Package: rxode2 (via r-universe)

October 14, 2026

Version 5.1.8

Title Facilities for Simulating from ODE-Based Models

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

Depends R (>= 4.1.0)

Suggests Matrix, DT, callr, covr, crayon, curl, digest, dplyr (>= 0.8.0), ggrepel, gridExtra, htmltools, knitr, learnr, microbenchmark, nlme, remotes, rlang, rmarkdown, scales, shiny, stringi, symengine, testthat, tidyr, usethis, withr, xgxr (>= 1.1.6), pillar, ps, tibble, units (>= 0.6-0), rsconnect, devtools, patchwork, nlmixr2data, lifecycle, kableExtra, pmxTools, rootSolve, arrow, fst, duckdb, DBI

Imports PreciseSums (>= 0.7), Rcpp (>= 0.12.3), backports, cli (>= 2.0.0), checkmate, compiler, ggplot2 (>= 3.4.0), inline, lotri (>= 1.0.4), memoise, methods, mirai, rex, sys, tools, utils, dparser (>= 1.3.1-14), qs2, rxode2ll (>= 2.0.18), rxode2lincmt (>= 0.1.0), data.table (>= 1.12.4), vctrs, stats

Description Facilities for running simulations from ordinary differential equation ('ODE') models, such as pharmacometrics and other compartmental models. A compilation manager translates the ODE model into C, compiles it, and dynamically loads the object code into R for improved computational efficiency. An event table object facilitates the specification of complex dosing regimens (optional) and sampling schedules. NB: The use of this package requires both C and Fortran compilers, for details on their use with R please see Section 6.3, Appendix A, and Appendix D in the ``R Administration and Installation'' manual. Also the code is mostly released under GPL. The 'VODE' and 'LSODA' are in the public domain. The vendored 'SUNDIALS' 'CVODE' sources and headers are released under the BSD-3-Clause license. The information is available in the inst/COPYRIGHTS.

BugReports https://github.com/nlmixr2/rxode2/issues/

NeedsCompilation yes

VignetteBuilder knitr

License GPL (>= 3)

URL https://nlmixr2.github.io/rxode2/,
https://github.com/nlmixr2/rxode2/

Biarch true

LinkingTo sitmo, rxode2lincmt (>= 0.1.0), lotri (>= 1.0.4),
PreciseSums (>= 0.7), Rcpp, RcppArmadillo (>= 0.9.300.2.0), BH,
dparser (>= 1.3.1-14)

Encoding UTF-8

LazyData true

Language en-US

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

Author Matthew L. Fidler [aut, cre] (ORCID:
<https://orcid.org/0000-0001-8538-6691>), Wenping Wang [aut],
Aaron Collier [ctb] (SUNDIALS/CVODE), Alan Hindmarsh [ctb],
Arun Srinivasan [ctb], Ashley Crawford [ctb] (SUNDIALS/CVODE),
Awad H. Al-Mohy [ctb], Bill Denney [ctb] (ORCID:
<https://orcid.org/0000-0002-5759-428X>), Brian Ripley [ctb],
Cleve Moler [ctb], Cody J. Balos [ctb] (SUNDIALS/CVODE), Dan
Shumaker [ctb] (SUNDIALS/CVODE), Daniel Kaschek [ctb], Daniel
R. Reynolds [ctb] (SUNDIALS/CVODE), David Cooley [ctb], David
J. Gardner [ctb] (SUNDIALS/CVODE), Drew Schmidt [ctb], Ernst
Hairer [ctb], Gabriel Staples [ctb], Gerhard Wanner [ctb],
Gilbert Stewart [ctb], Goro Fuji [ctb], Hadley Wickham [ctb],
Hidde van de Beek [ctb], Igor Kushnir [ctb], Imperial College
of Science, Technology and Medicine [cph], Jack Dongarra [ctb],
Jacob Williams [ctb], Jim Bunch [ctb], Karline Soetaert [ctb],
Kevin Ushey [ctb], Lawrence Livermore National Security [cph]
(SUNDIALS/CVODE), Linda Petzold [ctb], Mark M. Baum [ctb],
Martin Maechler [ctb], Matt Dowle [ctb], Matteo Fasiolo [ctb],
Melissa Hallow [aut], Michel Lang [ctb], Morwenn [ctb],
Nicholas J. Higham [ctb], Omar Elashkar [ctb], Peter N. Brown
[ctb], Radu Serban [ctb] (SUNDIALS/CVODE), Rich FitzJohn [ctb],
Richard Upton [ctb], Roger B. Sidje [ctb], Scott D. Cohen [ctb]
(SUNDIALS/CVODE), Simon Frost [ctb], Slaven Peles [ctb]
(SUNDIALS/CVODE), Southern Methodist University [cph]
(SUNDIALS/CVODE), Thomas Petzoldt [ctb], University of Maryland
Baltimore County [cph] (SUNDIALS/CVODE), Wes Hinsley [ctb],
Woodrow Setzer [ctb], Yu Feng [ctb], Zufar Mulyukov [ctb]

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

Date/Publication 2026-10-13 09:10:03 UTC

RemoteUrl https://github.com/cran/rxode2

RemoteRef HEAD

RemoteSha 915f6b41e7b01f5e96985e8e5a5acb1168aa93b7

Contents

.cbindOme . . . 9
.collectWarnings . . . 9
.copyUi . . . 10
.handleSingleErrTypeNormOrTFoceiBase . . . 10
.matchesLangTemplate . . . 11
.modelHandleModelLines . . . 12
.quoteCallInfoLines . . . 13
.rxLinCmtGen . . . 14
.rxode2ptrs . . . 14
.rxSensStrippable . . . 15
.rxWithOptions . . . 15
.rxWithWd . . . 16
.toClassicEvid . . . 17
.vecDf . . . 18
add.dosing . . . 18
add.sampling . . . 21
as.arrow . . . 24
as.data.table.rxEt . . . 25
as.et . . . 25
as.ini . . . 26
as.model . . . 28
as.rxUi . . . 30
as_tibble.rxEt . . . 31
assertCompartmentExists . . . 32
assertCompartmentName . . . 33
assertCompartmentNew . . . 34
assertRxUi . . . 34
assertVariableExists . . . 38
assertVariableNew . . . 39
binomProbs . . . 40
bolus . . . 42
boxCox . . . 43
coef.rxUi . . . 44
confint.rxSolve . . . 45
cvPost . . . 47
delay . . . 50
dELU . . . 52
dfWishart . . . 53
dGELU . . . 54
dlReLU . . . 55
dPReLU . . . 55
dReLU . . . 57

dSELU . . . 58
dsoftplus . . . 59
dSwish . . . 60
ELU . . . 61
erf . . . 62
et . . . 62
etExpand . . . 67
etRbind . . . 68
etRep . . . 71
etSeq . . . 74
eventTable . . . 77
evid_ . . . 79
gammap . . . 82
gammapDer . . . 83
gammapInv . . . 83
gammaq . . . 84
gammaqInv . . . 85
GELU . . . 86
genShinyApp.template . . . 87
getRxThreads . . . 88
indLin . . . 89
infuse . . . 90
infuseDur . . . 91
ini.rxUi . . . 93
ini<- . . . 95
is.rxEt . . . 96
is.rxStackData . . . 96
linCmtMicro . . . 97
linMod . . . 98
linToOde . . . 100
llikBeta . . . 101
llikBinom . . . 102
llikCauchy . . . 104
llikChisq . . . 105
llikExp . . . 106
llikF . . . 107
llikGamma . . . 108
llikGeom . . . 109
llikNbinom . . . 110
llikNbinomMu . . . 111
llikNorm . . . 112
llikPois . . . 114
llikT . . . 115
llikUnif . . . 116
llikWeibull . . . 117
logit . . . 118
lowergamma . . . 119
lReLU . . . 120

meanProbs . . . 121
mexpit . . . 123
mix . . . 124
mlogit . . . 125
model.function . . . 127
model<- . . . 129
modelExtract . . . 129
multiply . . . 132
obs . . . 133
odeMethodToInt . . . 134
odeToLin . . . 142
phantom . . . 143
phi . . . 144
phiU . . . 145
plot.rxSolve . . . 146
PReLU . . . 147
print.rxModelVars . . . 148
probit . . . 150
ReLU . . . 151
replace . . . 152
reset . . . 152
rinvchisq . . . 153
rxAllowUnload . . . 154
rxAppendModel . . . 154
rxAssignControlValue . . . 156
rxAssignPtr . . . 156
rxbeta . . . 157
rxbinom . . . 158
rxcauchy . . . 159
rxCbindStudyIndividual . . . 161
rxchisq . . . 162
rxClean . . . 163
rxCompile . . . 164
rxControlUpdateSens . . . 166
rxCreateCache . . . 167
rxD . . . 168
rxDelete . . . 169
rxDerived . . . 169
rxDfdy . . . 171
rxEtaDistExpand . . . 172
rxEtaDistMuRef . . . 173
rxEtDispatchSolve . . . 175
rxEventListen . . . 176
rxEventTableFile . . . 177
rxEvid . . . 178
rxexp . . . 179
rxf . . . 180
rxFixPop . . . 182

rxFixRes . . . 183
rxForcedPars . . . 185
rxFun . . . 186
rxgamma . . . 188
rxgeom . . . 190
rxGetControl . . . 191
rxGetDefaultSerialize . . . 192
rxGetLin . . . 192
rxGetrxode2 . . . 193
rxGetSeed . . . 193
rxHasAr . . . 194
rxHtml . . . 195
rxIndLinState . . . 195
rxIndLinStrategy . . . 196
rxInjectedPars . . . 197
rxIntToBase . . . 197
rxIntToLetter . . . 198
rxInv . . . 198
rxIsAutoSwitch . . . 199
rxIsCurrent . . . 199
rxIsDense . . . 200
rxIsImplicit . . . 201
rxIsNonStiff . . . 202
rxIsStiff . . . 203
rxLastCompile . . . 204
rxLhs . . . 205
rxLock . . . 205
rxMemoryEstimate . . . 206
rxMemSummary . . . 208
rxModelName . . . 209
rxModelNameLhs . . . 210
rxnbinom . . . 211
rxNorm . . . 213
rxnormV . . . 214
rxNumLoaded . . . 215
rxode2 . . . 215
rxode2<- . . . 243
rxode2parse . . . 245
rxode2parseAssignTranslation . . . 246
rxode2parseD . . . 247
rxode2parseGetPackagesToLoad . . . 247
rxode2parseGetPointerAssignment . . . 248
rxode2parseGetTranslation . . . 249
rxOmegaVarCovDeriv . . . 249
rxOptExpr . . . 250
rxord . . . 251
rxParams . . . 252
rxParLoader . . . 254

rxParseSuppressMsg . . . 255
rxPkg . . . 256
rxpois . . . 257
rxPp . . . 258
rxPreferredDistributionName . . . 260
rxPriorBuildSpec . . . 261
rxPriorLogDensity . . . 261
rxPriorOmegaToCholOmegaInvGrad . . . 264
rxProgress . . . 265
rxRateDur . . . 266
rxRegisterUiAssembled . . . 267
rxRegisterUiPrep . . . 268
rxRemoveControl . . . 268
rxRename . . . 269
rxReservedKeywords . . . 270
rxResidualError . . . 271
rxRmvn . . . 271
rxS . . . 274
rxSensMatExp . . . 275
rxSetActiveParLoader . . . 276
rxSetControl . . . 277
rxSetCovariateNamesForPiping . . . 277
rxSetIni0 . . . 278
rxSetPipingAuto . . . 279
rxSetProd . . . 280
rxSetProgressBar . . . 280
rxSetSeed . . . 281
rxSetSum . . . 282
rxShiny . . . 283
rxSimThetaOmega . . . 284
rxSolve . . . 288
rxSolveAdjoint . . . 320
rxSolveAdjointRk4 . . . 321
rxSolveChunked . . . 322
rxStack . . . 323
rxState . . . 324
rxStateOde . . . 325
rxSumProdModel . . . 326
rxSupportedFuns . . . 326
rxSuppressMsg . . . 327
rxSymInvChol . . . 328
rxSyncOptions . . . 329
rxSyntaxFunctions . . . 329
rxt . . . 330
rxTempDir . . . 331
rxTheme . . . 331
rxToSE . . . 332
rxTrans . . . 333

rxUdfUiControl . . . 335
rxUdfUiData . . . 336
rxUdfUiEst . . . 336
rxUdfUiExpr . . . 337
rxUdfUiFlag . . . 338
rxUdfUiIniDf . . . 338
rxUdfUiIniLhs . . . 339
rxUdfUiIsValue . . . 340
rxUdfUiLhs.dist . . . 340
rxUdfUiMv . . . 341
rxUdfUiNum . . . 342
rxUdfUiParsing . . . 343
rxUiDecompress . . . 343
rxUiDeparse . . . 344
rxUiDevelop . . . 345
rxUiEtaDistAnchors . . . 346
rxUiEtaDists . . . 347
rxUiGet.cmtLines . . . 348
rxUiPriors . . . 352
rxunif . . . 353
rxUnloadAll . . . 355
rxUse . . . 355
rxValidate . . . 356
rxweibull . . . 356
rxWithSeed . . . 358
SELU . . . 359
softplus . . . 360
splitBolus . . . 361
splitBolusInfusion-directive . . . 361
splitInfusion . . . 362
splitInfusionBolus-directive . . . 363
stat_amt . . . 364
stat_cens . . . 367
summary.rxode2 . . . 369
swapMatListWithCube . . . 370
Swish . . . 371
testIniDf . . . 372
testRxLinCmt . . . 373
testRxUnbounded . . . 374
toTrialDuration . . . 375
update.rxUi . . . 376
uppergamma . . . 377
zeroRe . . . 378

Index **380**

.cbindOme *cbind Ome*

Description

cbind Ome

Usage

```
.cbindOme(et, mat, n)
```

Arguments

et	The theta data frame
mat	The full matrix simulation from omegas
n	number of subject simulated

Value

data frame with et combined with simulated omega matrix values

Author(s)

Matthew Fidler

.collectWarnings *Collect warnings and just warn once.*

Description

Collect warnings and just warn once.

Usage

```
.collectWarnings(expr, lst = FALSE)
```

Arguments

expr	R expression
lst	When TRUE return a list with list(object,warnings) instead of issuing the warnings. Otherwise, when FALSE issue the warnings and return the object.

Value

The value of the expression or a list with the value of the expression and a list of warning messages

Author(s)

Matthew L. Fidler

.copyUi *This copies the rxode2 UI object so it can be modified*

Description

This copies the rxode2 UI object so it can be modified

Usage

```
.copyUi(ui)
```

Arguments

ui	Original UI object

Value

Copied UI object

Author(s)

Matthew L. Fidler

.handleSingleErrTypeNormOrTFoceiBase *Handle the single error for normal or t distributions*

Description

Handle the single error for normal or t distributions

Usage

```
.handleSingleErrTypeNormOrTFoceiBase(
  env,
  pred1,
  errNum = 1L,
  rxPredLlik = TRUE,
  arNorm = FALSE
)
```

Arguments

env	Environment for the parsed model
pred1	The data.frame of the current error
errNum	The number of the error specification in the nlmixr2 model
rxPredLlik	A boolean indicating if the log likelihood should be calculated for non-normal distributions. By default TRUE.
arNorm	A boolean; when TRUE, emit the AR(1) norm (conditional mean/variance) form used by the focei inner model. By default FALSE.

Value

A list of the lines added. The lines will contain

- rx_yj_ which is an integer that corresponds to the transformation type.
- rx_lambda_ is the transformation lambda
- rx_low_ The lower boundary of the transformation
- rx_hi_ The upper boundary of the transformation
- rx_pred_f_ The prediction function
- rx_pred_ The transformed prediction function
- rx_r_ The transformed variance

Author(s)

Matthew Fidler

.matchesLangTemplate *Check if a language object matches a template language object*

Description

- If template == str2lang("."), it will match anything.
- If template == str2lang(".name"), it will match any name.
- If template == str2lang(".call()"), it will match any call.

Usage

```
.matchesLangTemplate(x, template)
```

Arguments

x	The object to check
template	The template object it should match

Value

TRUE if it matches, FALSE, otherwise

Examples

```
.matchesLangTemplate(str2lang("d/dt(foo)"), str2lang("d/dt(.name)"))
.matchesLangTemplate(str2lang("d/dt(foo)"), str2lang("d/foo(.name)"))
.matchesLangTemplate(str2lang("d/dt(foo)"), str2lang("d/."))
```

.modelHandleModelLines

Handle model lines

Description

Handle model lines

Usage

```
.modelHandleModelLines(
  modelLines,
  rxui,
  modifyIni = FALSE,
  append = NULL,
  auto = getOption("rxode2.autoVarPiping", TRUE),
  cov = NULL,
  envir
)
```

Arguments

modelLines	The model lines that are being considered
rxui	The rxode2 UI object
modifyIni	Should the ini() be considered
append	This is a boolean to determine if the lines are appended in piping. The possible values for this is: • TRUE which is when the lines are appended to the model instead of replaced • FALSE when the lines are replaced in the model (default) • NA is when the lines are pre-pended to the model instead of replaced • lhs expression, which will append the lines after the last observed line of the expression lhs
auto	This boolean tells if piping automatically selects the parameters should be characterized as a population parameter, between subject variability, or a covariate. When TRUE this automatic selection occurs. When FALSE this automatic selection is turned off and everything is added as a covariate (which can be promoted to a parameter with the ini statement). By default this is TRUE, but it can be changed by options(rxode2.autoVarPiping=FALSE).

cov	is a character vector of variables that should be assumed to be covariates. This will override automatic promotion to a population parameter estimate (or an eta)
envir	Environment for evaluation

Value

New UI

Author(s)

Matthew L. Fidler

.quoteCallInfoLines *Returns quoted call information*

Description

Returns quoted call information

Usage

```
.quoteCallInfoLines(callInfo, envir = parent.frame(), iniDf = NULL)
```

Arguments

callInfo	Call information
envir	Environment for evaluation (if needed)
iniDf	The parent model iniDf when piping in a ini block (NULL otherwise)

Value

Quote call information. for name=expression, change to name<-expression in quoted call list. For expressions that are within brackets ie {}, unlist the brackets as if they were called in one single sequence.

Author(s)

Matthew L. Fidler

.rxLinCmtGen *Internal function to generate the model variables for a linCmt() model*

Description

Internal function to generate the model variables for a linCmt() model

Usage

```
.rxLinCmtGen(lenState, vars)
```

Arguments

lenState	Length of the state
vars	Variables in the model

Value

Model variables of expanded linCmt model

Author(s)

Matthew L. Fidler

.rxode2ptrs *Get the rxode2 function pointers*

Description

This function is used to get the function pointers for rxode2. This is used to allow rxode2 to have binary linkage to nlmixr2est.

Usage

```
.rxode2ptrs()
```

Value

a list of function pointers

Author(s)

Matthew L. Fidler

Examples

```
.rxode2ptrs()
```

.rxSensStrippable *Get model properties without compiling it.*

Description

Get model properties without compiling it.

Usage

```
.rxSensStrippable(mv)
```

Arguments

mv	rxode2 model variables Sensitivity-named states (rx__sens_<state>_BY_<var>__) are stripped of their d/dt() when calcJac/calcSens regenerate the sensitivity block. A state that is read by delay() is a load-bearing ODE (delay(X, T) requires d/dt(X)), so it must never be stripped – e.g. nlmixr2est's analytic-cov augmented model supplies delayed sensitivity ODEs directly. Returns the sens states safe to strip (those not referenced by any delay() term).

Value

rxode2 trans list

Author(s)

Matthew L. Fidler

.rxWithOptions *Temporarily set options then restore them while running code*

Description

Temporarily set options then restore them while running code

Usage

```
.rxWithOptions(ops, code)
```

Arguments

ops	list of options that will be temporarily set for the code
code	The code to run during the sink

Value

value of code

Examples

```
.rxWithOptions(list(digits = 21), {
  print(pi)
})

print(pi)
```

`.rxWithWd` *Temporarily set options then restore them while running code*

Description

Temporarily set options then restore them while running code

Usage

```
.rxWithWd(wd, code)
```

Arguments

wd	working directory to temporarily set the system to while evaluating the code
code	The code to run during the sink

Value

value of code

Examples

```
.rxWithWd(tempdir(), {
  getwd()
})

getwd()
```

`.toClassicEvid` *This converts NONMEM-style EVIDs to classic RxODE events*

Description

This converts NONMEM-style EVIDs to classic RxODE events

Usage

```
.toClassicEvid(cmt = 1L, amt = 0, rate = 0, dur = 0, ii = 0, evid = 0L, ss = 0)
```

Arguments

`cmt`	compartment flag
`amt`	dose amount
`rate`	dose rate
`dur`	dose duration
`ii`	inter-dose interval
`evid`	event id
`ss`	steady state

Value

classic evids, excluding evids that are added (you need to add them manually) or simply use etTran. This is mostly for testing and really shouldn't be used directly.

Author(s)

Matthew L. Fidler

Examples

```
.toClassicEvid(cmt=10, amt=3, evid=1)
.toClassicEvid(cmt=10, amt=3, rate=2, evid=1)
.toClassicEvid(cmt=10, amt=3, rate=-1, evid=1)
.toClassicEvid(cmt=10, amt=3, rate=-2, evid=1)
.toClassicEvid(cmt=10, amt=3, dur=2, evid=1)
.toClassicEvid(cmt=304, amt=3, dur=2, evid=1)
.toClassicEvid(cmt=7, amt=0, rate=2, evid=1, ss=1)
.toClassicEvid(cmt=-10, amt=3, evid=1)
.toClassicEvid(cmt=10, amt=3, evid=5)
.toClassicEvid(cmt=6, amt=3, evid=6)
.toClassicEvid(cmt=6, amt=3, evid=7)
.toClassicEvid(evid=2)
.toClassicEvid(evid=4)
```

.vecDf *Convert numeric vector to repeated data.frame*

Description

Convert numeric vector to repeated data.frame

Usage

```
.vecDf(vec, n)
```

Arguments

vec	Named input vector
n	Number of columns

Value

Data frame with repeated vec

Author(s)

Matthew Fidler

add.dosing *Add dosing to eventTable*

Description

This adds a dosing event to the event table. This is provided for piping syntax through magrittr. It can also be accessed by `eventTable$add.dosing(...)`

Usage

```
add.dosing(
  eventTable,
  dose,
  nbr.doses = 1L,
  dosing.interval = 24,
  dosing.to = 1L,
  rate = NULL,
  amount.units = NA_character_,
  start.time = 0,
  do.sampling = FALSE,
  time.units = NA_character_,
  ...
)
```

Arguments

eventTable	eventTable object; When accessed from object it would be eventTable$
dose	numeric scalar, dose amount in amount.units;
nbr.doses	integer, number of doses;
dosing.interval	required numeric scalar, time between doses in time.units, defaults to 24 of time.units="hours";
dosing.to	integer, compartment the dose goes into (first compartment by default);
rate	for infusions, the rate of infusion (default is NULL, for bolus dosing;
amount.units	optional string indicating the dosing units. Defaults to NA to indicate as per the original EventTable definition.
start.time	required dosing start time;
do.sampling	logical, should observation sampling records be added at the dosing times? Defaults to FALSE.
time.units	optional string indicating the time units. Defaults to "hours" to indicate as per the original EventTable definition.
...	Other parameters passed to et().

Value

eventTable with updated dosing (note the event table will be updated anyway)

Author(s)

Matthew L. Fidler

Matthew L Fidler, Wenping Wang

References

Wang W, Hallow K, James D (2015). "A Tutorial on rxode2: Simulating Differential Equation Pharmacometric Models in R." CPT: Pharmacometrics and Systems Pharmacology, 5(1), 3-10. ISSN 2163-8306

See Also

eventTable, add.sampling, add.dosing, et, etRep, etRbind, rxode2

Examples

```
## Not run:

library(rxode2)
library(units)

# Model from rxode2 tutorial
# Using a nlmixr2 style function
```

```
mod1 <-function(){
  ini({
    KA <- 2.94E-01
    CL <- 1.86E+01
    V2 <- 4.02E+01
    Q <- 1.05E+01
    V3 <- 2.97E+02
    Kin <- 1
    Kout <- 1
    EC50 <- 200
  })
 model({
    C2 <- centr/V2
    C3 <- peri/V3
    d/dt(depot) <- -KA*depot
    d/dt(centr) <- KA*depot - CL*C2 - Q*C2 + Q*C3
    d/dt(peri)  <-                   Q*C2 - Q*C3
    d/dt(eff)   <- Kin - Kout*(1-C2/(EC50+C2))*eff
 })
}

## These are making the more complex regimens of the rxode2 tutorial

## bid for 5 days
bid <- et(timeUnits="hr") |>
       et(amt=10000,ii=12,until=set_units(5, "days"))

## qd for 5 days
qd <- et(timeUnits="hr") |>
      et(amt=20000,ii=24,until=set_units(5, "days"))

## bid for 5 days followed by qd for 5 days

et <- seq(bid,qd) |>
      et(seq(0,11*24,length.out=100))

bidQd <- rxSolve(mod1, et)

plot(bidQd, C2)


## Now Infusion for 5 days followed by oral for 5 days

##  note you can dose to a named compartment instead of using the compartment number
infusion <- et(timeUnits = "hr") |>
      et(amt=10000, rate=5000, ii=24, until=set_units(5, "days"), cmt="centr")


qd <- et(timeUnits = "hr") |>
  et(amt=10000, ii=24, until=set_units(5, "days"), cmt="depot")

et <- seq(infusion,qd)
```

```
infusionQd <- rxSolve(mod1, et)

plot(infusionQd, C2)

## 2wk-on, 1wk-off

qd <- et(timeUnits = "hr") |>
      et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- seq(qd, set_units(1,"weeks"), qd) |>
     add.sampling(set_units(seq(0, 5.5,by=0.005),weeks))

wkOnOff <- rxSolve(mod1, et)

plot(wkOnOff, C2)

## You can also repeat the cycle easily with the rep function

qd <-et(timeUnits = "hr") |>
     et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- etRep(qd, times=4, wait=set_units(1,"weeks")) |>
      add.sampling(set_units(seq(0, 12.5,by=0.005),weeks))

repCycle4 <- rxSolve(mod1, et)

plot(repCycle4, C2)


## End(Not run)
```

`add.sampling` *Add sampling to eventTable*

Description

This adds a dosing event to the event table. This is provided for piping syntax through magrittr. It can also be accessed by `eventTable$add.sampling()`

Usage

```
add.sampling(eventTable, time, time.units = NA_character_)
```

Arguments

`eventTable`	An eventTable object. When accessed from object it would be `eventTable$`
`time`	a vector of time values (in `time.units`).
`time.units`	an optional string specifying the time units. Defaults to the units specified when the `EventTable` was initialized.

Value

eventTable with updated sampling. (Note the event table will be updated even if you don't reassign the eventTable)

Author(s)

Matthew L Fidler, Wenping Wang

References

Wang W, Hallow K, James D (2015). "A Tutorial on rxode2: Simulating Differential Equation Pharmacometric Models in R." CPT: Pharmacometrics and Systems Pharmacology, 5(1), 3-10. ISSN 2163-8306

See Also

eventTable, add.sampling, add.dosing, et, etRep, etRbind, rxode2

Examples

```
## Not run:

library(rxode2)
library(units)

# Model from rxode2 tutorial
# Using a nlmixr2 style function

mod1 <-function(){
  ini({
    KA <- 2.94E-01
    CL <- 1.86E+01
    V2 <- 4.02E+01
    Q <- 1.05E+01
    V3 <- 2.97E+02
    Kin <- 1
    Kout <- 1
    EC50 <- 200
  })
 model({
    C2 <- centr/V2
    C3 <- peri/V3
    d/dt(depot) <- -KA*depot
    d/dt(centr) <- KA*depot - CL*C2 - Q*C2 + Q*C3
    d/dt(peri)  <-                   Q*C2 - Q*C3
    d/dt(eff)   <- Kin - Kout*(1-C2/(EC50+C2))*eff
 })
}

## These are making the more complex regimens of the rxode2 tutorial
```

```
## bid for 5 days
bid <- et(timeUnits="hr") |>
       et(amt=10000,ii=12,until=set_units(5, "days"))

## qd for 5 days
qd <- et(timeUnits="hr") |>
      et(amt=20000,ii=24,until=set_units(5, "days"))

## bid for 5 days followed by qd for 5 days

et <- seq(bid,qd) |>
      et(seq(0,11*24,length.out=100))

bidQd <- rxSolve(mod1, et)

plot(bidQd, C2)


## Now Infusion for 5 days followed by oral for 5 days

##  note you can dose to a named compartment instead of using the compartment number
infusion <- et(timeUnits = "hr") |>
      et(amt=10000, rate=5000, ii=24, until=set_units(5, "days"), cmt="centr")


qd <- et(timeUnits = "hr") |>
  et(amt=10000, ii=24, until=set_units(5, "days"), cmt="depot")

et <- seq(infusion,qd)

infusionQd <- rxSolve(mod1, et)

plot(infusionQd, C2)

## 2wk-on, 1wk-off

qd <- et(timeUnits = "hr") |>
      et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- seq(qd, set_units(1,"weeks"), qd) |>
     add.sampling(set_units(seq(0, 5.5,by=0.005),weeks))

wkOnOff <- rxSolve(mod1, et)

plot(wkOnOff, C2)

## You can also repeat the cycle easily with the rep function

qd <-et(timeUnits = "hr") |>
     et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- etRep(qd, times=4, wait=set_units(1,"weeks")) |>
      add.sampling(set_units(seq(0, 12.5,by=0.005),weeks))
```

```
repCycle4 <- rxSolve(mod1, et)

plot(repCycle4, C2)


## End(Not run)
```

`as.arrow` *Convert an rxSolveOom result to a lazy Arrow Dataset*

Description

Opens all parquet chunk files as a single lazy `arrow::Dataset` using `arrow::open_dataset()`. The dataset can be filtered and selected with dplyr verbs before calling `dplyr::collect()` to materialise. Requires the `arrow` package.

Usage

```
as.arrow(x, ...)

## S3 method for class 'rxSolveOom'
as.arrow(x, ...)
```

Arguments

`x`	An `rxSolveOom` object.
`...`	Ignored.

Details

This is an rxode2 generic: **arrow** provides `open_dataset()` but no lazy-dataset coercion generic to dispatch on, so rxode2 defines its own.

Value

An `arrow::Dataset`.

as.data.table.rxEt *Convert an event table to a data.table*

Description

Convert an event table to a data.table

Usage

```
## S3 method for class 'rxEt'
as.data.table(x, keep.rownames = FALSE, ...)
```

Arguments

x	An R object.
keep.rownames	Default is FALSE. If TRUE, adds the input object's names as a separate column named "rn". keep.rownames = "id" names the column "id" instead. For lists and when calling data.table(), names from the first named vector are extracted and used as row names, similar to data.frame() behavior.
...	Additional arguments to be passed to or from other methods.

Value

data.table of event table

as.et *Coerce object to data.frame*

Description

Coerce object to data.frame

Usage

```
as.et(x, ...)

## Default S3 method:
as.et(x, ...)
```

Arguments

x	Object to coerce to et.
...	Other parameters

Value

An event table

as.ini *Turn into an ini block for initialization*

Description

Turn into an ini block for initialization

Usage

```
as.ini(x)

## S3 method for class 'character'
as.ini(x)

## S3 method for class 'data.frame'
as.ini(x)

## S3 method for class 'call'
as.ini(x)

## S3 method for class 'lotriFix'
as.ini(x)

## S3 method for class 'matrix'
as.ini(x)

## Default S3 method:
as.ini(x)
```

Arguments

x	Item to convert to a rxode2/nlmixr2 ui ini expression

Value

rxode2 ini expression

Author(s)

Matthew L. Fidler

Examples

```
ini <- quote(ini({
  tka <- log(1.57)
  tcl <- log(2.72)
  tv <- log(31.5)
  eta.ka ~ 0.6
```

```
  eta.cl ~ 0.3
  eta.v ~ 0.1
  add.sd <- 0.7
}))

as.ini(ini)

l <- quote(lotri({
  tka <- log(1.57)
  tcl <- log(2.72)
  tv <- log(31.5)
  eta.ka ~ 0.6
  eta.cl ~ 0.3
  eta.v ~ 0.1
  add.sd <- 0.7
 }))

as.ini(l)

m <- lotri({
   eta.ka ~ 0.6
   eta.cl ~ 0.3
   eta.v ~ 0.1
})

as.ini(m)

one.compartment <- function() {
  ini({
    tka <- log(1.57)
    tcl <- log(2.72)
    tv <- log(31.5)
    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)
    d/dt(depot) = -ka * depot
    d/dt(center) = ka * depot - cl / v * center
    cp = center / v
    cp ~ add(add.sd)
  })
}

as.ini(one.compartment)

ui <- one.compartment()

as.ini(ui)
```

```
ui$iniDf

as.ini(ui$iniDf)

ini <- c("ini({",
          "tka <- log(1.57)",
          "tcl <- log(2.72)",
          "tv <- log(31.5)",
          "eta.ka ~ 0.6",
          "eta.cl ~ 0.3",
          "eta.v ~ 0.1",
          "add.sd <- 0.7",
          "})")

as.ini(ini)

ini <- paste(ini, collapse="\n")

as.ini(ini)
```

`as.model` *Turn into a model expression*

Description

Turn into a model expression

Usage

```
as.model(x)

## S3 method for class 'character'
as.model(x)

## S3 method for class 'call'
as.model(x)

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

## Default S3 method:
as.model(x)
```

Arguments

`x`	item to convert to a `model({})` expression

Value

model expression

Author(s)

Matthew L. Fidler

Examples

```
model <- quote(model({
  ka <- exp(tka + eta.ka)
  cl <- exp(tcl + eta.cl)
  v <- exp(tv + eta.v)
  d/dt(depot) = -ka * depot
  d/dt(center) = ka * depot - cl / v * center
  cp = center / v
  cp ~ add(add.sd)
}))

as.model(model)

one.compartment <- function() {
  ini({
    tka <- log(1.57)
    tcl <- log(2.72)
    tv <- log(31.5)
    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)
    d/dt(depot) = -ka * depot
    d/dt(center) = ka * depot - cl / v * center
    cp = center / v
    cp ~ add(add.sd)
  })
}

as.model(one.compartment)

ui <- one.compartment()

as.model(ui)

model <- c("model({",
           "ka <- exp(tka + eta.ka)",
           "cl <- exp(tcl + eta.cl)",
           "v <- exp(tv + eta.v)",
```

```
            "d/dt(depot) = -ka * depot",
            "d/dt(center) = ka * depot - cl / v * center",
            "cp = center / v",
            "cp ~ add(add.sd)",
            "})")

as.model(model)

model <- paste(model, collapse="\n")

as.model(model)
```

`as.rxUi` *As rxode2 ui*

Description

As rxode2 ui

Usage

```
as.rxUi(x)

## S3 method for class 'rxode2'
as.rxUi(x)

## S3 method for class 'rxode2tos'
as.rxUi(x)

## S3 method for class 'rxModelVars'
as.rxUi(x)

## S3 method for class '`function`'
as.rxUi(x)

## S3 method for class 'rxUi'
as.rxUi(x)

## Default S3 method:
as.rxUi(x)
```

Arguments

`x`	Object to convert to `rxUi` object

Value

rxUi object (or error if it cannot be converted)

Author(s)

Matthew L. Fidler

Examples

```
mod1 <- function() {
 ini({
   # central
   KA=2.94E-01
   CL=1.86E+01
   V2=4.02E+01
   # peripheral
   Q=1.05E+01
   V3=2.97E+02
   # effects
   Kin=1
   Kout=1
   EC50=200
 })
 model({
   C2 <- centr/V2
   C3 <- peri/V3
   d/dt(depot) <- -KA*depot
   d/dt(centr) <- KA*depot - CL*C2 - Q*C2 + Q*C3
   d/dt(peri)  <- Q*C2 - Q*C3
   eff(0) <- 1
   d/dt(eff)   <- Kin - Kout*(1-C2/(EC50+C2))*eff
 })
}

as.rxUi(mod1)
```

---

as_tibble.rxEt *Convert to tbl*

Description

Convert to tbl

Usage

```
as_tibble.rxEt(x, ...)
```

Arguments

x	rxode2 event table
...	Other arguments to as_tibble

Value

tibble of event table

`assertCompartmentExists` *Verify that the compartment exists in a model*

Description

Verify that the compartment exists in a model

Usage

```
assertCompartmentExists(ui, x)

testCompartmentExists(ui, x)
```

Arguments

`ui`	is the model to test
`x`	The value to test (can be a vector of strings)

Value

the value of the compartment that exists; if it is a vector returns the first item that matches

Functions

- `testCompartmentExists()`: Test if compartment exists

Author(s)

Matthew Fidler & Bill Denney

See Also

Other Assertions: `assertCompartmentName()`, `assertCompartmentNew()`, `assertRxUi()`, `assertVariableExists()`, `assertVariableNew()`, `rxPriorBuildSpec()`, `rxPriorLogDensity()`, `rxPriorOmegaToCholOmegaInvGrad()`, `rxUiPriors()`, `testIniDf()`, `testRxUnbounded()`

`assertCompartmentName` *Verify that a value is a valid nlmixr2 compartment name*

Description

Verify that a value is a valid nlmixr2 compartment name

Usage

```
assertCompartmentName(x)

assertVariableName(x)

assertParameterValue(x)

assertExists(ui, x)

testExists(ui, x)
```

Arguments

`x`	The value to test
`ui`	when needed, this is the rxode2/nlmixr2 model

Value

The value or an error

Functions

- `assertVariableName()`: Verify that a value is a valid nlmixr2 variable name
- `assertParameterValue()`: Verify that a value is a valid nlmixr2 parameter value
- `assertExists()`: Assert compartment/variable exists
- `testExists()`: Test compartment/variable exists

Author(s)

Bill Denney

See Also

Other Assertions: `assertCompartmentExists()`, `assertCompartmentNew()`, `assertRxUi()`, `assertVariableExists()`, `assertVariableNew()`, `rxPriorBuildSpec()`, `rxPriorLogDensity()`, `rxPriorOmegaToCholOmegaInvGrad()`, `rxUiPriors()`, `testIniDf()`, `testRxUnbounded()`

assertCompartmentNew *Verify that a compartment would be new to the model*

Description

Verify that a compartment would be new to the model

Usage

```
assertCompartmentNew(ui, x)
```

Arguments

ui	is the model to test that a model paramet exists
x	The value to test

Value

The value or an error

Author(s)

Matthew Fidler & Bill Denney

See Also

Other Assertions: assertCompartmentExists(), assertCompartmentName(), assertRxUi(), assertVariableExists(), assertVariableNew(), rxPriorBuildSpec(), rxPriorLogDensity(), rxPriorOmegaToCholOmegaInvGrad(), rxUiPriors(), testIniDf(), testRxUnbounded()

assertRxUi *Assert properties of the rxUi models*

Description

Assert properties of the rxUi models

Usage

```
assertRxUi(ui, extra = "", .var.name = .vname(ui))

assertRxUiPrediction(ui, extra = "", .var.name = .vname(ui))

assertRxUiIovNoCor(ui, extra = "", .var.name = .vname(ui))

assertRxUiNoMix(ui, extra = "", .var.name = .vname(ui))

assertRxUiNoAutoregressive(ui, extra = "", .var.name = .vname(ui))

assertRxUiSingleEndpoint(ui, extra = "", .var.name = .vname(ui))

assertRxUiTransformNormal(ui, extra = "", .var.name = .vname(ui))

assertRxUiNormal(ui, extra = "", .var.name = .vname(ui))

testRxUiPriors(ui, extra = "", .var.name = .vname(ui))

testRxUiNormalPriors(ui, extra = "", .var.name = .vname(ui))

testRxUiOmegaDf(ui, extra = "", .var.name = .vname(ui))

testRxUiOmegaNormalPriors(ui, extra = "", .var.name = .vname(ui))

assertRxUiNoPriors(ui, extra = "", .var.name = .vname(ui))

assertRxUiNormalPriors(ui, extra = "", .var.name = .vname(ui))

assertRxUiNoOmegaDf(ui, extra = "", .var.name = .vname(ui))

assertRxUiNoOmegaNormalPriors(ui, extra = "", .var.name = .vname(ui))

assertRxUiMuRefOnly(ui, extra = "", .var.name = .vname(ui))

assertRxUiEstimatedResiduals(ui, extra = "", .var.name = .vname(ui))

assertRxUiNoFixedResiduals(ui, extra = "", .var.name = .vname(ui))

assertRxUiNoFixedOmega(ui, extra = "", .var.name = .vname(ui))

assertRxUiTransform(ui, transform, extra = "", .var.name = .vname(ui))

assertRxUiErrType(ui, errType, extra = "", .var.name = .vname(ui))

assertRxUiAddProp(ui, addProp, extra = "", .var.name = .vname(ui))

assertRxUiPopulationOnly(ui, extra = "", .var.name = .vname(ui))
```

```
assertRxUiMixedOnly(ui, extra = "", .var.name = .vname(ui))

assertRxUiRandomOnIdOnly(ui, extra = "", .var.name = .vname(ui))
```

