`normF` the Frobenius norm (`norm(x-X, "F")`) of the difference between the original and the resulting matrix.

`iterations` number of iterations needed.

`converged` logical indicating if iterations converged.

Author(s)

Jens Oehlschlägel donated a first version. Subsequent changes by the Matrix package authors.

References

Cheng, Sheung Hun and Higham, Nick (1998) A Modified Cholesky Algorithm Based on a Symmetric Indefinite Factorization; *SIAM J. Matrix Anal.\ Appl.*, **19**, 1097–1110.

Knol DL, ten Berge JMF (1989) Least-squares approximation of an improper correlation matrix by a proper one. *Psychometrika* **54**, 53–61.

Higham, Nick (2002) Computing the nearest correlation matrix - a problem from finance; *IMA Journal of Numerical Analysis* **22**, 329–343.

See Also

A first version of this (with non-optional corr=TRUE) has been available as nearcor(); and more simple versions with a similar purpose posdefify(), both from package **sfsmisc**.

Examples

```
set.seed(27)
m <- matrix(round(rnorm(25),2), 5, 5)
m <- m + t(m)
diag(m) <- pmax(0, diag(m)) + 1
(m <- round(cov2cor(m), 2))

near.m <- nmNearPD(m)
round(near.m, 2)
norm(m - near.m) # 1.102 / 1.08

round(nmNearPD(m, only.values=TRUE), 9)

## A longer example, extended from Jens' original,
## showing the effects of some of the options:

pr <- matrix(c(1,     0.477, 0.644, 0.478, 0.651, 0.826,
               0.477, 1,     0.516, 0.233, 0.682, 0.75,
               0.644, 0.516, 1,     0.599, 0.581, 0.742,
               0.478, 0.233, 0.599, 1,     0.741, 0.8,
               0.651, 0.682, 0.581, 0.741, 1,     0.798,
               0.826, 0.75,  0.742, 0.8,   0.798, 1),
               nrow = 6, ncol = 6)

nc  <- nmNearPD(pr)
```

---

nmObjGetControl.bobyqa *Get control object from fit*

Description

Get control object from fit

Usage

```
## S3 method for class 'bobyqa'
nmObjGetControl(x, ...)

## S3 method for class 'lbfgsb3c'
nmObjGetControl(x, ...)
```

```
## S3 method for class 'n1qn1'
nmObjGetControl(x, ...)

## S3 method for class 'newuoa'
nmObjGetControl(x, ...)

## S3 method for class 'nlm'
nmObjGetControl(x, ...)

## S3 method for class 'nlme'
nmObjGetControl(x, ...)

## S3 method for class 'nlminb'
nmObjGetControl(x, ...)

## S3 method for class 'nls'
nmObjGetControl(x, ...)

nmObjGetControl(x, ...)

## S3 method for class 'focei'
nmObjGetControl(x, ...)

## S3 method for class 'foce'
nmObjGetControl(x, ...)

## S3 method for class 'foi'
nmObjGetControl(x, ...)

## S3 method for class 'fo'
nmObjGetControl(x, ...)

## S3 method for class 'posthoc'
nmObjGetControl(x, ...)

## S3 method for class 'saem'
nmObjGetControl(x, ...)

## Default S3 method:
nmObjGetControl(x, ...)

## S3 method for class 'optim'
nmObjGetControl(x, ...)

## S3 method for class 'uobyqa'
nmObjGetControl(x, ...)
```

Arguments

`x`	nlmixr fit object
`...`	Other parameters

Value

Control object of estimation method

Author(s)

Matthew L. Fidler

`nmObjGetEstimationModel` *Get the estimation model for a fit object depending on the object type*

Description

By default it gets the focei models if available.

Usage

```
nmObjGetEstimationModel(x)
```

Arguments

`x`	nlmixr fit object

Value

returns the estimation '$model' for the estimation type

`nmObjGetFoceiControl.nlme` *Method for getting focei compatible control object from nlmixr object*

Description

Method for getting focei compatible control object from nlmixr object

Usage

```
## S3 method for class 'nlme'
nmObjGetFoceiControl(x, ...)

nmObjGetFoceiControl(x, ...)

## Default S3 method:
nmObjGetFoceiControl(x, ...)

## S3 method for class 'saem'
nmObjGetFoceiControl(x, ...)
```

Arguments

`x`	nlmixr composed fit object
`...`	Other parameters

Value

foceiControl translated from current control

`nmObjGetIpredModel` *Get the ipred model for a fit object depending on the object type*

Description

By default it gets the focei models if available.

Usage

```
nmObjGetIpredModel(x)

## S3 method for class 'saem'
nmObjGetIpredModel(x)

## Default S3 method:
nmObjGetIpredModel(x)

## S3 method for class 'saem'
nmObjGetEstimationModel(x)

## Default S3 method:
nmObjGetEstimationModel(x)
```

Arguments

`x`	nlmixr fit object

Value

ipred ‘rxode2‘ model

`nmObjGetPredOnly` *Get the pred-only model for a fit depending on the object type*

Description

By default it gets the focei models if available

Usage

```
nmObjGetPredOnly(x)

## S3 method for class 'saem'
nmObjGetPredOnly(x)

## Default S3 method:
nmObjGetPredOnly(x)
```