Arguments

ui	Model to check
extra	Extra text to append to the error message (like "for focei")
.var.name	[character(1)] Name of the checked object to print in assertions. Defaults to the heuristic implemented in vname.
transform	character vector of the residual transformations that are allowed (the levels of ui$predDf$transform, like "untransformed", "lnorm" or "boxCox")
errType	character vector of the residual error types that are allowed ("add", "prop", "pow", "add + prop" or "add + pow")
addProp	character vector of the add() + prop() and add() + pow() combinations that are allowed ("combined1" or "combined2")

Details

These functions have different types of assertions

- assertRxUi – Make sure this is a proper rxode2 model (if not throw error)
- assertRxUiSingleEndpoint – Make sure the rxode2 model is only a single endpoint model (if not throw error)
- assertRxUiTransformNormal – Make sure that the model residual distribution is normal or transformably normal
- assertRxUiNormal – Make sure that the model residual distribution is normal
- assertRxUiEstimatedResiduals – Make sure that the residual error parameters are estimated (not modeled).
- assertRxUiNoFixedResiduals – Make sure that none of the residual error parameters are fixed (ie add.sd <- fix(0.7), a literal add(0.7), or a fixed distribution parameter like the lambda of pois(lambda)); used by estimation methods that always estimate them, so a fixed value is an error instead of being silently estimated
- assertRxUiNoFixedOmega – Make sure that none of the between-subject variability parameters are fixed (ie eta.ka ~ fix(0.6)); used by estimation methods that always estimate them, so a fixed value is an error instead of being silently estimated
- assertRxUiTransform – Make sure that every endpoint with a residual error (normal, or with dnorm(), dt() or dcauchy()) uses one of the transform residual transformations (like "untransformed" or "lnorm"); used by estimation methods that support only some transformations. Endpoints without a residual error (like pois() or ll()) are not checked; see assertRxUiTransformNormal
- assertRxUiErrType – Make sure that every endpoint with a residual error uses one of the errType residual error types ("add", "prop", "pow", "add + prop", "add + pow"); used by estimation methods that support only some residual error types

- `assertRxUiAddProp` – Make sure that every `add()` + `prop()` or `add()` + `pow()` residual error uses one of the `addProp` combinations (`"combined1"`, where the standard deviations add, or `"combined2"`, where the variances add); an endpoint declared without `combined1()` or `combined2()` uses `rxControl(addProp=)` or, when that is not set, `getOption("rxode2.addProp", "combined2")`
- `assertRxUiPopulationOnly` – Make sure the model is the population only model (no mixed effects)
- `assertRxUiMixedOnly` – Make sure the model is a mixed effect model (not a population effect, only)
- `assertRxUiPrediction` – Make sure the model has predictions
- `assertRxUiMuRefOnly` – Make sure that all the parameters are mu-referenced
- `assertRxUiRandomOnIdOnly` – Make sure there are only random effects at the ID level
- `assertRxUiIovNoCor` – Make sure that the IOV model does not have any correlations
- `assertRxUiNoMix` – Make sure that the model does not have a mixture model inside it
- `assertRxUiNoAutoregressive` – Make sure the model does not have an autoregressive residual (ie `ar()`); used by estimation methods that do not support it
- `assertRxUiNoPriors` – Make sure the model does not specify any prior distributions; used by estimation methods that cannot use them, so that a specified prior is an error instead of being silently ignored
- `assertRxUiNormalPriors` – Make sure that every prior the model specifies is a normal prior (`dnorm()`, `stdNormal()`, or the multivariate `multiNormal()` that the `lotri` normal prior shorthand produces for correlated parameters); used by estimation methods that support priors but only normal ones
- `assertRxUiNoOmegaDf` – Make sure the model does not give prior degrees of freedom for an omega block (ie `invWishart(4)`, the `$OMEGAPD` of a NONMEM NWPRI model); used by estimation methods that cannot use them
- `assertRxUiNoOmegaNormalPriors` – Make sure the model does not put a normal prior on an omega parameter (ie `om.eta.cl ~ 0.01`, what a NONMEM TNPRI model needs); used by estimation methods that can put a prior on an omega but only a Wishart one

Value

the rxUi model

Author(s)

Matthew L. Fidler

See Also

Other Assertions: `assertCompartmentExists()`, `assertCompartmentName()`, `assertCompartmentNew()`, `assertVariableExists()`, `assertVariableNew()`, `rxPriorBuildSpec()`, `rxPriorLogDensity()`, `rxPriorOmegaToCholOmegaInvGrad()`, `rxUiPriors()`, `testIniDf()`, `testRxUnbounded()`

Examples

```
one.cmt <- function() {
 ini({
   tka <- 0.45; label("Ka")
   tcl <- log(c(0, 2.7, 100)); label("Cl")
   tv <- 3.45; label("V")
   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)
 })
}

assertRxUi(one.cmt)
# assertRxUi(rnorm) # will fail

assertRxUiSingleEndpoint(one.cmt)

assertRxUiNoFixedResiduals(one.cmt)

assertRxUiNoFixedOmega(one.cmt)

assertRxUiTransform(one.cmt, c("untransformed", "lnorm"))

assertRxUiErrType(one.cmt, c("add", "prop", "add + prop"))

assertRxUiAddProp(one.cmt, "combined2")
```

---

assertVariableExists *Assert a variable exists in the model*

Description

Assert a variable exists in the model

Usage

```
assertVariableExists(ui, x)

testVariableExists(ui, x)
```

Arguments

ui	rxode2 ui model
x	does the x variable exist in the model. If it is a vector of variable check to see if any exists, but all must be valid nlmixr2 variable names

Value

variable that matches, in the case of multiple variables, the first that matches. If nothing matches return error

Functions

- testVariableExists(): Test if variable exists

Author(s)

Matthew L. Fidler

See Also

Other Assertions: assertCompartmentExists(), assertCompartmentName(), assertCompartmentNew(), assertRxUi(), assertVariableNew(), rxPriorBuildSpec(), rxPriorLogDensity(), rxPriorOmegaToCholOmegaInvGr rxUiPriors(), testIniDf(), testRxUnbounded()

---

assertVariableNew *Assert a variable would be new to the model*

Description

Assert a variable would be new to the model

Usage

```
assertVariableNew(ui, x)
```

Arguments

ui	rxode2 ui model
x	would the variable x variable be new in the model

Value

nothing, but will error if x would not be new

Author(s)

Matthew L. Fidler

See Also

Other Assertions: assertCompartmentExists(), assertCompartmentName(), assertCompartmentNew(), assertRxUi(), assertVariableExists(), rxPriorBuildSpec(), rxPriorLogDensity(), rxPriorOmegaToCholOmegaIr rxUiPriors(), testIniDf(), testRxUnbounded()

`binomProbs` *Calculate expected confidence bands with binomial sampling distribution*

Description

This is meant to perform in the same way as `quantile()` so it can be a drop in replacement for code using `quantile()` but using distributional assumptions.

Usage

```
binomProbs(x, ...)

## Default S3 method:
binomProbs(
  x,
  probs = c(0.025, 0.05, 0.5, 0.95, 0.975),
  na.rm = FALSE,
  names = TRUE,
  onlyProbs = TRUE,
  n = 0L,
  m = 0L,
  pred = FALSE,
  piMethod = "lim",
  M = 5e+05,
  tol = .Machine$double.eps^0.25,
  ciMethod = c("wilson", "wilsonCorrect", "agrestiCoull", "wald", "wc", "ac"),
  ...
)
```

Arguments

`x`	numeric vector whose mean and probability based confidence values are wanted, NA and NaN values are not allowed in numeric vectors unless `na.rm` is `TRUE`.
`...`	Arguments passed to default method, allows many different methods to be applied.
`probs`	numeric vector of probabilities with values in the interval 0 to 1, inclusive. When 0, it represents the maximum observed, when 1, it represents the maximum observed. When 0.5 it represents the expected probability (mean).
`na.rm`	logical; if true, any NA and NaN's are removed from `x` before the quantiles are computed.

names	logical; if true, the result has a names attribute.
onlyProbs	logical; if true, only return the probability based confidence interval/prediction interval estimates, otherwise return extra statistics.
n	integer/integerish; this is the n used to calculate the prediction or confidence interval. When n=0 (default) use the number of non-NA observations. When calculating the prediction interval, this represents the number of observations used in the input ("true") distribution.
m	integer. When using the prediction interval this represents the number of samples that will be observed in the future for the prediction interval.
pred	Use a prediction interval instead of a confidence interval. By default this is FALSE.
piMethod	gives the prediction interval method (currently only lim) from Lu 2020
M	number of simulations to run for the LIM PI.
tol	tolerance of root finding in the LIM prediction interval
ciMethod	gives the method for calculating the confidence interval. Can be: • "argestiCoull" or "ac" – Agresti-Coull method. For a 95\ interval, this method does not use the concept of "adding 2 successes and 2 failures," but rather uses the formulas explicitly described in the following link: https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval#Agresti-Coull_Interval. • "wilson" – Wilson Method • "wilsonCorrect" or "wc" – Wilson method with continuity correction • "wald" – Wald confidence interval or standard z approximation.

Details

It is used for confidence intervals with rxode2 solved objects using `confint(mean="binom")`

Value

By default the return has the probabilities as names (if named) with the points where the expected distribution are located given the sampling mean and standard deviation. If `onlyProbs=FALSE` then it would prepend mean, variance, standard deviation, minimum, maximum and number of non-NA observations.

Author(s)

Matthew L. Fidler

References

- Newcombe, R. G. (1998). "Two-sided confidence intervals for the single proportion: comparison of seven methods". Statistics in Medicine. 17 (8): 857-872. doi:10.1002/(SICI)1097-0258(19980430)17:8<857::AID-SIM777>3.0.CO;2-E. PMID 9595616.

- Hezhi Lu, Hua Jin, A new prediction interval for binomial random variable based on inferential models, Journal of Statistical Planning and Inference, Volume 205, 2020, Pages 156-174, ISSN 0378-3758, https://doi.org/10.1016/j.jspi.2019.07.001.

Examples

```
x<- rbinom(7001, p=0.375, size=1)
binomProbs(x)

# you can also use the prediction interval

binomProbs(x, pred=TRUE)


# Can get some extra statistics if you request onlyProbs=FALSE
binomProbs(x, onlyProbs=FALSE)

x[2] <- NA_real_

binomProbs(x, onlyProbs=FALSE)

binomProbs(x, na.rm=TRUE)
```

`bolus` *Administer a bolus dose inside a rxode2 model*

Description

Administer a bolus dose inside a rxode2 model

Usage

```
bolus(amt, cmt = 1, ii = 0, addl = 0, ss = 0)
```

Arguments

`amt`	Numeric dose amount (for dose events) or `0` for observations. When `rate > 0` this is interpreted as the total infusion amount and the infusion duration is `amt / rate`.
`cmt`	Integer compartment number (1-based) to which the dose is applied. Default is 1
`ii`	Numeric inter-dose interval. Used together with `addl` to schedule repeat doses at `time`, `time + ii`, `time + 2*ii`, ..., `time + addl*ii`. Also required when `ss > 0`. Default 0
`addl`	Integer number of *additional* doses beyond the first. The total number of doses pushed is `addl + 1`, spaced `ii` apart. Each dose is pushed as a standalone event (not as a periodic schedule in the event table).

`ss` Integer steady-state flag applied to the *first* dose only (`addl` repetitions always use `ss = 0`).

`0` No steady-state (default).

`1` Steady-state additive: add SS solution to current state.

`2` Steady-state replace: replace current state with SS solution.

Details

Behavior inside a model:

`bolus()` is evaluated at every output time point (when the solver is exactly at a scheduled event time). The pushed event is inserted into the individual's event timeline and the solver visits it now or at the specified future time.

The number of events that may be pushed per individual is limited by the `maxExtra` argument of `rxSolve()`. When `maxExtra = 0` (the default) there is no limit. Exceeding the limit causes an error.

Past-time pushes (where `time < t`) are silently ignored and counted; a warning is issued after solving.

Value

This function is only meaningful inside an rxode2 model; it returns `NULL` invisibly if called from R directly (after signaling an error).

Author(s)

Matthew L. Fidler

---

boxCox *boxCox/yeoJohnson and inverse boxCox/yeoJohnson functions*

Description

boxCox/yeoJohnson and inverse boxCox/yeoJohnson functions

Usage

```
boxCox(x, lambda = 1)

boxCoxInv(x, lambda = 1)

yeoJohnson(x, lambda = 1)

yeoJohnsonInv(x, lambda = 1)
```

Arguments

`x`	input value(s) to transform
`lambda`	lambda value for the transformation

Value

values from boxCox and boxCoxInv

Examples

```
boxCox(10, 0.5)

boxCoxInv(4.32, 0.5)

yeoJohnson(10, 0.5)

yeoJohnsonInv(4.32, 0.5)
```

`coef.rxUi` *Extract the initial coefficients from an rxode2 model (rxUi)*

Description

For an rxode2 model specification (`rxUi`) there are no realized random effects (etas), so `coef()` returns the fixed-effect (`theta`) estimates by default. The random-effect variability matrix (`omega`) can be requested with `level="omega"`, or both can be returned together with `level="all"`.

Usage

```
## S3 method for class 'rxUi'
coef(
  object,
  level = c("theta", "fixed", "omega", "random", "all", "both"),
  ...
)

## S3 method for class '`function`'
coef(
  object,
  level = c("theta", "fixed", "omega", "random", "all", "both"),
  ...
)
```

Arguments

`object`	rxode2 ui model or a function that evaluates to one
`level`	which coefficients to return: `"theta"`/`"fixed"` (the default) returns the named fixed-effect estimates; `"omega"`/`"random"` returns the random-effect variability matrix (or `NULL` when the model has no random effects); `"all"`/`"both"` returns a list with `theta` and `omega`.
`...`	other parameters (currently ignored)

Value

depends on `level`: a named numeric vector of fixed effects, the `omega` matrix (or `NULL`), or a list with both `theta` and `omega`.

Author(s)

Matthew L. Fidler

Examples

```
one.compartment <- function() {
  ini({
    tka <- 0.45
    tcl <- 1
    tv <- 3.45
    eta.ka ~ 0.6
  })
  model({
    ka <- exp(tka + eta.ka)
    cl <- exp(tcl)
    v <- exp(tv)
    d/dt(depot) <- -ka * depot
    d/dt(center) <- ka * depot - cl / v * center
    cp <- center / v
  })
}

ui <- one.compartment()
coef(ui)
coef(ui, level="omega")
coef(ui, level="all")
```

---

confint.rxSolve *Simulated percentiles, with confidence bands, from a solved rxode2 object*

Description

Summarizes a solved object into the percentiles of the simulated values at each time and, when the simulation can support it, a confidence band around each of those percentiles.

Usage

```
## S3 method for class 'rxSolve'
confint(object, parm = NULL, level = 0.95, ...)
```

Arguments

object	solved rxode2 object
parm	compartments or calculated (lhs) variables to summarize; when NULL everything rxStack() returns is summarized
level	width of the interval taken over the simulated individuals, that is, which percentiles are reported at each time
...	other options: • ci – width of the confidence band placed around each percentile, defaulting to level. ci = FALSE (or 0) returns the percentiles with no band. • mean – when TRUE report the mean and its interval with meanProbs() instead of the empirical quantiles; mean = "binom" uses binomProbs() for a 0/1 variable. • by – character vector of extra columns of object to stratify by. • useT, pred – passed to meanProbs(); n, m, M, tol, pred, ciMethod – passed to binomProbs(). • doSim – passed to rxStack().

Details

The percentiles are always taken over the simulated individuals; how (and whether) the band around them is obtained depends on the simulation:

- ci = FALSE – no band; the pooled percentiles are returned.
- nStud > 1 – the percentiles are computed within each study and the band is the quantile of those study-level percentiles. This is the meaningful case, since the studies differ by the thetaMat/omega uncertainty draw.
- a single study of at least 2500 individuals – the individuals are split into round(sqrt(n)) sub-samples, and the band is the quantile of the sub-sample percentiles, that is, the sampling variability of the percentile itself. It does not include parameter uncertainty.
- anything smaller – no band, with a message saying so.

When the solve was given a thetaMat, confint() also says whether that thetaMat was actually drawn from – it is ignored unless the variability is being simulated (nStud > 1, or simVariability=TRUE) – so it is clear whether the summarized values carry parameter uncertainty. This describes the simulated values, not the band: a solve can carry parameter uncertainty and still have no study dimension left to place a band with (nSub = 1, or ci = FALSE).

Value

A data.frame (a tibble when **tibble** is present) with one row per time, endpoint and requested percentile. Without a band it is class rxSolveConfint1 with the percentile in p1 and its value in eff; with a band it is class rxSolveConfint2 with the percentile in p1 and the band in the p<lower>, p50 and p<upper> columns. Both carry a Percentile label used by plot().

Author(s)

Matthew L. Fidler

Examples

```
mod <- function() {
  ini({
    tka <- 0.45
    tcl <- 1
    tv <- 3.45
    eta.cl ~ 0.1
    add.sd <- 0.7
  })
  model({
    ka <- exp(tka)
    cl <- exp(tcl + eta.cl)
    v <- exp(tv)
    d/dt(depot) <- -ka * depot
    d/dt(center) <- ka * depot - cl / v * center
    cp <- center / v
    cp ~ add(add.sd)
  })
}

ev <- et(amt=100) |> et(seq(0, 24, length.out=25))

# 100 individuals in one study: percentiles only
s <- rxSolve(mod, ev, nSub=100)

confint(s, "cp", level=0.95, ci=FALSE)

# with 20 studies the percentiles get a confidence band, and the
# `thetaMat` is drawn from
s2 <- rxSolve(mod, ev, nSub=100, nStud=20, thetaMat=lotri(tka ~ 0.01))

confint(s2, "cp", level=0.95)
```

`cvPost` *Sample a covariance Matrix from the Posterior Inverse Wishart distribution.*

Description

Note this Inverse wishart rescaled to match the original scale of the covariance matrix.

Usage

```
cvPost(
  nu,
```

```
  omega,
  n = 1L,
  omegaIsChol = FALSE,
  returnChol = FALSE,
  type = c("invWishart", "lkj", "separation"),
  diagXformType = c("log", "identity", "variance", "nlmixrSqrt", "nlmixrLog",
    "nlmixrIdentity")
)
```

Arguments

nu	Degrees of Freedom (Number of Observations) for covariance matrix simulation.
omega	Either the estimate of covariance matrix or the estimated standard deviations in matrix form each row forming the standard deviation simulated values
n	Number of Matrices to sample. By default this is 1. This is only useful when omega is a matrix. Otherwise it is determined by the number of rows in the input omega matrix of standard deviations
omegaIsChol	is an indicator of if the omega matrix is in the Cholesky decomposition. This is only used when type="invWishart"
returnChol	Return the Cholesky decomposition of the covariance matrix sample. This is only used when type="invWishart"
type	The type of covariance posterior that is being simulated. This can be: • invWishart The posterior is an inverse wishart; This allows for correlations between parameters to be modeled. All the uncertainty in the parameter is captured in the degrees of freedom parameter. • lkj The posterior separates the standard deviation estimates (modeled outside and provided in the omega argument) and the correlation estimates. The correlation estimate is simulated with the rLKJ1(). This simulation uses the relationship eta=(nu-1)/2. This is relationship based on the proof of the relationship between the restricted LKJ-distribution and inverse wishart distribution (XXXXXX). Once the correlation posterior is calculated, the estimated standard deviations are then combined with the simulated correlation matrix to create the covariance matrix. • separation Like the lkj option, this separates out the estimation of the correlation and standard deviation. Instead of using the LKJ distribution to simulate the correlation, it simulates the inverse wishart of the identity matrix and converts the result to a correlation matrix. This correlation matrix is then used with the standard deviation to calculate the simulated covariance matrix.
diagXformType	Diagonal transformation type. These could be: • log The standard deviations are log transformed, so the actual standard deviations are exp(omega) • identity The standard deviations are not transformed. The standard deviations are not transformed; They should be positive.

- `variance` The variances are specified in the `omega` matrix; They are transformed into standard deviations.
- `nlmixrSqrt` These standard deviations come from an nlmixr omega matrix where diag(chol(inv(omega))) = x^2
- `nlmixrLog` These standard deviations come from a nlmixr omega matrix omega matrix where diag(chol(solve(omega))) = exp(x)
- `nlmixrIdentity` These standard deviations come from a nlmixr omega matrix omega matrix where diag(chol(solve(omega))) = x

The nlmixr transformations only make sense when there is no off-diagonal correlations modeled.

Details

If your covariance matrix is a 1x1 matrix, this uses an scaled inverse chi-squared which is equivalent to the Inverse Wishart distribution in the uni-directional case.

In general, the separation strategy is preferred for diagonal matrices. If the dimension of the matrix is below 10, `lkj` is numerically faster than `separation` method. However, the `lkj` method has densities too close to zero (XXXX) when the dimension is above 10. In that case, though computationally more expensive `separation` method performs better.

For matrices with modeled covariances, the easiest method to use is the inverse Wishart which allows the simulation of correlation matrices (XXXX). This method is more well suited for well behaved matrices, that is the variance components are not too low or too high. When modeling nonlinear mixed effects modeling matrices with too high or low variances are considered sub-optimal in describing a system. With these rules in mind, it is reasonable to use the inverse Wishart.

Value

a matrix (n=1) or a list of matrices (n > 1)

Author(s)

Matthew L.Fidler & Wenping Wang

References

Alvarez I, Niemi J and Simpson M. (2014) *Bayesian Inference for a Covariance Matrix*. Conference on Applied Statistics in Agriculture.

Wang1 Z, Wu Y, and Chu H. (2018) *On Equivalence of the LKJ distribution and the restricted Wishart distribution*. <doi:10.48550/arXiv.1809.047463

Examples

```
## Sample a single covariance.
draw1 <- cvPost(3, matrix(c(1, .3, .3, 1), 2, 2))

## Sample 3 covariances
set.seed(42)
draw3 <- cvPost(3, matrix(c(1, .3, .3, 1), 2, 2), n = 3)
```

```
## Sample 3 covariances, but return the cholesky decomposition
set.seed(42)
draw3c <- cvPost(3, matrix(c(1, .3, .3, 1), 2, 2), n = 3, returnChol = TRUE)

## Sample 3 covariances with lognormal standard deviations via LKJ
## correlation sample
cvPost(3, sapply(1:3, function(...) {
  rnorm(10)
}), type = "lkj")

## or return cholesky decomposition
cvPost(3, sapply(1:3, function(...) {
  rnorm(10)
}),
type = "lkj",
returnChol = TRUE
)

## Sample 3 covariances with lognormal standard deviations via separation
## strategy using inverse Wishart correlation sample
cvPost(3, sapply(1:3, function(...) {
  rnorm(10)
}), type = "separation")

## or returning the cholesky decomposition
cvPost(3, sapply(1:3, function(...) {
  rnorm(10)
}),
type = "separation",
returnChol = TRUE
)
```

`delay` *Delayed state for delay differential equations*

Description

`delay(state, T)` evaluates an ODE state at the past time `t - T`, turning an ordinary differential equation model into a delay differential equation (DDE). The semantics match the `delay()` function of Monolix.

Usage

```
delay(state, T)
```

Arguments

`state`	An ODE state (compartment) defined in the model whose past value is required.
`T`	The delay duration. May be a constant, a parameter, a covariate, or any model expression. The value returned is the state at time `t - T`.

Details

Delayed states are interpolated from the solver's dense output, so delay models are solved on a dense path: the default method becomes the dense AutoSwitch composite `"dop853+ros4"`, and dense methods such as `"dop853"` or `"ros4"` also work. The step size is capped to the smallest delay, and methods that cannot record dense history raise an error. The dense-output and delay-history machinery is adapted from the dde package by Rich FitzJohn and Wes Hinsley (Imperial College of Science, Technology and Medicine).

Value

Inside an rxode2 model, the value of `state` at the past time `t - T`. Before the start of integration the constant initial-condition history is used.

Author(s)

Matthew L. Fidler

See Also

`rxSolve()`

Examples

```
# Classic linear delay differential equation y'(t) = -y(t - 1)
dde <- rxode2({
  y(0) <- 1
  d/dt(y) <- -delay(y, 1)
})

s <- rxSolve(dde, et(seq(0, 5, by = 0.1)))

# Delayed (Hutchinson) logistic growth
hutch <- rxode2({
  r <- 0.5
  K <- 10
  tau <- 1
  N(0) <- 2
  d/dt(N) <- r * N * (1 - delay(N, tau) / K)
})

s2 <- rxSolve(hutch, et(seq(0, 40, by = 0.5)))
```

dELU *Derivatives of the Exponential Linear Unit (ELU) Activation Function*

Description

Derivatives of the Exponential Linear Unit (ELU) Activation Function

Usage

```
dELU(x, alpha = 1)

d2ELU(x, alpha = 1)

d2aELU(x, alpha = 1)

dELUa(x, alpha = 1)

d2ELUa(x, alpha = 1)
```

Arguments

x	A numeric vector. All elements must be finite and non-missing.
alpha	A numeric scalar. All elements must be finite and non-missing.

Value

A numeric vector where the derivative(s) of the ELU function has been applied to each element of x.

Author(s)

Matthew L. Fidler

See Also

Other Activation Functions: ELU(), GELU(), PReLU(), ReLU(), SELU(), Swish(), dGELU(), dPReLU(), dReLU(), dSELU(), dSwish(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
dELU(c(-1, 0, 1, 2), 2)
d2ELU(c(-1, 0, 1, 2), 2)
d2aELU(c(-1, 0, 1, 2), 2)
dELUa(c(-1, 0, 1, 2), 2)
d2ELUa(c(-1, 0, 1, 2), 2)


# Can also be used in rxode2:
```

```
r <- rxode2({
  r1=dELU(time, 2)
  r2=d2ELU(time, 2)
  r2a=d2aELU(time, 2)
  ra=dELUa(time, 2)
  r2a=d2ELUa(time, 2)
})

e <- et(c(-1, 0, 1, 2))
rxSolve(r, e)
```

`dfWishart` *This uses simulations to match the rse*

Description

This uses simulations to match the rse

Usage

```
dfWishart(omega, n, rse, upper, totN = 1000, diag = TRUE, seed = 1234)
```

Arguments

`omega`	represents the matrix for simulation
`n`	This represents the number of subjects/samples this comes from (used to calculate rse). When present it assumes the rse= sqrt(2)/sqrt(n)
`rse`	This is the rse that we try to match, if not specified, it is derived from `n`
`upper`	The upper boundary for root finding in terms of degrees of freedom. If not specified, it is n*200
`totN`	This represents the total number of simulated inverse wishart deviates
`diag`	When `TRUE`, represents the rse to match is the diagonals, otherwise it is the total matrix.
`seed`	to make the simulation reproducible, this represents the seed that is used for simulating the inverse Wishart distribution

Value

output from `uniroot()` to find the right estimate

Author(s)

Matthew L. Fidler

Examples

```
dfWishart(lotri::lotri(a+b~c(1, 0.5, 1)), 100)
```

dGELU *Derivatives of GELU*

Description

Derivatives of GELU

Usage

```
dGELU(x)

d2GELU(x)

d3GELU(x)

d4GELU(x)
```

Arguments

x	numeric vector

Value

numeric vector

See Also

Other Activation Functions: ELU(), GELU(), PReLU(), ReLU(), SELU(), Swish(), dELU(), dPReLU(), dReLU(), dSELU(), dSwish(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
dGELU(c(-2, -1, 0, 1, 2))
d2GELU(c(-2, -1, 0, 1, 2))
d3GELU(c(-2, -1, 0, 1, 2))
d4GELU(c(-2, -1, 0, 1, 2))

# you can use rxode2 as well
r <- rxode2({
   r1 <- dGELU(time)
   r2 <- d2GELU(time)
   r3 <- d3GELU(time)
   r4 <- d4GELU(time)
})
et <- et(c(-2, -1, 0, 1, 2))
rxSolve(r, et)
```

dlReLU *Derivative of Leaky ReLU activation function*

Description

Derivative of Leaky ReLU activation function

Usage

```
dlReLU(x)
```

Arguments

x	numeric vector

Value

numeric vector

See Also

Other Activation Functions: ELU(), GELU(), PReLU(), ReLU(), SELU(), Swish(), dELU(), dGELU(), dPReLU(), dReLU(), dSELU(), dSwish(), dsoftplus(), lReLU(), softplus()

Examples

```
dlReLU(c(-1, 0, 1))


# Can use in rxode2 as well

r <- rxode2({r <- dlReLU(time)})
e <- et(c(-1, 0, 1))
rxSolve(r, e)
```

dPReLU *Derivatives Parametric ReLU Activation Function*

Description

Derivatives Parametric ReLU Activation Function

Usage

```
dPReLU(x, alpha = 1)

dPReLUa(x, alpha = 1)

dPReLUa1(x, alpha = 1)
```

Arguments

x	A numeric vector. All elements must be finite and non-missing.
alpha	A numeric scalar. All elements must be finite and non-missing.

Value

A numeric vector where the derivative(s) of the ELU function has been applied to each element of x.

Author(s)

Matthew L. Fidler

See Also

Other Activation Functions: ELU(), GELU(), PReLU(), ReLU(), SELU(), Swish(), dELU(), dGELU(), dReLU(), dSELU(), dSwish(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
dPReLU(c(-1, 0, 1, 2), 2)
dPReLUa(c(-1, 0, 1, 2), 2)
dPReLUa1(c(-1, 0, 1, 2), 2)



# Can also be used in rxode2:
r <- rxode2({
  r1=dPReLU(time, 2)
  r2a=dPReLUa(time, 2)
  ra=dPReLUa1(time, 2)
})

e <- et(c(-1, 0, 1, 2))
rxSolve(r, e)
```

dReLU *Derivative of the Rectified Linear Unit (ReLU) Activation Function*

Description

This function applies the derivative of the Rectified Linear Unit (ReLU) activation function to the input numeric vector.

Usage

```
dReLU(x)
```

Arguments

x	A numeric vector. All elements must be finite and non-missing.

Value

A numeric vector where the derivative of the ReLU function

See Also

Other Activation Functions: ELU(), GELU(), PReLU(), ReLU(), SELU(), Swish(), dELU(), dGELU(), dPReLU(), dSELU(), dSwish(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
dReLU(c(-1, 0, 1, 2))


# Can also be used in rxode2:
x <- rxode2({
   r=dReLU(time)
})

e <- et(c(-1, 0, 1, 2))

rxSolve(x, e)
```

dSELU *Derivative of the Scaled Exponential Linear Unit (SELU) Activation Function*

Description

Derivative of the Scaled Exponential Linear Unit (SELU) Activation Function

Usage

```
dSELU(x)
```

Arguments

x	A numeric vector. All elements must be finite and non-missing.

Value

A numeric vector where the derivative of the SELU function has been applied to each element of x.

Author(s)

Matthew Fidler

See Also

Other Activation Functions: ELU(), GELU(), PReLU(), ReLU(), SELU(), Swish(), dELU(), dGELU(), dPReLU(), dReLU(), dSwish(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
dSELU(c(-1, 0, 1, 2))

# Can also be used in rxode2:
x <- rxode2({
  r=dSELU(time)
})
e <- et(c(-1, 0, 1, 2))
rxSolve(x, e)
```

dsoftplus *Default Softplus Activation Function*

Description

Default Softplus Activation Function

Usage

```
dsoftplus(x)

d2softplus(x)

d3softplus(x)

d4softplus(x)
```

Arguments

x	numeric vector

Value

numeric vector

Author(s)

Matthew L. Fidler

See Also

Other Activation Functions: ELU(), GELU(), PReLU(), ReLU(), SELU(), Swish(), dELU(), dGELU(), dPReLU(), dReLU(), dSELU(), dSwish(), dlReLU(), lReLU(), softplus()

Examples

```
dsoftplus(c(-1, 0, 1, 2))


# You can use rxode2 too:

r <- rxode2({
 s1 <- dsoftplus(time)
})

e <- et(c(-1, 0, 1, 2))

rxSolve(r, e)
```

dSwish *Derivative of the Swish Activation Function*

Description

Derivative of the Swish Activation Function

Usage

```
dSwish(x)
```

Arguments

x	A numeric vector. All elements must be finite and non-missing.

Value

A numeric vector where the derivative of the SELU function has been applied to each element of x.

Author(s)

Matthew Fidler

See Also

Other Activation Functions: ELU(), GELU(), PReLU(), ReLU(), SELU(), Swish(), dELU(), dGELU(), dPReLU(), dReLU(), dSELU(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
dSwish(c(-1, 0, 1, 2))


# Can also be used in rxode2:
x <- rxode2({
  r <- dSwish(time)
})
e <- et(c(-1, 0, 1, 2))
rxSolve(x, e)
```

ELU *Exponential Linear Unit (ELU) Activation Function*

Description

Exponential Linear Unit (ELU) Activation Function

Usage

```
ELU(x, alpha = 1)
```

Arguments

x	A numeric vector. All elements must be finite and non-missing.
alpha	A numeric scalar. All elements must be finite and non-missing.

Value

A numeric vector where the ReLU function has been applied to each element of x.