Arguments

x	nlmixr fit object

Value

rxode2 pred-only model

`nmObjHandleControlObject.bobyqaControl`
Handle the control object

Description

Handle the control object

Usage

```
## S3 method for class 'bobyqaControl'
nmObjHandleControlObject(control, env)

## S3 method for class 'lbfgsb3cControl'
nmObjHandleControlObject(control, env)

## S3 method for class 'n1qn1Control'
```

```
nmObjHandleControlObject(control, env)

## S3 method for class 'newuoaControl'
nmObjHandleControlObject(control, env)

## S3 method for class 'nlmControl'
nmObjHandleControlObject(control, env)

## S3 method for class 'nlmeControl'
nmObjHandleControlObject(control, env)

## S3 method for class 'nlminbControl'
nmObjHandleControlObject(control, env)

## S3 method for class 'nlsControl'
nmObjHandleControlObject(control, env)

nmObjHandleControlObject(control, env)

## S3 method for class 'foceiControl'
nmObjHandleControlObject(control, env)

## S3 method for class 'saemControl'
nmObjHandleControlObject(control, env)

## Default S3 method:
nmObjHandleControlObject(control, env)

## S3 method for class 'optimControl'
nmObjHandleControlObject(control, env)

## S3 method for class 'uobyqaControl'
nmObjHandleControlObject(control, env)
```

Arguments

`control`	Control object
`env`	fit environment

Value

Nothing, called for side effects

Author(s)

Matthew L. Fidler

nmObjHandleModelObject *Handle Model Object*

Description

Handle Model Object

Usage

```
nmObjHandleModelObject(model, env)

## S3 method for class 'saemModelList'
nmObjHandleModelObject(model, env)

## S3 method for class 'foceiModelList'
nmObjHandleModelObject(model, env)

## Default S3 method:
nmObjHandleModelObject(model, env)
```

Arguments

`model`	model list should have at least: - ‘predOnly‘ – this is the prediction model with all the left handed equations added so they will be added the table. The model should have ‘rx_pred_‘, the model based prediction, as the first defined lhs component. The second component should be ‘rx_r_‘, the variance of the prediction. These variables may change based on distribution type. In additional all interesting calculated variables should be included. - ‘predNoLhs‘ – This is the prediction model. It only has the prediction and no left handed equations.
`env`	Environment for the fit information

Value

This returns the ‘$model‘ object for a fit. It is a s3 method because it may be different between different model types

---

`nmObjUiSetCompressed` *Set if the nlmixr2 object will return a compressed ui*

Description

Set if the nlmixr2 object will return a compressed ui

Usage

```
nmObjUiSetCompressed(type)
```

Arguments

`type`	is a boolean indicating if the compressed ui will be returned (‘TRUE‘) or not be returned (‘FALSE‘)

Value

invisible logical type

Author(s)

Matthew L. Fidler

Examples

```
nmObjUiSetCompressed(FALSE) # now the $ui will return an environment
nmObjUiSetCompressed(TRUE) # now the $ui will return a compressed value
```

`nmsimplex` *Nelder-Mead simplex search*

Description

Nelder-Mead simplex search

Usage

```
nmsimplex(start, fr, rho = NULL, control = list())
```

Arguments

`start`	initials
`fr`	objective function
`rho`	evaluation environment
`control`	additional optimization options

Value

a list of ...

`ofv` *Return the objective function*

Description

Return the objective function

Usage

```
ofv(x, type, ...)
```

Arguments

`x`	object to return objective function value
`type`	Objective function type value to retrieve or add. • focei For most models you can specify "focei" and it will add the focei objective function. • nlme This switches/chooses the nlme objective function if applicable. This objective function cannot be added if it isn't present. • fo FO objective function value. Cannot be generated • foce FOCE object function value. Cannot be generated • laplace# This adds/retrieves the Laplace objective function value. The # represents the number of standard deviations requested when expanding the Gaussian Quadrature. This can currently only be used with saem fits. • gauss#.# This adds/retrieves the Gaussian Quadrature approximation of the objective function. The first number is the number of nodes to use in the approximation. The second number is the number of standard deviations to expand upon.
`...`	Other arguments sent to ofv for other methods.

Value

Objective function value

Author(s)

Matthew Fidler

optimControl *nlmixr2 optim defaults*

Description

nlmixr2 optim defaults

Usage

```
optimControl(
  method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN", "Brent"),
  trace = 0,
  fnscale = 1,
  parscale = 1,
  ndeps = 0.001,
  maxit = 10000,
  abstol = 1e-08,
  reltol = 1e-08,
  alpha = 1,
  beta = 0.5,
  gamma = 2,
  REPORT = NULL,
  warn.1d.NelderMead = TRUE,
  type = NULL,
  lmm = 5,
  factr = 1e+07,
  pgtol = 0,
  temp = 10,
  tmax = 10,
  stickyRecalcN = 4,
  maxOdeRecalc = 5,
  odeRecalcFactor = 10^(0.5),
  eventType = c("central", "forward"),
  shiErr = (.Machine$double.eps)^(1/3),
  shi21maxFD = 20L,
  solveType = c("grad", "fun"),
  useColor = crayon::has_color(),
  printNcol = floor((getOption("width") - 23)/12),
  print = 1L,
  normType = c("rescale2", "mean", "rescale", "std", "len", "constant"),
  scaleType = c("nlmixr2", "norm", "mult", "multAdd"),
  scaleCmax = 1e+05,
  scaleCmin = 1e-05,
  scaleC = NULL,
  scaleTo = 1,
  gradTo = 1,
  rxControl = NULL,
```

```
  optExpression = TRUE,
  sumProd = FALSE,
  literalFix = TRUE,
  returnOptim = FALSE,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
  compress = TRUE,
  covMethod = c("r", "optim", ""),
  adjObf = TRUE,
  ci = 0.95,
  sigdig = 4,
  sigdigTable = NULL,
  ...
)
```