Author(s)

Matthew Fidler

See Also

Other Activation Functions: GELU(), PReLU(), ReLU(), SELU(), Swish(), dELU(), dGELU(), dPReLU(), dReLU(), dSELU(), dSwish(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
ELU(c(-1, 0, 1, 2), 2)


# Can also be used in rxode2:
x <- rxode2({
   r=ELU(time, 2)
})

e <- et(c(-1, 0, 1, 2))

rxSolve(x, e)
```

erf *Error function*

Description

Error function

Usage

```
erf(x)
```

Arguments

x	vector or real values

Value

erf of x

Author(s)

Matthew L. Fidler

Examples

```
erf(1.0)
```

et *Event Table Function*

Description

Event Table Function

Usage

```
et(x, ..., envir = parent.frame())

## S3 method for class 'rxode2'
et(x, ..., envir = parent.frame())

## S3 method for class '`function`'
et(x, ..., envir = parent.frame())

## S3 method for class 'rxUi'
et(x, ..., envir = parent.frame())
```

```
## S3 method for class 'rxSolve'
et(x, ..., envir = parent.frame())

## S3 method for class 'rxParams'
et(x, ..., envir = parent.frame())

## Default S3 method:
et(
  x,
  ...,
  time = NULL,
  amt = NULL,
  evid = NULL,
  cmt = NULL,
  ii = NULL,
  addl = NULL,
  ss = NULL,
  rate = NULL,
  dur = NULL,
  until = NULL,
  id = NULL,
  amountUnits = NULL,
  timeUnits = NULL,
  addSampling = NULL,
  envir = parent.frame(),
  by = NULL,
  length.out = NULL
)
```

Arguments

x	This is the first argument supplied to the event table. This is named to allow et to be used in a pipe-line with arbitrary objects.
...	Times or event tables. They can also be one of the named arguments below.
envir	the environment in which expr is to be evaluated. May also be NULL, a list, a data frame, a pairlist or an integer as specified to sys.call.
time	Time is the time of the dose or the sampling times. This can also be unspecified and is determined by the object type (list or numeric/integer).
amt	Amount of the dose. If specified, this assumes a dosing record, instead of a sampling record.
evid	Event ID; This can be:

Numeric Value	Description
0	An observation. This can also be specified as evid=obs
1	A dose observation. This can also be specified as evid=dose
2	A non-dose event. This can also be specified as evid=other

3	A reset event. This can also be specified as `evid=reset`.
4	Dose and reset event. This can also be specified as `evid=doseReset` or `evid=resetDose`

Note a reset event resets all the compartment values to zero and turns off all infusions.

`cmt` Compartment name or number. If a number, this is an integer starting at 1. Negative compartments turn off a compartment. If the compartment is a name, the compartment name is changed to the correct state/compartment number before running the simulation. For a compartment named "-cmt" the compartment is turned off.

Can also specify `cmt` as `dosing.to`, `dose.to`, `doseTo`, `dosingTo`, and `state`.

`ii` When specifying a dose, this is the inter-dose interval for `ss`, `addl` and `until` options (described below).

`addl` The number of additional doses at a inter-dose interval after one dose.

`ss` Steady state flag; It can be one of:

Value	Description
0	This dose is not a steady state dose
1	This dose is a steady state dose with the between/inter-dose interval of `ii`
2	Superposition steady state

When `ss=2` the steady state dose that uses the super-position principle to allow more complex steady states, like 10 mg in the morning and 20 mg at night, or dosing at 8 am 12 pm and 8 pm instead of every 12 hours. Since it uses the super positioning principle, it only makes sense when you know the kinetics are linear.

All other values of `SS` are currently invalid.

`rate` When positive, this is the rate of infusion. Otherwise:

Value	Description
0	No infusion is on this record
-1	Modeled rate (in rxode2:`rate(cmt) =`); Can be `et(rate=model)`.
-2	Modeled duration (in rxode2: `dur(cmt) =`); Can be`et(dur=model)` or `et(rate=dur)`.

When a modeled bioavailability is applied to positive rates (`rate` > 0), the duration of infusion is changed. This is because the data specify the rate and amount, the only think that modeled bioavailability can affect is duration.

If instead you want the modeled bioavailability to increase the rate of infusion instead of the duration of infusion, specify the `dur` instead or model the duration with `rate=2`.

dur	Duration of infusion. When amt and dur are specified the rate is calculated from the two data items. When dur is specified instead of rate, the bioavailability changes will increase rate instead of duration.
until	This is the time until the dosing should end. It can be an easier way to figure out how many additional doses are needed over your sampling period.
id	A integer vector of ids to add or remove from the event table. If the event table is identical for each ID, then you may expand it to include all the ids in this vector. All the negative ids in this vector will be removed.
amountUnits	The units for the dosing records (amt)
timeUnits	The units for the time records (time)
addSampling	This is a boolean indicating if a sampling time should be added at the same time as a dosing time. By default this is FALSE.
by	number: increment of the sequence.
length.out	desired length of the sequence. A non-negative number, which for seq and seq.int will be rounded up if fractional.

Value

A new event table

Author(s)

Matthew L Fidler, Wenping Wang

References

Wang W, Hallow K, James D (2015). "A Tutorial on rxode2: Simulating Differential Equation Pharmacometric Models in R." CPT: Pharmacometrics and Systems Pharmacology, 5(1), 3-10. ISSN 2163-8306

See Also

eventTable, add.sampling, add.dosing, et, etRep, etRbind, rxode2

Examples

```
## Not run:

library(rxode2)
library(units)

# Model from rxode2 tutorial
# Using a nlmixr2 style function

mod1 <-function(){
  ini({
    KA <- 2.94E-01
    CL <- 1.86E+01
```

```
    V2 <- 4.02E+01
    Q <- 1.05E+01
    V3 <- 2.97E+02
    Kin <- 1
    Kout <- 1
    EC50 <- 200
  })
 model({
    C2 <- centr/V2
    C3 <- peri/V3
    d/dt(depot) <- -KA*depot
    d/dt(centr) <- KA*depot - CL*C2 - Q*C2 + Q*C3
    d/dt(peri)  <-                   Q*C2 - Q*C3
    d/dt(eff)   <- Kin - Kout*(1-C2/(EC50+C2))*eff
 })
}

## These are making the more complex regimens of the rxode2 tutorial

## bid for 5 days
bid <- et(timeUnits="hr") |>
       et(amt=10000,ii=12,until=set_units(5, "days"))

## qd for 5 days
qd <- et(timeUnits="hr") |>
      et(amt=20000,ii=24,until=set_units(5, "days"))

## bid for 5 days followed by qd for 5 days

et <- seq(bid,qd) |>
      et(seq(0,11*24,length.out=100))

bidQd <- rxSolve(mod1, et)

plot(bidQd, C2)


## Now Infusion for 5 days followed by oral for 5 days

##  note you can dose to a named compartment instead of using the compartment number
infusion <- et(timeUnits = "hr") |>
      et(amt=10000, rate=5000, ii=24, until=set_units(5, "days"), cmt="centr")


qd <- et(timeUnits = "hr") |>
  et(amt=10000, ii=24, until=set_units(5, "days"), cmt="depot")

et <- seq(infusion,qd)

infusionQd <- rxSolve(mod1, et)

plot(infusionQd, C2)
```

```
## 2wk-on, 1wk-off

qd <- et(timeUnits = "hr") |>
      et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- seq(qd, set_units(1,"weeks"), qd) |>
     add.sampling(set_units(seq(0, 5.5,by=0.005),weeks))

wkOnOff <- rxSolve(mod1, et)

plot(wkOnOff, C2)

## You can also repeat the cycle easily with the rep function

qd <-et(timeUnits = "hr") |>
     et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- etRep(qd, times=4, wait=set_units(1,"weeks")) |>
      add.sampling(set_units(seq(0, 12.5,by=0.005),weeks))

repCycle4 <- rxSolve(mod1, et)

plot(repCycle4, C2)


## End(Not run)
```

`etExpand` *Expand additional doses*

Description

Expand additional doses

Usage

```
etExpand(et)
```

Arguments

`et`	Event table to expand additional doses for.

Value

New event table with `addl` doses expanded

Author(s)

Matthew Fidler

Examples

```
ev <- et(amt = 3, ii = 24, until = 240)
print(ev)
etExpand(ev) # expands event table, but doesn't modify it

print(ev)

ev$expand() ## Expands the current event table and saves it in ev
```

`etRbind` *Combining event tables*

Description

Combining event tables

Usage

```
etRbind(
  ...,
  samples = c("use", "clear"),
  waitII = c("smart", "+ii"),
  id = c("merge", "unique")
)

## S3 method for class 'rxEt'
rbind(..., deparse.level = 1)
```

Arguments

`...`	The event tables and optionally time between event tables, called waiting times in this help document.
`samples`	How to handle samples when repeating an event table. The options are: • `"clear"` Clear sampling records before combining the datasets • `"use"` Use the sampling records when combining the datasets
`waitII`	This determines how waiting times between events are handled. The options are: • `"smart"` This "smart" handling of waiting times is the default option. In this case, if the waiting time is above the last observed inter-dose interval in the first combined event table, then the actual time between doses is given by the wait time. If it is smaller than the last observed inter-dose interval, the time between event tables is given by the inter-dose interval + the waiting time between event tables. • `"+ii"` In this case, the wait time is added to the inter-dose interval no matter the length of the wait time or inter-dose interval

`id` This is how rbind will handle ids. There are two different types of options:

- `merge` with `id="merge"`, the ids are merged together, overlapping ids would be merged into a single event table.
- `unique` with `id="unique"`, the ids will be renumbered so that the ids in all the event tables are not overlapping.

`deparse.level` The `deparse.level` of a traditional `rbind` is ignored.

Value

An event table

Author(s)

Matthew L Fidler

Matthew L Fidler, Wenping Wang

References

Wang W, Hallow K, James D (2015). "A Tutorial on rxode2: Simulating Differential Equation Pharmacometric Models in R." CPT: Pharmacometrics and Systems Pharmacology, 5(1), 3-10. ISSN 2163-8306

See Also

`eventTable`, `add.sampling`, `add.dosing`, `et`, `etRep`, `etRbind`, `rxode2`

Examples

```
## Not run:

library(rxode2)
library(units)

# Model from rxode2 tutorial
# Using a nlmixr2 style function

mod1 <-function(){
  ini({
    KA <- 2.94E-01
    CL <- 1.86E+01
    V2 <- 4.02E+01
    Q <- 1.05E+01
    V3 <- 2.97E+02
    Kin <- 1
    Kout <- 1
    EC50 <- 200
  })
 model({
    C2 <- centr/V2
    C3 <- peri/V3
```

```
    d/dt(depot) <- -KA*depot
    d/dt(centr) <- KA*depot - CL*C2 - Q*C2 + Q*C3
    d/dt(peri)  <-                   Q*C2 - Q*C3
    d/dt(eff)   <- Kin - Kout*(1-C2/(EC50+C2))*eff
 })
}

## These are making the more complex regimens of the rxode2 tutorial

## bid for 5 days
bid <- et(timeUnits="hr") |>
       et(amt=10000,ii=12,until=set_units(5, "days"))

## qd for 5 days
qd <- et(timeUnits="hr") |>
      et(amt=20000,ii=24,until=set_units(5, "days"))

## bid for 5 days followed by qd for 5 days

et <- seq(bid,qd) |>
      et(seq(0,11*24,length.out=100))

bidQd <- rxSolve(mod1, et)

plot(bidQd, C2)


## Now Infusion for 5 days followed by oral for 5 days

##  note you can dose to a named compartment instead of using the compartment number
infusion <- et(timeUnits = "hr") |>
      et(amt=10000, rate=5000, ii=24, until=set_units(5, "days"), cmt="centr")


qd <- et(timeUnits = "hr") |>
  et(amt=10000, ii=24, until=set_units(5, "days"), cmt="depot")

et <- seq(infusion,qd)

infusionQd <- rxSolve(mod1, et)

plot(infusionQd, C2)

## 2wk-on, 1wk-off

qd <- et(timeUnits = "hr") |>
      et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- seq(qd, set_units(1,"weeks"), qd) |>
     add.sampling(set_units(seq(0, 5.5,by=0.005),weeks))

wkOnOff <- rxSolve(mod1, et)
```

```
plot(wkOnOff, C2)

## You can also repeat the cycle easily with the rep function

qd <-et(timeUnits = "hr") |>
     et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- etRep(qd, times=4, wait=set_units(1,"weeks")) |>
      add.sampling(set_units(seq(0, 12.5,by=0.005),weeks))

repCycle4 <- rxSolve(mod1, et)

plot(repCycle4, C2)


## End(Not run)
```

---

etRep *Repeat an rxode2 event table*

Description

Repeat an rxode2 event table

Usage

```
etRep(
  x,
  times = 1,
  length.out = NA,
  each = NA,
  n = NULL,
  wait = 0,
  id = integer(0),
  samples = c("clear", "use"),
  waitII = c("smart", "+ii"),
  ii = 24
)

## S3 method for class 'rxEt'
rep(x, ...)
```

Arguments

x	An rxode2 event table
times	Number of times to repeat the event table
length.out	Invalid with rxode2 event tables, will throw an error if used.
each	Invalid with rxode2 event tables, will throw an error if used.

n	The number of times to repeat the event table. Overrides times.
wait	Waiting time between each repeated event table. By default there is no waiting, or wait=0
id	A integer vector of ids to add or remove from the event table. If the event table is identical for each ID, then you may expand it to include all the ids in this vector. All the negative ids in this vector will be removed.
samples	How to handle samples when repeating an event table. The options are: • "clear" Clear sampling records before combining the datasets • "use" Use the sampling records when combining the datasets
waitII	This determines how waiting times between events are handled. The options are: • "smart" This "smart" handling of waiting times is the default option. In this case, if the waiting time is above the last observed inter-dose interval in the first combined event table, then the actual time between doses is given by the wait time. If it is smaller than the last observed inter-dose interval, the time between event tables is given by the inter-dose interval + the waiting time between event tables. • "+ii" In this case, the wait time is added to the inter-dose interval no matter the length of the wait time or inter-dose interval
ii	When specifying a dose, this is the inter-dose interval for ss, addl and until options (described below).
...	Times or event tables. They can also be one of the named arguments below.

Value

An event table

Author(s)

Matthew L Fidler, Wenping Wang

References

Wang W, Hallow K, James D (2015). "A Tutorial on rxode2: Simulating Differential Equation Pharmacometric Models in R." CPT: Pharmacometrics and Systems Pharmacology, 5(1), 3-10. ISSN 2163-8306

See Also

eventTable, add.sampling, add.dosing, et, etRep, etRbind, rxode2

Examples

```
## Not run:

library(rxode2)
library(units)
```

```
# Model from rxode2 tutorial
# Using a nlmixr2 style function

mod1 <-function(){
  ini({
    KA <- 2.94E-01
    CL <- 1.86E+01
    V2 <- 4.02E+01
    Q <- 1.05E+01
    V3 <- 2.97E+02
    Kin <- 1
    Kout <- 1
    EC50 <- 200
  })
 model({
    C2 <- centr/V2
    C3 <- peri/V3
    d/dt(depot) <- -KA*depot
    d/dt(centr) <- KA*depot - CL*C2 - Q*C2 + Q*C3
    d/dt(peri)  <-                   Q*C2 - Q*C3
    d/dt(eff)   <- Kin - Kout*(1-C2/(EC50+C2))*eff
 })
}

## These are making the more complex regimens of the rxode2 tutorial

## bid for 5 days
bid <- et(timeUnits="hr") |>
       et(amt=10000,ii=12,until=set_units(5, "days"))

## qd for 5 days
qd <- et(timeUnits="hr") |>
      et(amt=20000,ii=24,until=set_units(5, "days"))

## bid for 5 days followed by qd for 5 days

et <- seq(bid,qd) |>
      et(seq(0,11*24,length.out=100))

bidQd <- rxSolve(mod1, et)

plot(bidQd, C2)


## Now Infusion for 5 days followed by oral for 5 days

##  note you can dose to a named compartment instead of using the compartment number
infusion <- et(timeUnits = "hr") |>
      et(amt=10000, rate=5000, ii=24, until=set_units(5, "days"), cmt="centr")


qd <- et(timeUnits = "hr") |>
```

```
  et(amt=10000, ii=24, until=set_units(5, "days"), cmt="depot")

et <- seq(infusion,qd)

infusionQd <- rxSolve(mod1, et)

plot(infusionQd, C2)

## 2wk-on, 1wk-off

qd <- et(timeUnits = "hr") |>
      et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- seq(qd, set_units(1,"weeks"), qd) |>
     add.sampling(set_units(seq(0, 5.5,by=0.005),weeks))

wkOnOff <- rxSolve(mod1, et)

plot(wkOnOff, C2)

## You can also repeat the cycle easily with the rep function

qd <-et(timeUnits = "hr") |>
     et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- etRep(qd, times=4, wait=set_units(1,"weeks")) |>
      add.sampling(set_units(seq(0, 12.5,by=0.005),weeks))

repCycle4 <- rxSolve(mod1, et)

plot(repCycle4, C2)


## End(Not run)
```

---

etSeq *Sequence of event tables*

Description

This combines a sequence of event tables.

Usage

```
etSeq(..., samples = c("clear", "use"), waitII = c("smart", "+ii"), ii = 24)

## S3 method for class 'rxEt'
seq(...)
```

Arguments

`...`	The event tables and optionally time between event tables, called waiting times in this help document.
`samples`	How to handle samples when repeating an event table. The options are: • `"clear"` Clear sampling records before combining the datasets • `"use"` Use the sampling records when combining the datasets
`waitII`	This determines how waiting times between events are handled. The options are: • `"smart"` This "smart" handling of waiting times is the default option. In this case, if the waiting time is above the last observed inter-dose interval in the first combined event table, then the actual time between doses is given by the wait time. If it is smaller than the last observed inter-dose interval, the time between event tables is given by the inter-dose interval + the waiting time between event tables. • `"+ii"` In this case, the wait time is added to the inter-dose interval no matter the length of the wait time or inter-dose interval
`ii`	If there was no inter-dose intervals found in the event table, assume that the interdose interval is given by this `ii` value. By default this is `24`.

Details

This sequences all the event tables in added in the argument list `...`. By default when combining the event tables the offset is at least by the last inter-dose interval in the prior event table (or `ii`). If you separate any of the event tables by a number, the event tables will be separated at least the wait time defined by that number or the last inter-dose interval.

Value

An event table

Author(s)

Matthew L Fidler, Wenping Wang

References

Wang W, Hallow K, James D (2015). "A Tutorial on rxode2: Simulating Differential Equation Pharmacometric Models in R." CPT: Pharmacometrics and Systems Pharmacology, 5(1), 3-10. ISSN 2163-8306

See Also

`eventTable`, `add.sampling`, `add.dosing`, `et`, `etRep`, `etRbind`, `rxode2`

Examples

```
## Not run:

library(rxode2)
library(units)

# Model from rxode2 tutorial
# Using a nlmixr2 style function

mod1 <-function(){
  ini({
    KA <- 2.94E-01
    CL <- 1.86E+01
    V2 <- 4.02E+01
    Q <- 1.05E+01
    V3 <- 2.97E+02
    Kin <- 1
    Kout <- 1
    EC50 <- 200
  })
 model({
    C2 <- centr/V2
    C3 <- peri/V3
    d/dt(depot) <- -KA*depot
    d/dt(centr) <- KA*depot - CL*C2 - Q*C2 + Q*C3
    d/dt(peri)  <-                   Q*C2 - Q*C3
    d/dt(eff)   <- Kin - Kout*(1-C2/(EC50+C2))*eff
 })
}

## These are making the more complex regimens of the rxode2 tutorial

## bid for 5 days
bid <- et(timeUnits="hr") |>
       et(amt=10000,ii=12,until=set_units(5, "days"))

## qd for 5 days
qd <- et(timeUnits="hr") |>
      et(amt=20000,ii=24,until=set_units(5, "days"))

## bid for 5 days followed by qd for 5 days

et <- seq(bid,qd) |>
      et(seq(0,11*24,length.out=100))

bidQd <- rxSolve(mod1, et)

plot(bidQd, C2)


## Now Infusion for 5 days followed by oral for 5 days
```

```
##  note you can dose to a named compartment instead of using the compartment number
infusion <- et(timeUnits = "hr") |>
      et(amt=10000, rate=5000, ii=24, until=set_units(5, "days"), cmt="centr")


qd <- et(timeUnits = "hr") |>
  et(amt=10000, ii=24, until=set_units(5, "days"), cmt="depot")

et <- seq(infusion,qd)

infusionQd <- rxSolve(mod1, et)

plot(infusionQd, C2)

## 2wk-on, 1wk-off

qd <- et(timeUnits = "hr") |>
      et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- seq(qd, set_units(1,"weeks"), qd) |>
     add.sampling(set_units(seq(0, 5.5,by=0.005),weeks))

wkOnOff <- rxSolve(mod1, et)

plot(wkOnOff, C2)

## You can also repeat the cycle easily with the rep function

qd <-et(timeUnits = "hr") |>
     et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- etRep(qd, times=4, wait=set_units(1,"weeks")) |>
      add.sampling(set_units(seq(0, 12.5,by=0.005),weeks))

repCycle4 <- rxSolve(mod1, et)

plot(repCycle4, C2)


## End(Not run)
```

---

eventTable *Create an event table object*

Description

Initializes an object of class 'EventTable' with methods for adding and querying dosing and observation records

Usage

```
eventTable(amount.units = NA, time.units = NA)
```

Arguments

amount.units	string denoting the amount dosing units, e.g., "mg", "ug". Default to NA to denote unspecified units. It could also be a solved rxode2 object. In that case, eventTable(obj) returns the eventTable that was used to solve the rxode2 object.
time.units	string denoting the time units, e.g., "hours", "days". Default to "hours".
	An eventTable is an object that consists of a data.frame storing ordered time-stamped events of an (unspecified) PK/PD dynamic system, units (strings) for dosing and time records, plus a list of functions to add and extract event records.
	Currently, events can be of two types: dosing events that represent inputs to the system and sampling time events that represent observations of the system with 'amount.units' and 'time.units', respectively.

Value

A modified data.frame with the following accessible functions:

- get.EventTable() returns the current event table
- add.dosing() adds dosing records to the event table.
- get.dosing() returns a data.frame of dosing records.
- clear.dosing() clears or deletes all dosing from event table
- 'add.sampling() adds sampling time observation records to the event table.
- get.sampling()returns a data.frame of sampled observation records.
- clear.sampling() removes all sampling from event table.
- get.obs.rec() returns a logical vector indicating whether each event record represents an observation or not.
- get.nobs() returns the number of observation (not dosing) records.
- get.units() returns a two-element character vector with the dosing and time units, respectively
- copy() makes a copy of the current event table. To create a copy of an event table object use qd2 <- qd$copy()
- expand() Expands the event table for multi-subject solving. This is done by qd$expand(400) for a 400 subject data expansion

Author(s)

Matthew Fidler, Melissa Hallow and Wenping Wang

See Also

et()

Examples

```
# create dosing and observation (sampling) events
# QD 50mg dosing, 5 days followed by 25mg 5 days
#
qd <- eventTable(amount.units = "mg", time.units = "days")
#
qd$add.dosing(dose = 50, nbr.doses = 5, dosing.interval = 1, do.sampling = FALSE)
#
# sample the system's drug amounts hourly the first day, then every 12 hours
# for the next 4 days
qd$add.sampling(seq(from = 0, to = 1, by = 1 / 24))
qd$add.sampling(seq(from = 1, to = 5, by = 12 / 24))
#
# print(qd$get.dosing())     # table of dosing records
print(qd$get.nobs()) # number of observation (not dosing) records
#
# BID dosing, 5 days
bid <- eventTable("mg", "days") # only dosing
bid$add.dosing(
  dose = 10000, nbr.doses = 2 * 5,
  dosing.interval = 12, do.sampling = FALSE
)
#
# Use the copy() method to create a copy (clone) of an existing
# event table (simple assignments just create a new reference to
# the same event table object (closure)).
#
bid.ext <- bid$copy() # three-day extension for a 2nd cohort
bid.ext$add.dosing(
  dose = 5000, nbr.doses = 2 * 3,
  start.time = 120, dosing.interval = 12, do.sampling = FALSE
)

# You can also use the Piping operator to create a table

qd2 <- eventTable(amount.units = "mg", time.units = "days") |>
  add.dosing(dose = 50, nbr.doses = 5, dosing.interval = 1, do.sampling = FALSE) |>
  add.sampling(seq(from = 0, to = 1, by = 1 / 24)) |>
  add.sampling(seq(from = 1, to = 5, by = 12 / 24))
# print(qd2$get.dosing())     # table of dosing records
print(qd2$get.nobs()) # number of observation (not dosing) records

# Note that piping with |> will update the original table.

qd3 <- qd2 |> add.sampling(seq(from = 5, to = 10, by = 6 / 24))
print(qd2$get.nobs())
print(qd3$get.nobs())
```

`evid_` *Push a future dose or observation event from within an rxode2 model*

Description

`evid_()` is a model-only function that schedules a current or future dosing or observation event during ODE solving. The model body is run once per distinct record time, at the first record of that time, and the event is inserted immediately after it – so an event pushed at the current time is applied before the solver advances, exactly like the same event written in the data. The final record of the timeline does not push (it starts no integration interval).

This function has no meaning outside an rxode2 model block and will throw an error if called directly from R.

Usage

```
evid_(time, evid, amt, cmt = 1, rate = 0, ii = 0, addl = 0, ss = 0)
```

Arguments

`time`	Numeric. The time at which the event should occur. Must be greater than the current model time `t`; events in the past are silently counted and a warning is issued after solving.
`evid`	Integer event ID. Follows NONMEM/rxode2 conventions: `0` Observation (adds an output row, no dose). `1` Dose (bolus or infusion depending on `rate`). `2` Other type observation (passed through). `3` Reset all compartments. `4` Reset then dose. `5` Replace compartment amount. `6` Multiply compartment amount. `7` Phantom/transit dose. `>= 100` Classic rxode2 internal evid; passed through verbatim.
`amt`	Numeric dose amount (for dose events) or `0` for observations. When `rate > 0` this is interpreted as the total infusion amount and the infusion duration is `amt / rate`.
`cmt`	Integer compartment number (1-based) to which the dose is applied. Default is 1
`rate`	Numeric infusion rate. `0` Bolus dose (default). `> 0` Fixed infusion rate; duration = `amt / rate`. `-1` Rate defined by the model (`rate_<cmt>` variable). `-2` Duration defined by the model (`dur_<cmt>` variable).
`ii`	Numeric inter-dose interval. Used together with `addl` to schedule repeat doses at `time`, `time + ii`, `time + 2*ii`, ..., `time + addl*ii`. Also required when `ss > 0`. Default 0
`addl`	Integer number of *additional* doses beyond the first. The total number of doses pushed is `addl + 1`, spaced `ii` apart. Each dose is pushed as a standalone event (not as a periodic schedule in the event table).

ss — Integer steady-state flag applied to the *first* dose only (addl repetitions always use ss = 0).

- 0 No steady-state (default).
- 1 Steady-state additive: add SS solution to current state.
- 2 Steady-state replace: replace current state with SS solution.

Details

Behavior inside a model:

evid_() is evaluated at every output time point (when the solver is exactly at a scheduled event time). The pushed event is inserted into the individual's event timeline and the solver visits it at the specified future time.

The number of events that may be pushed per individual is limited by the maxExtra argument of rxSolve(). When maxExtra = 0 (the default) there is no limit. Exceeding the limit causes an error.

Past-time pushes (where time <= t) are silently ignored and counted; a warning is issued after solving.

Relationship to NONMEM event columns:

The argument order and names mirror the standard NONMEM dataset columns: TIME, EVID, AMT, CMT, RATE, II, ADDL, SS.

Value

This function is only meaningful inside an rxode2 model; it returns NULL invisibly if called from R directly (after signaling an error).

See Also

rxSolve() for the maxExtra control argument. Other more convenient ways to interact with adaptive observations and events include bolus() infuse(), infuseDur(), reset().

Examples

```
# Push a single bolus of 50 mg to compartment 1 at t + 12
m <- rxode2({
  d/dt(depot)   <- -ka * depot
  d/dt(central) <- ka * depot - cl / vd * central
  cp <- central / vd
  if (t < 24) {
    evid_(t + 12, 1, 50, 1, 0, 0, 0, 0)
  }
})

# Push three boluses (addl = 2) at t+6, t+18, t+30 (ii = 12)
m2 <- rxode2({
  d/dt(depot)   <- -ka * depot
  d/dt(central) <- ka * depot - cl / vd * central
  cp <- central / vd
  if (t < 1) {
```

```
    evid_(t + 6, 1, 50, 1, 0, 12, 2, 0)
  }
})
```

---

gammap *Gammap: normalized lower incomplete gamma function*

Description

This is the gamma_p from the boost library

Usage

```
gammap(a, z)
```

Arguments

a	The numeric 'a' parameter in the normalized lower incomplete gamma
z	The numeric 'z' parameter in the normalized lower incomplete gamma

Details

The gamma p function is given by:

gammap = lowergamma(a, z)/gamma(a)

Value

gammap results

Author(s)

Matthew L. Fidler

Examples

```
gammap(1, 3)
gammap(1:3, 3)
gammap(1, 1:3)
```

gammapDer *gammapDer: derivative of gammap*

Description

This is the gamma_p_derivative from the boost library

Usage

```
gammapDer(a, z)
```

Arguments

a	The numeric 'a' parameter in the upper incomplete gamma
z	The numeric 'z' parameter in the upper incomplete gamma

Value

lowergamma results

Author(s)

Matthew L. Fidler

Examples

```
gammapDer(1:3, 3)

gammapDer(1, 1:3)
```

gammapInv *gammapInv and gammapInva: Inverses of normalized gammap function*

Description

gammapInv and gammapInva: Inverses of normalized gammap function

Usage

```
gammapInv(a, p)

gammapInva(x, p)
```

Arguments

a	The numeric 'a' parameter in the upper incomplete gamma
p	The numeric 'p' parameter in the upper incomplete gamma
x	The numeric 'x' parameter in the upper incomplete gamma

Details

With the equation:

p = gammap(a, x)

The 'gammapInv' function returns a value 'x' that satisfies the equation above

The 'gammapInva' function returns a value 'q' that satisfies the equation above

NOTE: gammapInva is slow

Value

inverse gammap results

Author(s)

Matthew L. Fidler

Examples

```
gammapInv(1:3, 0.5)

gammapInv(1, 1:3 / 3.1)

gammapInv(1:3, 1:3 / 3.1)

gammapInva(1:3, 1:3 / 3.1)
```

---

gammaq *Gammaq: normalized upper incomplete gamma function*

Description

This is the gamma_q from the boost library

Usage

```
gammaq(a, z)
```

Arguments

a	The numeric 'a' parameter in the normalized upper incomplete gamma
z	The numeric 'z' parameter in the normalized upper incomplete gamma

Details

The gamma q function is given by:

gammaq = uppergamma(a, z)/gamma(a)

Value

gammaq results

Author(s)

Matthew L. Fidler

Examples

```
gammaq(1, 3)
gammaq(1:3, 3)
gammaq(1, 1:3)
```

`gammaqInv` *gammaqInv and gammaqInva: Inverses of normalized gammaq function*

Description

gammaqInv and gammaqInva: Inverses of normalized gammaq function

Usage

```
gammaqInv(a, q)

gammaqInva(x, q)
```

Arguments

a	The numeric 'a' parameter in the upper incomplete gamma
q	The numeric 'q' parameter in the upper incomplete gamma
x	The numeric 'x' parameter in the upper incomplete gamma

Details

With the equation:

q = gammaq(a, x)

The 'gammaqInv' function returns a value 'x' that satisfies the equation above

The 'gammaqInva' function returns a value 'a' that satisfies the equation above

NOTE: gammaqInva is slow

Value

inverse gammaq results

Author(s)

Matthew L. Fidler

Examples

```
gammaqInv(1:3, 0.5)

gammaqInv(1, 1:3 / 3)

gammaqInv(1:3, 1:3 / 3.1)

gammaqInva(1:3, 1:3 / 3.1)
```

GELU *GELU activation function*

Description

GELU activation function

Usage

```
GELU(x)
```

Arguments

x	numeric vector

Value

numeric vector

See Also

Other Activation Functions: ELU(), PReLU(), ReLU(), SELU(), Swish(), dELU(), dGELU(), dPReLU(), dReLU(), dSELU(), dSwish(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
GELU(c(-2, -1, 0, 1, 2))


# you can use rxode2 as well
r <- rxode2({
  r = GELU(time)
})
et <- et(c(-2, -1, 0, 1, 2))
rxSolve(r, et)
```

---

genShinyApp.template *Generate an example (template) of a dosing regimen shiny app*

Description

Create a complete shiny application for exploring dosing regimens given a (hardcoded) PK/PD model.

Usage

```
genShinyApp.template(
  appDir = "shinyExample",
  verbose = TRUE,
 ODE.config = list(ode = "model", params = c(KA = 0.294), inits = c(eff = 1), method =
    "lsoda", atol = 1e-08, rtol = 1e-06)
)

write.template.server(appDir)

write.template.ui(appDir, statevars)
```

Arguments

appDir	a string with a directory where to store the shiny app, by default is "shinyExample". The directory appDir will be created if it does not exist.
verbose	logical specifying whether to write messages as the shiny app is generated. Defaults to TRUE.
ODE.config	model name compiled and list of parameters sent to rxSolve().
statevars	List of statevars passed to to the write.template.ui() function. This usually isn't called directly. A PK/PD model is defined using rxode2(), and a set of parameters and initial values are defined. Then the appropriate R scripts for the shiny's user interface ui.R and the server logic server.R are created in the directory appDir. The function evaluates the following PK/PD model by default:

```
C2 = centr/V2;
C3 = peri/V3;
d/dt(depot) =-KA*depot;
d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
d/dt(peri)  =                   Q*C2 - Q*C3;
d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
```

This can be changed by the `ODE.config` parameter.

To launch the shiny app, simply issue the `runApp(appDir)` R command.

Value

None, these functions are used for their side effects.

Note

These functions create a simple, but working example of a dosing regimen simulation web application. Users may want to modify the code to experiment creating shiny applications for their specific `rxode2` models.

See Also

`rxode2()`,`eventTable()`, and the package **shiny** (`https://shiny.posit.co`).

Examples

```
# create in a temp dir so nothing is left in the working directory
.appDir <- tempfile("myapp")
on.exit(unlink(.appDir, recursive = TRUE, force = TRUE))
# create the shiny app example (template)
genShinyApp.template(appDir = .appDir)
# run the shiny app
if (requireNamespace("shiny", quietly=TRUE)) {
  library(shiny)
  # runApp(.appDir) # Won't launch in environments without browsers
}
```

---

getRxThreads *Get/Set the number of threads that rxode2 uses*

Description

Get/Set the number of threads that rxode2 uses

Usage

```
getRxThreads(verbose = FALSE)

setRxThreads(threads = NULL, percent = NULL, throttle = NULL)

rxCores(verbose = FALSE)
```

Arguments

verbose	Display the value of relevant OpenMP settings
threads	NULL (default) rereads environment variables specified by OpenMP (OMP_THREAD_LIMIT). If they are not specified, this determines based the logical cores available. 0 means to use all logical CPU threads available. Otherwise a number >= 1
percent	If provided it should be a number between 2 and 100; the percentage of logical CPUs to use. By default on startup, 50 percent.
throttle	2 (default) means that, roughly speaking, a single thread will be used when number subjects solved for is <=2, 2 threads when the number of all points is <=4, etc. The throttle is to speed up small data tasks (especially when repeated many times) by not incurring the overhead of managing multiple threads. The throttle will also suppress sorting which ID will be solved first when there are (nsubject solved)*throttle <= nthreads. In rxode2 this sorting occurs to minimize the time for waiting for another thread to finish. If the last item solved is has a long solving time, all the other solving have to wait for that last costly solving to occur. If the items which are likely to take more time are solved first, this wait is less likely to have an impact on the overall solving time. rxode2 itself solves the ids in data order; the sort is reached through a C interface that resorts them by the total time each id has spent solving so far (largest first). This allows packages like nlmixr2 to sort by solving time between iterations of a fit. Overall the the number of threads is throttled (restricted) for small tasks and sorting for ids are suppressed.

Value

number of threads that rxode2 uses

---

indLin *Transition ODEs written in d/dt() format to matrix exponential / inductive linearization format*

Description

Transition ODEs written in d/dt() format to matrix exponential / inductive linearization format

Usage

```
indLin(model, doConst = FALSE, calcSens = NULL)

rxOdeToIndLin(model, doConst = FALSE, calcSens = NULL)

rxToIndLin(model, doConst = FALSE, calcSens = NULL)
```

Arguments

model	rxode2 model, text, or function
doConst	Replace constants with values; By default this is FALSE.
calcSens	A character vector of parameter names for which sensitivities should be calculated.

Value

A character string representing the matrix exponential model code

Author(s)

Matthew L. Fidler

---

infuse *Administer a infusion (with rate being fixed) inside a rxode2 model*

Description

Administer a infusion (with rate being fixed) inside a rxode2 model

Usage

```
infuse(amt, rate, cmt = 1, ii = 0, addl = 0, ss = 0)
```

Arguments

amt	Numeric dose amount (for dose events) or 0 for observations. When rate > 0 this is interpreted as the total infusion amount and the infusion duration is amt / rate.
rate	Numeric infusion rate. 0 Bolus dose (default). > 0 Fixed infusion rate; duration = amt / rate. -1 Rate defined by the model (rate_<cmt> variable). -2 Duration defined by the model (dur_<cmt> variable).
cmt	Integer compartment number (1-based) to which the dose is applied. Default is 1

`ii` Numeric inter-dose interval. Used together with `addl` to schedule repeat doses at `time`, `time + ii`, `time + 2*ii`, ..., `time + addl*ii`. Also required when `ss > 0`. Default 0

`addl` Integer number of *additional* doses beyond the first. The total number of doses pushed is `addl + 1`, spaced `ii` apart. Each dose is pushed as a standalone event (not as a periodic schedule in the event table).

`ss` Integer steady-state flag applied to the *first* dose only (`addl` repetitions always use `ss = 0`).

- `0` No steady-state (default).
- `1` Steady-state additive: add SS solution to current state.
- `2` Steady-state replace: replace current state with SS solution.

Details

Behavior inside a model:

`infuse()` is evaluated at every output time point (when the solver is exactly at a scheduled event time). The pushed event is inserted into the individual's event timeline and the solver visits it now or at the specified future time.

The number of events that may be pushed per individual is limited by the `maxExtra` argument of `rxSolve()`. When `maxExtra = 0` (the default) there is no limit. Exceeding the limit causes an error.

Past-time pushes (where `time < t`) are silently ignored and counted; a warning is issued after solving.

Value

This function is only meaningful inside an rxode2 model; it returns `NULL` invisibly if called from R directly (after signaling an error).

Author(s)

Matthew L. Fidler

---

infuseDur *Administer a infusion (with duration being fixed) inside a rxode2 model*

Description

Administer a infusion (with duration being fixed) inside a rxode2 model

Usage

```
infuseDur(amt, dur, cmt = 1, ii = 0, addl = 0, ss = 0)
```

Arguments

amt	Numeric dose amount (for dose events) or 0 for observations. When rate > 0 this is interpreted as the total infusion amount and the infusion duration is amt / rate.
dur	Numeric infusion duration.
cmt	Integer compartment number (1-based) to which the dose is applied. Default is 1
ii	Numeric inter-dose interval. Used together with addl to schedule repeat doses at time, time + ii, time + 2*ii, ..., time + addl*ii. Also required when ss > 0. Default 0
addl	Integer number of *additional* doses beyond the first. The total number of doses pushed is addl + 1, spaced ii apart. Each dose is pushed as a standalone event (not as a periodic schedule in the event table).
ss	Integer steady-state flag applied to the *first* dose only (addl repetitions always use ss = 0). 0 No steady-state (default). 1 Steady-state additive: add SS solution to current state. 2 Steady-state replace: replace current state with SS solution.

Details

Behavior inside a model:

infuseDur() is evaluated at every output time point (when the solver is exactly at a scheduled event time). The pushed event is inserted into the individual's event timeline and the solver visits it now or at the specified future time.

The number of events that may be pushed per individual is limited by the maxExtra argument of rxSolve(). When maxExtra = 0 (the default) there is no limit. Exceeding the limit causes an error.

Past-time pushes (where time < t) are silently ignored and counted; a warning is issued after solving.

Value

This function is only meaningful inside an rxode2 model; it returns NULL invisibly if called from R directly (after signaling an error).

Author(s)

Matthew L. Fidler

ini.rxUi *Ini block for rxode2/nlmixr models*

Description

The ini block controls initial conditions for 'theta' (fixed effects), 'omega' (random effects), and 'sigma' (residual error) elements of the model.

Usage

```
## S3 method for class 'rxUi'
ini(x, ..., envir = parent.frame(), append = NULL)

## Default S3 method:
ini(x, ..., envir = parent.frame(), append = NULL)

ini(x, ..., envir = parent.frame(), append = NULL)
```

Arguments

x	expression
...	Other expressions for ini() function
envir	the environment in which unevaluated model expressions is to be evaluated. May also be NULL, a list, a data frame, a pairlist or an integer as specified to sys.call.
append	Reorder theta parameters. NULL means no change to parameter order. A parameter name (as a character string) means to put the new parameter after the named parameter. A number less than or equal to zero means to put the parameter at the beginning of the list. A number greater than the last parameter number means to put the parameter at the end of the list.

Details

The ini() function is used in two different ways. The main way that it is used is to set the initial conditions and associated attributes (described below) in a model. The other way that it is used is for updating the initial conditions in a model, often using the pipe operator.

'theta' and 'sigma' can be set using either <- or = such as tvCL <- 1 or equivalently tvCL = 1. 'omega' can be set with a ~ such as etaCL ~ 0.1.

Parameters can be named or unnamed (though named parameters are preferred). A named parameter is set using the name on the left of the assignment while unnamed parameters are set without an assignment operator. tvCL <- 1 would set a named parameter of tvCL to 1. Unnamed parameters are set using just the value, such as 1.

For some estimation methods, lower and upper bounds can be set for 'theta' and 'sigma' values. To set a lower and/or upper bound, use a vector of values. The vector is c(lower, estimate, upper). The vector may be given with just the estimate (estimate), the lower bound and estimate

(c(lower, estimate)), or all three (c(lower, estimate,upper)). To set an estimate and upper bound without a lower bound, set the lower bound to -Inf, c(-Inf, estimate, upper). When an estimation method does not support bounds, the bounds will be ignored with a warning.

'omega' values can be set as a single value or as the values of a lower-triangular matrix. The values may be set as either a variance-covariance matrix (the default) or as a correlation matrix for the off-diagonals with the standard deviations on the diagonals. Names may be set on the left side of the ~. To set a variance-covariance matrix with variance values of 2 and 3 and a covariance of -2.5 use ~c(2, 2.5, 3). To set the same matrix with names of iivKa and iivCL, use iivKa + iivCL~c(2, 2.5, 3). To set a correlation matrix with standard deviations on the diagonal, use cor() like iivKa + iivCL~cor(2, -0.5, 3). As of rxode2 3.0 you can also use iivKa ~ 2, iivCL ~ c(2.5, 3) for covariance matrices as well.

Values may be fixed (and therefore not estimated) using either the name fixed at the end of the assignment or by calling fixed() as a function for the value to fix. For 'theta' and 'sigma', either the estimate or the full definition (including lower and upper bounds) may be included in the fixed setting. For example, the following are all effectively equivalent to set a 'theta' or 'sigma' to a fixed value (because the lower and upper bounds are ignored for a fixed value): tvCL <- fixed(1), tvCL <-fixed(0, 1), tvCL <- fixed(0, 1, 2), tvCL <- c(0, fixed(1),2), or tvCL <- c(0, 1, fixed). For 'omega' assignment, the full block or none of the block must be set as fixed. Examples of setting an 'omega' value as fixed are: iivKa~fixed(1), iivKa + iivCL~fixed(1, 2, 3), or iivKa + iivCL~c(1, 2, 3, fixed). Anywhere that fixed is used, FIX, FIXED, or fix may be used equivalently.

For any value, standard mathematical operators or functions may be used to define the value. For example, log(2) and 24*30 may be used to define a value anywhere that a number can be used (e.g. lower bound, estimate, upper bound, variance, etc.).

Values may be labeled using the label() function after the assignment. Labels are are used to make reporting easier by giving a human-readable description of the parameter, but the labels do not have any effect on estimation. The typical way to set a label so that the parameter tvCL has a label of "Typical Value of Clearance (L/hr)" is tvCL <- 1; label("Typical Value of Clearance (L/hr)").

Off diagonal values of 'omega' can be set to zero using the diag() to remove all off-diagonals can be removed with ini(diag()). To remove covariances of 'omega' item with iivKa, you can use |> ini(diag(iivKa)). Or to remove covariances that contain either iivKa or iivCl you can use |> ini(diag(iivKa, iivCl)). For finer control you can remove the covariance between two items (like iivKa and iivCl) by '|> ini(-cov(iivKa, iivCl))

rxode2/nlmixr2 will attempt to determine some back-transformations for the user. For example, CL <- exp(tvCL) will detect that tvCL must be back-transformed by exp() for easier interpretation. When you want to control the back-transformation, you can specify the back-transformation using backTransform() after the assignment. For example, to set the back-transformation to exp(), you can use tvCL <- 1; backTransform(exp()).