Arguments

`method`	The method to be used. See 'Details'. Can be abbreviated.
`trace`	Non-negative integer. If positive, tracing information on the progress of the optimization is produced. Higher values may produce more tracing information: for method '"L-BFGS-B"', there are six levels of tracing. See 'optim()' for more information
`fnscale`	An overall scaling to be applied to the value of 'fn' and 'gr' during optimization. If negative, turns the problem into a maximization problem. Optimization is performed on 'fn(par)/fnscale'
`parscale`	A vector of scaling values for the parameters. Optimization is performed on 'par/parscale' and these should be comparable in the sense that a unit change in any element produces about a unit change in the scaled value. Not used (nor needed) for 'method = "Brent"'
`ndeps`	A vector of step sizes for the finite-difference approximation to the gradient, on 'par/parscale' scale. Defaults to '1e-3'
`maxit`	The maximum number of iterations. Defaults to '100' for the derivative-based methods, and '500' for '"Nelder-Mead"'.
`abstol`	The absolute convergence tolerance. Only useful for non-negative functions, as a tolerance for reaching zero.
`reltol`	Relative convergence tolerance. The algorithm stops if it is unable to reduce the value by a factor of 'reltol * (abs(val) + reltol)' at a step
`alpha`	Reflection factor for the '"Nelder-Mead"' method.
`beta`	Contraction factor for the '"Nelder-Mead"' method
`gamma`	Expansion factor for the '"Nelder-Mead"' method
`REPORT`	The frequency of reports for the '"BFGS"', '"L-BFGS-B"' and '"SANN"' methods if 'control$trace' is positive. Defaults to every 10 iterations for '"BFGS"' and '"L-BFGS-B"', or every 100 temperatures for '"SANN"'
`warn.1d.NelderMead`	a logical indicating if the (default) '"Nelder-Mead"' method should signal a warning when used for one-dimensional minimization. As the warning is sometimes inappropriate, you can suppress it by setting this option to 'FALSE'

`type` for the conjugate-gradients method. Takes value '1' for the Fletcher-Reeves update, '2' for Polak-Ribiere and '3' for Beale-Sorenson.

`lmm` is an integer giving the number of BFGS updates retained in the '"L-BFGS-B"' method, It defaults to '5'

`factr` controls the convergence of the '"L-BFGS-B"' method. Convergence occurs when the reduction in the objective is within this factor of the machine tolerance. Default is '1e7', that is a tolerance of about '1e-8'.

`pgtol` helps control the convergence of the '"L-BFGS-B"' method. It is a tolerance on the projected gradient in the current search direction. This defaults to zero, when the check is suppressed

`temp` controls the '"SANN"' method. It is the starting temperature for the cooling schedule. Defaults to '10'.

`tmax` is the number of function evaluations at each temperature for the '"SANN"' method. Defaults to '10'.

`stickyRecalcN` The number of bad ODE solves before reducing the atol/rtol for the rest of the problem.

`maxOdeRecalc` Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.

`odeRecalcFactor`
The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced

`eventType` Event gradient type for dosing events; Can be "central" or "forward"

`shiErr` This represents the epsilon when optimizing the ideal step size for numeric differentiation using the Shi2021 method

`shi21maxFD` The maximum number of steps for the optimization of the forward difference step size when using dosing events (lag time, modeled duration/rate and bioavailability)

`solveType` tells if 'optim' will use nlmixr2's analytical gradients when available (finite differences will be used for event-related parameters like parameters controlling lag time, duration/rate of infusion, and modeled bioavailability). This can be:

- '"gradient"' which will use the gradient and let 'optim' calculate the finite difference hessian

- '"fun"' where optim will calculate both the finite difference gradient and the finite difference Hessian

When using nlmixr2's finite differences, the "ideal" step size for either central or forward differences are optimized for with the Shi2021 method which may give more accurate derivatives

These are only applied in the gradient based methods: "BFGS", "CG", "L-BFGS-B"

`useColor` Boolean indicating if focei can use ASCII color codes

`printNcol` Number of columns to printout before wrapping parameter estimates/gradient

`print` Integer representing when the outer step is printed. When this is 0 or do not print the iterations. 1 is print every function evaluation (default), 5 is print every 5 evaluations.

`normType` This is the type of parameter normalization/scaling used to get the scaled initial values for nlmixr2. These are used with `scaleType` of.

With the exception of `rescale2`, these come from Feature Scaling. The `rescale2` The rescaling is the same type described in the OptdesX software manual.

In general, all all scaling formula can be described by:

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

Where

The other data normalization approaches follow the following formula

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

- `rescale2` This scales all parameters from (-1 to 1). The relative differences between the parameters are preserved with this approach and the constants are:

  $$C_1$$

  = (max(all unscaled values)+min(all unscaled values))/2

  $$C_2$$

  = (max(all unscaled values) - min(all unscaled values))/2
- `rescale` or min-max normalization. This rescales all parameters from (0 to 1). As in the `rescale2` the relative differences are preserved. In this approach:

  $$C_1$$

  = min(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `mean` or mean normalization. This rescales to center the parameters around the mean but the parameters are from 0 to 1. In this approach:

  $$C_1$$

  = mean(all unscaled values)

$$C_2$$

= max(all unscaled values) - min(all unscaled values)

- `std` or standardization. This standardizes by the mean and standard deviation. In this approach:

$$C_1$$

= mean(all unscaled values)

$$C_2$$

= sd(all unscaled values)

- `len` or unit length scaling. This scales the parameters to the unit length. For this approach we use the Euclidean length, that is:

$$C_1$$

= 0

$$C_2$$

=

$$\sqrt(v_1^2 + v_2^2 + \cdots + v_n^2)$$

- `constant` which does not perform data normalization. That is

$$C_1$$

= 0

$$C_2$$

= 1

`scaleType` The scaling scheme for nlmixr2. The supported types are:

- `nlmixr2` In this approach the scaling is performed by the following equation:

$$v_{scaled}$$

= (

$$v_{current} - v_{init}$$

)*scaleC[i] + scaleTo

The `scaleTo` parameter is specified by the `normType`, and the scales are specified by `scaleC`.

- `norm` This approach uses the simple scaling provided by the `normType` argument.

- `mult` This approach does not use the data normalization provided by `normType`, but rather uses multiplicative scaling to a constant provided by the `scaleTo` argument.
  In this case:

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo
- `multAdd` This approach changes the scaling based on the parameter being specified. If a parameter is defined in an exponential block (ie exp(theta)), then it is scaled on a linearly, that is:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

) + scaleTo
  Otherwise the parameter is scaled multiplicatively.

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo

`scaleCmax` Maximum value of the scaleC to prevent overflow.

`scaleCmin` Minimum value of the scaleC to prevent underflow.

`scaleC` The scaling constant used with `scaleType=nlmixr2`. When not specified, it is based on the type of parameter that is estimated. The idea is to keep the derivatives similar on a log scale to have similar gradient sizes. Hence parameters like log(exp(theta)) would have a scaling factor of 1 and log(theta) would have a scaling factor of ini_value (to scale by 1/value; ie d/dt(log(ini_value)) = 1/ini_value or scaleC=ini_value)

- For parameters in an exponential (ie exp(theta)) or parameters specifying powers, boxCox or yeoJohnson transformations , this is 1.
- For additive, proportional, lognormal error structures, these are given by 0.5*abs(initial_estimate)
- Factorials are scaled by abs(1/digamma(initial_estimate+1))
- parameters in a log scale (ie log(theta)) are transformed by log(abs(initial_estimate))*abs(initial_estir

These parameter scaling coefficients are chose to try to keep similar slopes among parameters. That is they all follow the slopes approximately on a log-scale.

While these are chosen in a logical manner, they may not always apply. You can specify each parameters scaling factor by this parameter if you wish.

`scaleTo` Scale the initial parameter estimate to this value. By default this is 1. When zero or below, no scaling is performed.

`gradTo` this is the factor that the gradient is scaled to before optimizing. This only works with scaleType="nlmixr2".

`rxControl` ‘rxode2‘ ODE solving options during fitting, created with ‘rxControl()‘

`optExpression` Optimize the rxode2 expression to speed up calculation. By default this is turned on.

`sumProd` Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is `FALSE`.

`literalFix` boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is ‘TRUE‘.

`returnOptim` logical; when TRUE this will return the optim list instead of the nlmixr2 fit object

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2 \times c}} \times \varepsilon$$

Where:

- y represents the observed value

- f represents the predicted value

- a is the additive standard deviation

- b is the proportional/power standard deviation

- c is the power exponent (in the proportional case c=1)

`calcTables` This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`

`compress` Should the object have compressed items

`covMethod` allows selection of "r", which uses nlmixr2’s ‘nlmixr2Hess()‘ for the hessian calculation or "optim" which uses the hessian from ‘stats::optim(.., hessian=TRUE)‘

`adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM’s default objective function. By default this is `TRUE`

`ci` Confidence level for some tables. By default this is 0.95 or 95% confidence.

sigdig
: Optimization significant digits. This controls:

  - The tolerance of the inner and outer optimization is `10^-sigdig`
  - The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)
  - The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`

sigdigTable
: Significant digits in the final output table. If not specified, then it matches the significant digits in the ‘sigdig‘ optimization algorithm. If ‘sigdig‘ is NULL, use 3.

...
: Further arguments to be passed to `fn` and `gr`.

Value

optimControl object for nlmixr2

Author(s)

Matthew L. Fidler

Examples

```
# A logit regression example with emax model

dsn <- data.frame(i=1:1000)
dsn$time <- exp(rnorm(1000))
dsn$DV=rbinom(1000,1,exp(-1+dsn$time)/(1+exp(-1+dsn$time)))

mod <- function() {
 ini({
   E0 <- 0.5
   Em <- 0.5
   E50 <- 2
   g <- fix(2)
 })
 model({
   v <- E0+Em*time^g/(E50^g+time^g)
   ll(bin) ~ DV * v - log(1 + exp(v))
 })
}

fit2 <- nlmixr(mod, dsn, est="optim", optimControl(method="BFGS"))
fit2
```

print.saemFit — *Print an SAEM model fit summary*

Description

Print an SAEM model fit summary

Usage

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

Arguments

`x`	a saemFit object
`...`	others

Value

a list

residuals.nlmixr2FitData — *Extract residuals from the FOCEI fit*

Description

Extract residuals from the FOCEI fit

Usage

```
## S3 method for class 'nlmixr2FitData'
residuals(
  object,
  ...,
  type = c("ires", "res", "iwres", "wres", "cwres", "cpred", "cres")
)
```

Arguments

`object`	focei.fit object
`...`	Additional arguments
`type`	Residuals type fitted.

Value

residuals

Author(s)

Matthew L. Fidler

---

saemControl *Control Options for SAEM*

Description

Control Options for SAEM

Usage

```
saemControl(
  seed = 99,
  nBurn = 200,
  nEm = 300,
  nmc = 3,
  nu = c(2, 2, 2),
  print = 1,
  trace = 0,
  covMethod = c("linFim", "fim", "r,s", "r", "s", ""),
  calcTables = TRUE,
  logLik = FALSE,
  nnodesGq = 3,
  nsdGq = 1.6,
  optExpression = TRUE,
  literalFix = TRUE,
  adjObf = TRUE,
  sumProd = FALSE,
  addProp = c("combined2", "combined1"),
  tol = 1e-06,
  itmax = 30,
  type = c("nelder-mead", "newuoa"),
  powRange = 10,
  lambdaRange = 3,
  odeRecalcFactor = 10^(0.5),
  maxOdeRecalc = 5L,
  perSa = 0.75,
  perNoCor = 0.75,
  perFixOmega = 0.1,
  perFixResid = 0.1,
  compress = TRUE,
  rxControl = NULL,
```

```
  sigdig = NULL,
  sigdigTable = NULL,
  ci = 0.95,
  muRefCov = TRUE,
  muRefCovAlg = TRUE,
  handleUninformativeEtas = TRUE,
  ...
)
```

Arguments

`seed` Random Seed for SAEM step. (Needs to be set for reproducibility.) By default this is 99.

`nBurn` Number of iterations in the first phase, ie the MCMC/Stochastic Approximation steps. This is equivalent to Monolix's `K_0` or `K_b`.