Value

ini block

Author(s)

Matthew Fidler

See Also

Other Initial conditions: zeroRe()

Examples

```
# Set the ini() block in a model
one.compartment <- function() {
  ini({
    tka <- log(1.57); label("Ka")
    tcl <- log(2.72); label("Cl")
    tv <- log(31.5); label("V")
    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)
    d/dt(depot) = -ka * depot
    d/dt(center) = ka * depot - cl / v * center
    cp = center / v
    cp ~ add(add.sd)
  })
}

# Use piping to update initial conditions
one.compartment |> ini(tka <- log(2))
one.compartment |> ini(tka <- label("Absorption rate, Ka (1/hr)"))
# Move the tka parameter to be just below the tv parameter (affects parameter
# summary table, only)
one.compartment |> ini(tka <- label("Absorption rate, Ka (1/hr)"), append = "tv")
# When programming with rxode2/nlmixr2, it may be easier to pass strings in
# to modify the ini
one.compartment |> ini("tka <- log(2)")
```

---

ini<- *Assign the ini block in the rxode2 related object*

Description

Assign the ini block in the rxode2 related object

Usage

```
ini(x, envir = environment(x)) <- value
```

Arguments

x	rxode2 related object
envir	Environment where assignment occurs
value	Value of the object

Value

rxode2 related object

Author(s)

Matthew L. Fidler

---

is.rxEt *Check if object is an rxEt event table*

Description

Check if object is an rxEt event table

Usage

```
is.rxEt(x)
```

Arguments

x	object to test

Value

logical

---

is.rxStackData *Return if the object can be stacked*

Description

Return if the object can be stacked

Usage

```
is.rxStackData(object)
```

Arguments

object	object to test if it can be stacked

Value

boolean to tell if an object can be stacked using rxode2

Author(s)

Matthew L. Fidler

Examples

```
is.rxStackData(NULL)
```

`linCmtMicro` *Get the micro-constant parameterization of a linCmt() model*

Description

This extracts the solved linear compartment model(s) from a model and returns them as micro-constants (`k`, `k12`, `k21`, `k13`, `k31`), the central volume `v` and the absorption rate `ka`, each as an R expression in terms of the model variables. This is useful for translating a `linCmt()` model to software that has its own closed-form linear compartment solutions (like NONMEM's `ADVAN1-4`, `ADVAN11-12` or Monolix's `pkmodel()`).

Usage

```
linCmtMicro(ui)
```

Arguments

`ui`	rxUi-like model object

Value

A list with one element per linear compartment system (endpoints that share the same `linCmt()`, like conditional `linCmt() ~ ... | cond` endpoints, share one element). Each element is a list with `ncmt` (number of compartments), `oral0` (1 when there is a depot compartment, 0 otherwise), and the expressions `ka`, `v`, `k`, `k12`, `k21`, `k13` and `k31`. `ka` is 0 for a model without a depot, and the transfer rates a model does not have (like `k13` and `k31` for 2 compartments) are `NULL`. A model without `linCmt()` returns an empty list.

Author(s)

Matthew L. Fidler

Examples

```
oneCmt <- function() {
  ini({
    tka <- 0.45
    tcl <- log(2.7)
    tv <- 3.45
    add.sd <- 0.7
  })
  model({
    ka <- exp(tka)
    cl <- exp(tcl)
    v <- exp(tv)
    cp <- linCmt()
    cp ~ add(add.sd)
  })
}

linCmtMicro(oneCmt)
```

---

linMod *Linear model to replace in rxode2 ui model*

Description

Linear model to replace in rxode2 ui model

Usage

```
linMod(
  variable,
  power,
  dv = "dv",
  intercept = TRUE,
  type = c("replace", "before", "after"),
  num = NULL,
  iniDf = NULL,
  data = FALSE,
  mv = FALSE
)

linMod0(..., intercept = FALSE)

linModB(..., type = "before")

linModB0(..., intercept = FALSE, type = "before")

linModA(..., type = "after")
```

```
linModA0(..., intercept = FALSE, type = "after")

linModD(..., intercept = TRUE, data = TRUE)

linModD0(..., intercept = FALSE, data = TRUE)

linModM(..., intercept = TRUE, mv = TRUE)

linModM0(..., intercept = FALSE, mv = TRUE)
```

Arguments

variable	The variable that the rxode2 will be made on.
power	The power of the polynomial that will be generated.
dv	the dependent variable to use to generate the initial estimates from the data. If NULL query using rxUdfUiData().
intercept	Boolean that tells if the intercept be generated.
type	the type of linear model replacement to be used.
num	the number the particular model is being generated. If unspecified, query using rxUdfUiNum().
iniDf	the initialization data.frame, if NULL query using rxUdfUiIniDf()
data	logical that tells if the initial estimates of the linear model should be estimated from the data.
mv	logical that tell if the model variables need to be used to generate model variables.
...	arguments that are passed to linMod() for the other abbreviations of linMod()

Value

a list for use in when generating the rxode2 ui model see rxUdfUi() for details.

Functions

- linMod0(): linear model without intercept
- linModB(): linear model before where it occurs
- linModB0(): linear model before where the user function occurs
- linModA(): linear model after where the user function occurs
- linModA0(): liner model without an intercept placed after where the user function occurs
- linModD(): linear model where initial estimates are generated from the data
- linModD0(): linear model where initial estimates are generated from the data (no intercept)
- linModM(): linear model where the model variables are used to generate the model variables
- linModM0(): linear model where the model variables are used to generate the model variables (no intercept)

Author(s)

Matthew L. Fidler

See Also

Other User functions: `rxUdfUiControl()`, `rxUdfUiData()`, `rxUdfUiEst()`, `rxUdfUiIniLhs()`, `rxUdfUiMv()`, `rxUdfUiNum()`, `rxUdfUiParsing()`

Examples

```
linMod(x, 3)
```

---

linToOde *Convert linCmt rxUi models to ODE rxUi models*

Description

Convert linCmt rxUi models to ODE rxUi models

Usage

```
linToOde(ui)
```

Arguments

ui	rxUi-like model object

Value

rxUi model with `linCmt()` translated to explicit ODEs

Author(s)

Matthew Fidler

Examples

```
oneCmt <- function() {
  ini({
    tka <- 0.45
    tcl <- log(2.7)
    tv <- 3.45
    add.sd <- 0.7
  })
  model({
    ka <- exp(tka)
    cl <- exp(tcl)
    v <- exp(tv)
    cp <- linCmt()
```

```
    cp ~ add(add.sd)
  })
}

oneCmtOde <- linToOde(oneCmt)



pkpd <- function() {
  ini({
    ka <- 1
    cl <- 2
    v <- 20
    kin <- 1
    kout <- 1
    ec50 <- 2
  })
  model({
    cp <- linCmt(ka, cl, v)
    eff(0) <- 1
    d/dt(eff) <- kin - kout * (1 - cp/(ec50 + cp)) * eff
  })
}

pkpdOde <- linToOde(pkpd)
```

`llikBeta` *Calculate the log likelihood of the binomial function (and its derivatives)*

Description

Calculate the log likelihood of the binomial function (and its derivatives)

Usage

```
llikBeta(x, shape1, shape2, full = FALSE)
```

Arguments

`x`	Observation
`shape1`, `shape2`	non-negative parameters of the Beta distribution.
`full`	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikBeta()` but you have to use all arguments. You can also get the derivative of `shape1` and `shape2` with `llikBetaDshape1()` and `llikBetaDshape2()`.

Value

data frame with `fx` for the log pdf value of with `dShape1` and `dShape2` that has the derivatives with respect to the parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
x <- seq(1e-4, 1 - 1e-4, length.out = 21)

llikBeta(x, 0.5, 0.5)

llikBeta(x, 1, 3, TRUE)

et <- et(seq(1e-4, 1-1e-4, length.out=21))
et$shape1 <- 0.5
et$shape2 <- 1.5

model <- function() {
  model({
    fx <- llikBeta(time, shape1, shape2)
    dShape1 <- llikBetaDshape1(time, shape1, shape2)
    dShape2 <- llikBetaDshape2(time, shape1, shape2)
  })
}

rxSolve(model, et)
```

---

llikBinom *Calculate the log likelihood of the binomial function (and its derivatives)*

Description

Calculate the log likelihood of the binomial function (and its derivatives)

Usage

```
llikBinom(x, size, prob, full = FALSE)
```

Arguments

x	Number of successes
size	Size of trial
prob	probability of success
full	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an rxode2() model, you can use llikBinom() but you have to use all arguments. You can also get the derivative of prob with llikBinomDprob()

Value

data frame with fx for the pdf value of with dProb that has the derivatives with respect to the parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikBinom(46:54, 100, 0.5)

llikBinom(46:54, 100, 0.5, TRUE)

# In rxode2 you can use:

et <- et(46:54)
et$size <- 100
et$prob <-0.5

model <- function() {
  model({
    fx <- llikBinom(time, size, prob)
    dProb <- llikBinomDprob(time, size, prob)
  })
}

rxSolve(model, et)
```

`llikCauchy` *log likelihood of Cauchy distribution and it's derivatives (from stan)*

Description

log likelihood of Cauchy distribution and it's derivatives (from stan)

Usage

```
llikCauchy(x, location = 0, scale = 1, full = FALSE)
```

Arguments

`x`	Observation
`location`, `scale`	location and scale parameters.
`full`	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikCauchy()` but you have to use all arguments. You can also get the derivative of `location` and `scale` with `llikCauchyDlocation()` and `llikCauchyDscale()`.

Value

data frame with `fx` for the log pdf value of with `dLocation` and `dScale` that has the derivatives with respect to the parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
x <- seq(-3, 3, length.out = 21)

llikCauchy(x, 0, 1)

llikCauchy(x, 3, 1, full=TRUE)

et <- et(-3, 3, length.out=10)
et$location <- 0
et$scale <- 1

model <- function() {
  model({
    fx <- llikCauchy(time, location, scale)
    dLocation <- llikCauchyDlocation(time, location, scale)
    dScale <- llikCauchyDscale(time, location, scale)
```

```
  })
}

rxSolve(model, et)
```

`llikChisq` *log likelihood and derivatives for chi-squared distribution*

Description

log likelihood and derivatives for chi-squared distribution

Usage

```
llikChisq(x, df, full = FALSE)
```

Arguments

`x`	variable that is distributed by chi-squared distribution
`df`	degrees of freedom (non-negative, but can be non-integer).
`full`	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikChisq()` but you have to use the x and df arguments. You can also get the derivative of `df` with `llikChisqDdf()`.

Value

data frame with `fx` for the log pdf value of with `dDf` that has the derivatives with respect to the `df` parameter the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikChisq(1, df = 1:3, full=TRUE)

llikChisq(1, df = 6:9)

et <- et(1:3)
et$x <- 1

model <- function() {
  model({
```

```
   fx <- llikChisq(x, time)
   dDf <- llikChisqDdf(x, time)
  })
}

rxSolve(model, et)
```

`llikExp` *log likelihood and derivatives for exponential distribution*

Description

log likelihood and derivatives for exponential distribution

Usage

```
llikExp(x, rate, full = FALSE)
```

Arguments

`x`	variable that is distributed by exponential distribution
`rate`	vector of rates.
`full`	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikExp()` but you have to use the x and rate arguments. You can also get the derivative of `rate` with `llikExpDrate()`.

Value

data frame with `fx` for the log pdf value of with `dRate` that has the derivatives with respect to the `rate` parameter the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikExp(1, 1:3)

llikExp(1, 1:3, full=TRUE)

# You can use rxode2 for these too:

et <- et(1:3)
et$x <- 1
```

```
model <- function() {
  model({
    fx <- llikExp(x, time)
    dRate <- llikExpDrate(x, time)
  })
}

rxSolve(model, et)
```

`llikF` *log likelihood and derivatives for F distribution*

Description

log likelihood and derivatives for F distribution

Usage

```
llikF(x, df1, df2, full = FALSE)
```

Arguments

`x`	variable that is distributed by f distribution
`df1`, `df2`	degrees of freedom. `Inf` is allowed.
`full`	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikF()` but you have to use the x and rate arguments. You can also get the derivative of `df1` and `df2` with `llikFDdf1()` and `llikFDdf2()`.

Value

data frame with `fx` for the log pdf value of with `dDf1` and `dDf2` that has the derivatives with respect to the `df1`/`df2` parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
x <- seq(0.001, 5, length.out = 100)

llikF(x^2, 1, 5)

model <- function(){
  model({
    fx <- llikF(time, df1, df2)
    dMean <- llikFDdf1(time, df1, df2)
    dSd <- llikFDdf2(time, df1, df2)
  })
}

et <- et(x)
et$df1 <- 1
et$df2 <- 5

rxSolve(model, et)
```

---

llikGamma *log likelihood and derivatives for Gamma distribution*

Description

log likelihood and derivatives for Gamma distribution

Usage

```
llikGamma(x, shape, rate, full = FALSE)
```

Arguments

x	variable that is distributed by gamma distribution
shape	this is the distribution's shape parameter. Must be positive.
rate	this is the distribution's rate parameters. Must be positive.
full	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikGamma()` but you have to use the x and rate arguments. You can also get the derivative of `shape` or `rate` with `llikGammaDshape()` and `llikGammaDrate()`.

Value

data frame with `fx` for the log pdf value of with `dProb` that has the derivatives with respect to the `prob` parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikGamma(1, 1, 10)

# You can use this in `rxode2` too:

et  <- et(seq(0.001, 1, length.out=10))
et$shape <- 1
et$rate <- 10

model <- function() {
  model({
    fx <- llikGamma(time, shape, rate)
    dShape<- llikGammaDshape(time, shape, rate)
    dRate <- llikGammaDrate(time, shape, rate)
  })
}

rxSolve(model, et)
```

`llikGeom` *log likelihood and derivatives for Geom distribution*

Description

log likelihood and derivatives for Geom distribution

Usage

```
llikGeom(x, prob, full = FALSE)
```

Arguments

`x`	variable distributed by a geom distribution
`prob`	probability of success in each trial. `0 < prob <= 1`.
`full`	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikGeom()` but you have to use the x and rate arguments. You can also get the derivative of `prob` with `llikGeomDprob()`.

Value

data frame with `fx` for the log pdf value of with `dProb` that has the derivatives with respect to the `prob` parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikGeom(1:10, 0.2)

et  <- et(1:10)
et$prob <- 0.2

model <- function() {
  model({
    fx <- llikGeom(time, prob)
    dProb <- llikGeomDprob(time, prob)
  })
}

rxSolve(model, et)
```

---

llikNbinom *Calculate the log likelihood of the negative binomial function (and its derivatives)*

Description

Calculate the log likelihood of the negative binomial function (and its derivatives)

Usage

```
llikNbinom(x, size, prob, full = FALSE)
```

Arguments

`x`	Number of successes
`size`	Size of trial
`prob`	probability of success
`full`	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an rxode2() model, you can use llikNbinom() but you have to use all arguments. You can also get the derivative of prob with llikNbinomDprob()

Value

data frame with fx for the pdf value of with dProb that has the derivatives with respect to the parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikNbinom(46:54, 100, 0.5)

llikNbinom(46:54, 100, 0.5, TRUE)

# In rxode2 you can use:

et <- et(46:54)
et$size <- 100
et$prob <-0.5

model <- function() {
  model({
    fx <- llikNbinom(time, size, prob)
    dProb <- llikNbinomDprob(time, size, prob)
  })
}

rxSolve(model, et)
```

---

llikNbinomMu *Calculate the log likelihood of the negative binomial function (and its derivatives)*

Description

Calculate the log likelihood of the negative binomial function (and its derivatives)

Usage

```
llikNbinomMu(x, size, mu, full = FALSE)
```

Arguments

x	Number of successes
size	Size of trial
mu	mu parameter for negative binomial
full	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an rxode2() model, you can use llikNbinomMu() but you have to use all arguments. You can also get the derivative of mu with llikNbinomMuDmu()

Value

data frame with fx for the pdf value of with dProb that has the derivatives with respect to the parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikNbinomMu(46:54, 100, 40)

llikNbinomMu(46:54, 100, 40, TRUE)

et <- et(46:54)
et$size <- 100
et$mu <- 40

model <- function() {
  model({
    fx <- llikNbinomMu(time, size, mu)
    dProb <- llikNbinomMuDmu(time, size, mu)
  })
}

rxSolve(model, et)
```

---

llikNorm *Log likelihood for normal distribution*

Description

Log likelihood for normal distribution

Usage

```
llikNorm(x, mean = 0, sd = 1, full = FALSE)
```

Arguments

x	Observation
mean	Mean for the likelihood
sd	Standard deviation for the likelihood
full	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an rxode2() model, you can use llikNorm() but you have to use all arguments. You can also get the derivatives with llikNormDmean() and llikNormDsd()

Value

data frame with fx for the pdf value of with dMean and dSd that has the derivatives with respect to the parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikNorm(0)

llikNorm(seq(-2,2,length.out=10), full=TRUE)

# With rxode2 you can use:

et <- et(-3, 3, length.out=10)
et$mu <- 0
et$sigma <- 1

model <- function(){
  model({
    fx <- llikNorm(time, mu, sigma)
    dMean <- llikNormDmean(time, mu, sigma)
    dSd <- llikNormDsd(time, mu, sigma)
  })
 }

ret <- rxSolve(model, et)
ret
```

`llikPois` *log-likelihood for the Poisson distribution*

Description

log-likelihood for the Poisson distribution

Usage

```
llikPois(x, lambda, full = FALSE)
```

Arguments

`x`	non negative integers
`lambda`	non-negative means
`full`	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikPois()` but you have to use all arguments. You can also get the derivatives with `llikPoisDlambda()`

Value

data frame with `fx` for the pdf value of with `dLambda` that has the derivatives with respect to the parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikPois(0:7, lambda = 1)

llikPois(0:7, lambda = 4, full=TRUE)

# In rxode2 you can use:

et <- et(0:10)
et$lambda <- 0.5

model <- function() {
  model({
    fx <- llikPois(time, lambda)
    dLambda <- llikPoisDlambda(time, lambda)
  })
}
```

```
rxSolve(model, et)
```

`llikT` *Log likelihood of T and it's derivatives (from stan)*

Description

Log likelihood of T and it's derivatives (from stan)

Usage

```
llikT(x, df, mean = 0, sd = 1, full = FALSE)
```

Arguments

`x`	Observation
`df`	degrees of freedom (> 0, maybe non-integer). `df = Inf` is allowed.
`mean`	Mean for the likelihood
`sd`	Standard deviation for the likelihood
`full`	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikT()` but you have to use all arguments. You can also get the derivative of `df`, `mean` and `sd` with `llikTDdf()`, `llikTDmean()` and `llikTDsd()`.

Value

data frame with `fx` for the log pdf value of with `dDf` `dMean` and `dSd` that has the derivatives with respect to the parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
x <- seq(-3, 3, length.out = 21)

llikT(x, 7, 0, 1)

llikT(x, 15, 0, 1, full=TRUE)

et <- et(-3, 3, length.out=10)
et$nu <- 7
```

```
et$mean <- 0
et$sd <- 1

model <- function() {
  model({
    fx <- llikT(time, nu, mean, sd)
    dDf <- llikTDdf(time, nu, mean, sd)
    dMean <- llikTDmean(time, nu, mean, sd)
    dSd   <- llikTDsd(time, nu, mean, sd)
  })
}

rxSolve(model, et)
```

`llikUnif` *log likelihood and derivatives for Unif distribution*

Description

log likelihood and derivatives for Unif distribution

Usage

```
llikUnif(x, alpha, beta, full = FALSE)
```

Arguments

`x`	variable distributed by a uniform distribution
`alpha`	is the lower limit of the uniform distribution
`beta`	is the upper limit of the distribution
`full`	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikUnif()` but you have to use the x and rate arguments. You can also get the derivative of `alpha` or `beta` with `llikUnifDalpha()` and `llikUnifDbeta()`.

Value

data frame with `fx` for the log pdf value of with `dProb` that has the derivatives with respect to the `prob` parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikUnif(1, -2, 2)

et  <- et(seq(1,1, length.out=4))
et$alpha <- -2
et$beta <- 2

model <- function() {
  model({
    fx <- llikUnif(time, alpha, beta)
    dAlpha<- llikUnifDalpha(time, alpha, beta)
    dBeta <- llikUnifDbeta(time, alpha, beta)
  })
}

rxSolve(model, et)
```

---

llikWeibull *log likelihood and derivatives for Weibull distribution*

Description

log likelihood and derivatives for Weibull distribution

Usage

```
llikWeibull(x, shape, scale, full = FALSE)
```

Arguments

x	variable distributed by a Weibull distribution
shape, scale	shape and scale parameters, the latter defaulting to 1.
full	Add the data frame showing x, mean, sd as well as the fx and derivatives

Details

In an `rxode2()` model, you can use `llikWeibull()` but you have to use the x and rate arguments. You can also get the derivative of `shape` or `scale` with `llikWeibullDshape()` and `llikWeibullDscale()`.

Value

data frame with `fx` for the log pdf value of with `dProb` that has the derivatives with respect to the `prob` parameters at the observation time-point

Author(s)

Matthew L. Fidler

Examples

```
llikWeibull(1, 1, 10)

# rxode2 can use this too:

et  <- et(seq(0.001, 1, length.out=10))
et$shape <- 1
et$scale <- 10

model <- function() {
  model({
    fx <- llikWeibull(time, shape, scale)
    dShape<- llikWeibullDshape(time, shape, scale)
    dScale <- llikWeibullDscale(time, shape, scale)
  })
}

rxSolve(model, et)
```

`logit` *logit and inverse logit (expit) functions*

Description

logit and inverse logit (expit) functions

Usage

```
logit(x, low = 0, high = 1)

expit(alpha, low = 0, high = 1)

logitNormInfo(mean = 0, sd = 1, low = 0, high = 1, abs.tol = 1e-06, ...)

probitNormInfo(mean = 0, sd = 1, low = 0, high = 1, abs.tol = 1e-06, ...)
```

Arguments

`x`	Input value(s) in range [low,high] to translate -Inf to Inf
`low`	Lowest value in the range
`high`	Highest value in the range
`alpha`	Infinite value(s) to translate to range of [low, high]

mean	logit-scale mean
sd	logit-scale standard deviation
abs.tol	absolute accuracy requested.
...	other parameters passed to integrate()

Details

logit is given by:

logit(p) = -log(1/p-1)

where:

p = x-low/high-low

expit is given by:

expit(p, low, high) = (high-low)/(1+exp(-alpha)) + low

The logitNormInfo() gives the mean, variance and coefficient of variability on the untransformed scale.

Value

values from logit and expit

Examples

```
logit(0.25)

expit(-1.09)

logitNormInfo(logit(0.25), sd = 0.1)

logitNormInfo(logit(1, 0, 10), sd = 1, low = 0, high = 10)
```

---

lowergamma *lowergamma: upper incomplete gamma function*

Description

This is the tgamma_lower from the boost library

Usage

```
lowergamma(a, z)
```

Arguments

a	The numeric 'a' parameter in the upper incomplete gamma
z	The numeric 'z' parameter in the upper incomplete gamma

Details

The lowergamma function is given by:

$$lowergamma(a, z) = \int_0^z t^{a-1} \cdot e^{-t} dt$$

Value

lowergamma results

Author(s)

Matthew L. Fidler

Examples

```
lowergamma(1, 3)

lowergamma(1:3, 3)

lowergamma(1, 1:3)
```

`lReLU` *Leaky ReLU activation function*

Description

Leaky ReLU activation function

Usage

```
lReLU(x)
```

Arguments

x	numeric vector

Value

numeric vector

See Also

Other Activation Functions: `ELU()`, `GELU()`, `PReLU()`, `ReLU()`, `SELU()`, `Swish()`, `dELU()`, `dGELU()`, `dPReLU()`, `dReLU()`, `dSELU()`, `dSwish()`, `dlReLU()`, `dsoftplus()`, `softplus()`

Examples

```
lReLU(c(-1, 0, 1))


# Can use in rxode2 as well

r <- rxode2({r <- lReLU(time)})
e <- et(c(-1, 0, 1))
rxSolve(r, e)
```

`meanProbs` *Calculate expected confidence bands or prediction intreval with normal or t sampling distribution*

Description

The generic function `meanProbs` produces expected confidence bands under either the t distribution or the normal sampling distribution. This uses `qnorm()` or `qt()` with the mean and standard deviation.

Usage

```
meanProbs(x, ...)

## Default S3 method:
meanProbs(
  x,
  probs = seq(0, 1, 0.25),
  na.rm = FALSE,
  names = TRUE,
  useT = TRUE,
  onlyProbs = TRUE,
  pred = FALSE,
  n = 0L,
  ...
)
```

Arguments

`x`	numeric vector whose mean and probability based confidence values are wanted, NA and NaN values are not allowed in numeric vectors unless 'na.rm' is 'TRUE'.
`...`	Arguments passed to default method, allows many different methods to be applied.
`probs`	numeric vector of probabilities with values in the interval from 0 to 1 .
`na.rm`	logical; if true, any NA and NaN's are removed from `x` before the quantiles are computed.

names	logical; if true, the result has a names attribute.
useT	logical; if true, use the t-distribution to calculate the confidence-based estimates. If false use the normal distribution to calculate the confidence based estimates.
onlyProbs	logical; if true, only return the probability based confidence interval estimates, otherwise return
pred	logical; if true use the prediction interval instead of the confidence interval
n	integer/integerish; this is the n used to calculate the prediction or confidence interval. When n=0 (default) use the number of non-NA observations.

Details

For a single probability, p, it uses either:

mean + qt(p, df=n)*sd/sqrt(n)

or

mean + qnorm(p)*sd/sqrt(n)

The smallest observation corresponds to a probability of 0 and the largest to a probability of 1 and the mean corresponds to 0.5.

The mean and standard deviation of the sample is calculated based on Welford's method for a single pass.

This is meant to perform in the same way as quantile() so it can be a drop in replacement for code using quantile() but using distributional assumptions.

Value

By default the return has the probabilities as names (if named) with the points where the expected distribution are located given the sampling mean and standard deviation. If onlyProbs=FALSE then it would prepend mean, variance, standard deviation, minimum, maximum and number of non-NA observations.

Author(s)

Matthew L. Fidler

Examples

```
quantile(x<- rnorm(1001))
meanProbs(x)

# Can get some extra statistics if you request onlyProbs=FALSE
meanProbs(x, onlyProbs=FALSE)

x[2] <- NA_real_

meanProbs(x, onlyProbs=FALSE)

quantile(x<- rnorm(42))
```

```
meanProbs(x)

meanProbs(x, useT=FALSE)
```

`mexpit` *mexpit – Convert log-scale numbers to probabilities*

Description

This function converts log-scale numbers to probabilities.

Usage

```
mexpit(...)

dmexpit(...)
```

Arguments

`...`	numeric log-scale numbers to convert to probabilities.

Details

The probabilities are calculated using the following equation:

$$p_i = \frac{e^{x_i}}{1 + \sum_{j=1}^{N-1} e^{x_j}}$$

This ensures one remaining probability will add to one, that is

$$p_N = \frac{1}{1 + \sum_{j=1}^{N-1} e^{x_j}}$$

For the function `dmexpit()`, the element-wise derivatives are calculated; that is, it returns the diagonal of the Jacobian matrix, dp_i/dx_i, not the full Jacobian with off-diagonal terms.

Value

Probabilities that add up to a number less than 1.

Author(s)

Matthew L. Fidler

Examples

```
m <- mlogit(0.1, 0.2, 0.3)
mexpit(m)

# derivatives
dmexpit(m)

p <- mexpit(-3, 0.5, 3)
mlogit(p)
```

`mix` *Specify a mixture model of variables*

Description

Specify a mixture model of variables

Usage

```
mix(...)
```

Arguments

`...`	Arguments to the mixture model. The first call to the mixture model function takes an odd number of arguments (at least 3). For example the model we could have: cl = mix(cl1, p1, cl2, p2, cl3) Here there is a mixture of three clearance variables, `cl1`, `cl2`, and `cl3`, at a probability of `p1`, `p2`, and the last one is assumed to be `1 - p1 - p2`. For simulations this is selected randomly. For estimations this is selected by the data for each individual. After the first call when the number of populations has been established, you can also call the mixture model with the number of populations, for example: v = mix(v1, v2, v3) The ui function will translate this to the following model: v = mix(v1, p1, v2, p2, v3) This is because the first call to `mix()` sets the probabilities. In rxode2/nlmixr2 these probabilities should be conserved between the models. These probabilities also have to be defined in the ini block directly.

Value

The mixture model replacement for the underlying rxode2 model.

Author(s)

Matthew L. Fidler

Examples

```
# This is an example of a mixture model
# Where there are 2 different clearance populations

one.cmt <- function() {
  ini({
    tka <- 0.45 # Log Ka
    tcl1 <- log(c(0, 2.7, 100)) # Log Cl
    tcl2 <- log(c(0, 0.1, 120)) # Log Cl
    tv <- 3.45; label("log V")
    p1 <- 0.3
    eta.ka ~ 0.6
    eta.cl ~ 0.3
    eta.v ~ 0.1
    add.sd <- 0.7
  })
  model({
    ka <- exp(tka + eta.ka)
    # This is the example mixture model
    cl <- mix(exp(tcl1 + eta.cl), p1, exp(tcl2 + eta.cl))
    v <- exp(tv + eta.v)
    me <- mixest # This is the assigned mixture estimate
    mn <- mixnum # This is the number of mixture estimate in the model
    # This is the uniform mixture estimate used in simulation to
    # determine the population
    mu <- mixunif
    linCmt() ~ add(add.sd)
  })
}



s <- rxSolve(one.cmt, et(amt=320, ii=12, addl=2, cmt=1) |>
                      et(seq(0, 72)) |>
                      et(id=1:20))

plot(s, ipredSim)
```

`mlogit` *mlogit – Convert multiple probabilities to log-scale numbers*

Description

These multiple probabilities need to add up to be less than 1.

Usage

```
mlogit(
  ...,
  maxiter = 10000,
  rtol = 1e-10,
  atol = 1e-12,
  ctol = 1e-12,
  returnRoot = FALSE
)
```

Arguments

`...`	numeric probabilities to convert to log-scale numbers. These probabilities must add to a number less than 1 and are used in the mix() estimation algorithm.
`maxiter`	maximal number of iterations allowed.
`rtol`	relative error tolerance, either a scalar or a vector, one value for each element in the unknown x.
`atol`	absolute error tolerance, either a scalar or a vector, one value for each element in x.
`ctol`	a scalar. If between two iterations, the maximal change in the variable values is less than this amount, then it is assumed that the root is found.
`returnRoot`	logical; If TRUE, return the root object, otherwise return the root itself.

Details

Once converted to log-scale numbers, they can be used in the in the `mexpit()` to get the probabilities back with the following equation

$$p_i = \frac{e^{x_i}}{1 + \sum_{j=1}^{N-1} e^{x_j}}$$

This ensures one remaining probability will add to one, that is

$$p_N = \frac{1}{1 + \sum_{j=1}^{N-1} e^{x_j}}$$

Unfortunately, the log-scale inverse cannot be solved analytically, so it is solved with the `rootSolve::multiroot()` function.

You may adjust some of the root finding options when using this function.

When running `nlmixr2` with mixture models (ie. `mix()` models), the `mlogit()` function is called in the probabilities and the log-based values are used in the optimization problem. The probabilities are determined by the `mexpit()` function.

Value

A numeric vector of the log-scale numbers for use in regressions where the sum of a set of probabilities must add to be one.

Author(s)

Matthew L. Fidler

Examples

```
mlogit(0.1, 0.2, 0.3)

mlogit(0.1, 0.2, 0.3, returnRoot = TRUE)
```

`model.function` *Model block for rxode2/nlmixr models*

Description

Model block for rxode2/nlmixr models

Usage

```
## S3 method for class '`function`'
model(
  x,
  ...,
  append = NULL,
  auto = getOption("rxode2.autoVarPiping", TRUE),
  cov = NULL,
  envir = parent.frame()
)

## S3 method for class 'rxUi'
model(
  x,
  ...,
  append = NULL,
  auto = getOption("rxode2.autoVarPiping", TRUE),
  cov = NULL,
  envir = parent.frame()
)

## S3 method for class 'rxode2'
model(
  x,
  ...,
  append = NULL,
  auto = getOption("rxode2.autoVarPiping", TRUE),
  cov = NULL,
  envir = parent.frame()
```

```
)

## S3 method for class 'rxModelVars'
model(
  x,
  ...,
  append = NULL,
  auto = getOption("rxode2.autoVarPiping", TRUE),
  cov = NULL,
  envir = parent.frame()
)

model(
  x,
  ...,
  append = FALSE,
  auto = getOption("rxode2.autoVarPiping", TRUE),
  cov = NULL,
  envir = parent.frame()
)

## Default S3 method:
model(x, ..., append = FALSE, cov = NULL, envir = parent.frame())
```

Arguments

x	model expression
...	Other arguments
append	This is a boolean to determine if the lines are appended in piping. The possible values for this is: • TRUE which is when the lines are appended to the model instead of replaced • FALSE when the lines are replaced in the model (default) • NA is when the lines are pre-pended to the model instead of replaced • lhs expression, which will append the lines after the last observed line of the expression lhs
auto	This boolean tells if piping automatically selects the parameters should be characterized as a population parameter, between subject variability, or a covariate. When TRUE this automatic selection occurs. When FALSE this automatic selection is turned off and everything is added as a covariate (which can be promoted to a parameter with the ini statement). By default this is TRUE, but it can be changed by options(rxode2.autoVarPiping=FALSE).
cov	is a character vector of variables that should be assumed to be covariates. This will override automatic promotion to a population parameter estimate (or an eta)
envir	the environment in which unevaluated model expressions is to be evaluated. May also be NULL, a list, a data frame, a pairlist or an integer as specified to sys.call.

Value

Model block with ini information included. `ini` must be called before `model` block

Author(s)

Matthew Fidler

`model<-` *Assign the model block in the rxode2 related object*

Description

Assign the model block in the rxode2 related object

Usage

```
model(x, envir = environment(x)) <- value
```

Arguments

`x`	rxode2 related object
`envir`	Environment where assignment occurs
`value`	Value of the object

Value

rxode2 related object

Author(s)

Matthew L. Fidler

`modelExtract` *Extract model lines from a rxui model*

Description

Extract model lines from a rxui model

Usage

```
modelExtract(
  x,
  ...,
  expression = FALSE,
  endpoint = FALSE,
  lines = FALSE,
  envir = parent.frame()
)

## S3 method for class '`function`'
modelExtract(
  x,
  ...,
  expression = FALSE,
  endpoint = FALSE,
  lines = FALSE,
  envir = parent.frame()
)

## S3 method for class 'rxUi'
modelExtract(
  x,
  ...,
  expression = FALSE,
  endpoint = FALSE,
  lines = FALSE,
  envir = parent.frame()
)

## S3 method for class 'rxode2'
modelExtract(
  x,
  ...,
  expression = FALSE,
  endpoint = FALSE,
  lines = FALSE,
  envir = parent.frame()
)

## S3 method for class 'rxModelVars'
modelExtract(
  x,
  ...,
  expression = FALSE,
  endpoint = FALSE,
  lines = FALSE,
  envir = parent.frame()
```

```
)

## Default S3 method:
modelExtract(
  x,
  ...,
  expression = FALSE,
  endpoint = FALSE,
  lines = FALSE,
  envir = parent.frame()
)
```

Arguments

x	model to extract lines from
...	variables to extract. When it is missing, it will extract the entire model (conditioned on the endpoint option below)
expression	return expressions (if TRUE) or strings (if FALSE)
endpoint	include endpoint. This can be: • NA – Missing means include both the endpoint and non-endpoint lines • TRUE – Only include endpoint lines • FALSE – Only include non-endpoint lines
lines	is a boolean. When TRUE this will add the lines as an attribute to the output value ie attr(, "lines")
envir	Environment for evaluating variables

Value

expressions or strings of extracted lines. Note if there is a duplicated lhs expression in the line, it will return both lines

Author(s)

Matthew L. Fidler

Examples

```
one.compartment <- function() {
  ini({
    tka <- 0.45 # Log Ka
    tcl <- 1 # Log Cl
    tv <- 3.45    # Log V
    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)
   d/dt(depot)  <- -ka * depot
   d/dt(center) <-  ka * depot - cl / v * center
   cp <- center / v
   cp ~ add(add.sd)
 })
}

f <- one.compartment()

modelExtract(f, cp)

modelExtract(one.compartment, d/dt(depot))

# from variable
var <- "d/dt(depot)"

modelExtract(one.compartment, var)

modelExtract(f, endpoint=NA, lines=TRUE, expression=TRUE)
```

`multiply` *Multiply dose the rxode2 system in the model*

Description

Multiply dose the rxode2 system in the model

Usage

```
multiply(amt, cmt = 1)
```

Arguments

`amt`	Numeric dose amount (for dose events) or `0` for observations. When `rate > 0` this is interpreted as the total infusion amount and the infusion duration is `amt / rate`.
`cmt`	Integer compartment number (1-based) to which the dose is applied. Default is 1

Details

Behavior inside a model:

`multiply()` is evaluated at every output time point (when the solver is exactly at a scheduled event time). The pushed event is inserted into the individual's event timeline and the solver visits it now or at the specified future time.

The number of events that may be pushed per individual is limited by the `maxExtra` argument of `rxSolve()`. When `maxExtra = 0` (the default) there is no limit. Exceeding the limit causes an error.

Past-time pushes (where `time < t`) are silently ignored and counted; a warning is issued after solving.

Value

This function is only meaningful inside an rxode2 model; it returns `NULL` invisibly if called from R directly (after signaling an error).

`obs` *Add observations to a model*

Description

Add observations to a model

Usage

```
obs(...)
```

Arguments

`...`	Numeric values to be added as observations after the current time `t`. They would be added as events with `evid = 0` and `amt = 0` at the specified future time points. The time points are relative to the current model time `t` (i.e., `obs(12)` adds an observation at `t + 12`). If this is a ui model, you can also specify observations with obs(seq(0, 24, by=0.5)) to add observations every 0.5 time units from t to t+24 since it will resolve to the obs() inside of the final rxode2 model.

Details

Behavior inside a model:

`obs()` is evaluated at every output time point (when the solver is exactly at a scheduled event time). The pushed event is inserted into the individual's event timeline and the solver visits it now or at the specified future time.

The number of events that may be pushed per individual is limited by the `maxExtra` argument of `rxSolve()`. When `maxExtra = 0` (the default) there is no limit. Exceeding the limit causes an error.

Past-time pushes (where `time < t`) are silently ignored and counted; a warning is issued after solving.

Value

This function is only meaningful inside an rxode2 model; it returns `NULL` invisibly if called from R directly (after signaling an error).

Author(s)

Matthew L. Fidler

`odeMethodToInt` *Conversion between character and integer ODE integration methods for rxode2*

Description

If `NULL` is given as the method, all choices are returned as a named vector.

Usage

```
odeMethodToInt(
 method = c("liblsoda", "lsoda", "dop853", "indLin", "f78", "rk4", "ck54", "ab", "abm",
  "dop5", "bs", "ros4", "iem", "sem", "sb3a", "sb3am4", "vv", "mm", "em", "cvode",
  "trapz", "ssp3", "f32", "rk43", "dop54", "vern65", "vern76", "dop87", "vern98",
  "ros43", "ros6", "backwardEuler", "gauss6", "iiic6", "radauiia5", "geng5", "sdirk43",
  "euler", "midpoint", "heun", "ssp22", "rk3", "ssp53", "s4", "r4", "ls44", "ls54",
   "ssp54", "s5", "rk5", "c5", "l5", "lk5a", "lk5b", "b6", "s7", "s8_10", "cv8",
    "s8_12", "s10",
     "z10", "o10", "h10", "dp54", "v65e", "v76e", "dp87", "v98e",
  "ssp33", "bs32", "ssp43", "f45", "t54", "s54", "pp54", "pp54b", "bs54", "ss54",
  "dp65", "c65", "tp64", "v65r", "v65", "dverk65", "tf65", "tp75", "tmy7", "tmy7s",
  "v76r", "ss76", "v78", "dverk78", "dp85", "tp86", "v87e", "v87r", "ev87", "k87",
  "f89", "v89", "t98a", "v98r", "s98", "f108", "c108", "b109", "s1110a", "f1210",
  "o129", "f1412", "lsode", "bdf", "rk4s", "eulers", "midpoints", "heuns", "dop5s",
   "dop853s", "ck54s", "bs32s", "vern65s",
    "vern76s", "dop87s", "f78s", "ros4s",
  "radauiia5s", "backwardEulers", "gauss6s", "sdirk43s", "iiic6s", "ros43s", "ros6s",
  "geng5s", "rk3s", "rk43s", "cvodesadj", "liblsodaadj", "abs", "dop54s", "dp54s",
  "vern98s", "f45s", "t54s", "pp54s", "pp54bs", "bs54s", "ss54s", "dp65s", "c65s",
  "tp64s", "v65rs", "dverk65s", "tf65s", "tp75s", "tmy7sadj", "tmy7adj", "v76rs",
  "ss76s", "v78s", "dverk78s", "dp85s", "tp86s", "v87es", "v87rs", "ev87s", "k87s",
   "v89s", "t98as", "v98rs", "s98s", "c108s", "b109s",
    "s1110as", "o129s")
)
```

Arguments

`method`	The method for solving ODEs. Currently this supports: • `"liblsoda"` thread safe lsoda. This supports parallel thread-based solving, and ignores user Jacobian specification. • `"lsoda"` – LSODA solver. Does not support parallel thread-based solving, but allows user Jacobian specification.

- `"dop853"` – DOP853 solver. Supports parallel thread-based solving; does not support user Jacobian specification.
- `"indLin"` – Solving through inductive linearization. The rxode2 dll must be setup specially to use this solving routine. Supports parallel thread-based solving and honors `cores`.
- `"f78"` – Runge-Kutta Fehlberg 78 solver using Boost's odeint library.
- `"rk4"` – Runge-Kutta 4 solver using Boost's odeint library.
- `"ck54"` – Cash-Karp 5(4) solver using Boost's odeint library.
- `"ab"` – Adams-Bashforth multi-step solver with a direct implementation (order 1-8, default 5). Uses RK4 to initialize the derivative history and then applies the Adams-Bashforth extrapolation formula. Is a fixed-step method (step size controlled by `hmin`).
- `"abm"` – Adams-Bashforth-Moulton solver using Boost's odeint library.
- `"dop5"` – DOPRI5 solver using Boost's odeint library (supports dense output).
- `"bs"` – Bulirsch-Stoer solver using Boost's odeint library (supports dense output).
- `"ros4"` **(implicit)** – Rosenbrock 4 solver using Boost's odeint library (supports dense output). Requires an analytical Jacobian (auto-computed from the model when not provided); falls back to `liblsoda` if Jacobian generation fails.
- `"ros43"` **(implicit)** – Kaps-Rentrop 4th-order A-stable Rosenbrock method (GRK4A) from libode. Adaptive step size via embedded 3rd-order error estimate. Requires an analytical Jacobian (auto-computed from the model when not provided); falls back to `liblsoda` if Jacobian generation fails.
- `"ros6"` **(implicit)** – Kaps-Wanner 6th-order A-stable Rosenbrock method (ROW6A) from libode. Fixed step size (set with `hmin`). Requires Jacobian; falls back to `liblsoda` if unavailable.
- `"backwardEuler"` **(implicit)** – Backward Euler (BDF-1), 1st-order L-stable fully implicit method from libode. Fixed step size (set with `hmin`). Requires Jacobian; falls back to `liblsoda` if unavailable.
- `"gauss6"` **(implicit)** – Gauss-Legendre 6th-order A-stable fully-implicit method (3 stages) from libode. Fixed step size. Requires Jacobian; falls back to `liblsoda` if unavailable.
- `"iiic6"` **(implicit)** – Lobatto IIIC 6th-order L-stable fully-implicit method (4 stages) from libode. Fixed step size. Requires Jacobian; falls back to `liblsoda` if unavailable.
- `"radauiia5"` **(implicit)** – Radau IIA 5th-order L-stable fully-implicit method (3 stages) from libode. Fixed step size (set with `hmin`). Requires Jacobian; falls back to `liblsoda` if unavailable.
- `"geng5"` **(implicit)** – Geng 5th-order symplectic fully-implicit method (3 stages) from libode. Fixed step size. Requires Jacobian; falls back to `liblsoda` if unavailable.
- `"sdirk43"` **(implicit)** – L-stable SDIRK 4(3) pair from libode. Adaptive step size via embedded 3rd-order error estimate. Requires Jacobian; falls back to `liblsoda` if unavailable.

- `"iem"` **(implicit)** – Implicit Euler solver using Boost's odeint library. Requires an explicit Jacobian (auto-generated via `calcJac=TRUE` when not provided). Uses Boost.uBLAS vectors and is a fixed-step method (step size controlled by `hmin`).
- `"sem"` – Symplectic Euler solver using Boost's odeint library. Requires splitting the state into coordinate-momentum pairs (and automatically pads odd-dimensional systems). Is a fixed-step method (step size controlled by `hmin`).
- `"sb3a"` – Symplectic Runge-Kutta-Nystrom SB3A McLachlan stepper using Boost's odeint library. Requires splitting the state into coordinate-momentum pairs (and automatically pads odd-dimensional systems). Is a fixed-step method (step size controlled by `hmin`).
- `"sb3am4"` – Symplectic Runge-Kutta-Nystrom SB3A M4 McLachlan stepper using Boost's odeint library. Requires splitting the state into coordinate-momentum pairs (and automatically pads odd-dimensional systems). Is a fixed-step method (step size controlled by `hmin`).
- `"vv"` – Velocity Verlet stepper using Boost's odeint library. Requires splitting the state into coordinate-momentum pairs (and automatically pads odd-dimensional systems). Is a fixed-step method (step size controlled by `hmin`).
- `"mm"` – Modified Midpoint stepper using Boost's odeint library. Is a fixed-step method (step size controlled by `hmin`) and uses the `order` parameter to configure the number of intermediate steps.
- `"em"` – Explicit Euler stepper using Boost's odeint library. Is a fixed-step method (step size controlled by `hmin`).
- `"cvode"` – CVODE BDF stiff solver from the SUNDIALS library (sources and headers vendored into rxode2). Supports thread-parallel solving and per-compartment absolute tolerances.
- `"trapz"` – Explicit trapezoidal rule (Heun's method), a 2nd-order fixed-step Runge-Kutta method implemented via the libode library. Each inter-event interval is integrated with fixed steps of size `hmin` (default `0.01` when `hmin=0`). If `hmin` would require more than `maxsteps` steps to span the interval, the step size is automatically enlarged to stay within that limit. When an inter-event interval is shorter than the nominal step size (e.g., two doses very close together), the step is silently clamped to the interval length so that short intervals are always handled correctly. Supports parallel thread-based solving and steady-state (`ss=1`) dosing. The method does not use `atol` or `rtol` (fixed-step; no error control). Steady-state convergence is governed by `ssAtol`, `ssRtol`, `minSS`, `maxSS`, and `strictSS`.
- `"ssp3"` – Strong Stability-Preserving Runge-Kutta of order 3 (SSP-RK3, Shu-Osher method), implemented via the libode library. A 3-stage, 3rd-order explicit fixed-step method with superior non-oscillatory properties for problems with discontinuities or sharp fronts. Uses the Butcher tableau c2=1, a21=1; c3=1/2, a31=1/4, a32=1/4; b1=1/6, b2=1/6, b3=2/3. The step-size behavior, clamping, and steady-state options (`hmin`, `maxsteps`, `ssAtol`, `ssRtol`, `minSS`, `maxSS`, `strictSS`) are identical to `"trapz"`. Does not use `atol` or `rtol` (fixed-step; no error control). Supports parallel thread-based solving and steady-state (`ss=1`) dosing.

- "f32" – Fehlberg 3(2) embedded pair from libode (class OdeRKF32). A 3-stage, 3rd-order adaptive method with a built-in 2nd-order error estimate for automatic step-size control. Tableau: c2=1, a21=1; c3=1/2, a31=1/4, a32=1/4; primary (3rd-order) weights d1=1/6, d2=1/6, d3=4/6; embedded (2nd-order) weights b1=1/2, b2=1/2. Uses atol and rtol for error control. The hmin parameter sets the initial step size (default 0.01); subsequent steps are chosen adaptively. The total number of accepted and rejected steps is bounded by maxsteps. Supports parallel thread-based solving and steady-state (ss=1) dosing with convergence governed by ssAtol, ssRtol, minSS, maxSS, and strictSS.
- "rk43" – Runge-Kutta 4(3) pair from libode. A 5-stage, 4th-order adaptive method with a built-in 3rd-order embedded error estimate for automatic step-size control. Tableau: c2=1/3, a21=1/3; c3=2/3, a31=-1/3, a32=1; c4=1, a41=1, a42=-1, a43=1; a5j=bj (FSAL). Primary (4th-order) weights b1=b4=1/8, b2=b3=3/8; embedded (3rd-order) weights d1=1/12, d2=1/2, d3=1/4, d5=1/6. Since a5j=bj the 5th stage evaluation is reused as the 1st stage of the next step (FSAL). Uses atol and rtol for error control. The hmin parameter sets the initial step size (default 0.01); subsequent steps are chosen adaptively. The total number of steps is bounded by maxsteps. Supports parallel thread-based solving and steady-state (ss=1) dosing with convergence governed by ssAtol, ssRtol, minSS, maxSS, and strictSS.
- "dop54" – Dormand-Prince 5(4) FSAL method (Dormand & Prince 1980), implemented via the libode library. The same algorithm as MATLAB's ode45. A 7-stage, 5th-order adaptive method with an embedded 4th-order error estimate for automatic step-size control (FSAL: the 7th stage evaluation is reused as the 1st stage of the next step, giving effectively 6 function evaluations per accepted step). Uses atol and rtol for error control. The hmin parameter sets the initial step size (default 0.01); subsequent steps are chosen adaptively. The total number of steps is bounded by maxsteps. Supports parallel thread-based solving and steady-state (ss=1) dosing with convergence governed by ssAtol, ssRtol, minSS, maxSS, and strictSS. NaN/Inf in derivatives (e.g. from NA parameters) is detected immediately and the solve exits with NA output for the affected subject.
- "vern65" – Jim Verner's "most efficient" 6(5) FSAL pair (9 stages), implemented via the libode library using coefficients from Verner's own website (RKV65.IIIXb.Efficient). A 6th-order adaptive method with an embedded 5th-order error estimate. The sparse tableau (many zero a-coefficients) reduces function evaluations; the FSAL property means the 9th stage evaluation is reused as the 1st stage of the next step, giving effectively 8 function evaluations per accepted step. Uses atol and rtol for error control. The hmin parameter sets the initial step size (default 0.01); subsequent steps are chosen adaptively. The total number of steps is bounded by maxsteps. Supports parallel thread-based solving and steady-state (ss=1) dosing with convergence governed by ssAtol, ssRtol, minSS, maxSS, and strictSS. NaN/Inf in derivatives is detected immediately and the solve exits with NA output.
- "vern76" – Jim Verner's "most efficient" 7(6) pair (10 stages), implemented via the libode library using coefficients from Verner's own website (RKV76.IIa.Efficient).

A 7th-order adaptive method with an embedded 6th-order error estimate. The sparse tableau reduces function evaluations per step. Uses `atol` and `rtol` for error control. The `hmin` parameter sets the initial step size (default `0.01`); subsequent steps are chosen adaptively. The total number of steps is bounded by `maxsteps`. Supports parallel thread-based solving and steady-state (`ss=1`) dosing with convergence governed by `ssAtol`, `ssRtol`, `minSS`, `maxSS`, and `strictSS`. NaN/Inf in derivatives is detected immediately and the solve exits with NA output.

- `"dop87"` – Dormand-Prince 8(7) pair (13 stages), implemented via the libode library using coefficients from Hairer, Norsett and Wanner (1993) "Solving ODEs I" (2nd ed.). An 8th-order adaptive method with an embedded 7th-order error estimate. Both solutions are computed from the original state in the final step loop. Uses `atol` and `rtol` for error control. The `hmin` parameter sets the initial step size (default `0.01`); subsequent steps are chosen adaptively. The total number of steps is bounded by `maxsteps`. Supports parallel thread-based solving and steady-state (`ss=1`) dosing with convergence governed by `ssAtol`, `ssRtol`, `minSS`, `maxSS`, and `strictSS`. NaN/Inf in derivatives is detected immediately and the solve exits with NA output.
- `"vern98"` – Jim Verner's "most efficient" 9(8) pair (16 stages), implemented via the libode library using coefficients from Verner's own website (RKV98.IIa.Efficient). A 9th-order adaptive method with an embedded 8th-order error estimate. The highly sparse tableau (stages 8-16 use only k_0 and $k_{5..}$) minimizes function evaluations per step. Both solutions are computed from the original state in the final step loop. Uses `atol` and `rtol` for error control. The `hmin` parameter sets the initial step size (default `0.01`); subsequent steps are chosen adaptively. The total number of steps is bounded by `maxsteps`. Supports parallel thread-based solving and steady-state (`ss=1`) dosing with convergence governed by `ssAtol`, `ssRtol`, `minSS`, `maxSS`, and `strictSS`. NaN/Inf in derivatives is detected immediately and the solve exits with NA output.

Aliases for existing methods (no additional C++ code; `hmin`, `atol`, `rtol`, and `maxsteps` follow the aliased method):

- `"dp54"` – alias for `"dop54"` (Dormand-Prince 5(4) FSAL, libode).
- `"v65e"` – alias for `"vern65"` (Verner 6(5) efficient, libode).
- `"v76e"` – alias for `"vern76"` (Verner 7(6) efficient, libode).
- `"dp87"` – alias for `"dop87"` (Dormand-Prince 8(7), libode).
- `"v98e"` – alias for `"vern98"` (Verner 9(8) efficient, libode).
- `"ssp33"` – alias for `"ssp3"` (Strong Stability-Preserving RK3, libode).

libode fixed-step explicit methods. All use `hmin` as the fixed step size (default `0.01` when `hmin=0`); `atol` and `rtol` are ignored (no error control); `maxsteps` bounds the total number of steps; NaN/Inf in derivatives sets `ind$err` and the subject exits with NA output. All support parallel thread-based solving.

- `"euler"` – Forward (explicit) Euler, 1st-order, 1 stage. Requires a very small step size for accuracy (e.g., `hmin=1e-4`); intended for pedagogical use or method-comparison baselines.

- "midpoint" – Explicit midpoint rule, 2nd-order, 2 stages.
- "heun" – Heun's method (explicit trapezoid), 2nd-order, 2 stages. Identical in structure to "trapz" but uses the libode fixed-step driver.
- "ssp22" – Strong Stability-Preserving 2-stage 2nd-order method (SSP-RK22), 2 stages. Superior non-oscillatory properties near discontinuities.
- "rk3" – Classical 3rd-order Runge-Kutta (Kutta 1901), 3 stages.
- "ssp53" – Strong Stability-Preserving 5-stage 3rd-order method (SSP-RK53), 5 stages. High SSP coefficient for hyperbolic PDEs or event-heavy ODE systems.
- "s4" – Shanks 4th-order method, 4 stages.
- "r4" – Ralston's 4th-order method, 4 stages. Minimizes local truncation error among classical 4-stage 4th-order methods.
- "ls44" – Low-storage 4th-order method, 4 stages. Uses a 2-register update scheme that minimizes memory bandwidth at the cost of a less general tableau structure.
- "ls54" – Low-storage 4th-order method, 5 stages. Five-stage variant of the 2-register low-storage scheme.
- "ssp54" – Strong Stability-Preserving 5-stage 4th-order method (SSP-RK54), 5 stages.
- "s5" – Shanks 5th-order method, 5 stages.
- "rk5" – Classical 5th-order Runge-Kutta, 6 stages.
- "c5" – Cassity 5th-order method, 6 stages.
- "l5" – Lawson 5th-order method, 6 stages.
- "lk5a" – Luther-Konen 5th-order method, variant A, 6 stages.
- "lk5b" – Luther-Konen 5th-order method, variant B, 6 stages.
- "b6" – Butcher 6th-order method, 7 stages.
- "s7" – Shanks 7th-order method, 9 stages.
- "s8_10" – Shanks 8th-order method, 10 stages.
- "cv8" – Cooper-Verner 8th-order method, 11 stages.
- "s8_12" – Shanks 8th-order method, 12 stages.
- "s10" – Stepanov 10th-order method, 15 stages. Requires a moderate step size (e.g., hmin=1.0) because a single step is accurate to very high order.
- "z10" – Zhang 10th-order method, 16 stages.
- "o10" – Ono 10th-order method, 17 stages.
- "h10" – Hairer 10th-order method, 17 stages.

libode adaptive (variable-step) explicit methods. All use atol and rtol for error control; hmin sets the initial step size (default 0.01 when hmin=0); hmax sets the maximum step size; maxsteps bounds total steps; NaN/Inf in derivatives sets ind$err and the subject exits with NA output. All support parallel thread-based solving and steady-state (ss=1) dosing (convergence governed by ssAtol, ssRtol, minSS, maxSS, strictSS).

- "bs32" – Bogacki-Shampine 3(2) FSAL pair, 4 stages (Bogacki & Shampine 1989). 3rd-order primary with 2nd-order embedded error estimate. FSAL: the 4th-stage evaluation is reused as the 1st stage of the next step. The same algorithm as Julia BS3() and MATLAB ode23.

- "ssp43" – Strong Stability-Preserving 4(3) pair, 4 stages. Adaptive SSP method with a 3rd-order embedded error estimate.
- "f45" – Fehlberg 4(5) pair, 6 stages (Fehlberg 1970). 4th-order primary solution with a 5th-order embedded estimate used for error control.
- "t54" – Tsitouras 5(4) FSAL pair, 7 stages (Tsitouras 2011). 5th-order primary with 4th-order embedded error estimate. FSAL: the 7th stage is reused as the 1st stage of the next step. The same Butcher tableau as Julia's `Tsit5()`.
- "s54" – Stepanov 5(4) FSAL pair, 7 stages. 5th-order primary with 4th-order embedded error estimate.
- "pp54" – Papakostas-Papageorgiou 5(4) FSAL pair, 7 stages.
- "pp54b" – Papakostas-Papageorgiou 5(4) variant B FSAL pair, 7 stages.
- "bs54" – Bogacki-Shampine 5(4) pair, 8 stages. 5th-order primary with 4th-order embedded error estimate (non-FSAL).
- "ss54" – Sharp-Smart 5(4) pair, 7 stages.
- "dp65" – Dormand-Prince 6(5) pair, 8 stages. 6th-order primary with 5th-order embedded error estimate.
- "c65" – Calvo 6(5) pair, 9 stages.
- "tp64" – Tsitouras-Papakostas 6(4) pair, 7 stages. 6th-order primary with 4th-order embedded error estimate.
- "v65r" – Verner "robust" 6(5) FSAL pair, 9 stages. Robust variant of Verner's 6(5) family with wider stability region.
- "v65" – Verner 6(5) pair, 8 stages (non-FSAL).
- "dverk65" – Verner DVERK 6(5) pair, 8 stages. Coefficients from the classic DVERK Fortran code distributed by Hull and Enright.
- "tf65" – Tsitouras-Famelis 6(5) FSAL pair, 9 stages.
- "tp75" – Tsitouras-Papakostas 7(5) pair, 9 stages. 7th-order primary with 5th-order embedded error estimate.
- "tmy7" – Tanaka-Muramatsu-Yamashita 7th-order pair, 10 stages. The same family as Julia's `TanYam7()`.
- "tmy7s" – Tanaka-Muramatsu-Yamashita 7th-order stable variant, 10 stages. Alternative coefficient set with wider stability region.
- "v76r" – Verner "robust" 7(6) pair, 10 stages.
- "ss76" – Sharp-Smart 7(6) pair, 11 stages.
- "v78" – Verner 7(8) pair, 13 stages. 7th-order primary with 8th-order embedded error estimate. Closest rxode2 analog to Julia `Vern8()`.
- "dverk78" – Verner DVERK 7(8) pair, 13 stages. Coefficients from the classic DVERK Fortran code; companion to "dverk65". Also close to Julia `Vern8()`.
- "dp85" – Dormand-Prince 8(5) pair, 12 stages. 8th-order primary with 5th-order embedded error estimate.
- "tp86" – Tsitouras-Papakostas 8(6) pair, 12 stages. The same family as Julia's `TsitPap8()`.
- "v87e" – Verner "efficient" 8(7) pair, 13 stages.
- "v87r" – Verner "robust" 8(7) pair, 13 stages.

- "ev87" – Enright-Verner 8(7) pair, 13 stages.
- "k87" – Kovalnogov-Fedorov-Karpukhina-Simos 8(7) pair, 13 stages.
- "f89" – Fehlberg 8(9) pair, 17 stages. 8th-order primary with 9th-order embedded estimate. Same family as MATLAB ode89.
- "v89" – Verner 8(9) pair, 16 stages. Alternative to "f89" in the same order bracket. Also close to MATLAB ode89.
- "t98a" – Tsitouras 9(8) variant A pair, 16 stages.
- "v98r" – Verner "robust" 9(8) pair, 16 stages.
- "s98" – Sharp 9(8) pair, 16 stages.
- "f108" – Feagin 10(8) pair, 17 stages (Feagin 2007). 10th-order primary with 8th-order embedded error estimate. The same method as Julia's Feagin10(). The large number of stages carries elevated transcription-error risk; verified against Williams' libode canonical order test.
- "c108" – Curtis 10(8) pair, 21 stages.
- "b109" – Baker 10(9) pair, 21 stages.
- "s1110a" – Stone 11(10) variant A pair, 26 stages. 11th-order primary; very few published references.
- "f1210" – Feagin 12(10) pair, 25 stages (Feagin 2007). 12th-order primary with 10th-order embedded error estimate. The same method as Julia's Feagin12(). Elevated transcription-error risk due to many stages; verified against Williams' libode canonical order test.
- "o129" – Ono 12(9) pair, 29 stages.
- "f1412" – Feagin 14(12) pair, 35 stages (Feagin 2007). 14th-order primary with 12th-order embedded error estimate. The same method as Julia's Feagin14(). The highest-order method in rxode2; elevated transcription-error risk due to 35 stages; verified against Williams' libode canonical order test.

deSolve-derived Fortran solvers (from the deSolve R package). Both use non-reentrant Fortran COMMON blocks and therefore run single-threaded only, regardless of the cores setting. They are BDF (Backward Differentiation Formula) stiff solvers and are most useful when you want behavior comparable to deSolve's lsode() or bdf()/vode() functions.

- "lsode" – DLSODE (Hindmarsh 1983, ODEPACK) in Adams (non-stiff) mode, MF=10. Variable-order Adams method, order 1-12; no Jacobian evaluation. Corresponds to deSolve's lsode(method="adams"). Adaptive step-size; error controlled by atol and rtol. **Not thread-safe** – always runs on a single core.
- "bdf" – DLSODE (Hindmarsh 1983, ODEPACK) in BDF (stiff) mode, MF=22. Variable-order BDF method, order 1-5; internally generated dense Jacobian. Corresponds to deSolve's bdf() / vode(method="bdf"). Adaptive step-size; error controlled by atol and rtol. **Not thread-safe** – always runs on a single core.

Value

An integer for the method (unless the input is NULL, in which case, see the details)

odeToLin *Convert ODE-based linear compartment models to analytical linCmt() form*

Description

Detects whether a model's ODE equations form a standard 1-3 compartment PK system and, if so, replaces the `d/dt()` equations and the central output assignment (e.g. `cp <- central/v`) with `cp <- linCmt()`, preserving all other model lines. Detection requires linear ODE right-hand sides, 1-4 compartments in a standard depot/central/peripheral topology, and one output line of the form `var <- central_cmt / v_expr`. Named parameter assignments are retained so `linCmt()` can infer the parameterization. Every right-hand side term must be proportional to a compartment; an exogenous input (`transit()` absorption, a zero-order or endogenous production rate, a dose carried in a covariate column) has no `linCmt()` representation and declines the conversion. So does a rate coefficient, or a concentration line, that is not what its named parameters imply – a scale factor, an inverted ratio or a covariate factor written into the ODE – since `linCmt()` rebuilds the rates from those names alone.

Usage

```
odeToLin(ui)
```

Arguments

`ui`	rxUi-like model object (function, rxUi, or anything accepted by `as.rxUi`).

Value

An rxUi model with ODE equations replaced by `linCmt()`, or (with a message) the original model unchanged if conversion is not possible.

Author(s)

Matthew Fidler

See Also

`linToOde` for the inverse transformation.

Examples

```
oneCmtOde <- function() {
  ini({
    tka <- 0.45
    tcl <- log(2.7)
    tv <- 3.45
    add.sd <- 0.7
  })
```

```
  model({
    ka  <- exp(tka)
    cl  <- exp(tcl)
    v   <- exp(tv)
    d/dt(depot)   <- -ka * depot
    d/dt(central) <- ka * depot - cl/v * central
    cp <- central / v
    cp ~ add(add.sd)
  })
}
linCmtModel <- odeToLin(oneCmtOde)
```

---

phantom *Administer a phantom/transit dose inside a rxode2 model*

Description

Administer a phantom/transit dose inside a rxode2 model

Usage

```
phantom(amt, cmt = 1, ii = 0, addl = 0, ss = 0)
```

Arguments

amt	Numeric dose amount (for dose events) or `0` for observations. When `rate > 0` this is interpreted as the total infusion amount and the infusion duration is `amt / rate`.
cmt	Integer compartment number (1-based) to which the dose is applied. Default is 1
ii	Numeric inter-dose interval. Used together with `addl` to schedule repeat doses at `time`, `time + ii`, `time + 2*ii`, ..., `time + addl*ii`. Also required when `ss > 0`. Default 0
addl	Integer number of *additional* doses beyond the first. The total number of doses pushed is `addl + 1`, spaced `ii` apart. Each dose is pushed as a standalone event (not as a periodic schedule in the event table).
ss	Integer steady-state flag applied to the *first* dose only (`addl` repetitions always use `ss = 0`). `0` No steady-state (default). `1` Steady-state additive: add SS solution to current state. `2` Steady-state replace: replace current state with SS solution.

Details

Behavior inside a model:

`phantom()` is evaluated at every output time point (when the solver is exactly at a scheduled event time). The pushed event is inserted into the individual's event timeline and the solver visits it now or at the specified future time.

The number of events that may be pushed per individual is limited by the `maxExtra` argument of `rxSolve()`. When `maxExtra = 0` (the default) there is no limit. Exceeding the limit causes an error.

Past-time pushes (where `time < t`) are silently ignored and counted; a warning is issued after solving.

Value

This function is only meaningful inside an rxode2 model; it returns `NULL` invisibly if called from R directly (after signaling an error).

Author(s)

Matthew L. Fidler

`phi` *Cumulative distribution of standard normal*

Description

Cumulative distribution of standard normal

Usage

```
phi(q)
```

Arguments

`q`	vector of quantiles

Value

cumulative distribution of standard normal distribution

Author(s)

Matthew Fidler

Examples

```
# phi is equivalent to pnorm(x)
phi(3)

# See
pnorm(3)

# This is provided for NONMEM-like compatibility in rxode2 models
```

---

phiU *Inverse CDFs for declared non-normal random effects*

Description

These are the pieces a declared non-normal random effect is built from (see the `dist()` line of an `ini({})` block). The latent random effect stays standard normal, `phiU()` maps it to a uniform, and the family's inverse CDF maps that uniform to the random effect.

Usage

```
phiU(q)

ibeta(a, b, x)

ibetaDer(a, b, x)

ibetaInv(a, b, p)

ibetaDera(a, b, x)

ibetaDerb(a, b, x)

gammapDera(a, x)

studentTDen(x, nu)

studentTCdf(x, nu)

studentTCdfDnu(x, nu)

studentTInv(p, nu)
```

Arguments

q	normal deviate (`phiU()`)
a	first shape parameter
b	second shape parameter

x	value at which the CDF (or its derivative) is taken
p	probability
nu	degrees of freedom

Details

phiU() is phi() bounded away from 0 and 1 by 1e-15. phi() saturates to exactly 0 or 1 in double precision around |q| = 8.3, where an inverse CDF would return an infinity.

ibeta(), ibetaDer() and ibetaInv() are the regularized incomplete beta function, its derivative in x and its inverse – ie pbeta(), dbeta() and qbeta(). studentTCdf(), studentTDen() and studentTInv() are the Student t CDF, density and quantile, written on the incomplete beta so the quantile is exactly the inverse of the CDF at the same tolerance.

gammapDera(), ibetaDera(), ibetaDerb() and studentTCdfDnu() are the derivatives of those CDFs with respect to their SHAPE parameters. None has an elementary closed form, so they are central differences with one Richardson extrapolation. They exist so that rxode2's derivative table is complete for these functions: without them a model using an inverse CDF silently degrades to a one sided finite difference, which is precisely wrong in the tails.

Value

numeric vector, recycled to the longest argument

Author(s)

Matthew L. Fidler

Examples

```
phiU(c(-9, 0, 9))

ibetaInv(2, 3, 0.5) ## == qbeta(0.5, 2, 3)

studentTInv(0.975, 6) ## == qt(0.975, 6)
```

---

plot.rxSolve *Plot rxode2 objects*

Description

Plot rxode2 objects

Usage

```
## S3 method for class 'rxSolve'
plot(x, y, ..., log = "", xlab = "Time", ylab = "")

## S3 method for class 'rxSolveConfint1'
plot(x, y, ..., xlab = "Time", ylab = "", log = "")

## S3 method for class 'rxSolveConfint2'
plot(x, y, ..., xlab = "Time", ylab = "", log = "")
```

Arguments

x	rxode2 object to plot
y	Compartments or left-hand-side values to plot either as a bare name or as a character vector
...	Ignored
log	Should "" (neither x nor y), "x", "y", or "xy" (or "yx") be log-scale?
xlab, ylab	The x and y axis labels

Value

A ggplot2 object

See Also

Other rxode2 plotting: `rxTheme()`

PReLU *Parametric ReLU Activation Function*

Description

Parametric ReLU Activation Function

Usage

```
PReLU(x, alpha = 1)
```

Arguments

x	A numeric vector. All elements must be finite and non-missing.
alpha	A numeric scalar. All elements must be finite and non-missing.

Value

A numeric vector where the ReLU function has been applied to each element of x.

Author(s)

Matthew Fidler

See Also

Other Activation Functions: ELU(), GELU(), ReLU(), SELU(), Swish(), dELU(), dGELU(), dPReLU(), dReLU(), dSELU(), dSwish(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
PReLU(c(-1, 0, 1, 2), 2)


# Can also be used in rxode2:
x <- rxode2({
   r=PReLU(time, 2)
})

e <- et(c(-1, 0, 1, 2))

rxSolve(x, e)
```

---

print.rxModelVars *Print Values*

Description

print prints its argument and returns it *invisibly* (via invisible(x)). It is a generic function which means that new printing methods can be easily added for new classes.

Usage

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

Arguments

x	an object used to select a method.
...	further arguments passed to or from other methods.

Details

The default method, print.default has its own help page. Use methods("print") to get all the methods for the print generic.

print.factor allows some customization and is used for printing ordered factors as well.

print.table for printing tables allows other customization. As of R 3.0.0, it only prints a description in case of a table with 0-extents (this can happen if a classifier has no valid data).

See noquote as an example of a class whose main purpose is a specific print method.

Value

This returns invisibly the model variables object

References

Chambers JM, Hastie TJ (1992). *Statistical Models in S*. Chapman & Hall, London. ISBN 9780412830402.

See Also

The default method print.default, and help for the methods above; further options, noquote.

For more customizable (but cumbersome) printing, see cat, format or also write. For a simple prototypical print method, see .print.via.format in package **tools**.

Examples

```
require(stats)

ts(1:20)  #-- print is the "Default function" --> print.ts(.) is called
for(i in 1:3) print(1:i)

## Printing of factors
attenu$station ## 117 levels -> 'max.levels' depending on width

## ordered factors: levels  "l1 < l2 < .."
esoph$agegp[1:12]
esoph$alcgp[1:12]

## Printing of sparse (contingency) tables
set.seed(521)
t1 <- round(abs(rt(200, df = 1.8)))
t2 <- round(abs(rt(200, df = 1.4)))
table(t1, t2) # simple
print(table(t1, t2), zero.print = ".") # nicer to read

## same for non-integer "table":
T <- table(t2,t1)
T <- T * (1+round(rlnorm(length(T)))/4)
print(T, zero.print = ".") # quite nicer,
print.table(T[,2:8] * 1e9, digits=3, zero.print = ".")
## still slightly inferior to  Matrix::Matrix(T)  for larger T
```

```
## Corner cases with empty extents:
table(1, NA) # < table of extent 1 x 0 >
```

`probit` *probit and inverse probit functions*

Description

probit and inverse probit functions

Usage

```
probit(x, low = 0, high = 1)

probitInv(x, low = 0, high = 1)
```

Arguments

`x`	Input value(s) in range [low,high] to translate -Inf to Inf
`low`	Lowest value in the range
`high`	Highest value in the range

Value

values from probit, probitInv and probitNormInfo

Examples

```
probit(0.25)

probitInv(-0.674)

probitNormInfo(probit(0.25), sd = 0.1)

probitNormInfo(probit(1, 0, 10), sd = 1, low = 0, high = 10)
```

ReLU *Rectified Linear Unit (ReLU) Activation Function*

Description

This function applies the Rectified Linear Unit (ReLU) activation function to the input numeric vector. The ReLU function is defined as the positive part of its argument: $f(x) = max(0, x)$.

Usage

```
ReLU(x)
```

Arguments

x	A numeric vector. All elements must be finite and non-missing.

Value

A numeric vector where the ReLU function has been applied to each element of x.

Author(s)

Matthew Fidler

See Also

Other Activation Functions: ELU(), GELU(), PReLU(), SELU(), Swish(), dELU(), dGELU(), dPReLU(), dReLU(), dSELU(), dSwish(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
ReLU(c(-1, 0, 1, 2))


# Can also be used in rxode2:
x <- rxode2({
   r=ReLU(time)
})

e <- et(c(-1, 0, 1, 2))

rxSolve(x, e)
```

replace *Replace dose the rxode2 system in the model*

Description

Replace dose the rxode2 system in the model

Usage

```
replace(amt, cmt = 1)
```

Arguments

amt	Numeric dose amount (for dose events) or `0` for observations. When `rate > 0` this is interpreted as the total infusion amount and the infusion duration is `amt / rate`.
cmt	Integer compartment number (1-based) to which the dose is applied. Default is 1

Details

Behavior inside a model:

`replace()` is evaluated at every output time point (when the solver is exactly at a scheduled event time). The pushed event is inserted into the individual's event timeline and the solver visits it now or at the specified future time.

The number of events that may be pushed per individual is limited by the `maxExtra` argument of `rxSolve()`. When `maxExtra = 0` (the default) there is no limit. Exceeding the limit causes an error.

Past-time pushes (where `time < t`) are silently ignored and counted; a warning is issued after solving.

Value

This function is only meaningful inside an rxode2 model; it returns `NULL` invisibly if called from R directly (after signaling an error).

reset *Reset the rxode2 system in the model*

Description

Reset the rxode2 system in the model

Usage

```
reset()
```

Details

Behavior inside a model:

`reset()` is evaluated at every output time point (when the solver is exactly at a scheduled event time). The pushed event is inserted into the individual's event timeline and the solver visits it now or at the specified future time.

The number of events that may be pushed per individual is limited by the `maxExtra` argument of `rxSolve()`. When `maxExtra = 0` (the default) there is no limit. Exceeding the limit causes an error.

Past-time pushes (where `time < t`) are silently ignored and counted; a warning is issued after solving.

Value

This function is only meaningful inside an rxode2 model; it returns `NULL` invisibly if called from R directly (after signaling an error).

---

rinvchisq *Scaled Inverse Chi Squared distribution*

Description

Scaled Inverse Chi Squared distribution

Usage

```
rinvchisq(n = 1L, nu = 1, scale = 1)
```

Arguments

`n`	Number of random samples
`nu`	degrees of freedom of inverse chi square
`scale`	Scale of inverse chi squared distribution (default is 1).

Value

a vector of inverse chi squared deviates.

Examples

```
rinvchisq(3, 4, 1) ## Scale = 1, degrees of freedom = 4
rinvchisq(2, 4, 2) ## Scale = 2, degrees of freedom = 4
```

rxAllowUnload *Allow unloading of dlls*

Description

Allow unloading of dlls

Usage

```
rxAllowUnload(allow)
```

Arguments

allow	boolean indicating if garbage collection will unload of rxode2 dlls.

Value

Boolean allow; called for side effects

Author(s)

Matthew Fidler

Examples

```
# Garbage collection will not unload un-used rxode2 dlls
rxAllowUnload(FALSE);

# Garbage collection will unload unused rxode2 dlls
rxAllowUnload(TRUE);
```

rxAppendModel *Append two rxui models together*

Description

Append two rxui models together

Usage

```
rxAppendModel(..., common = TRUE)
```

Arguments

...	models to append together
common	boolean; when TRUE warn if the models have no variables in common

Value

New model with both models appended together

Author(s)

Matthew L. Fidler

Examples

```
ocmt <- function() {
  ini({
    tka <- exp(0.45) # Ka
    tcl <- exp(1) # Cl
    tv <- exp(3.45); # log V
    ## the label("Label name") works with all models
    add.sd <- 0.7
  })
  model({
    ka <- tka
    cl <- tcl
    v <- tv
    d/dt(depot) <- -ka * depot
    d/dt(center) <- ka * depot - cl / v * center
    cp <- center / v
    cp ~ add(add.sd)
  })
}

idr <- function() {
  ini({
    tkin <- log(1)
    tkout <- log(1)
    tic50 <- log(10)
    gamma <- fix(1)
    idr.sd <- 1
  })
  model({
    kin <- exp(tkin)
    kout <- exp(tkout)
    ic50 <- exp(tic50)
    d/dt(eff) <- kin - kout*(1-ceff^gamma/(ic50^gamma+ceff^gamma))
    eff ~ add(idr.sd)
  })
}

rxAppendModel(ocmt |>
  model(ceff=cp,append=TRUE), idr)
```

rxAssignControlValue *Assign Control Variable*

Description

Assign Control Variable

Usage

```
rxAssignControlValue(ui, option, value)
```

Arguments

ui	rxode2 ui function
option	Option name in the control to modify
value	Value of control to modify

Value

Nothing; called for the side effects

Author(s)

Matthew L. Fidler

rxAssignPtr *Assign pointer based on model variables*

Description

Assign pointer based on model variables

Usage

```
rxAssignPtr(object = NULL)
```

Arguments

object	rxode2 family of objects

Value

nothing, called for side effects

rxbeta *Simulate beta variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the sitmo threefry, this currently generates one random deviate from the uniform distribution to seed the engine threefry and then run the code.