`nEm` Number of iterations in the Expectation-Maximization (EM) Step. This is equivalent to Monolix's `K_1`.

`nmc` Number of Markov Chains. By default this is 3. When you increase the number of chains the numerical integration by MC method will be more accurate at the cost of more computation. In Monolix this is equivalent to `L`.

`nu` This is a vector of 3 integers. They represent the numbers of transitions of the three different kernels used in the Hasting-Metropolis algorithm. The default value is `c(2,2,2)`, representing 40 for each transition initially (each value is multiplied by 20).

The first value represents the initial number of multi-variate Gibbs samples are taken from a normal distribution.

The second value represents the number of uni-variate, or multi- dimensional random walk Gibbs samples are taken.

The third value represents the number of bootstrap/reshuffling or uni-dimensional random samples are taken.

`print` The number it iterations that are completed before anything is printed to the console. By default, this is 1.

`trace` An integer indicating if you want to trace(1) the SAEM algorithm process. Useful for debugging, but not for typical fitting.

`covMethod` Method for calculating covariance. In this discussion, R is the Hessian matrix of the objective function. The S matrix is the sum of each individual's gradient cross-product (evaluated at the individual empirical Bayes estimates).

"`linFim`" Use the Linearized Fisher Information Matrix to calculate the covariance.

"`fim`" Use the SAEM-calculated Fisher Information Matrix to calculate the covariance.

"`r,s`" Uses the sandwich matrix to calculate the covariance, that is: $R^-1 \times S \times R^-1$

"`r`" Uses the Hessian matrix to calculate the covariance as $2 \times R^-1$

"`s`" Uses the crossproduct matrix to calculate the covariance as $4 \times S^-1$

"" Does not calculate the covariance step.

`calcTables` This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`

`logLik` boolean indicating that log-likelihood should be calculate by Gaussian quadrature.

`nnodesGq` number of nodes to use for the Gaussian quadrature when computing the likelihood with this method (defaults to 1, equivalent to the Laplacian likelihood)

`nsdGq` span (in SD) over which to integrate when computing the likelihood by Gaussian quadrature. Defaults to 3 (eg 3 times the SD)

`optExpression` Optimize the rxode2 expression to speed up calculation. By default this is turned on.

`literalFix` boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is 'TRUE'.

`adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM's default objective function. By default this is `TRUE`

`sumProd` Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is `FALSE`.

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

Where:

- y represents the observed value
- f represents the predicted value
- a is the additive standard deviation
- b is the proportional/power standard deviation
- c is the power exponent (in the proportional case c=1)

`tol` This is the tolerance for the regression models used for complex residual errors (ie add+prop etc)

`itmax` This is the maximum number of iterations for the regression models used for complex residual errors. The number of iterations is itmax*number of parameters

`type` indicates the type of optimization for the residuals; Can be one of c("nelder-mead", "newuoa")

`powRange` This indicates the range that powers can take for residual errors; By default this is 10 indicating the range is c(-10, 10)

`lambdaRange` This indicates the range that Box-Cox and Yeo-Johnson parameters are constrained to be; The default is 3 indicating the range c(-3,3)

odeRecalcFactor
: The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced

maxOdeRecalc
: Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.

perSa
: This is the percent of the time the ‘nBurn‘ iterations in phase runs runs a simulated annealing.

perNoCor
: This is the percentage of the MCMC phase of the SAEM algorithm where the variance/covariance matrix has no correlations. By default this is 0.75 or 75 Monte-carlo iteration.

perFixOmega
: This is the percentage of the ‘nBurn‘ phase where the omega values are unfixed to allow better exploration of the likelihood surface. After this time, the omegas are fixed during optimization.

perFixResid
: This is the percentage of the ‘nBurn‘ phase where the residual components are unfixed to allow better exploration of the likelihood surface.

compress
: Should the object have compressed items

rxControl
: ‘rxode2‘ ODE solving options during fitting, created with ‘rxControl()‘

sigdig
: Specifies the "significant digits" that the ode solving requests. When specified this controls the relative and absolute tolerances of the ODE solvers. By default the tolerance is `0.5*10^(-sigdig-2)` for regular ODEs. For the sensitivity equations the default is `0.5*10\^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda). This also controls the `atol`/`rtol` of the steady state solutions. The `ssAtol`/`ssRtol` is `0.5*10\^(-sigdig)` and for the sensitivities `0.5*10\^(-sigdig+0.625)`. By default this is unspecified (`NULL`) and uses the standard `atol`/`rtol`.

sigdigTable
: Significant digits in the final output table. If not specified, then it matches the significant digits in the ‘sigdig‘ optimization algorithm. If ‘sigdig‘ is NULL, use 3.

ci
: Confidence level for some tables. By default this is 0.95 or 95% confidence.

muRefCov
: This controls if mu-referenced covariates in ‘saem‘ are handled differently than non mu-referenced covariates. When ‘TRUE‘, mu-referenced covariates have special handling. When ‘FALSE‘ mu-referenced covariates are treated the same as any other input parameter.

muRefCovAlg
: This controls if algebraic expressions that can be mu-referenced are treated as mu-referenced covariates by:

1. Creating a internal data-variable ‘nlmixrMuDerCov#‘ for each algebraic mu-referenced expression

2. Change the algebraic expression to ‘nlmixrMuDerCov# * mu_cov_theta‘

3. Use the internal mu-referenced covariate for saem

4. After optimization is completed, replace ‘model()‘ with old ‘model()‘ expression

5. Remove ‘nlmixrMuDerCov#‘ from nlmix2 output

In general, these covariates should be more accurate since it changes the system to a linear compartment model. Therefore, by default this is ‘TRUE‘.

`handleUninformativeEtas`
: boolean that tells nlmixr2's saem to calculate uninformative etas and handle them specially (default is 'TRUE').

`...`
: Other arguments to control SAEM.

Value

List of options to be used in `nlmixr2` fit for SAEM.

Author(s)

Wenping Wang & Matthew L. Fidler

See Also

Other Estimation control: `foceiControl()`, `nlmixr2NlmeControl()`

`setCov` *Set the covariance type based on prior calculated covariances*

Description

Set the covariance type based on prior calculated covariances

Usage

```
setCov(fit, method)
```

Arguments

`fit`	nlmixr2 fit
`method`	covariance method (see the 'covMethod' argument for the control options for the choices)

Value

Fit object with covariance updated

Author(s)

Matt Fidler

See Also

`foceiControl()`, `saemControl()`

setOfv — *Set/get Objective function type for a nlmixr2 object*

Description

Set/get Objective function type for a nlmixr2 object

Usage

```
setOfv(x, type)

getOfvType(x)
```

Arguments

`x`	nlmixr2 fit object
`type`	Type of objective function to use for AIC, BIC, and $objective

Value

Nothing

Author(s)

Matthew L. Fidler

sqrtm — *Return the square root of general square matrix A*

Description

Return the square root of general square matrix A

Usage

```
sqrtm(m)
```

Arguments

`m`	Matrix to take the square root of.

Value

A square root general square matrix of m

summary.saemFit *Print an SAEM model fit summary*

Description

Print an SAEM model fit summary

Usage

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

Arguments

`object`	a saemFit object
`...`	others

Value

a list

tableControl *Output table/data.frame options*

Description

Output table/data.frame options

Usage

```
tableControl(
  npde = NULL,
  cwres = NULL,
  nsim = 300,
  ties = TRUE,
  censMethod = c("truncated-normal", "cdf", "ipred", "pred", "epred", "omit"),
  seed = 1009,
  cholSEtol = (.Machine$double.eps)^(1/3),
  state = TRUE,
  lhs = TRUE,
  eta = TRUE,
  covariates = TRUE,
  addDosing = FALSE,
  subsetNonmem = TRUE,
  cores = NULL,
```

```
  keep = NULL,
  drop = NULL
)
```

Arguments

`npde` When TRUE, request npde regardless of the algorithm used.

`cwres` When TRUE, request CWRES and FOCEi likelihood regardless of the algorithm used.

`nsim` represents the number of simulations. For rxode2, if you supply single subject event tables (created with `[eventTable()]`)

`ties` When ‘TRUE‘ jitter prediction-discrepancy points to discourage ties in cdf.

`censMethod` Handle censoring method:

- ‘"truncated-normal"‘ Simulates from a truncated normal distribution under the assumption of the model and censoring.

- ‘"cdf"‘ Use the cdf-method for censoring with npde and use this for any other residuals (‘cwres‘ etc)

- ‘"omit"‘ omit the residuals for censoring

`seed` an object specifying if and how the random number generator should be initialized

`cholSEtol` The tolerance for the ‘rxode2::choleSE‘ function

`state` is a Boolean indicating if ‘state‘ values will be included (default ‘TRUE‘)

`lhs` is a Boolean indicating if remaining ‘lhs‘ values will be included (default ‘TRUE‘)

`eta` is a Boolean indicating if ‘eta‘ values will be included (default ‘TRUE‘)

`covariates` is a Boolean indicating if covariates will be included (default ‘TRUE‘)

`addDosing` Boolean indicating if the solve should add rxode2 EVID and related columns. This will also include dosing information and estimates at the doses. Be default, rxode2 only includes estimates at the observations. (default `FALSE`). When `addDosing` is `NULL`, only include `EVID=0` on solve and exclude any model-times or `EVID=2`. If `addDosing` is `NA` the classic `rxode2` EVID events are returned. When `addDosing` is `TRUE` add the event information in NONMEM-style format; If `subsetNonmem=FALSE` rxode2 will also include extra event types (`EVID`) for ending infusion and modeled times:

- `EVID=-1` when the modeled rate infusions are turned off (matches `rate=-1`)
- `EVID=-2` When the modeled duration infusions are turned off (matches `rate=-2`)
- `EVID=-10` When the specified `rate` infusions are turned off (matches `rate>0`)
- `EVID=-20` When the specified `dur` infusions are turned off (matches `dur>0`)
- `EVID=101,102,103,...` Modeled time where 101 is the first model time, 102 is the second etc.

`subsetNonmem` subset to NONMEM compatible EVIDs only. By default `TRUE`.

`cores` Number of cores used in parallel ODE solving. This is equivalent to calling `setRxThreads()`

`keep` is the keep sent to the table

`drop` is the dropped variables sent to the table

Details

If you ever want to add CWRES/FOCEi objective function you can use the `addCwres`

If you ever want to add NPDE/EPRED columns you can use the `addNpde`

Value

A list of table options for nlmixr2

Author(s)

Matthew L. Fidler

`uobyqaControl` *Control for uobyqa estimation method in nlmixr2*

Description

Control for uobyqa estimation method in nlmixr2

Usage

```
uobyqaControl(
  npt = NULL,
  rhobeg = NULL,
  rhoend = NULL,
  iprint = 0L,
  maxfun = 100000L,
  returnUobyqa = FALSE,
  stickyRecalcN = 4,
  maxOdeRecalc = 5,
  odeRecalcFactor = 10^(0.5),
  useColor = crayon::has_color(),
  printNcol = floor((getOption("width") - 23)/12),
  print = 1L,
  normType = c("rescale2", "mean", "rescale", "std", "len", "constant"),
  scaleType = c("nlmixr2", "norm", "mult", "multAdd"),
  scaleCmax = 1e+05,
  scaleCmin = 1e-05,
  scaleC = NULL,
  scaleTo = 1,
  rxControl = NULL,
  optExpression = TRUE,
  sumProd = FALSE,
  literalFix = TRUE,
  addProp = c("combined2", "combined1"),
  calcTables = TRUE,
```

```
  compress = TRUE,
  covMethod = c("r", ""),
  adjObf = TRUE,
  ci = 0.95,
  sigdig = 4,
  sigdigTable = NULL,
  ...
)
```