Usage

```
rxbeta(shape1, shape2, n = 1L, ncores = 1L)
```

Arguments

shape1, shape2	non-negative parameters of the Beta distribution.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
ncores	Number of cores for the simulation rxnorm simulates using the threefry sitmo generator. rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the rxode2 environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with rxSetSeed()

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

beta random deviates

Examples

```
## Use threefry engine

rxbeta(0.5, 0.5, n = 10) # with rxbeta you have to explicitly state n
```

```
rxbeta(5, 1, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxbeta(1, 3)


## This example uses `rxbeta` directly in the model

rx <- function() {
  model({
    a <- rxbeta(2, 2)
  })
}

et <- et(1, id = 1:2)


s <- rxSolve(rx, et)
```

`rxbinom` *Simulate Binomial variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxbinom(size, prob, n = 1L, ncores = 1L)
```

Arguments

`size`	number of trials (zero or more).
`prob`	probability of success on each trial.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator. `rxnormV` used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of `rxnorm`. It is no longer supported in `rxode2({})` blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

binomial random deviates

Examples

```
## Use threefry engine


rxbinom(10, 0.9, n = 10) # with rxbinom you have to explicitly state n
rxbinom(3, 0.5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxbinom(4, 0.7)


## This example uses `rxbinom` directly in the model

rx <- function() {
  model({
    a <- rxbinom(1, 0.5)
  })
}

et <- et(1, id = 1:2)

s <- rxSolve(rx, et)
```

`rxcauchy` *Simulate Cauchy variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxcauchy(location = 0, scale = 1, n = 1L, ncores = 1L)
```

Arguments

location, scale	location and scale parameters.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
ncores	Number of cores for the simulation rxnorm simulates using the threefry sitmo generator. rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the rxode2 environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with rxSetSeed()

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

Cauchy random deviates

Examples

```
## Use threefry engine

rxcauchy(0, 1, n = 10) # with rxcauchy you have to explicitly state n
rxcauchy(0.5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxcauchy(3)


## This example uses `rxcauchy` directly in the model

rx <- function() {
  model({
    a <- rxcauchy(2)
  })
}

et <- et(1, id = 1:2)
```

```
s <- rxSolve(rx, et)
```

`rxCbindStudyIndividual` *Bind the study parameters and individual parameters*

Description

Bind the study parameters and individual parameters

Usage

```
rxCbindStudyIndividual(studyParameters, individualParameters)
```

Arguments

`studyParameters`
: These are the study parameters, often can be generated by sampling from a population. This can be either a matrix or a data frame

`individualParameters`
: A data frame of individual parameters

Value

Data frame that can be used in rxode2 simulations

Author(s)

Matthew Fidler

Examples

```
# Function for converting coefficient of covariance into a variance
logncv <- function(x){log((x/100)^2+1)}

set.seed(32)

nSub  <-  100
nStud  <-  10

#define theta
theta <- c(lka=log(0.5), # log ka
          lCl=log(5), # log Cl
          lV=log(300) # log V
          )
```

```
#define theta Matrix
thetaMat <- lotri(lCl ~ lognCv(5),
                  lV  ~ lognCv(5),
                  lka ~ lognCv(5))


nev <- nSub*nStud

ev1 <- data.frame(COV1=rnorm(nev,50,30),COV2=rnorm(nev,75,10),
                  COV3=sample(c(1.0,2.0),nev,replace=TRUE))


tmat <-rxRmvn(nStud, theta[dimnames(thetaMat)[[1]]], thetaMat)

rxCbindStudyIndividual(tmat, ev1)
```

`rxchisq` *Simulate chi-squared variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxchisq(df, n = 1L, ncores = 1L)
```

Arguments

`df`	degrees of freedom (non-negative, but can be non-integer).
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator. `rxnormV` used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of `rxnorm`. It is no longer supported in `rxode2({})` blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

chi squared random deviates

Examples

```
## Use threefry engine

rxchisq(0.5, n = 10) # with rxchisq you have to explicitly state n
rxchisq(5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxchisq(1)


## This example uses `rxchisq` directly in the model

rx <- function() {
  model({
    a <- rxchisq(2)
  })
}

et <- et(1, id = 1:2)


s <- rxSolve(rx, et)
```

`rxClean` *Cleanup anonymous DLLs by unloading them*

Description

This cleans up any rxode2 loaded DLLs

Usage

```
rxClean(wd)
```

Arguments

wd	What directory should be cleaned; (DEPRECIATED), this no longer does anything. This unloads all rxode2 anonymous dlls.

Value

TRUE if successful

Author(s)

Matthew L. Fidler

---

rxCompile *Compile a model if needed*

Description

This is the compilation workhorse creating the rxode2 model DLL files.

Usage

```
rxCompile(
  model,
  dir,
  prefix,
  force = FALSE,
  modName = NULL,
  package = NULL,
  ...
)

## S3 method for class 'rxModelVars'
rxCompile(
  model,
  dir = NULL,
  prefix = NULL,
  force = FALSE,
  modName = NULL,
  package = NULL,
  eventSensCode = rep("", 13L),
  ...
)
```

```
## S3 method for class 'character'
rxCompile(
  model,
  dir = NULL,
  prefix = NULL,
  force = FALSE,
  modName = NULL,
  package = NULL,
  eventSensCode = rep("", 13L),
  ...
)

## S3 method for class 'rxDll'
rxCompile(model, ...)

## S3 method for class 'rxode2'
rxCompile(model, ...)
```

Arguments

model	This is the ODE model specification. It can be: • a string containing the set of ordinary differential equations (ODE) and other expressions defining the changes in the dynamic system. • a file name where the ODE system equation is contained An ODE expression enclosed in \{\} (see also the filename argument). For details, see the sections "Details" and rxode2 Syntax below.
dir	This is the model directory where the C file will be stored for compiling. If unspecified, the C code is stored in a temporary directory, then the model is compiled and moved to the current directory. Afterwards the C code is removed. If specified, the C code is stored in the specified directory and then compiled in that directory. The C code is not removed after the DLL is created in the same directory. This can be useful to debug the c-code outputs.
prefix	is a string indicating the prefix to use in the C based functions. If missing, it is calculated based on file name, or md5 of parsed model.
force	is a boolean stating if the (re)compile should be forced if rxode2 detects that the models are the same as already generated.
modName	a string to be used as the model name. This string is used for naming various aspects of the computations, including generating C symbol names, dynamic libraries, etc. Therefore, it is necessary that modName consists of simple ASCII alphanumeric characters starting with a letter.

package	Package name for pre-compiled binaries.
...	Other arguments sent to the rxTrans() function.
eventSensCode	character vector of length 13 giving the C body lines for the event ("jump") sensitivity helper functions dLag/dF/dRate/dDur/d2F/d2Lag/d2Rate/d2Dur/d3F/dFQ/dLagJac/dLagQ/dDurQ (in that order). These are the per-model dosing-parameter total-derivative assignments produced by the event-sensitivity code generator and are emitted verbatim into the corresponding generated functions. The default of thirteen empty strings produces trivial (no-op) helpers, which is what non-sensitivity models and eventSens = "fd" models use.

Value

An rxDll object that has the following components

- dll DLL path
- model model specification
- .c A function to call C code in the correct context from the DLL using the .C() function.
- .call A function to call C code in the correct context from the DLL using the .Call() function.
- args A list of the arguments used to create the rxDll object.

Author(s)

Matthew L.Fidler

See Also

rxode2()

---

rxControlUpdateSens *This updates the tolerances based on the sensitivity equations*

Description

This assumes the normal ODE equations are the first equations and the ODE is expanded by the forward sensitivities or other type of sensitivity (like adjoint)

Usage

```
rxControlUpdateSens(rxControl, sensCmt = NULL, ncmt = NULL)
```

Arguments

rxControl	Input list or rxControl type of list
sensCmt	Number of sensitivity compartments
ncmt	Number of compartments

Value

Updated rxControl where `$atol`, `$rtol`, `$ssAtol` `$ssRtol` are updated with different sensitivities for the normal ODEs (first) and a different sensitivity for the larger compartments (sensitivities).

Author(s)

Matthew L. Fidler

Examples

```
tmp <- rxControl()

tmp2 <- rxControlUpdateSens(tmp, 3, 6)

tmp2$atol
tmp2$rtol
tmp2$ssAtol
tmp2$ssRtol
```

---

rxCreateCache *This will create the cache directory for rxode2 to save between sessions*

Description

When run, if the `R_user_dir` for rxode2's cache isn't present, create the cache

Usage

```
rxCreateCache()
```

Value

nothing

Author(s)

Matthew Fidler

rxD *Add to rxode2's derivative tables*

Description

Add to rxode2's derivative tables

Usage

```
rxD(name, derivatives)
```

Arguments

name	Function Name
derivatives	A list of functions. Each function takes the same number of arguments as the original function. The first function will construct the derivative with respect to the first argument; The second function will construct the derivative with respect to the second argument, and so on.

Value

nothing

Author(s)

Matthew Fidler

Examples

```
## Add an arbitrary list of derivative functions
## In this case the fun(x,y) is assumed to be 0.5*x^2+0.5*y^2

rxD("fun", list(
  function(x, y) {
    return(x)
  },
  function(x, y) {
    return(y)
  }
))
```

`rxDelete` *Delete the DLL for the model*

Description

This function deletes the DLL, but doesn't delete the model information in the object.

Usage

```
rxDelete(obj)
```

Arguments

`obj`	rxode2 family of objects

Value

A boolean stating if the operation was successful.

Author(s)

Matthew L.Fidler

`rxDerived` *Calculate derived parameters for the 1-, 2-, and 3- compartment linear models.*

Description

This calculates the derived parameters based on what is provided in a data frame or arguments

Usage

```
rxDerived(..., verbose = FALSE, digits = 0)
```

Arguments

`...`	The input can be: • A data frame with PK parameters in it; This should ideally be a data frame with one pk parameter per row since it will output a data frame with one PK parameter per row. • PK parameters as either a vector or a scalar
`verbose`	boolean that when TRUE provides a message about the detected pk parameters and the detected compartmental model. By default this is `FALSE`.
`digits`	represents the number of significant digits for the output; If the number is zero or below (default), do not round.

Value

Return a data.frame of derived PK parameters for a 1-, 2-, or 3-compartment linear model given provided clearances and volumes based on the inferred model type.

The model parameters that will be provided in the data frame are:

- `vc`: Central Volume (for 1-, 2- and 3- compartment models)
- `kel`: First-order elimination rate (for 1-, 2-, and 3-compartment models)
- `k12`: First-order rate of transfer from central to first peripheral compartment; (for 2- and 3-compartment models)
- `k21`: First-order rate of transfer from first peripheral to central compartment, (for 2- and 3-compartment models)
- `k13`: First-order rate of transfer from central to second peripheral compartment; (3-compartment model)
- `k31`: First-order rate of transfer from second peripheral to central compartment (3-compartment model)
- `vp`: Peripheral Volume (for 2- and 3- compartment models)
- `vp2`: Peripheral Volume for 3rd compartment (3- compartment model)
- `vss`: Volume of distribution at steady state; (1-, 2-, and 3-compartment models)
- `t12alpha`: $t_{1/2,\alpha}$; (1-, 2-, and 3-compartment models)
- `t12beta`: $t_{1/2,\beta}$; (2- and 3-compartment models)
- `t12gamma`: $t_{1/2,\gamma}$; (3-compartment model)
- `alpha`: α; (1-, 2-, and 3-compartment models)
- `beta`: β; (2- and 3-compartment models)
- `gamma`: β; (3-compartment model)
- `A`: true `A`; (1-, 2-, and 3-compartment models)
- `B`: true `B`; (2- and 3-compartment models)
- `C`: true `C`; (3-compartment model)
- `fracA`: fractional A; (1-, 2-, and 3-compartment models)
- `fracB`: fractional B; (2- and 3-compartment models)
- `fracC`: fractional C; (3-compartment model)

Author(s)

Matthew Fidler and documentation from Justin Wilkins, <justin.wilkins@occams.com>

References

Shafer S. L. `CONVERT.XLS`

Rowland M, Tozer TN. Clinical Pharmacokinetics and Pharmacodynamics: Concepts and Applications (4th). Clipping Williams & Wilkins, Philadelphia, 2010.

Examples

```
## Note that rxode2 parses the names to figure out the best PK parameter

params <- rxDerived(cl = 29.4, v = 23.4, Vp = 114, vp2 = 4614, q = 270, q2 = 73)

## That is why this gives the same results as the value before

params <- rxDerived(CL = 29.4, V1 = 23.4, V2 = 114, V3 = 4614, Q2 = 270, Q3 = 73)

## You may also use micro-constants alpha/beta etc.

params <- rxDerived(k12 = 0.1, k21 = 0.2, k13 = 0.3, k31 = 0.4, kel = 10, v = 10)

## or you can mix vectors and scalars

params <- rxDerived(CL = 29.4, V = 1:3)

## If you want, you can round to a number of significant digits
## with the `digits` argument:

params <- rxDerived(CL = 29.4, V = 1:3, digits = 2)
```

---

rxDfdy *Jacobian and parameter derivatives*

Description

Return Jacobain and parameter derivatives

Usage

```
rxDfdy(obj)
```

Arguments

obj	rxode2 family of objects

Value

A list of the jacobian parameters defined in this rxode2 object.

Author(s)

Matthew L. Fidler

See Also

Other Query model information: rxInits(), rxLhs(), rxModelVars(), rxParams(), rxState()

`rxEtaDistExpand` *Expand declared non-normal random effect distributions into a model*

Description

Rewrites a ui that carries `dist()` declarations into an ordinary ui: the declared random effects become latent standard normals (a fixed identity omega) plus unconstrained correlation thetas, and the model block gains the `phiU()` + inverse CDF lines that recreate them under their original names. Everything downstream – `rxSolve()`, and every nlmixr2est estimation method – then sees a model it already knows how to handle.

Usage

```
rxEtaDistExpand(ui, param = c("cdf", "direct"))
```

Arguments

`ui`	rxode2 ui
`param`	How a declared random effect is represented for estimation. `"cdf"` (default) is the construction described above: a standard normal latent, `phiU()`, the family's inverse CDF, and `rxCor.*` carrying a Gaussian copula. The estimator then sees an ordinary model with a fixed identity omega, and the non-normality lives in a decoder line. `"direct"` leaves the declared random effect ALONE: no latent, no decoder, no `phiU()`. The eta itself is what the estimator samples or optimizes, and it carries the declared family as its prior rather than a variance. Its omega entry is a FIXED placeholder that an estimator on this route must ignore – an estimator that reads it as a Gaussian variance will silently fit the wrong model, so the route is opt-in per estimator. The two are not interchangeable everywhere. For a CORRELATED block of declared random effects the CDF construction is not an alternative, it is the definition: a Gaussian copula with non-normal marginals IS `eta = Q(phi(z))`. Expressing that directly needs an explicit joint distribution (NoLimits.jl reaches for `Copulas.SklarDist`), which this does not have, so `"direct"` refuses a correlated declared block by name rather than dropping the correlation silently.

Details

A ui with no declaration is returned unchanged.

Value

the rewritten rxode2 ui, or `ui` itself when there is nothing to expand

Author(s)

Matthew L. Fidler

Examples

```
if (requireNamespace("lotri", quietly = TRUE) &&
      "lotriEtaDists" %in% getNamespaceExports("lotri")) {
one.cmt <- function() {
  ini({
    lclm <- log(5)
    lclrv <- log(0.09)
    tv <- 3.45
    eta.v ~ 0.1
    dist(eta.cl) ~ dgamma(shape=1/exp(lclrv),
                          rate=1/(exp(lclrv)*exp(lclm)))
    add.sd <- 0.7
  })
  model({
    cl <- eta.cl
    v <- exp(tv + eta.v)
    linCmt() ~ add(add.sd)
  })
}

rxEtaDistExpand(one.cmt())
}
```

`rxEtaDistMuRef` *Mu-reference the parameters of a declared eta distribution*

Description

`rxEtaDistExpand()` writes the declared distribution's parameters into the model as bare thetas inside an inverse-CDF call. Nothing about that shape is `theta + eta`, so every one of them comes out **non**-mu-referenced – which is the case both `saem` and the FOCEi family handle worst, and it is why a cold-started fit of a declared-distribution model tends to settle a long way from the answer.

Usage

```
rxEtaDistMuRef(ui, variance = 0.1)
```

Arguments

`ui`	rxode2 model with at least one `dist()` declaration
`variance`	variance to fix each helper random effect at. `variance = 0` **is the preferred spelling**, and it is NONMEM's own: Bauer's control streams mu-reference every distribution parameter and put each helper on `$OMEGA (0.0 FIXED)`. It declares what is true – the helper carries no between-subject variability – and hands the question of what to do about that to the estimation method, where `nlmixr2est::saemControl(zeroOmegaTune=, zeroOmegaAnneal=, zeroOmegaDirect=)` can act on it.

A NONZERO value writes a sampling width into the model itself and bypasses that machinery entirely. It is what this function did before saem had a direct-maximization M-step for these thetas, and it is kept because it still works: the helper has to MOVE, or the conditional mean saem shifts its theta by is identically zero and the theta never budges.

Wider is not generally better. Measured on Bauer's gamma model (300 subjects, cold start) widening degraded every parameter monotonically: at 0.1 / 1 / 4 the residual SD came out 0.150 / 0.162 / 0.170 against a truth of 0.141, and Q came out 2.29 / 2.48 / 2.60 against 2.13. 0.1 recovered CL 5.60 and V1 4.77 against truths of 5.03 and 4.66. That a constant cannot be right twice – wide enough early to explore, tight enough late to settle – is what `zeroOmegaAnneal=` addresses.

Details

This carries each of those parameters on its own random effect with a small FIXED variance, which is what puts them back into a `theta + eta` form and so back onto the mu-referenced path. It is the same structure NONMEM control streams get from `MU_5 = THETA(5)` with `$OMEGA (0.0 FIXED)`, with one important difference: the helper variance must **not** be ~0 here. nlmixr2's mu-theta M-step is weighted by `omega^-1`, so a ~0 variance pins the parameter at its starting value instead of freeing it (NONMEM updates such a parameter by direct maximization, so the idiom works there).

The result is a **different model** – the helper variance is real between-subject variability on the distribution's parameters – so this is a way to travel, not a way to finish. Use it as the first stage of a chain and refit the model you actually mean from its estimates:

```
stage1 <- nlmixr2(rxEtaDistMuRef(mod), data, est = "saem")
final  <- nlmixr2(mod |> ini(stage1), data, est = "focei",
                  control = foceiControl(mceta = 100))
```

Measured on Bauer's gamma-distributed CL/V1 data (300 subjects), that chain recovers the structural parameters essentially exactly (CL 5.04 against a simulation truth of 5.03, Q 2.15 against 2.13) where a cold start of either method alone does not. Stage two has to be a gradient method: `saem` as the second stage moved the residual error further from the truth than stage one had it.

Value

an rxode2 model, already expanded, whose declared-distribution parameters are mu-referenced

Author(s)

Matthew L. Fidler

Examples

```
if (requireNamespace("lotri", quietly = TRUE) &&
      "lotriEtaDists" %in% getNamespaceExports("lotri")) {
mod <- function() {
  ini({
    lclm <- log(5)
```

```
    lclrv <- log(0.09)
    tv <- 3.45
    dist(eta.cl) ~ dgamma(shape = 1 / exp(lclrv),
                          rate = 1 / (exp(lclrv) * exp(lclm)))
    add.sd <- 0.7
  })
  model({
    cl <- eta.cl
    v <- exp(tv)
    linCmt() ~ add(add.sd)
  })
}
rxEtaDistMuRef(mod)
}
```

`rxEtDispatchSolve` *Dispatch solve to 'rxode2' solve*

Description

Dispatch solve to 'rxode2' solve

Usage

```
rxEtDispatchSolve(x, ...)

## Default S3 method:
rxEtDispatchSolve(x, ...)
```

Arguments

`x`	rxode2 solve dispatch object
`...`	other arguments

Value

if 'rxode2' is loaded, a solved object, otherwise an error

Author(s)

Matthew L. Fidler

rxEventListen *rxode2 event bus*

Description

A minimal publish/subscribe mechanism. Packages that finish a top-level fit, simulation or result call `rxEventEmit()`; listeners registered with `rxEventListen()` are told about it. Events are only delivered from the outermost operation: work done inside `rxEventScope()` (or between `.rxEventEnter()` and `.rxEventExit()`) emits nothing.

Usage

```
rxEventListen(id, fun)

rxEventUnlisten(id)

rxEventListeners()

rxEventDepth()

rxEventSeq()

rxEventScope(expr)

.rxEventEnter()

.rxEventExit(event = NULL, ..., fun = NULL)

.rxEventCall(call, fun = NULL)

rxEventEmit(event, ..., fun = NULL)
```

Arguments

`id`	A listener id; registering the same id again replaces it.
`fun`	For `rxEventListen()`, the listener, a function `function(event, ...)`. For `rxEventEmit()`, `.rxEventExit()` and `.rxEventCall()`, the name of the emitting function, used to normalize a `call` payload field (see Details).
`expr`	Code to evaluate inside an operation scope.
`event`	The event name, e.g. `"fitComplete"` or `"solveComplete"`.
`...`	Payload fields (named).
`call`	A call to normalize.

Details

Listeners are called as `fun(event, ...)` with the event name and the payload fields as named arguments. A listener that errors gives a warning and never stops the other listeners or the caller. Events emitted while a listener runs are dropped.

A `call` payload field is passed through `.rxEventCall()`: its head becomes `fun` (when given; otherwise a function object in the head becomes `` `<fun>` ``). Arguments that are not language (values inlined by `do.call()`) become `` `<value>` ``, except single constants; when there are more than five such values they are all dropped and replaced by one `` `<...>` `` marker. A recorded call therefore never embeds a large object.

Value

`rxEventListen()`, `rxEventUnlisten()` and `rxEventEmit()` return `NULL` invisibly; `rxEventListeners()` returns the listener ids; `rxEventDepth()` the current depth; `rxEventSeq()` the number of events delivered in this process; `rxEventScope()` the value of `expr`.

Author(s)

Matthew L. Fidler

Examples

```
got <- NULL
rxEventListen("example", function(event, ...) got <<- event)
rxEventEmit("myEvent", value = 1)
got
rxEventScope(rxEventEmit("ignored"))
rxEventUnlisten("example")
```

---

rxEventTableFile *Create a file-backed event table reference*

Description

Create a file-backed event table reference

Usage

```
rxEventTableFile(
  path,
  format = c("auto", "parquet", "csv", "fst", "rds"),
  id = "id"
)
```

Arguments

path	path to the file containing the event table
format	file format: "auto" (detect from extension), "parquet", "csv", "fst", or "rds"
id	name of the subject ID column

Value

An rxEtFile object

---

rxEvid *EVID formatting for tibble and other places.*

Description

This is to make an EVID more readable by non pharmacometricians. It displays what each means and allows it to be displayed in a tibble.

Usage

```
rxEvid(x)

as.rxEvid(x)

## S3 method for class 'rxEvid'
c(x, ...)

## S3 method for class 'rxEvid'
x[...]

## S3 method for class 'rxEvid'
as.character(x, ...)

## S3 method for class 'rxEvid'
x[[...]]

## S3 replacement method for class 'rxEvid'
units(x) <- value

## S3 method for class 'rxRateDur'
c(x, ...)

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

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

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

Arguments

x	Item to be converted to a rxode2 EVID specification.
...	Other parameters
value	It will be an error to set units for evid

Value

rxEvid specification

Examples

```
rxEvid(1:7)
```

---

rxexp *Simulate exponential variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the sitmo threefry, this currently generates one random deviate from the uniform distribution to seed the engine threefry and then run the code.

Usage

```
rxexp(rate, n = 1L, ncores = 1L)
```

Arguments

rate	vector of rates.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
ncores	Number of cores for the simulation
	rxnorm simulates using the threefry sitmo generator.
	rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

exponential random deviates

Examples

```
## Use threefry engine

rxexp(0.5, n = 10) # with rxexp you have to explicitly state n
rxexp(5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxexp(1)


## This example uses `rxexp` directly in the model

rx <- function() {
  model({
    a <- rxexp(2)
  })
}

et <- et(1, id = 1:2)


s <- rxSolve(rx, et)
```

`rxf` *Simulate F variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the sitmo threefry, this currently generates one random deviate from the uniform distribution to seed the engine threefry and then run the code.

Usage

```
rxf(df1, df2, n = 1L, ncores = 1L)
```

Arguments

df1, df2	degrees of freedom. Inf is allowed.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
ncores	Number of cores for the simulation rxnorm simulates using the threefry sitmo generator. rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the rxode2 environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with rxSetSeed()

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

f random deviates

Examples

```
## Use threefry engine

rxf(0.5, 0.5, n = 10) # with rxf you have to explicitly state n
rxf(5, 1, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxf(1, 3)


## This example uses `rxf` directly in the model
```

```
rx <- function() {
  model({
    a <- rxf(2, 2)
  })
}

et <- et(1, id = 1:2)


s <- rxSolve(rx, et)
```

---

rxFixPop *Apply the fixed population estimated parameters*

Description

Apply the fixed population estimated parameters

Usage

```
rxFixPop(ui, returnNull = FALSE)
```

Arguments

ui	rxode2 ui function
returnNull	boolean for if unchanged values should return the original ui (FALSE) or null (TRUE)

Value

when returnNull is TRUE, NULL if nothing was changed, or the changed model ui. When returnNull is FALSE, return a ui no matter if it is changed or not.

Author(s)

Matthew L. Fidler

Examples

```
One.comp.transit.allo <- function() {
 ini({
   # Where initial conditions/variables are specified
   lktr <- log(1.15)  #log k transit (/h)
   lcl  <- log(0.15)  #log Cl (L/hr)
   lv   <- log(7)     #log V (L)
   ALLC <- fix(0.75)  #allometric exponent cl
```

```
    ALLV <- fix(1.00)  #allometric exponent v
    prop.err <- 0.15   #proportional error (SD/mean)
    add.err <- 0.6     #additive error (mg/L)
    eta.ktr ~ 0.5
    eta.cl ~ 0.1
    eta.v ~ 0.1
  })
  model({
    #Allometric scaling on weight
    cl <- exp(lcl + eta.cl + ALLC * logWT70)
    v  <- exp(lv + eta.v + ALLV * logWT70)
    ktr <- exp(lktr + eta.ktr)
    # RxODE-style differential equations are supported
    d/dt(depot)   = -ktr * depot
    d/dt(central) =  ktr * trans - (cl/v) * central
    d/dt(trans)   =  ktr * depot - ktr * trans
    ## Concentration is calculated
    cp = central/v
    # And is assumed to follow proportional and additive error
    cp ~ prop(prop.err) + add(add.err)
  })
}

m <- rxFixPop(One.comp.transit.allo)
m

# now everything is already fixed, so calling again will do nothing

rxFixPop(m)

# if you call it with returnNull=TRUE when no changes have been
# performed, the function will return NULL

rxFixPop(m, returnNull=TRUE)
```

`rxFixRes` *Literally fix residual parameters*

Description

Literally fix residual parameters

Usage

```
rxFixRes(ui, returnNull = FALSE)
```

Arguments

ui	rxode2 ui function
returnNull	boolean for if unchanged values should return the original ui (FALSE) or null (TRUE)

Value

model with residual parameters literally fixed in the model

Author(s)

Matthew L. Fidler

Examples

```
One.comp.transit.allo <- function() {
 ini({
   # Where initial conditions/variables are specified
   lktr <- log(1.15)  #log k transit (/h)
   lcl  <- log(0.15)  #log Cl (L/hr)
   lv   <- log(7)     #log V (L)
   ALLC <- 0.75  #allometric exponent cl
   ALLV <- 1.00  #allometric exponent v
   prop.err <- fix(0.15)   #proportional error (SD/mean)
   add.err <- fix(0.6)     #additive error (mg/L)
   eta.ktr ~ 0.5
   eta.cl ~ 0.1
   eta.v ~ 0.1
 })
 model({
   #Allometric scaling on weight
   cl <- exp(lcl + eta.cl + ALLC * logWT70)
   v  <- exp(lv + eta.v + ALLV * logWT70)
   ktr <- exp(lktr + eta.ktr)
   # RxODE-style differential equations are supported
   d/dt(depot)   = -ktr * depot
   d/dt(central) =  ktr * trans - (cl/v) * central
   d/dt(trans)   =  ktr * depot - ktr * trans
   ## Concentration is calculated
   cp = central/v
   # And is assumed to follow proportional and additive error
   cp ~ prop(prop.err) + add(add.err)
 })
}

m <- rxFixRes(One.comp.transit.allo)
```

`rxForcedPars` *Forced parameters carried by a model / ui*

Description

Get or set a named-numeric vector of *forced* parameters stored in a hidden slot of an rxode2 ui. Forced parameters override the values supplied in `params`/data and the model initial estimates on **every** solve of that ui (they are written into every subject/simulation column at solve setup, the same injection point used by par-loader hooks). Because the values live on the ui they travel with it – through model piping and into any `nlmixr2` fit built from it – so the model stays self-contained and portable (e.g. a fit that carries trained neural-network weights re-solves, predicts and simulates with those weights with no external state).

Usage

```
rxForcedPars(ui)

rxForcedPars(ui) <- value
```

Arguments

`ui`	an rxode2 ui / model function.
`value`	named numeric vector of forced parameter values, or `NULL`.

Details

The value is stored directly in the ui environment (not in the printed `meta` block, so it stays hidden) and registered as a `sticky` model item so it survives model piping. Names must be model parameters; names that are not model parameters are ignored at solve time. Set to `NULL` (or an empty vector) to clear.

Value

the getter returns the named numeric vector (or `NULL`); the setter returns the (modified) ui.

Author(s)

Matthew L. Fidler

rxFun *Add/Create C functions for use in rxode2*

Description

Add/Create C functions for use in rxode2

Usage

```
rxFun(name, args, cCode)

rxRmFun(name)
```

Arguments

name	This can either give the name of the user function or be a simple R function that you wish to convert to C. If you have rxode2 convert the R function to C, the name of the function will match the function name provided and the number of arguments will match the R function provided. Hence, if you are providing an R function for conversion to C, the rest of the arguments are implied.
args	This gives the arguments of the user function
cCode	This is the C-code for the new function

Examples

```
# Right now rxode2 is not aware of the function fun
# Therefore it cannot translate it to symengine or
# Compile a model with it.

try(rxode2("a=fun(a,b,c)"))

# Note for this approach to work, it cannot interfere with C
# function names or reserved rxode2 special terms.  Therefore
# f(x) would not work since f is an alias for bioavailability.

fun <- "
double fun(double a, double b, double c) {
  return a*a+b*a+c;
}
" # C-code for function

rxFun("fun", c("a", "b", "c"), fun) ## Added function

# Now rxode2 knows how to translate this function to symengine

rxToSE("fun(a,b,c)")

# And will take a central difference when calculating derivatives
```

```
rxFromSE("Derivative(fun(a,b,c),a)")

## Of course, you could specify the derivative table manually
rxD("fun", list(
  function(a, b, c) {
    paste0("2*", a, "+", b)
  },
  function(a, b, c) {
    return(a)
  },
  function(a, b, c) {
    return("0.0")
  }
))

rxFromSE("Derivative(fun(a,b,c),a)")

# You can also remove the functions by `rxRmFun`

rxRmFun("fun")

# you can also use R functions directly in rxode2


gg <- function(x, y) {
  x + y
}

f <- rxode2({
 z = gg(x, y)
})


e <- et(1:10) |> as.data.frame()

e$x <- 1:10
e$y <- 21:30

rxSolve(f, e)

# Note that since it touches R, it can only run single-threaded.
# There are also requirements for the function:
#
# 1. It accepts one value per argument (numeric)
#
# 2. It returns one numeric value

# If it is a simple function (like gg) you can also convert it to C
# using rxFun and load it into rxode2

rxFun(gg)

rxSolve(f, e)
```

```
# to stop the recompile simply reassign the function
f <- rxode2(f)

rxSolve(f, e)

rxRmFun("gg")
rm(gg)
rm(f)


# You can also automatically convert a R function to R code (and
# calculate first derivatives)

fun <- function(a, b, c) {
  a^2+b*a+c
}

rxFun(fun)

# You can see the R code if you want with rxC

message(rxC("fun"))

# you can also remove both the function and the
# derivatives with rxRmFun("fun")

rxRmFun("fun")
```

`rxgamma` *Simulate gamma variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxgamma(shape, rate = 1, n = 1L, ncores = 1L)
```

Arguments

`shape`	The shape of the gamma random variable
`rate`	an alternative way to specify the scale.

n — number of observations. If `length(n) > 1`, the length is taken to be the number required.

ncores — Number of cores for the simulation

`rxnorm` simulates using the threefry sitmo generator.

`rxnormV` used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of `rxnorm`. It is no longer supported in `rxode2({})` blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

gamma random deviates

Examples

```
## Use threefry engine

rxgamma(0.5, n = 10) # with rxgamma you have to explicitly state n
rxgamma(5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxgamma(1)


## This example uses `rxbeta` directly in the model

rx <- function() {
  model({
    a <- rxgamma(2)
  })
}

et <- et(1, id = 1:2)


s <- rxSolve(rx, et)
```

rxgeom *Simulate geometric variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the sitmo threefry, this currently generates one random deviate from the uniform distribution to seed the engine threefry and then run the code.

Usage

```
rxgeom(prob, n = 1L, ncores = 1L)
```

Arguments

prob	probability of success in each trial. 0 < prob <= 1.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
ncores	Number of cores for the simulation rxnorm simulates using the threefry sitmo generator. rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the rxode2 environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with rxSetSeed()

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

geometric random deviates

Examples

```
## Use threefry engine

rxgeom(0.5, n = 10) # with rxgeom you have to explicitly state n
```

```
rxgeom(0.25, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxgeom(0.75)


## This example uses `rxgeom` directly in the model

rx <- function() {
  model({
    a <- rxgeom(0.24)
  })
}

et <- et(1, id = 1:2)


s <- rxSolve(rx, et)
```

`rxGetControl` *rxGetControl option from ui*

Description

rxGetControl option from ui

Usage

```
rxGetControl(ui, option, default)
```

Arguments

`ui`	rxode2 ui object
`option`	Option to get
`default`	Default value

Value

Option (if present) or default value

Author(s)

Matthew L. Fidler

rxGetDefaultSerialize *Get the Default Serialization Type*

Description

Get the Default Serialization Type

Usage

```
rxGetDefaultSerialize()
```

Value

string indicating the default serialization type

Author(s)

Matthew L. Fidler

Examples

```
rxGetDefaultSerialize()
```

rxGetLin *Get the linear compartment model true function*

Description

Get the linear compartment model true function

Usage

```
rxGetLin(model, linCmtSens = c("linCmtA", "linCmtB"), verbose = FALSE)
```

Arguments

model	This is the ODE model specification. It can be: • a string containing the set of ordinary differential equations (ODE) and other expressions defining the changes in the dynamic system. • a file name where the ODE system equation is contained An ODE expression enclosed in \{\} (see also the filename argument). For details, see the sections "Details" and rxode2 Syntax below.
linCmtSens	The method to calculate the linCmt() solutions
verbose	When TRUE be verbose with the linear compartmental model

Value

model with linCmt() replaced with linCmtA()

Author(s)

Matthew Fidler

---

rxGetrxode2 *Get rxode2 model from object*

Description

Get rxode2 model from object

Usage

```
rxGetrxode2(obj)
```

Arguments

obj	rxode2 family of objects

Value

rxode2 model

---

rxGetSeed *Get the rxode2 seed*

Description

Get the rxode2 seed

Usage

```
rxGetSeed()
```

Value

rxode2 seed state or -1 when the seed isn't set

See Also

rxSetSeed, rxWithSeed, rxWithPreserveSeed

Examples

```
# without setting seed

rxGetSeed()
# Now set the seed
rxSetSeed(42)

rxGetSeed()

rxnorm()

rxGetSeed()

# don't use the rxode2 seed again

rxSetSeed(-1)

rxGetSeed()

rxnorm()

rxGetSeed()
```

---

rxHasAr *Test if a model uses an autoregressive (*ar()*) residual*

Description

Test if a model uses an autoregressive (ar()) residual

Usage

```
rxHasAr(ui)
```

Arguments

ui	rxode2 user interface model

Value

logical, TRUE if any endpoint carries an ar() correlation (either an estimated correlation or a modeled/literal one)

Author(s)

Matthew L. Fidler

rxHtml *Format rxSolve and related objects as html.*

Description

Format rxSolve and related objects as html.

Usage

```
rxHtml(x, ...)