Arguments

`npt` The number of points used to approximate the objective function via a quadratic approximation for bobyqa. The value of npt must be in the interval [n+2,(n+1)(n+2)/2] where n is the number of parameters in par. Choices that exceed 2*n+1 are not recommended. If not defined, it will be set to 2*n + 1. (bobyqa)

`rhobeg` Beginning change in parameters for bobyqa algorithm (trust region). By default this is 0.2 or 20 parameters when the parameters are scaled to 1. rhobeg and rhoend must be set to the initial and final values of a trust region radius, so both must be positive with 0 < rhoend < rhobeg. Typically rhobeg should be about one tenth of the greatest expected change to a variable. Note also that smallest difference abs(upper-lower) should be greater than or equal to rhobeg*2. If this is not the case then rhobeg will be adjusted. (bobyqa)

`rhoend` The smallest value of the trust region radius that is allowed. If not defined, then 10^(-sigdig-1) will be used. (bobyqa)

`iprint` The value of ‘iprint‘ should be set to an integer value in ‘0, 1, 2, 3, ...‘, which controls the amount of printing. Specifically, there is no output if ‘iprint=0‘ and there is output only at the start and the return if ‘iprint=1‘. Otherwise, each new value of ‘rho‘ is printed, with the best vector of variables so far and the corresponding value of the objective function. Further, each new value of the objective function with its variables are output if ‘iprint=3‘. If ‘iprint > 3‘, the objective function value and corresponding variables are output every ‘iprint‘ evaluations. Default value is ‘0‘.

`maxfun` The maximum allowed number of function evaluations. If this is exceeded, the method will terminate.

`returnUobyqa` return the uobyqa output instead of the nlmixr2 fit

`stickyRecalcN` The number of bad ODE solves before reducing the atol/rtol for the rest of the problem.

`maxOdeRecalc` Maximum number of times to reduce the ODE tolerances and try to resolve the system if there was a bad ODE solve.

`odeRecalcFactor` The ODE recalculation factor when ODE solving goes bad, this is the factor the rtol/atol is reduced

`useColor` Boolean indicating if focei can use ASCII color codes

`printNcol` Number of columns to printout before wrapping parameter estimates/gradient

`print` Integer representing when the outer step is printed. When this is 0 or do not print the iterations. 1 is print every function evaluation (default), 5 is print every 5 evaluations.

`normType` This is the type of parameter normalization/scaling used to get the scaled initial values for nlmixr2. These are used with `scaleType` of.

With the exception of `rescale2`, these come from Feature Scaling. The `rescale2` The rescaling is the same type described in the OptdesX software manual.

In general, all all scaling formula can be described by:

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

Where

The other data normalization approaches follow the following formula

$$v_{scaled}$$

= (

$$v_{unscaled} - C_1$$

)/

$$C_2$$

- `rescale2` This scales all parameters from (-1 to 1). The relative differences between the parameters are preserved with this approach and the constants are:

  $$C_1$$

  = (max(all unscaled values)+min(all unscaled values))/2

  $$C_2$$

  = (max(all unscaled values) - min(all unscaled values))/2
- `rescale` or min-max normalization. This rescales all parameters from (0 to 1). As in the `rescale2` the relative differences are preserved. In this approach:

  $$C_1$$

  = min(all unscaled values)

  $$C_2$$

  = max(all unscaled values) - min(all unscaled values)
- `mean` or mean normalization. This rescales to center the parameters around the mean but the parameters are from 0 to 1. In this approach:

  $$C_1$$

  = mean(all unscaled values)

$$C_2$$

= max(all unscaled values) - min(all unscaled values)

- `std` or standardization. This standardizes by the mean and standard deviation. In this approach:

$$C_1$$

= mean(all unscaled values)

$$C_2$$

= sd(all unscaled values)

- `len` or unit length scaling. This scales the parameters to the unit length. For this approach we use the Euclidean length, that is:

$$C_1$$

= 0

$$C_2$$

=

$$\sqrt(v_1^2 + v_2^2 + \cdots + v_n^2)$$

- `constant` which does not perform data normalization. That is

$$C_1$$

= 0

$$C_2$$

= 1

`scaleType` The scaling scheme for nlmixr2. The supported types are:

- `nlmixr2` In this approach the scaling is performed by the following equation:

$$v_{scaled}$$

= (

$$v_{current} - v_{init}$$

)*scaleC[i] + scaleTo
The `scaleTo` parameter is specified by the `normType`, and the scales are specified by `scaleC`.

- `norm` This approach uses the simple scaling provided by the `normType` argument.

- `mult` This approach does not use the data normalization provided by `normType`, but rather uses multiplicative scaling to a constant provided by the `scaleTo` argument.
  In this case:

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo
- `multAdd` This approach changes the scaling based on the parameter being specified. If a parameter is defined in an exponential block (ie exp(theta)), then it is scaled on a linearly, that is:

  $$v_{scaled}$$

  = (

  $$v_{current} - v_{init}$$

) + scaleTo
  Otherwise the parameter is scaled multiplicatively.

  $$v_{scaled}$$

  =

  $$v_{current}$$

  /

  $$v_{init}$$

  *scaleTo

`scaleCmax` Maximum value of the scaleC to prevent overflow.

`scaleCmin` Minimum value of the scaleC to prevent underflow.

`scaleC` The scaling constant used with `scaleType=nlmixr2`. When not specified, it is based on the type of parameter that is estimated. The idea is to keep the derivatives similar on a log scale to have similar gradient sizes. Hence parameters like log(exp(theta)) would have a scaling factor of 1 and log(theta) would have a scaling factor of ini_value (to scale by 1/value; ie d/dt(log(ini_value)) = 1/ini_value or scaleC=ini_value)

- For parameters in an exponential (ie exp(theta)) or parameters specifying powers, boxCox or yeoJohnson transformations , this is 1.
- For additive, proportional, lognormal error structures, these are given by 0.5*abs(initial_estimate)
- Factorials are scaled by abs(1/digamma(initial_estimate+1))
- parameters in a log scale (ie log(theta)) are transformed by log(abs(initial_estimate))*abs(initial_estir

These parameter scaling coefficients are chose to try to keep similar slopes among parameters. That is they all follow the slopes approximately on a log-scale.

While these are chosen in a logical manner, they may not always apply. You can specify each parameters scaling factor by this parameter if you wish.

`scaleTo` Scale the initial parameter estimate to this value. By default this is 1. When zero or below, no scaling is performed.

`rxControl` ‘rxode2‘ ODE solving options during fitting, created with ‘rxControl()‘

`optExpression` Optimize the rxode2 expression to speed up calculation. By default this is turned on.

`sumProd` Is a boolean indicating if the model should change multiplication to high precision multiplication and sums to high precision sums using the PreciseSums package. By default this is `FALSE`.

`literalFix` boolean, substitute fixed population values as literals and re-adjust ui and parameter estimates after optimization; Default is ‘TRUE‘.

`addProp` specifies the type of additive plus proportional errors, the one where standard deviations add (combined1) or the type where the variances add (combined2).

The combined1 error type can be described by the following equation:

$$y = f + (a + b \times f^c) \times \varepsilon$$

The combined2 error model can be described by the following equation:

$$y = f + \sqrt{a^2 + b^2 \times f^{2\times c}} \times \varepsilon$$

Where:

- y represents the observed value

- f represents the predicted value

- a is the additive standard deviation

- b is the proportional/power standard deviation

- c is the power exponent (in the proportional case c=1)

`calcTables` This boolean is to determine if the foceiFit will calculate tables. By default this is `TRUE`

`compress` Should the object have compressed items

`covMethod` Method for calculating covariance. In this discussion, R is the Hessian matrix of the objective function. The S matrix is the sum of individual gradient cross-product (evaluated at the individual empirical Bayes estimates).

- "`r,s`" Uses the sandwich matrix to calculate the covariance, that is: `solve(R) %*% S %*% solve(R)`
- "`r`" Uses the Hessian matrix to calculate the covariance as `2 %*% solve(R)`
- "`s`" Uses the cross-product matrix to calculate the covariance as `4 %*% solve(S)`
- "" Does not calculate the covariance step.

`adjObf` is a boolean to indicate if the objective function should be adjusted to be closer to NONMEM’s default objective function. By default this is `TRUE` ci — Confidence level for some tables. By default this is 0.95 or 95% confidence.

sigdig — Optimization significant digits. This controls:

- The tolerance of the inner and outer optimization is `10^-sigdig`
- The tolerance of the ODE solvers is `0.5*10^(-sigdig-2)`; For the sensitivity equations and steady-state solutions the default is `0.5*10^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda)
- The tolerance of the boundary check is `5 * 10 ^ (-sigdig + 1)`

sigdigTable — Significant digits in the final output table. If not specified, then it matches the significant digits in the ‘sigdig‘ optimization algorithm. If ‘sigdig‘ is NULL, use 3.

... — Ignored parameters

Value

uobyqa control structure

Author(s)

Matthew L. Fidler

Examples

```
# A logit regression example with emax model

dsn <- data.frame(i=1:1000)
dsn$time <- exp(rnorm(1000))
dsn$DV=rbinom(1000,1,exp(-1+dsn$time)/(1+exp(-1+dsn$time)))

mod <- function() {
 ini({
   E0 <- 0.5
   Em <- 0.5
   E50 <- 2
   g <- fix(2)
 })
 model({
   v <- E0+Em*time^g/(E50^g+time^g)
   ll(bin) ~ DV * v - log(1 + exp(v))
 })
}

fit2 <- nlmixr(mod, dsn, est="uobyqa")

print(fit2)

# you can also get the nlm output with fit2$nlm

fit2$uobyqa
```

```
# The nlm control has been modified slightly to include
# extra components and name the parameters
```

`vpcSim` *VPC simulation*

Description

VPC simulation

Usage

```
vpcSim(
  object,
  ...,
  keep = NULL,
  n = 300,
  pred = FALSE,
  seed = 1009,
  nretry = 50,
  minN = 10,
  normRelated = TRUE
)
```

Arguments

`object`	This is the nlmixr2 fit object
`...`	Other arguments sent to 'rxSolve()'
`keep`	Column names to keep in the output simulated dataset
`n`	Number of simulations
`pred`	Should predictions be added to the simulation
`seed`	Seed to set for the VPC simulation
`nretry`	Number of times to retry the simulation if there is NA values in the simulation
`minN`	With retries, the minimum number of studies to restimulate (by default 10)
`normRelated`	should the VPC style simulation be for normal related variables only

Value

data frame of the VPC simulation

Author(s)

Matthew L. Fidler

Examples

```
if (rxode2parse::.linCmtSens()) {

one.cmt <- function() {
 ini({
   ## You may label each parameter with a comment
   tka <- 0.45 # Log Ka
   tcl <- log(c(0, 2.7, 100)) # Log Cl
   ## This works with interactive models
   ## You may also label the preceding line with label("label text")
   tv <- 3.45; label("log V")
   ## the label("Label name") works with all models
   eta.ka ~ 0.6
   eta.cl ~ 0.3
   eta.v ~ 0.1
   add.sd <- 0.7
 })
 model({
   ka <- exp(tka + eta.ka)
   cl <- exp(tcl + eta.cl)
   v <- exp(tv + eta.v)
   linCmt() ~ add(add.sd)
 })
}

fit <- nlmixr(one.cmt, theo_sd, est="focei")

head(vpcSim(fit, pred=TRUE))

}
```

Index