## S3 method for class 'rxSolve'
rxHtml(x, ...)
```

Arguments

x	rxode2 object
...	Extra arguments sent to kable

Value

html code for rxSolve object

Author(s)

Matthew L. Fidler

rxIndLinState *Set the preferred factoring by state*

Description

Set the preferred factoring by state

Usage

```
rxIndLinState(preferred = NULL)
```

Arguments

preferred	A list of each state's preferred factorization

Details

Like `rxIndLinStrategy()`, this no longer affects the conversion: a multi-state product cannot yield a state-free rate constant whichever state is factored out, so it becomes an `indLin()` forcing instead. The setting is kept so existing code keeps working.

Value

Nothing

Author(s)

Matthew Fidler

`rxIndLinStrategy` *This sets the inductive linearization strategy for matrix building*

Description

When there is more than one state in a ODE that cannot be separated this specifies how it is incorporated into the matrix exponential.

Usage

```
rxIndLinStrategy(strategy = c("curState", "split"))
```

Arguments

`strategy`	The strategy for inductive linearization matrix building • `curState` Prefer parameterizing in terms of the current state, followed by the first state observed in the term. • `split` Split the parameterization between all states in the term by dividing each by the number of states in the term and then adding a matrix term for each state.

Details

This no longer affects the conversion. The `matExp()` rate matrix has to be constant in the states, and dividing any one state out of a multi-state product always leaves a coefficient that still reads a state, so such a term goes to the `indLin()` forcing whichever state is preferred. The setting is kept so existing code keeps working.

Value

Nothing

Author(s)

Matthew L. Fidler

rxInjectedPars *Parameters injected into a solve by a par-loader hook*

Description

Returns the parameters (e.g. trained neural-network weights) that a registered par-loader wrote into the solve parameter vector, saved on the solved object so re-solving from it restores them.

Usage

```
rxInjectedPars(obj)
```

Arguments

obj	a solved rxode2 object.

Value

a named numeric vector, or NULL if nothing was injected.

rxIntToBase *Convert a positive base*

Description

Convert a positive base

Usage

```
rxIntToBase(x, base = 36L)
```

Arguments

x	integer to convert
base	can be 2 to 36

Value

a sequence of letters and representing the number(s) input

Author(s)

Matthew L. Fidler

Examples

```
rxIntToBase(1:100)
```

rxIntToLetter *Convert a positive integer to a letter series*

Description

Convert a positive integer to a letter series

Usage

```
rxIntToLetter(x, base = 26L)
```

Arguments

x	integer to convert
base	can be 2 to 26

Value

a sequence of letters representing the number(s) input

Author(s)

Matthew L. Fidler

Examples

```
rxIntToLetter(1:100)
```

rxInv *Invert matrix using RcppArmadillo.*

Description

Invert matrix using RcppArmadillo.

Usage

```
rxInv(matrix)
```

Arguments

matrix	matrix to be inverted.

Value

inverse or pseudo inverse of matrix.

rxIsAutoSwitch *Check whether an ODE method string is an AutoSwitch composite pair*

Description

Returns TRUE for strings containing "+", such as "dop853+ros43".

Usage

```
rxIsAutoSwitch(method)
```

Arguments

method	Character vector of method names.

Value

Logical vector the same length as method.

See Also

odeMethodToInt(), rxIsNonStiff(), rxIsStiff()

Examples

```
rxIsAutoSwitch("dop853+ros43")  # TRUE
rxIsAutoSwitch("dop853")        # FALSE
rxIsAutoSwitch(c("dop853+ros43", "liblsoda"))  # TRUE FALSE
```

rxIsCurrent *Checks if the rxode2 object was built with the current build*

Description

Checks if the rxode2 object was built with the current build

Usage

```
rxIsCurrent(obj)
```

Arguments

obj	rxode2 family of objects

Value

boolean indicating if this was built with current rxode2

`rxIsDense` *Check whether an ODE solving method supports dense output*

Description

Dense output (continuous interpolation) allows the solver to reconstruct the solution at any point within a completed step without extra function evaluations. This lets the solver take large internal steps that span many requested output times, then interpolate cheaply – improving both speed and accuracy on densely sampled grids.

Usage

```
rxIsDense(method)
```

Arguments

`method`	A character vector of method names or an integerish vector of method codes (as returned by `odeMethodToInt()`). Vectorised.

Details

Dense-capable single methods are: `"dop853"` (0), `"dop5"` (10), `"bs"` (11), `"ros4"` (13).

For composite AutoSwitch methods (e.g. `"dop5+ros4"`), `TRUE` is returned only when **both** the primary and stiff secondary are dense-capable. `"ros4"` is the only stiff method with dense support, so the valid dense composites are `"dop853+ros4"`, `"dop5+ros4"`, and `"bs+ros4"`.

Value

A logical vector the same length as `method`. `TRUE` if the corresponding method supports dense output.

See Also

`rxIsStiff()`, `rxIsNonStiff()`, `odeMethodToInt()`

Examples

```
rxIsDense("dop853")                    # TRUE
rxIsDense("ros4")                      # TRUE
rxIsDense("liblsoda")                  # FALSE
rxIsDense(c("dop5", "bs", "cvode"))    # TRUE TRUE FALSE
rxIsDense("dop5+ros4")                 # TRUE  (both dense)
rxIsDense("dop5+ros43")                # FALSE (ros43 not dense)
```

`rxIsImplicit` *Check whether an ODE solving method requires a Jacobian (is implicit)*

Description

Implicit ODE solvers linearise the system at each step and therefore require the Jacobian df/dy. rxode2 auto-computes a symbolic Jacobian when one of these methods is requested and falls back to `"liblsoda"` if symbolic differentiation fails. Use this function to test whether a method string or integer code corresponds to an implicit solver.

Usage

```
rxIsImplicit(method)
```

Arguments

`method`	A character vector of method names or an integerish vector of method codes (as returned by `odeMethodToInt()`). Vectorised.

Details

Implicit methods are: `"ros4"` (13), `"iem"` (14), `"ros43"` (31), `"ros6"` (32), `"backwardEuler"` (33), `"gauss6"` (34), `"iiic6"` (35), `"radauiia5"` (36), `"geng5"` (37), `"sdirk43"` (38).

Value

A logical vector the same length as `method`. `TRUE` if the corresponding method is implicit and requires a Jacobian.

See Also

`odeMethodToInt()`

Examples

```
rxIsImplicit("ros4")              # TRUE
rxIsImplicit("liblsoda")          # FALSE
rxIsImplicit(c("ros43", "dop853", "iem"))  # TRUE FALSE TRUE
rxIsImplicit(13L)                 # TRUE  (ros4)
rxIsImplicit(0L)                  # FALSE (dop853)
```

`rxIsNonStiff` *Check whether an ODE solving method is a purely non-stiff solver*

Description

Returns `TRUE` for methods that are designed exclusively for non-stiff systems. Solvers like `"lsoda"` and `"liblsoda"` that automatically switch between stiff and non-stiff algorithms return `FALSE`, as do all stiff-only methods.

Usage

```
rxIsNonStiff(method)
```

Arguments

`method`	A character vector of method names or an integerish vector of method codes (as returned by `odeMethodToInt()`). Vectorised.

Details

Switchers (`"lsoda"` = 1, `"liblsoda"` = 2) and the inductive linearisation solver (`"indLin"` = 3) are excluded from both `rxIsStiff()` and `rxIsNonStiff()`.

Value

A logical vector the same length as `method`. `TRUE` if the corresponding method is a purely non-stiff solver.

See Also

`rxIsStiff()`, `rxIsImplicit()`, `odeMethodToInt()`

Examples

```
rxIsNonStiff("dop853")                      # TRUE
rxIsNonStiff("lsoda")                       # FALSE (switches)
rxIsNonStiff("bdf")                         # FALSE (stiff-only)
rxIsNonStiff(c("lsode", "cvode", "bs32"))   # TRUE FALSE TRUE
```

rxIsStiff *Check whether an ODE solving method is a purely stiff solver*

Description

Returns TRUE for methods that are designed exclusively for stiff systems and will never switch to a non-stiff algorithm. Solvers like "lsoda" and "liblsoda" that automatically switch between stiff and non-stiff algorithms return FALSE.

Usage

```
rxIsStiff(method)
```

Arguments

method	A character vector of method names or an integerish vector of method codes (as returned by odeMethodToInt()). Vectorised.

Details

Stiff-only methods are: "ros4" (13), "iem" (14), "cvode" (21), "ros43" (31), "ros6" (32), "backwardEuler" (33), "gauss6" (34), "iiic6" (35), "radauiia5" (36), "geng5" (37), "sdirk43" (38), "bdf" (107).

Value

A logical vector the same length as method. TRUE if the corresponding method is a purely stiff solver.

See Also

rxIsNonStiff(), rxIsImplicit(), odeMethodToInt()

Examples

```
rxIsStiff("bdf")                          # TRUE
rxIsStiff("lsoda")                        # FALSE (switches)
rxIsStiff("liblsoda")                     # FALSE (switches)
rxIsStiff(c("cvode", "dop853", "ros43"))  # TRUE FALSE TRUE
```

`rxLastCompile` *Get the last compiled model information as alist*

Description

Get the last compiled model information as alist

Usage

```
rxLastCompile(what = c("msg", "stderr", "stdout", "c"))
```

Arguments

`what`	Which elements to message to the console; by default all of them. Use `what="stderr"` for the compiler error alone, or `what=character(0)` to message nothing and only take the returned list.

Value

A list contains the following elements:

- `msg` the message for a bad compilation, or NULL if successful.
- `stdout` the standard output from the compilation
- `stderr` the standard error from the compilation
- `c` the code code that was used

This list will be returned invisibly, but the function will also message the contents to the console.

Author(s)

Matthew L. Fidler

Examples

```
rxode2({
  a <- b
})
rxLastCompile()
```

rxLhs *Left handed Variables*

Description

This returns the model calculated variables

Usage

```
rxLhs(obj)
```

Arguments

obj	rxode2 family of objects

Value

a character vector listing the calculated parameters

Author(s)

Matthew L.Fidler

See Also

rxode2

Other Query model information: rxDfdy(), rxInits(), rxModelVars(), rxParams(), rxState()

rxLock *Lock/unlocking of rxode2 dll file*

Description

Lock/unlocking of rxode2 dll file

Usage

```
rxLock(obj)

rxUnlock(obj)
```

Arguments

obj	A rxode2 family of objects

Value

nothing; called for side effects

`rxMemoryEstimate` *Estimate memory required by rxSolve() for a given dataset and model*

Description

Accepts either a pre-summarized per-ID table (an `rxMemSummary` or any data.frame with `nobs` and `ndoses` columns) or a full event-table data.frame with an `evid` column. Model dimensions can be supplied via a compiled rxode2 model object or overridden individually.

Usage

```
rxMemoryEstimate(
  dat,
  model = NULL,
  control = NULL,
  neq = 1L,
  stateSize = neq,
  nlhs = 0L,
  npars = neq,
  neta = 0L,
  neps = 0L,
  ncov = 0L,
  nsim = 1L,
  cores = 1L,
  nMtime = 0L,
  extraCmt = 0L,
  linB = FALSE,
  nLlik = 0L,
  nIndSim = NULL,
  numLinSens = 0L,
  numLin = 0L,
  stiff = NA_integer_,
  doIndLin = 0L,
  indOwnAlloc = 0L,
  nDelayState = 0L
)
```

Arguments

`dat`	A `rxMemSummary`, a data.frame with `nobs`/`ndoses` columns, or a full event-table data.frame with an `evid` column. This can also be a serialized solve state file path, a solve state bundle list containing an `events` element, or an `rxSolve` object (using its stored solve events).
`model`	Optional rxode2 model object. When supplied, `neq`, `nlhs`, `npars`, `extraCmt`, `linB`, `nMtime`, `nLlik`, and `nIndSim` are extracted automatically.

control	Optional rxControl object. When supplied, cores, nsim, neta (from omega), neps (from sigma), and nLlik (adjusted by nLlikAlloc) are overridden automatically.
neq	Number of ODE states.
stateSize	Effective state.size() seen by the solver. Equals neq for pure ODE models; may differ for linCmt-only models. Defaults to neq.
nlhs	Number of LHS (calculated) output variables.
npars	Number of model parameters (drives gpars size).
neta	Number of random effects (etas).
neps	Number of residual-error levels (epsilons).
ncov	Number of time-varying covariates.
nsim	Number of simulations. nStud from control is the replicate count, so it lands here rather than in the subject count.
cores	Number of parallel OMP threads.
nMtime	Number of model measurement times.
extraCmt	Extra compartments (0, 1 = depot, 2 = depot+central).
linB	TRUE/1 if using a linear-compartment model.
nLlik	Number of log-likelihood terms (FOCEi use).
nIndSim	Per-individual simulation count. Defaults to neta + neps when not supplied explicitly.
numLinSens	Number of linear sensitivity parameters (FOCEi + linCmt).
numLin	Number of linear compartment terms (FOCEi + linCmt).
stiff	Solving method as an integer (odeMethodToInt); only 3 ("indLin") allocates anything extra. Taken from control when given, and overridden to 3 for any matExp() model, since rxSolve() force-selects that method for one whatever the control says.
doIndLin	Which matrix-exponential driver runs: 0 not a matExp() model, 1 pure matrix exponential, 2 plus a state-free indLin() forcing, 3/4 true inductive linearization. Taken from model when given. These cost very different amounts, so it is worth getting right.
indOwnAlloc	TRUE/1 when every individual gets its own event and solve arrays rather than pointing into the global buffers. These are allocated on top of gsolve, so they are real extra memory. Taken from the model's evid_ flag when model is given, and overridden by control$indOwnAlloc when that is set.
nDelayState	Number of ODE states delay() looks back on. Drives the per-individual dense-history buffer, which grows by doubling inside the solve, so it is charged as a documented bound rather than an exact mirror. Taken from model when given.

Details

The byte counts are computed by rxMemoryComponents_() which calls the same rxFillMemLayout() used by the real allocator, so any change to the allocation formulas propagates here automatically.

Value

A named list of class "rxMemoryEstimate" whose elements are raw byte counts plus outputData, ramBytes, freeRamBytes, total, sizeofInd, and rxLlikSaveSize. total is the number of bytes actually allocated, so it excludes gsolve_n0 (the ODE state output matrix), which is reported separately but is already part of gsolve.

Examples

```
mod <- rxode2::rxode2({
  d/dt(depot)  <- -ka * depot
  d/dt(center) <- ka * depot - cl / v * center
  cp           <- center / v
})

ev <- rxode2::et(amt = 100, ii = 24, until = 168) |>
  rxode2::et(seq(0, 168, by = 1))

# Basic estimate from event table and model
rxMemoryEstimate(as.data.frame(ev), model = mod)

# With rxControl: population simulation with omega and 4 cores
ctrl <- rxode2::rxControl(
  cores = 4L,
  omega = lotri::lotri(eta.ka ~ 0.09, eta.cl ~ 0.04)
)
rxMemoryEstimate(as.data.frame(ev), model = mod, control = ctrl)
```

---

rxMemSummary *Create a per-ID event summary for memory estimation*

Description

Create a per-ID event summary for memory estimation

Usage

```
rxMemSummary(nobs, ndoses, id = seq_along(nobs))
```

Arguments

nobs	Integer vector of observation counts per ID.
ndoses	Integer vector of dose event counts per ID.
id	Integer or character vector of subject IDs (optional).

Value

A data.frame with class "rxMemSummary".

Examples

```
# Three subjects with known observation and dose counts
rxMemSummary(nobs = c(48L, 96L, 48L), ndoses = c(7L, 14L, 7L))

# Explicit subject IDs
rxMemSummary(nobs = c(10L, 20L), ndoses = c(5L, 5L), id = c(101L, 102L))
```

---

rxModelName *Name a model from the function that created it*

Description

When a model comes from a call, like rxode2(nlmixr2lib::readModelDb("PK_1cmt")), the text of the call is a poor model name; the function that produced the model knows a better one. This is dispatched on the name of the called function, so a rxModelName.readModelDb() method names every model readModelDb() produces. Without a method the call is named by its own (deparsed) text.

Usage

```
rxModelName(x, ...)

## Default S3 method:
rxModelName(x, ...)
```

Arguments

x	the call that creates the model. Its class is the name of the called function (without any pkg:: qualifier) followed by "rxModelNameCall".
...	the arguments of that call, unevaluated.

Details

A method is given the call as x and the call's arguments in ..., matched to the argument names of the function being called whenever that function can be found. The arguments are unevaluated: one a method does not look at is never evaluated, and one it does look at is evaluated again, in the frame the call came from, so a method should only read arguments that are cheap to evaluate. A method that cannot name the model should return NULL; anything that is not a single non-empty string is ignored the same way.

Value

single character string naming the model, or NULL when the model cannot be named this way

Author(s)

Matthew L. Fidler

See Also

Other Model names: `rxModelNameFromExpr()`, `rxModelNameLhs()`

Examples

```
# a function that creates models from a database of them:
readMyModelDb <- function(name) {
  # ...look `name` up and return the model function...
  function() {
    ini({a <- 1})
    model({b <- a})
  }
}

# names the model for the database entry it came from
rxModelName.readMyModelDb <- function(x, ...) {
  list(...)$name
}

registerS3method("rxModelName", "readMyModelDb", rxModelName.readMyModelDb)

rxode2(readMyModelDb("one.cmt"))$modelName
```

---

rxModelNameLhs *Name a model from the left hand side of an assignment*

Description

Registers the name an assignment operator like `nlmixr2save`'s `:=` is assigning to, which rxode2 uses when the model expression itself names nothing (an anonymous model function) and when the expression is a call without a `rxModelName()` method. It is not consumed when it is used, so one assignment can name every model it builds; the operator that set it clears it with `rxModelNameLhs(NULL)`, which is what keeps the name from leaking into an unrelated later model.

Usage

```
rxModelNameLhs(value)
```

Arguments

`value`	when specified, a single non-empty character naming the model, or `NULL` to clear the name. When missing the current name is returned.

Value

the registered name, or NULL when there is none

Author(s)

Matthew L. Fidler

See Also

Other Model names: rxModelName(), rxModelNameFromExpr()

Examples

```
# an assignment operator registers the name it is assigning to before it
# forces the model, and clears it afterward:
#
# `:=` <- function(x, value) {
#   rxode2::rxModelNameLhs(as.character(substitute(x)))
#   on.exit(rxode2::rxModelNameLhs(NULL))
#   assign(as.character(substitute(x)), force(value), envir=parent.frame())
# }

rxModelNameLhs()

rxModelNameLhs("mod")

rxModelNameLhs()

rxModelNameLhs(NULL)
```

---

rxnbinom *Simulate Binomial variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the sitmo threefry, this currently generates one random deviate from the uniform distribution to seed the engine threefry and then run the code.

Usage

```
rxnbinom(size, prob, mu, n = 1L, ncores = 1L)

rxnbinomMu(size, mu, n = 1L, ncores = 1L)
```

Arguments

size	target for number of successful trials, or dispersion parameter (the shape parameter of the gamma mixing distribution). Must be strictly positive, need not be integer.
prob	probability of success in each trial. 0 < prob <= 1.
mu	alternative parametrization via mean: see 'Details'.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
ncores	Number of cores for the simulation rxnorm simulates using the threefry sitmo generator. rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the rxode2 environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with rxSetSeed()

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

negative binomial random deviates. Note that rxbinom2 uses the mu parameterization an the rxbinom uses the prob parameterization (mu=size/(prob+size))

Examples

```
## Use threefry engine

rxnbinom(10, 0.9, n = 10) # with rxbinom you have to explicitly state n
rxnbinom(3, 0.5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxnbinom(4, 0.7)

# use mu parameter
rxnbinomMu(40, 40, n=10)

## This example uses `rxbinom` directly in the model

rx <- function() {
  model({
    a <- rxnbinom(10, 0.5)
  })
```

```
}

et <- et(1, id = 1:100)


s <- rxSolve(rx, et)


rx <- function() {
  model({
    a <- rxnbinomMu(10, 40)
  })
}


s <- rxSolve(rx, et)
```

`rxNorm` *Get the normalized model*

Description

This get the syntax preferred model for processing

Usage

```
rxNorm(obj, condition = NULL, removeInis, removeJac, removeSens)
```

Arguments

`obj`	rxode2 family of objects
`condition`	Character string of a logical condition to use for subsetting the normalized model. When missing, and a condition is not set via `rxCondition`, return the whole code with all the conditional settings intact. When a condition is set with `rxCondition`, use that condition.
`removeInis`	A boolean indicating if parameter initialization will be removed from the model
`removeJac`	A boolean indicating if the Jacobians will be removed.
`removeSens`	A boolean indicating if the sensitivities will be removed.

Value

Normalized Normal syntax (no comments)

Author(s)

Matthew L. Fidler

rxnormV *Simulate random normal variable from threefry generator*

Description

Simulate random normal variable from threefry generator

Usage

```
rxnormV(mean = 0, sd = 1, n = 1L, ncores = 1L)

rxnorm(mean = 0, sd = 1, n = 1L, ncores = 1L)
```

Arguments

mean	vector of means.
sd	vector of standard deviations.
n	number of observations
ncores	Number of cores for the simulation
	rxnorm simulates using the threefry sitmo generator.
	rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Value

normal random number deviates

Examples

```
## Use threefry engine

rxnorm(n = 10) # with rxnorm you have to explicitly state n
rxnorm(n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxnorm(2, 3) ## The first 2 arguments are the mean and standard deviation


## This example uses `rxnorm` directly in the model

rx <- function() {
  model({
    a <- rxnorm()
  })
}

et <- et(1, id = 1:2)
```

```
s <- rxSolve(rx, et)
```

`rxNumLoaded` *Get the number of loaded rxode2 DLLs*

Description

Get the number of loaded rxode2 DLLs

Usage

```
rxNumLoaded()
```

Value

Number of loaded rxode2 DLLs

Author(s)

Matthew L. Fidler

Examples

```
rxNumLoaded()
```

`rxode2` *Create an ODE-based model specification*

Description

Create a dynamic ODE-based model object suitably for translation into fast C code

Usage

```
rxode2(
  model,
  modName = basename(wd),
  wd = getwd(),
  filename = NULL,
  extraC = NULL,
  debug = FALSE,
  calcJac = NULL,
  calcSens = NULL,
  calcSens2 = NULL,
```

```
  calcSens3 = NULL,
  collapseModel = FALSE,
  package = NULL,
  ...,
  linCmtSens = c("linCmtA", "linCmtB"),
  indLin = FALSE,
  eventSens = NULL,
  verbose = FALSE,
  fullPrint = getOption("rxode2.fullPrint", FALSE),
  envir = parent.frame()
)

RxODE(
  model,
  modName = basename(wd),
  wd = getwd(),
  filename = NULL,
  extraC = NULL,
  debug = FALSE,
  calcJac = NULL,
  calcSens = NULL,
  calcSens2 = NULL,
  calcSens3 = NULL,
  collapseModel = FALSE,
  package = NULL,
  ...,
  linCmtSens = c("linCmtA", "linCmtB"),
  indLin = FALSE,
  eventSens = NULL,
  verbose = FALSE,
  fullPrint = getOption("rxode2.fullPrint", FALSE),
  envir = parent.frame()
)

rxode(
  model,
  modName = basename(wd),
  wd = getwd(),
  filename = NULL,
  extraC = NULL,
  debug = FALSE,
  calcJac = NULL,
  calcSens = NULL,
  calcSens2 = NULL,
  calcSens3 = NULL,
  collapseModel = FALSE,
  package = NULL,
  ...,
```

```
  linCmtSens = c("linCmtA", "linCmtB"),
  indLin = FALSE,
  eventSens = NULL,
  verbose = FALSE,
  fullPrint = getOption("rxode2.fullPrint", FALSE),
  envir = parent.frame()
)
```

Arguments

model	This is the ODE model specification. It can be: • a string containing the set of ordinary differential equations (ODE) and other expressions defining the changes in the dynamic system. • a file name where the ODE system equation is contained An ODE expression enclosed in \{\} (see also the filename argument). For details, see the sections "Details" and rxode2 Syntax below.
modName	a string to be used as the model name. This string is used for naming various aspects of the computations, including generating C symbol names, dynamic libraries, etc. Therefore, it is necessary that modName consists of simple ASCII alphanumeric characters starting with a letter.
wd	character string with a working directory where to create a subdirectory according to modName. When specified, a subdirectory named after the "modName.d" will be created and populated with a C file, a dynamic loading library, plus various other working files. If missing, the files are created (and removed) in the temporary directory, and the rxode2 DLL for the model is created in the current directory named rx_????_platform, for example rx_129f8f97fb94a87ca49ca8dafe691e1e_i386.dll
filename	A file name or connection object where the ODE-based model specification resides. Only one of model or filename may be specified.
extraC	Extra c code to include in the model. This can be useful to specify functions in the model. These C functions should usually take double precision arguments, and return double precision values.
debug	is a boolean indicating if the executable should be compiled with verbose debugging information turned on.
calcJac	boolean indicating if rxode2 will calculate the Jacobain according to the specified ODEs.
calcSens	boolean indicating if rxode2 will calculate the sensitivities according to the specified ODEs. May also be a character vector of the states/parameters whose first-order sensitivities (rx__sens_<state>_BY_<param>__) should be generated.
calcSens2	character vector (or NULL) requesting second-order sensitivities in addition to the first-order ones from calcSens. When supplied, rxode2 also generates the rx__sens_<state>_BY_<p>_BY_<q>__ compartments (the Hessian path), where p ranges over calcSens and q over calcSens2. Used, for example, for population (THETA) second-order event sensitivities. NULL (the default) skips the second-order generation.

calcSens3	character vector (or NULL) requesting third-order sensitivities in addition to the first- and second-order ones. Requires calcSens2 to also be supplied (every calcSens3 parameter needs its own already-built second-order sensitivity compartment, from the pairing of calcSens2 with calcSens3, to reference – so calcSens3 should be a subset of calcSens2, which itself should be a subset of calcSens, mirroring how calcSens2 is used everywhere else in rxode2 today). When supplied, generates the rx__sens_<state>_BY_<p>_BY_<q>_BY_<r>__ compartments, where p ranges over calcSens, q over calcSens2, and r over calcSens3, via rxExpandSens3_(). NULL (the default) skips the third-order generation.
collapseModel	boolean indicating if rxode2 will remove all LHS variables when calculating sensitivities.
package	Package name for pre-compiled binaries.
...	ignored arguments.
linCmtSens	The method to calculate the linCmt() solutions
indLin	Calculate inductive linearization matrices and compile with inductive linearization support.
eventSens	controls how dosing/event-parameter (alag, F, rate, dur, amt) sensitivities are computed when sensitivities are generated: "jump" injects the analytic event ("jump") sensitivities into the sensitivity states at each dosing event, "fd" keeps the legacy finite-difference behavior (the backward-compatible opt-out), and "both" computes both for cross-checking. NULL (the default) uses getOption("rxode2.eventSens", "jump") for any model with sensitivities. When not "fd" and calcSens is supplied, calcJac is forced to TRUE so the Jacobian is available for the jump injection.
verbose	When TRUE be verbose with the linear compartmental model
fullPrint	When using printf within the model, if TRUE print on every step (except ME/indLin), otherwise when FALSE print only when calculating the d/dt
envir	is the environment to look for R user functions (defaults to parent environment)

Details

The Rx in the name rxode2 is meant to suggest the abbreviation *Rx* for a medical prescription, and thus to suggest the package emphasis on pharmacometrics modeling, including pharmacokinetics (PK), pharmacodynamics (PD), disease progression, drug-disease modeling, etc.

Creating rxode2 models:

The ODE-based model specification may be coded inside four places:

- Inside a rxode2({}) block statements:

```
library(rxode2)
mod <- rxode2({
  # simple assignment
  C2 <- centr/V2

  # time-derivative assignment
```

```
  d/dt(centr) <- F*KA*depot - CL*C2 - Q*C2 + Q*C3;
})
```

- Inside a `rxode2("")` string statement:

```
mod <- rxode2("
  # simple assignment
  C2 <- centr/V2

  # time-derivative assignment
  d/dt(centr) <- F*KA*depot - CL*C2 - Q*C2 + Q*C3;
")
```

- In a file name to be loaded by rxode2:

```
writeLines("
  # simple assignment
  C2 <- centr/V2

  # time-derivative assignment
  d/dt(centr) <- F*KA*depot - CL*C2 - Q*C2 + Q*C3;
", "modelFile.rxode2")
mod <- rxode2(filename='modelFile.rxode2')
unlink("modelFile.rxode2")
```

- In a model function which can be parsed by `rxode2`:

```
mod <- function() {
  model({
    # simple assignment
    C2 <- centr/V2

    # time-derivative assignment
    d/dt(centr) <- F*KA*depot - CL*C2 - Q*C2 + Q*C3;
  })
}

mod <- rxode2(mod) # or simply mod() if the model is at the end of the function

# These model functions often have residual components and initial
# (`ini({})`) conditions attached as well. For example the
# theophylline model can be written as:

one.compartment <- function() {
  ini({
    tka <- 0.45 # Log Ka
    tcl <- 1 # Log Cl
    tv <- 3.45    # Log V
    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)
    d/dt(depot) = -ka * depot
    d/dt(center) = ka * depot - cl / v * center
    cp = center / v
    cp ~ add(add.sd)
  })
}

# after parsing the model
mod <- one.compartment()
```

For the block statement, character string or text file an internal `rxode2` compilation manager translates the ODE system into C, compiles it and loads it into the R session. The call to `rxode2` produces an object of class `rxode2` which consists of a list-like structure (environment) with various member functions.

For the last type of model (a model function), a call to `rxode2` creates a parsed `rxode2` ui that can be translated to the `rxode2` compilation model.

```
mod$simulationModel

# or
mod$simulationIniModel
```

This is the same type of function required for `nlmixr2` estimation and can be extended and modified by model piping. For this reason will be focused on in the documentation.

This basic model specification consists of one or more statements optionally terminated by semicolons `;` and optional comments (comments are delimited by # and an end-of-line).

A block of statements is a set of statements delimited by curly braces, `{ ... }`.

Statements can be either assignments, conditional `if`/`else if`/`else`, `while` loops (can be exited by `break`), special statements, or printing statements (for debugging/testing).

Assignment statements can be:

- **simple** assignments, where the left hand is an identifier (i.e., variable). This includes string assignments
- special **time-derivative** assignments, where the left hand specifies the change of the amount in the corresponding state variable (compartment) with respect to time e.g., `d/dt(depot)`:
- special **initial-condition** assignments where the left hand specifies the compartment of the initial condition being specified, e.g. `depot(0) = 0`
- special model event changes including **bioavailability** (`f(depot)=1`), **lag time** (`alag(depot)=0`), **modeled rate** (`rate(depot)=2`) and **modeled duration** (`dur(depot)=2`). An example of these model features and the event specification for the modeled infusions the rxode2 data specification is found in rxode2 events vignette.

- special **change point syntax, or model times**. These model times are specified by `mtime(var)=time`
- special **Jacobian-derivative** assignments, where the left hand specifies the change in the compartment ode with respect to a variable. For example, if `d/dt(y) = dy`, then a Jacobian for this compartment can be specified as `df(y)/dy(dy) = 1`. There may be some advantage to obtaining the solution or specifying the Jacobian for very stiff ODE systems. However, for the few stiff systems we tried with LSODA, this actually slightly slowed down the solving.
- Special **string value declarations** which tell what values a string variable will take within a `rxode2` solving structure. These values will then cause a factor to be created for this variable on solving the `rxode2` model. As such, they are declared in much the same way as `R`, that is: `labels(a) <- c("a1", "a2")`.

Note that assignment can be done by `=`, `<-` or `~`.

When assigning with the `~` operator, the **simple assignments** and **time-derivative** assignments will not be output. Note that with the `rxode2` model functions assignment with `~` can also be overloaded with a residual distribution specification.

Special statements can be:

- **Compartment declaration statements**, which can change the default dosing compartment and the assumed compartment number(s) as well as add extra compartment names at the end (useful for multiple-endpoint nlmixr models); These are specified by `cmt(compartmentName)`
- **Parameter declaration statements**, which can make sure the input parameters are in a certain order instead of ordering the parameters by the order they are parsed. This is useful for keeping the parameter order the same when using 2 different ODE models. These are specified by `param(par1, par2,...)`
- **Variable interpolation statements**, which tells the interpolation for specific covariates. These include `locf(cov1, cov2, ...)` for last observation carried forward, `nocb(cov1, cov2, ...)` for next observation carried backward, `linear(cov1, cov2, ...)` for linear interpolation and `midpoint(cov1, cov2, ...)` for midpoint interpolation.

An example model is shown below:

```
# simple assignment
C2 <- centr/V2

# time-derivative assignment
d/dt(centr) <- F*KA*depot - CL*C2 - Q*C2 + Q*C3;
```

Expressions in assignment and `if` statements can be numeric or logical.

Numeric expressions can include the following numeric operators `+, -, *, /, ^, %%` and those mathematical functions defined in the C or the R math libraries (e.g., `fabs`, `exp`, `log`, `sin`, `abs`).

You may also access the R's functions in the R math libraries, like `lgammafn` for the log gamma function.

The `rxode2` syntax is case-sensitive, i.e., `ABC` is different than `abc`, `Abc`, `ABc`, etc.

Identifiers:

Like R, Identifiers (variable names) may consist of one or more alphanumeric, underscore _ or period . characters, but the first character cannot be a digit or underscore _.

Identifiers in a model specification can refer to:

- State variables in the dynamic system (e.g., compartments in a pharmacokinetics model).
- Implied input variable, `t` (time), `tlast` (last time point), and `podo` (oral dose, in the undocumented case of absorption transit models).
- Special constants like `pi` or R's predefined constants.
- Model parameters (e.g., `ka` rate of absorption, `CL` clearance, etc.)
- Others, as created by assignments as part of the model specification; these are referred as *LHS* (left-hand side) variable.

Currently, the `rxode2` modeling language only recognizes system state variables and "parameters", thus, any values that need to be passed from R to the ODE model (e.g., `age`) should be either passed in the `params` argument of the integrator function `rxSolve()` or be in the supplied event data-set.

There are certain variable names that are in the `rxode2` event tables. To avoid confusion, the following event table-related items cannot be assigned, or used as a state but can be accessed in the rxode2 code:

- `cmt`
- `dvid`
- `addl`
- `ss`
- `amt`
- `dur`
- `rate`
- `Rprintf`
- `print`
- `printf`
- `id`

However the following variables are cannot be used in a model specification:

- `evid`
- `ii`

Sometimes rxode2 generates variables that are fed back to rxode2. Similarly, nlmixr2 generates some variables that are used in nlmixr estimation and simulation. These variables start with the either the `rx` or `nlmixr` prefixes. To avoid any problems, it is suggested to not use these variables starting with either the `rx` or `nlmixr` prefixes.

Logical Operators:

Logical operators support the standard R operators `==`, `!=` `>=` `<=` `>` and `<`. Like R these can be in `if()` or `while()` statements, `ifelse()` expressions. Additionally they can be in a standard assignment. For instance, the following is valid:

```
cov1 = covm*(sexf == "female") + covm*(sexf != "female")
```

Notice that you can also use character expressions in comparisons. This convenience comes at a cost since character comparisons are slower than numeric expressions. Unlike R, `as.numeric` or `as.integer` for these logical statements is not only not needed, but will cause an syntax error if you try to use the function.

Supported functions:

All the supported functions in rxode2 can be seen with the `rxSupportedFuns()`.

A brief description of the built-in functions are in the following table:

Function	Description
gamma(x)	The Gamma function
lgamma(x)	Natural logarithm of absolute value of gamma function
digamma(x)	First derivative of lgamma
trigamma(x)	Second derivative of lgamma
tetragamma(x)	Third derivative of lgamma
pentagamma(x)	Fourth derivative of lgamma
psigamma(x, deriv)	n-th derivative of Psi, the digamma function, which is the derivative of lgammafn. In othe
cospi(x)	cos(pi*x)
sinpi(x)	sin(pi*x)
tanpi(x)	tan(pi*x)
beta(a, b)	Beta function
lbeta(a, b)	log Beta function
bessel_i(x, nu, expo)	Bessel function type I with index nu
bessel_j(x, nu)	Bessel function type J with index nu
bessel_k(x, ku, expo)	Bessel function type K with index nu
bessel_y(x, nu)	Bessel function type Y with index nu
R_pow(x, y)	x^y
R_pow_di(x, I)	x^y
log1pmx	log(1+x) - x
log1pexp	log(1+exp(x))
expm1(x)	exp(x)-1
lgamma1p(x)	log(gamma(x+1))
sign(x)	Compute the signum function where sign(x) is 1, 0 -1
fsign(x, y)	abs(x) carrying the sign of y, where y equal to 0 counts as positive
fprec(x, digits)	x rounded to digits (after the decimal point, used by signif()
fround(x, digits)	Round, used by R's round()
ftrunc(x)	Truncated towards zero
abs(x)	absolute value of x
sin(x)	sine of x
cos(x)	cos of x
tan(x)	tan of x
factorial(x)	factorial of x
lfactorial(x)	log(factorial(x))
log10(x)	log base 10
log2(x)	log base 2
dnorm(x, mean=0, sd=1)	Normal density (pdf) of x
pnorm(q, mean=0, sd=1)	Normal CDF of q
qnorm(p, mean=0, sd=1)	Normal quantile function (inverse CDF) of p
probit(x, low=0, hi=1)	Probit (normal quantile function) of x transforming into a range
probitInv(q, low=0, hi=1)	Inverse probit of x transforming into a range
acos(x)	Inverse cosine
asin(x)	Inverse sine
atan(x)	Inverse tangent

atan2(a, b)	Four quadrant inverse tangent
sinh(x)	Hyperbolic sine
cosh(x)	Hyperbolic cosine
tanh(x)	Hyperbolic tangent
floor(x)	Downward rounding
ceil(x)	Upward rounding
logit(x, low=0, hi=1)	Logit transformation of x transforming into a range
expit(x, low=0, hi=1)	expit transofmration in range
gammaq(a, z)	Normalized incomplete gamma from boost
gammaqInv(a, q)	Normalized incomplete gamma inverse from boost
ifelse(cond, trueValue, falseValue)	if else function
gammap(a, z)	Normalized lower incomplete gamma from boost
gammapInv(a, p)	Inverse of Normalized lower incomplete gamma from boost
gammapInva(x, p)	Inverse of Normalized lower incomplete gamma from boost
gammapDer(a, z)	Derivative of gammap(a, z) with respect to z; the gamma density with rate one
gammapDera(a, z)	Derivative of gammap(a, z) with respect to the shape a. No elementary closed form; a cer
phiU(q)	Normal CDF of q bounded away from 0 and 1 by 1e-15, so an inverse CDF of it cannot r
ibeta(a, b, x)	Regularized incomplete beta function I_x(a, b), ie pbeta(x, a, b)
ibetaDer(a, b, x)	Derivative of ibeta with respect to x, ie dbeta(x, a, b)
ibetaInv(a, b, p)	Inverse of ibeta, ie qbeta(p, a, b)
ibetaDera(a, b, x)	Derivative of ibeta with respect to the shape a. No elementary closed form; a central diffe
ibetaDerb(a, b, x)	Derivative of ibeta with respect to the shape b. No elementary closed form; a central diffe
studentTDen(x, nu)	Student t density with nu degrees of freedom, ie dt(x, nu)
studentTCdf(x, nu)	Student t CDF, ie pt(x, nu)
studentTCdfDnu(x, nu)	Derivative of studentTCdf with respect to nu. No elementary closed form; a central differ
studentTInv(p, nu)	Student t quantile, ie qt(p, nu); exactly the inverse of studentTCdf
rxnorm(x)	Generate one deviate of from a normal distribution for each observation scale
rxnormV(x)	Generate one deviate from low discrepancy normal for each observation
rxcauchy	Generate one deviate from the cauchy distribution for each observation
rxchisq	Generate one deviate from the chisq distribution for each observation
rxexp	Generate one deviate from the exponential distribution for each observation
rxf	Generate one deviate from low discrepancy normal for each observation
rxgamma	Generate one deviate from the gamma distribution for each observation
rxbeta	Generate one deviate from the beta distribution for each observation
rxgeom	Generate one deviate from the geometric distribution for each observation
rxpois	Generate one deviate from the poission distribution for each observation
rxt	Generate one deviate from the t distribution for each observation
tad() or tad(x)	Time after dose (tad()) or time after dose for a compartment tad(cmt); no dose=NA
tad0() or tad0(x)	Time after dose (tad0()) or time after dose for a compartment tad0(cmt); no dose=0
tafd() or tafd(x)	Time after first dose (tafd()) or time after first dose for a compartment tafd(cmt); no dose
tafd0() or tafd0(x)	Time after first dose (tafd()) or time after first dose for a compartment tafd(cmt); no dose
dosenum()	Dose Number
tlast() or tlast(cmt)	Time of Last dose; This takes into consideration any lag time, so if there is a dose at time
tlast0() or tlast0(cmt)	Time of Last dose; This takes into consideration any lag time, so if there is a dose at time
tfirst() or tfirst(cmt)	Time since first dose or time since first dose of a compartment; no dose=NA
tfirst0() or tfirst0(cmt)	Time since first dose or time since first dose of a compartment; no dose=0
prod(...)	product of terms; This uses PreciseSums so the product will not have as much floating po
sum(...)	sum of terms; This uses PreciseSums so the product will not have as much floating point

max(...)	maximum of a group of numbers
min(...)	Min of a group of numbers
lag(parameter, number=1)	Get the lag of an input parameter; You can specify a number of lagged observations
lead(parameter, number=2)	Get the lead of an input parameter; You can specify a number of lead observation
diff(par, number=1)	Get the difference between the current parameter and the last parameter; Can change the
lag0(parameter, number=1)	Like lag() but returns 0 instead of NA when there is no prior value (the first record)
lead0(parameter, number=1)	Like lead() but returns 0 instead of NA when there is no following value (the last record)
diff0(par, number=1)	Like diff() but returns the value itself (previous value treated as 0) when there is no prior
first(par)	Get the first value of an input parameter
last(par)	Get the last value of an input parameter
transit()	The transit compartment pseudo function
delay(state, T)	Value of an ODE state delayed by T time units (delay differential equations); requires a d
is.na()	Determine if a value is NA
is.nan()	Determine if a value is NaN
is.infinite()	Check to see if the value is infinite
rinorm(x)	Generate one deviate of from a normal distribution for each individual
rinormV(x)	Generate one deviate from low discrepancy normal for each individual
ricauchy	Generate one deviate from the cauchy distribution for each individual
richisq	Generate one deviate from the chisq distribution for each individual
riexp	Generate one deviate from the exponential distribution for each individual
rif	Generate one deviate from low discrepancy normal for each individual
rigamma	Generate one deviate from the gamma distribution for each individual
ribeta	Generate one deviate from the beta distribution for each individual
rigeom	Generate one deviate from the geometric distribution for each individual
ropois	Generate one deviate from the poission distribution for each individual
rit	Generate one deviate from the t distribution for each individual
simeps	Simulate EPS from possibly truncated sigma matrix. Will take sigma matrix from the cur
simeta	Simulate ETA from possibly truncated omega matrix. Will take the omega matrix from th

Note that `lag(cmt) =` is equivalent to `alag(cmt) =` and not the same as `= lag(wt)`

Inverse CDFs:

A group of these deserves a word, because they exist for one purpose: giving a random effect a distribution other than normal. The technique (Bauer, NONMEM 7.5.1) keeps the latent random effect standard normal and changes the CDF,

```
z   ~ N(0, 1)
u   = phiU(z)
eta = Q(u; args)
```

where `Q` is the inverse CDF of the family you want. `phiU()` is the normal CDF bounded away from 0 and 1, since `phi()` saturates to exactly 0 or 1 around `|q| = 8.3` and an inverse CDF of that would return an infinity.

Most families need nothing beyond ordinary arithmetic for `Q` – a Weibull random effect is `scale*(-log1p(-phiU(z)))^(1/shape)`, a lognormal one is just `exp(meanlog + sdlog*z)`. The three that are not elementary are served by `gammapInv()` (gamma, chi-square, inverse gamma), `ibetaInv()` (beta) and `studentTInv()` (Student t).

Every one of these is differentiable through rxode2's symbolic machinery, which is what lets such a random effect be *estimated* and not merely simulated. The derivative with respect to the probability argument – the one an inner (eta) problem needs – is exact and elementary, one over

the density at the quantile. The derivatives with respect to the shape parameters have no elementary closed form, so `gammapDera()`, `ibetaDera()`, `ibetaDerb()` and `studentTCdfDnu()` supply them as central differences with one Richardson extrapolation. They exist so the derivative table is *complete*: a missing rule does not error, it silently substitutes a one sided finite difference, which is precisely wrong in the tails where an inverse CDF is steepest.
You rarely write any of this by hand. Declaring the distribution in an `ini({})` block writes it for you:

```
ini({
  lclm  <- log(5)
  lclrv <- log(0.09)
  dist(eta.cl) ~ dgamma(shape=1/exp(lclrv), rate=1/(exp(lclrv)*exp(lclm)))
})
```

The declaration fixes the random effect's marginal, and the latent scale it is built on is standard normal, so no variance is written for it – a `dist()` implies `~ 1`. A block is written out only when there is a correlation between two declared random effects, since that has nowhere else to go:

```
ini({
  lclm  <- log(5);  lclrv <- log(0.09)
  lvm   <- log(50); lvrv  <- log(0.09)
  eta.cl + eta.v ~ c(1,
                     0.5, 1)
  dist(eta.cl) ~ dgamma(shape=1/exp(lclrv), rate=1/(exp(lclrv)*exp(lclm)))
  dist(eta.v)  ~ dgamma(shape=1/exp(lvrv),  rate=1/(exp(lvrv)*exp(lvm)))
})
```

That correlation is on the LATENT scale:
The `0.5` above is **not** the correlation you will measure between `eta.cl` and `eta.v`. Two declared random effects are joined by a Gaussian copula: each is built as `Q(Phi(z))` from its own quantile function, and the number you write is the correlation of the *latent* normals `z`, not of the random effects themselves.
Those two agree only when both marginals are normal. For anything else the correlation you observe is smaller in magnitude, by an amount that depends on the two families **and on their parameters** – a heavier tail loses more of it. Simulating 200,000 pairs from a latent correlation of `0.50`:

marginals	observed
`dnorm` with `dnorm`	0.500
`dgamma(shape 2)` with `dgamma(shape 2)`	0.474
`dgamma(shape 2)` with `dlnorm(sdlog 0.7)`	0.454
`dlnorm(sdlog 0.7)` with `dlnorm(sdlog 0.7)`	0.441
`dgamma(shape 2)` with `dgamma(shape 0.5)`	0.437

Read those as illustrations, not as constants for the families: change a shape or an `sdlog` and the number moves. The one row that holds generally is the first – normal with normal is the only case where the latent correlation and the observed one coincide.
So writing `0.5` and then measuring `0.44` on the fitted random effects is the construction working, not a bug. A fit reports the same latent quantity it was given – in `$etaDistCor`, and as a `cor()` row in `$parFixed` – so the value you read back is comparable with the value you wrote.

If you need a target correlation *between the random effects*, simulate from the declaration and adjust: there is no closed form for the mapping except in the all-normal case, because it depends on both quantile functions.
See `lotri::lotriEtaDists()` for the families that can be declared this way, and `rxEtaDistExpand()` for the model such a declaration becomes.

Reserved keywords:

There are a few reserved keywords in a rxode2 model. They are in the following table:

Reserved Name	Meaning
time	solver time
podo	In Transit compartment models, last dose amount
tlast	Time of Last dose
M_E	Exp(1)
M_LOG2E	log2(e)
M_LOG10E	log10(e)
M_LN2	log(2)
M_LN10	log(10)
M_PI	pi
M_PI_2	pi/2
M_PI_4	pi/4
M_1_PI	1/pi
M_2_PI	2/pi
M_2_SQRTPI	2/sqrt(pi)
M_SQRT2	sqrt(2)
M_SQRT1_2	1/sqrt(2)
M_SQRT_3	sqrt(3)
M_SQRT_32	sqrt(32)
M_LOG10_2	Log10(2)
M_2PI	2*pi
M_SQRT_PI	sqrt(pi)
M_1_SQRT_2PI	1/(sqrt(2*pi))
M_LN_SQRT_PI	log(sqrt(pi))
M_LN_SQRT_2PI	log(sqrt(2*pi))
M_LN_SQRT_PId2	log(sqrt(pi/2))
pi	pi
NA	R's NA value
NaN	Not a Number Value
Inf	Infinite Value
newind	1: First record of individual; 2: Subsequent record of individual
rxFlag	Flag for what part of the rxode2 model is being run; 1: ddt; 2: jac; 3: ini; 4: F; 5: lag; 6: rate; 7: dur; 8:

Note that `rxFlag` will always output `11` or `calc_lhs` since that is where the final variables are calculated, though you can tweak or test certain parts of `rxode2` by using this flag.

Residual functions when using rxode2 functions:

In addition to `~` hiding output for certain types of output, it also is used to specify a residual output or endpoint when the input is an `rxode2` model function (that includes the residual in the `model({})` block).

These specifications are of the form:

```
var ~ add(add.sd)
```

Indicating the variable `var` is the variable that represents the individual central tendencies of the model and it also represents the compartment specification in the data-set.
You can also change the compartment name using the `|` syntax, that is:

```
var ~ add(add.sd) | cmt
```

In the above case `var` represents the central tendency and `cmt` represents the compartment or `dvid` specification.

Transformations:
For normal and related distributions, you can apply the transformation on both sides by using some keywords/functions to apply these transformations.

Transformation	rxode2/nlmixr2 code
Box-Cox	+boxCox(lambda)
Yeo-Johnson	+yeoJohnson(lambda)
logit-normal	+logitNorm(logit.sd, low, hi)
probit-normal	+probitNorm(probid.sd, low, hi)
log-normal	+lnorm(lnorm.sd)

By default for the likelihood for all of these transformations is calculated on the **untransformed** scale.
For bounded variables like logit-normal or probit-normal the low and high values are defaulted to 0 and 1 if missing.
For models where you wish to have a proportional model on one of these transformation you can replace the standard deviation with `NA`
To allow for more transformations, `lnorm()`, `probitNorm()` and `logitNorm()` can be combined the variance stabilizing `yeoJohnson()` transformation.

Normal and t-related distributions:
For the normal and t-related distributions, we wanted to keep the ability to use skewed distributions additive and proportional in the t/cauchy-space, so these distributions are specified differently in comparison to the other supported distributions within `nlmixr2`:

Distribution	How to Add	Example
Normal (log-likelihood)	+dnorm()	cc ~ add(add.sd) + dnorm()
T-distribution	+dt(df)	cc ~a dd(add.sd) + dt(df)
Cauchy (t with df=1)	+dcauchy()	cc ~ add(add.sd) + dcauchy()

Note that with the normal and t-related distributions `nlmixr2` will calculate `cwres` and `npde` under the normal assumption to help assess the goodness of the fit of the model.
Also note that the `+dnorm()` is mostly for testing purposes and will slow down the estimation procedure in `nlmixr2`. We suggest not adding it (except for explicit testing). When there are multiple endpoint models that mix non-normal and normal distributions, the whole problem is shifted to a log-likelihood method for estimation in `nlmixr2`.

Notes on additive + proportional models:
There are two different ways to specify additive and proportional models, which we will call **combined1** and **combined2**, the same way that Monolix calls the two distributions (to avoid between software differences in naming).
The first, **combined1**, assumes that the additive and proportional differences are on the standard deviation scale, or:
y=f+(a+b* f^c)*err
The second, **combined2**, assumes that the additive and proportional differences are combined on a variance scale:
y=f+$$sqrt(a^2+b^2 *f^(2c))$$*err
The default in `nlmixr2`/`rxode2` if not otherwise specified is **combined2** since it mirrors how adding 2 normal distributions in statistics will add their variances (not the standard deviations). However, the **combined1** can describe the data possibly even better than **combined2** so both are possible options in `rxode2`/`nlmixr2`.

Autocorrelated residuals (`ar()`):
Adding `ar(cor)` to a normal residual makes successive residuals of the same endpoint follow a continuous-time AR(1) process:

```
cp ~ add(add.sd) + ar(ar1.cor)
```

Here `ar1.cor` is the lag-one correlation and must be in `[0, 1)`. The correlation between two residuals separated by a time gap `dt` is `cor^dt`, so irregular sampling is handled directly and the marginal (stationary) residual variance is unchanged; the first observation of each subject/endpoint uses the marginal distribution. `ar()` combines with `add()`/`prop()` normal residuals and uses the same `| cmt` endpoint syntax as the other residual functions.
The `ar()` specification is identical in `rxode2` and `nlmixr2`: the same model line simulates the autocorrelated residual in `rxode2` and estimates the correlation in `nlmixr2` (the `nlm`, `focei`/`foce` and `saem` families support it; `fo`, `foi`, `nlme` and `nls` do not). See Karlsson et al. (1995) for the continuous-time AR(1) residual model.

Distributions of known likelihoods:
For residuals that are not related to normal, t-distribution or cauchy, often the residual specification is of the form:

```
cmt ~ dbeta(alpha, beta)
```

Where the compartment specification is on the left handed side of the specification.
For generalized likelihood you can specify:

```
ll(cmt) ~ llik specification
```

Ordinal likelihoods:
Finally, ordinal likelihoods/simulations can be specified in 2 ways. The first is:

```
err ~ c(p0, p1, p2)
```

Here `err` represents the compartment and `p0` is the probability of being in a specific category:

Category	Probability
1	p0
2	p1
3	p2
4	1-p0-p1-p2

It is up to the model to ensure that the sum of the p values are less than 1. Additionally you can write an arbitrary number of categories in the ordinal model described above.
It seems a little off that p0 is the probability for category 1 and sometimes scores are in non-whole numbers. This can be modeled as follows:

```
err ~ c(p0=0, p1=1, p2=2, 3)
```

Here the numeric categories are specified explicitly, and the probabilities remain the same:

Category	Probability
0	p0
1	p1
2	p2
3	1-p0-p1-p2

General table of supported residual distributions:
In general all the that are supported are in the following table (available in `rxode2::rxResidualError`)

Error model	Functional Form	Transformation	code
constant		None	var ~ add(add.sd)
proportional		None	var ~ prop(prop.sd)
power		None	var ~ pow(pow.sd, exponent)
additive+proportional	combined1	None	var ~ add(add.sd) + prop(prop.sd) + coml
additive+proportional	combined2	None	var ~ add(add.sd) + prop(prop.sd) + coml
additive+power	combined1	None	var ~ add(add.sd) + pow(pow.sd, exponer
additive+power	combined2	None	var ~ add(add.sd) + pow(pow.sd, exponer
constant		log	var ~ lnorm(add.sd)
proportional		log	var ~ lnorm(NA) + prop(prop.sd)
power		log	var ~ lnorm(NA) + pow(pow.sd, exponen
additive+proportional	combined1	log	var ~ lnorm(add.sd) + prop(prop.sd) + co
additive+proportional	combined2	log	var ~ lnorm(add.sd) + prop(prop.sd) + co
additive+power	combined1	log	var ~ lnorm(add.sd) + pow(pow.sd, expor
additive+power	combined2	log	var ~ lnorm(add.sd) + pow(pow.sd, expor
constant		boxCox	var ~ boxCox(lambda) + add(add.sd)
proportional		boxCox	var ~ boxCox(lambda) + prop(prop.sd)
power		boxCox	var ~ boxCox(lambda) + pow(pow.sd, ex
additive+proportional	combined1	boxCox	var ~ boxCox(lambda) + add(add.sd) + p
additive+proportional	combined2	boxCox	var ~ boxCox(lambda) + add(add.sd) + p
additive+power	combined1	boxCox	var ~ boxCox(lambda) + add(add.sd) + p
additive+power	combined2	boxCox	var ~ boxCox(lambda) + add(add.sd) + p
constant		yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
proportional		yeoJohnson	var ~ yeoJohnson(lambda) + prop(prop.s
power		yeoJohnson	var ~ yeoJohnson(lambda) + pow(pow.sd
additive+proportional	combined1	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
additive+proportional	combined2	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
additive+power	combined1	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
additive+power	combined2	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
constant		logit	var ~ logitNorm(logit.sd)
proportional		logit	var ~ logitNorm(NA) + prop(prop.sd)

power		logit	var ~ logitNorm(NA) + pow(pow.sd, expo
additive+proportional	combined1	logit	var ~ logitNorm(logit.sd) + prop(prop.sd)
additive+proportional	combined2	logit	var ~ logitNorm(logit.sd) + prop(prop.sd)
additive+power	combined1	logit	var ~ logitNorm(logit.sd) + pow(pow.sd,
additive+power	combined2	logit	var ~ logitNorm(logit.sd) + pow(pow.sd,
additive		yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
proportional		yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(N
power		yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(N
additive+proportional	combined1	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
additive+proportional	combined2	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
additive+power	combined1	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
additive+power	combined2	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
constant		logit	var ~ probitNorm(probit.sd)
proportional		probit	var ~ probitNorm(NA) + prop(prop.sd)
power		probit	var ~ probitNorm(NA) + pow(pow.sd, ex
additive+proportional	combined1	probit	var ~ probitNorm(probit.sd) + prop(prop.
additive+proportional	combined2	probit	var ~ probitNorm(probit.sd) + prop(prop.
additive+power	combined1	probit	var ~ probitNorm(probit.sd) + pow(pow.s
additive+power	combined2	probit	var ~ probitNorm(probit.sd) + pow(pow.s
additive		yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
proportional		yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
power		yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+proportional	combined1	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+proportional	combined2	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+power	combined1	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+power	combined2	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
constant+t		None	var ~ add(add.sd) + dt(df)
proportional+t		None	var ~ prop(prop.sd) + dt(df)
power+t		None	var ~ pow(pow.sd, exponent) + dt(df)
additive+proportional+t	combined1	None	var ~ add(add.sd) + prop(prop.sd) + dt(df
additive+proportional+t	combined2	None	var ~ add(add.sd) + prop(prop.sd) + dt(df
additive+power+t	combined1	None	var ~ add(add.sd) + pow(pow.sd, exponen
additive+power+t	combined2	None	var ~ add(add.sd) + pow(pow.sd, exponen
constant+t		log	var ~ lnorm(add.sd) + dt(df)
proportional+t		log	var ~ lnorm(NA) + prop(prop.sd) + dt(df)
power+t		log	var ~ lnorm(NA) + pow(pow.sd, exponen
additive+proportional+t	combined1	log	var ~ lnorm(add.sd) + prop(prop.sd) + dt(
additive+proportional+t	combined2	log	var ~ lnorm(add.sd) + prop(prop.sd) + dt(
additive+power+t	combined1	log	var ~ lnorm(add.sd) + pow(pow.sd, expon
additive+power+t	combined2	log	var ~ lnorm(add.sd) + pow(pow.sd, expon
constant+t		boxCox	var ~ boxCox(lambda) + add(add.sd)+dt(
proportional+t		boxCox	var ~ boxCox(lambda) + prop(prop.sd)+d
power+t		boxCox	var ~ boxCox(lambda) + pow(pow.sd, exp
additive+proportional+t	combined1	boxCox	var ~ boxCox(lambda) + add(add.sd) + pr
additive+proportional+t	combined2	boxCox	var ~ boxCox(lambda) + add(add.sd) + pr
additive+power+t	combined1	boxCox	var ~ boxCox(lambda) + add(add.sd) + po
additive+power+t	combined2	boxCox	var ~ boxCox(lambda) + add(add.sd) + po
constant+t		yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)

proportional+t		yeoJohnson	var ~ yeoJohnson(lambda) + prop(prop.s
power+t		yeoJohnson	var ~ yeoJohnson(lambda) + pow(pow.sd
additive+proportional+t	combined1	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
additive+proportional+t	combined2	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
additive+power+t	combined1	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
additive+power+t	combined2	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
constant+t		logit	var ~ logitNorm(logit.sd)+dt(df)
proportional+t		logit	var ~ logitNorm(NA) + prop(prop.sd)+dt
power+t		logit	var ~ logitNorm(NA) + pow(pow.sd, expo
additive+proportional+t	combined1	logit	var ~ logitNorm(logit.sd) + prop(prop.sd)
additive+proportional+t	combined2	logit	var ~ logitNorm(logit.sd) + prop(prop.sd)
additive+power+t	combined1	logit	var ~ logitNorm(logit.sd) + pow(pow.sd,
additive+power+t	combined2	logit	var ~ logitNorm(logit.sd) + pow(pow.sd,
additive+t		yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
proportional+t		yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(N
power+t		yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(N
additive+proportional+t	combined1	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
additive+proportional+t	combined2	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
additive+power+t	combined1	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
additive+power+t	combined2	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
constant+t		logit	var ~ probitNorm(probit.sd) + dt(df)
proportional+t		probit	var ~ probitNorm(NA) + prop(prop.sd) +
power+t		probit	var ~ probitNorm(NA) + pow(pow.sd, ex
additive+proportional+t	combined1	probit	var ~ probitNorm(probit.sd) + prop(prop.
additive+proportional+t	combined2	probit	var ~ probitNorm(probit.sd) + prop(prop.
additive+power+t	combined1	probit	var ~ probitNorm(probit.sd) + pow(pow.s
additive+power+t	combined2	probit	var ~ probitNorm(probit.sd) + pow(pow.s
additive+t		yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
proportional+t		yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
power+t		yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+proportional+t	combined1	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+proportional+t	combined2	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+power+t	combined1	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+power+t	combined2	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
constant+cauchy		None	var ~ add(add.sd) + dcauchy()
proportional+cauchy		None	var ~ prop(prop.sd) + dcauchy()
power+cauchy		None	var ~ pow(pow.sd, exponent) + dcauchy()
additive+proportional+cauchy	combined1	None	var ~ add(add.sd) + prop(prop.sd) + dcau
additive+proportional+cauchy	combined2	None	var ~ add(add.sd) + prop(prop.sd) + dcau
additive+power+cauchy	combined1	None	var ~ add(add.sd) + pow(pow.sd, exponen
additive+power+cauchy	combined2	None	var ~ add(add.sd) + pow(pow.sd, exponen
constant+cauchy		log	var ~ lnorm(add.sd) + dcauchy()
proportional+cauchy		log	var ~ lnorm(NA) + prop(prop.sd) + dcauc
power+cauchy		log	var ~ lnorm(NA) + pow(pow.sd, exponen
additive+proportional+cauchy	combined1	log	var ~ lnorm(add.sd) + prop(prop.sd) + dc
additive+proportional+cauchy	combined2	log	var ~ lnorm(add.sd) + prop(prop.sd) + dc
additive+power+cauchy	combined1	log	var ~ lnorm(add.sd) + pow(pow.sd, expon
additive+power+cauchy	combined2	log	var ~ lnorm(add.sd) + pow(pow.sd, expon

constant+cauchy		boxCox	var ~ boxCox(lambda) + add(add.sd)+dc
proportional+cauchy		boxCox	var ~ boxCox(lambda) + prop(prop.sd)+c
power+cauchy		boxCox	var ~ boxCox(lambda) + pow(pow.sd, ex
additive+proportional+cauchy	combined1	boxCox	var ~ boxCox(lambda) + add(add.sd) + p
additive+proportional+cauchy	combined2	boxCox	var ~ boxCox(lambda) + add(add.sd) + p
additive+power+cauchy	combined1	boxCox	var ~ boxCox(lambda) + add(add.sd) + p
additive+power+cauchy	combined2	boxCox	var ~ boxCox(lambda) + add(add.sd) + p
constant+cauchy		yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
proportional+cauchy		yeoJohnson	var ~ yeoJohnson(lambda) + prop(prop.s
power+cauchy		yeoJohnson	var ~ yeoJohnson(lambda) + pow(pow.sd
additive+proportional+cauchy	combined1	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
additive+proportional+cauchy	combined2	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
additive+power+cauchy	combined1	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
additive+power+cauchy	combined2	yeoJohnson	var ~ yeoJohnson(lambda) + add(add.sd)
constant+cauchy		logit	var ~ logitNorm(logit.sd)+dcauchy()
proportional+cauchy		logit	var ~ logitNorm(NA) + prop(prop.sd)+dc
power+cauchy		logit	var ~ logitNorm(NA) + pow(pow.sd, exp
additive+proportional+cauchy	combined1	logit	var ~ logitNorm(logit.sd) + prop(prop.sd)
additive+proportional+cauchy	combined2	logit	var ~ logitNorm(logit.sd) + prop(prop.sd)
additive+power+cauchy	combined1	logit	var ~ logitNorm(logit.sd) + pow(pow.sd,
additive+power+cauchy	combined2	logit	var ~ logitNorm(logit.sd) + pow(pow.sd,
additive+cauchy		yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
proportional+cauchy		yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(N
power+cauchy		yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(N
additive+proportional+cauchy	combined1	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
additive+proportional+cauchy	combined2	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
additive+power+cauchy	combined1	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
additive+power+cauchy	combined2	yeoJohnson(logit())	var ~ yeoJohnson(lambda) + logitNorm(l
constant+cauchy		logit	var ~ probitNorm(probit.sd) + dcauchy()
proportional+cauchy		probit	var ~ probitNorm(NA) + prop(prop.sd) +
power+cauchy		probit	var ~ probitNorm(NA) + pow(pow.sd, ex
additive+proportional+cauchy	combined1	probit	var ~ probitNorm(probit.sd) + prop(prop.
additive+proportional+cauchy	combined2	probit	var ~ probitNorm(probit.sd) + prop(prop.
additive+power+cauchy	combined1	probit	var ~ probitNorm(probit.sd) + pow(pow.s
additive+power+cauchy	combined2	probit	var ~ probitNorm(probit.sd) + pow(pow.s
additive+cauchy		yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
proportional+cauchy		yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
power+cauchy		yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+proportional+cauchy	combined1	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+proportional+cauchy	combined2	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+power+cauchy	combined1	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
additive+power+cauchy	combined2	yeoJohnson(probit())	var ~ yeoJohnson(lambda) + probitNorm
autocorrelated (constant)		None	var ~ add(add.sd) + ar(cor)
autocorrelated (additive+proportional)		None	var ~ add(add.sd) + prop(prop.sd) + ar(co
poission		none	cmt ~ dpois(lamba)
binomial		none	cmt ~ dbinom(n, p)
beta		none	cmt ~ dbeta(alpha, beta)
chisq		none	cmt ~ dchisq(nu)

exponential	none	cmt ~ dexp(r)
uniform	none	cmt ~ dunif(a, b)
weibull	none	cmt ~ dweibull(a, b)
gamma	none	cmt ~ dgamma(a, b)
geometric	none	cmt ~ dgeom(a)
negative binomial form #1	none	cmt ~ dnbinom(n, p)
negative binomial form #2	none	cmt ~ dnbinomMu(size, mu)
ordinal probability	none	cmt ~ c(p0=0, p1=1, p2=2, 3)
log-likelihood	none	ll(cmt) ~ log likelihood expression

Prior distributions in the `ini({})` block:

When an estimation method supports priors, the prior on a parameter is given in the `ini({})` block with:

```
prior(name) ~ dist(...)
```

Because the statement names the parameter it applies to, prior lines are order independent; they can be put anywhere in the `ini({})` block.

```
one.compartment <- function() {
  ini({
    tka <- 0.45
    tcl <- log(c(0, 2.7, 100))
    tv  <- 3.45
    add.sd <- c(0, 0.7)

    eta.cl + eta.v ~ c(0.3,
                       0.01, 0.1)
    eta.ka ~ 0.6

    prior(tka)           ~ dnorm(0, 10)
    prior(tcl)           ~ dnorm(1, 10)
    prior(add.sd)        ~ dcauchy(0, 5)
    prior(eta.ka)        ~ dgamma(2, 1)
    prior(eta.cl, eta.v) ~ lkjCorr(2)
  })
  model({
    ka <- exp(tka + eta.ka)
    cl <- exp(tcl + eta.cl)
    v  <- exp(tv + eta.v)
    d/dt(depot)  <- -ka * depot
    d/dt(center) <-  ka * depot - cl / v * center
    cp <- center / v
    cp ~ add(add.sd)
  })
}
```

A prior can be put on:

- a population parameter, ie `prior(tka)`

- a single between subject variability term, ie `prior(eta.ka)`
- a whole covariance block, using one of the matrix-valued distributions, ie `prior(eta.cl, eta.v) ~ lkjCorr(2)`. The names given have to be exactly one block of the matrix.

The normal prior shorthand:

Normal priors are by far the most common, so they also have a shorthand that reuses the matrix syntax. Putting a *population parameter* on the left of a `~` gives it a normal prior with a **zero mean** and the given variance:

```
ini({
  tka <- 1
  tcl <- 3
  tv  <- 4

  tka ~ 4          # tka ~ N(0, sd = 2)

  tcl + tv ~ c(1,  # (tcl, tv) ~ MVN(0, Sigma)
               0.01, 1)
})
```

The number on the right is a variance, so `tka ~ 4` has a standard deviation of 2. The value given with `<-` stays the initial estimate; it is *not* the prior mean.

This is unambiguous because a name cannot be both a population parameter and an eta – that combination used to be a "duplicated parameter" error. A name that is not a population parameter still specifies an eta exactly as before.

Every matrix spelling works here, including the per-row line form, which builds up the block just as it does for etas:

```
tcl ~ 1
tv  ~ c(0.01, 1)
```

and the `sd()`, `var()`, `cor()`, `cov()` and `chol()` transformations:

```
tcl + tv ~ sd(2,
              0.5, 3)   # variances of 4 and 9
```

The matrix means exactly what it means for an eta block: the off-diagonal is a **covariance**, not a correlation. A prior block and an eta block written the same way give the same matrix, so `cor()` is how you give a correlation here just as it is there.

An uncorrelated block is simply independent normal priors, since that is what a multivariate normal with a diagonal covariance is. A zero variance is a point mass rather than a prior, so it is an error.

Priors on the omega values:

The two NONMEM prior flavours want different things from an omega.

An `NWPRI` model gives the omega block degrees of freedom. The scale matrix of the Wishart family is optional, because the block it is put on already *is* that matrix, so only the degrees of freedom are needed – this is the `$OMEGAP`/`$OMEGAPD` pair:

```
ini({
  eta.cl + eta.v ~ c(0.3,
                     0.01, 0.1)
  prior(eta.cl, eta.v) ~ invWishart(4)
})
```

It works on a 1x1 block as well, since an inverse Wishart of dimension one is an inverse gamma. An improper `nu <= p - 1` is an error.

When every block shares the same degrees of freedom, a one-sided ~ sets them all at once instead of naming each block:

```
ini({
  eta.cl + eta.v ~ c(0.3,
                     0.01, 0.1)
  eta.ka ~ 0.5
  ~invWishart(4)      # every omega block gets 4
})
```

Each block is still checked individually, and naming a block as well is a duplicate rather than an override.

A TNPRI model instead puts a normal prior on the omega *elements*, jointly with the thetas. Prepending om. to a between subject variability names its omega element, so the normal prior shorthand can be used on it:

```
ini({
  eta.cl ~ 0.3
  eta.v ~ 0.1
  om.eta.cl ~ 0.01     # normal prior on the omega element of eta.cl
  om.eta.v ~ 0.04
})
```

The omega itself is untouched; only the prior is added. Correlated omega priors work the same way, including the per-row line form. An om. name has to match a real between subject variability – it never quietly creates one.

Naming the eta directly means the same thing, so prior(om.eta.cl) and prior(eta.cl) are interchangeable. The om. spelling exists so the shorthand has a name to put on the left of a ~, since eta.cl ~ ... already means the omega value itself.

Prior distribution names:

There are three spellings of every distribution, and all of them are accepted:

1. the **R** name, when R parameterizes the distribution the same way 'Stan' does – dnorm(), dlnorm(), dgamma(), dbeta(), dcauchy(), dunif()
2. the **camelCase** name, which is the 'Stan' name written the way the rest of the package is written – invWishart(), lkjCorr(), studentT()
3. the **'Stan'** name itself – inv_wishart(), lkj_corr(), student_t()

The canonical one, which is what gets stored on the iniDf and printed back, is the R name where there is a faithful one and the camelCase name otherwise. The arguments may be given positionally or by name, so these are all the same prior:

```
prior(tka) ~ dnorm(0, 10)
prior(tka) ~ normal(0, 10)
prior(tka) ~ dnorm(sd=10, mean=0)
```

dt() is deliberately **not** accepted as a spelling of studentT(): R's dt() is the standardized (or noncentral) t, while studentT(nu, mu, sigma) (the 'Stan' student_t) is a location-scale t, so treating them as the same would silently change the prior. Use studentT() with its own parameterization.

The full list of supported distributions, including the 'Stan' name for each, is returned by lotri::lotriPriorDists().

Bounds and truncated priors:

The bounds are not repeated in the prior; they come from the parameter itself. A parameter that is already bounded below by zero therefore gets a half distribution:

```
add.sd <- c(0, 0.7)
prior(add.sd) ~ dcauchy(0, 5)   # half-Cauchy, since add.sd has lower=0
```

The prior is checked against those bounds, so putting a distribution with positive support on a parameter that allows negative values is an error.

Where the prior is stored:
Priors are kept in the `prior` column of the model's `$iniDf` and are printed back as part of the `ini({})` block, so they survive printing and piping.
A prior you specify is never silently dropped. An estimation method that cannot use priors is expected to reject such a model rather than quietly ignore the prior, and rxode2 supplies the assertions for that:

- `assertRxUiNoPriors()` – for a method that cannot use priors at all; a model that specifies one is an error
- `assertRxUiNormalPriors()` – for a method that supports priors but only normal ones. That covers `dnorm()`/`normal()`, `stdNormal()`, and the `multiNormal()` family, which is what the shorthand above produces for correlated parameters. Anything else, including a covariance matrix prior such as `lkjCorr()` or `invWishart()`, is an error
- `assertRxUiNoOmegaDf()` – for a method that cannot use prior degrees of freedom on an omega block, ie the `NWPRI` form above
- `assertRxUiNoOmegaNormalPriors()` – for a method that can put a prior on an omega but only a Wishart one, ie it supports `NWPRI` but not `TNPRI`

A method that *implements* priors asks instead of asserts. The predicates return `TRUE`/`FALSE` rather than throwing:

- `testRxUiPriors()` – does the model specify any prior at all
- `testRxUiNormalPriors()` – is every prior a normal one
- `testRxUiOmegaDf()` – does an omega carry degrees of freedom (`NWPRI`)
- `testRxUiOmegaNormalPriors()` – does an omega carry a normal prior (`TNPRI`)

The last two are mutually exclusive: an omega prior is either degrees of freedom or a normal prior, never both, and giving both is an error at specification time. So a method can branch on which one it got.
`rxUiPriors()` returns the priors themselves – `name`, `prior`, `neta1`/`neta2` (`NA` for a population parameter) and the parameter's `lower`/`upper`, which is what a truncated prior needs for its bounds.
So if a method silently ignored your prior, that is a bug in the method, not expected behaviour.

Simulating from the priors:

`rxSolve()` uses the priors for its uncertainty simulation, which is what NONMEM does with `$PRIOR NWPRI` and `$PRIOR TNPRI`. Nothing extra has to be turned on: a model that carries priors uses them whenever variability is being simulated, that is when `nStud` is greater than one (or whatever `simVariability` forces).

```
one.cmt <- function() {
  ini({
    tka <- 0.45
    tcl <- 1
    tv <- 3.45
    eta.cl + eta.v ~ c(0.3,
                       0.01, 0.1)
```

```
    eta.ka ~ 0.6
    add.sd <- 0.7
    tka ~ 0.01                            # normal prior on tka
    prior(eta.cl, eta.v) ~ invWishart(20) # degrees of freedom, per block
    prior(eta.ka) ~ invWishart(4)
  })
  model({
    ka <- exp(tka + eta.ka)
    cl <- exp(tcl + eta.cl)
    v <- exp(tv + eta.v)
    linCmt() ~ add(add.sd)
  })
}

s <- rxSolve(one.cmt, ev, nSub = 10, nStud = 100)
s$thetaMat   # the per study population parameter draws
s$omegaList  # the per study omega, one draw per study
```

Each omega block is drawn with **its own** degrees of freedom, which the single `dfSub` argument cannot express – above, the 2x2 block is drawn with 20 and the 1x1 with 4. A block with no prior is left at its point estimate. `dfWishart()` helps pick a degrees of freedom that matches a target relative standard error.

The prior mean is the estimate:
A draw is added to the value the model already gives, so a normal prior on a population parameter has to be centered on its estimate, and one on an omega element on that omega value. `tka ~ 0.01` is exactly that – the number is the variance and the mean comes from `tka <- 0.45`. Writing a prior centered somewhere else is an error rather than a simulation that quietly differs from what the model says:

```
tka <- 0.45
prior(tka) ~ dnorm(0, 0.1)
#> Error: cannot simulate from the prior on 'tka' (dnorm(0, 0.1)): the
#>   prior mean (0) is not the initial estimate (0.45)
```

A joint prior over the thetas and the omegas:
A `TNPRI` variance matrix covers the population parameters and the omega values together, with covariances between them. Prefix an omega element with `om.` to name it, and put both in one block:

```
tcl + om.eta.cl ~ c(0.02,
                    0.004, 0.005)
```

Drawn that way the omega is not guaranteed positive definite. A study whose draw is not gets the whole joint vector redrawn, up to `priorPdRetry` times (10 by default). If none of them is, the nearest positive definite matrix of the kept draws is used and `rxSolve()` warns – that projection lands on the boundary of the positive definite cone, so those studies are not draws from the stated prior. A model that reaches the fallback often wants a larger `priorPdRetry` or a tighter prior.

Turning it off, and precedence:
`usePrior = FALSE` ignores the priors and falls back to a supplied `thetaMat`/`dfSub`; `usePrior = TRUE` requires them and says why if it cannot honor them. A `thetaMat`/`dfSub` carried in the

model's `meta` block loses to the priors, with a warning. One given at the call site wins over them instead, also with a warning – an explicit argument is never silently discarded.
Prior simulation covers nested/occasion models (`| id`, `| occ`). Each prior's degrees of freedom go on the nesting level that holds its block, and a level with no prior stays at its estimate. One draw is shared across the occasions of a study, since the level is drawn once.
A nesting level is drawn as a whole, so a prior has to cover its level. A prior on part of a level – one of two independent blocks at `| id`, say – is an error, because drawing the level would redraw the other block too and correlate blocks the model declared independent, and there is nowhere to put a second degrees of freedom. Put the prior on the whole level instead.
Under a chunked solve (`file =`/`chunkSize =`) a prior on the population parameters is simulated, since that half of a prior is a `thetaMat`, which the one draw every chunk shares covers. A prior on an omega block is not, and is a clear error rather than a solve that quietly drops it.

A thetaMat that already carries the omegas:
The same joint draw works without an `ini({})` prior, for a `thetaMat` that came from a covariance step. `nonmem2rx` gives one directly, and so does a nlmixr2 fit:

```
mod <- nonmem2rx("run3.lst")
colnames(mod$thetaMat)
#> "t.CL" "GFRCL" ... "IIVCL" "omega1.2" "omega1.3" "IIVV1" ... "eps1" "sigma1.2"

rxSolve(mod, data.sim, nStud = 100, omegaSeparation = "tnpri")
```

`omegaSeparation = "tnpri"` says those entries are the omega itself and should be drawn from the `thetaMat` jointly with the thetas, instead of having their correlations redrawn by the `lkj`/`separation` strategies – which discard the off-diagonal entries the covariance step measured, and cannot carry a covariance between a theta and an omega entry at all. `sigmaSeparation = "tnpri"` does the same for the residual entries.
Columns are matched to entries by name, in whichever spelling produced them:

	diagonal	off-diagonal
nlmixr2 fit `$cov`	`om.eta.cl`	`cov.eta.cl.eta.v`
`nonmem2rx`	`eta.cl`	`omega1.2`, `omega.2.1`

This is opt-in rather than automatic: an eta-named column already means that eta's variance under the existing separation strategy, so it cannot change meaning on its own. `omega` has to be a matrix, since the drawn entries are added to it.

Value

An object (environment) of class `rxode2` (see Chambers and Temple Lang (2001)) consisting of the following list of strings and functions:

```
* `model` a character string holding the source model specification.
* `get.modelVars`a function that returns a list with 3 character
    vectors, `params`, `state`, and `lhs` of variable names used in the model
   specification. These will be output when the model is computed (i.e., the ODE solved by integration).

  * `solve`{this function solves (integrates) the ODE. This
      is done by passing the code to [rxSolve()].
```

```
    This is as if you called `rxSolve(rxode2object, ...)`,
    but returns a matrix instead of a rxSolve object.

    `params`: a numeric named vector with values for every parameter
    in the ODE system; the names must correspond to the parameter
    identifiers used in the ODE specification;

    `events`: an `eventTable` object describing the
    input (e.g., doses) to the dynamic system and observation
    sampling time points (see  [eventTable()]);

    `inits`: a vector of initial values of the state variables
    (e.g., amounts in each compartment), and the order in this vector
    must be the same as the state variables (e.g., PK/PD compartments);


    `stiff`: a logical (`TRUE` by default) indicating whether
    the ODE system is stiff or not.

    For stiff ODE systems (`stiff = TRUE`), `rxode2` uses
    the LSODA (Livermore Solver for Ordinary Differential Equations)
    Fortran package, which implements an automatic method switching
    for stiff and non-stiff problems along the integration interval,
    authored by Hindmarsh and Petzold (2003).

    For non-stiff systems (`stiff = FALSE`), `rxode2` uses `DOP853`,
    an explicit Runge-Kutta method of order 8(5, 3) of Dormand and Prince
    as implemented in C by Hairer and Wanner (1993).

    `trans_abs`: a logical (`FALSE` by default) indicating
    whether to fit a transit absorption term
    (TODO: need further documentation and example);

    `atol`: a numeric absolute tolerance (1e-08 by default);

    `rtol`: a numeric relative tolerance (1e-06 by default).

    The output of \dQuote{solve} is a matrix with as many rows as there
    are sampled time points and as many columns as system variables
    (as defined by the ODEs and additional assignments in the rxode2 model
        code).}

* `isValid` a function that (naively) checks for model validity,
    namely that the C object code reflects the latest model
    specification.
* `version` a string with the version of the `rxode2`
    object (not the package).
* `dynLoad` a function with one `force = FALSE` argument
```

```
    that dynamically loads the object code if needed.
* `dynUnload` a function with no argument that unloads
    the model object code.
* `delete` removes all created model files, including C and DLL files.
    The model object is no longer valid and should be removed, e.g.,
    `rm(m1)`.
* `run` deprecated, use `solve`.
* `get.index` deprecated.
* `getObj` internal (not user callable) function.
```

Note on strings in rxode2

Strings are converted to double values inside of rxode2, hence you can refer to them as an integer corresponding to the string value or the string value itself. For covariates these are calculated on the fly based on your data and you should likely not try this, though you should be aware. For strings defined in the model, this is fixed and both could be used.

For example:

```
if (APGAR == 10 || APGAR == 8 || APGAR == 9) {
    tAPGAR <- "High"
  } else if (APGAR == 1 || APGAR == 2 || APGAR == 3) {
    tAPGAR <- "Low"
  } else if (APGAR == 4 || APGAR == 5 || APGAR == 6 || APGAR == 7) {
    tAPGAR <- "Med"
  } else {
    tAPGAR<- "Med"
  }
```

Could also be replaced by:

```
if (APGAR == 10 || APGAR == 8 || APGAR == 9) {
    tAPGAR <- "High"
  } else if (APGAR == 1 || APGAR == 2 || APGAR == 3) {
    tAPGAR <- "Low"
  } else if (APGAR == 4 || APGAR == 5 || APGAR == 6 || APGAR == 7) {
    tAPGAR <- "Med"
  } else {
    tAPGAR<- 3
  }
```

Since "Med" is already defined

If you wanted you can pre-declare what levels it has (and the order) to give you better control of this:

```
levels(tAPGAR) <- c("Med", "Low", "High")
if (APGAR == 10 || APGAR == 8 || APGAR == 9) {
    tAPGAR <- 3
  } else if (APGAR == 1 || APGAR == 2 || APGAR == 3) {
```

```
  tAPGAR <- 2
} else if (APGAR == 4 || APGAR == 5 || APGAR == 6 || APGAR == 7) {
  tAPGAR <- 1
} else {
  tAPGAR<- 1
}
```

You can see that the number changed since the declaration change the numbers in each variable for `tAPGAR`. These `levels()` statements need to be declared before the variable occurs to ensure the numbering is consistent with what is declared.

Author(s)

Melissa Hallow, Wenping Wang and Matthew Fidler

References

Chamber, J. M. and Temple Lang, D. (2001) *Object Oriented Programming in R*. R News, Vol. 1, No. 3, September 2001. https://cran.r-project.org/doc/Rnews/Rnews_2001-3.pdf.

Hindmarsh, A. C. *ODEPACK, A Systematized Collection of ODE Solvers*. Scientific Computing, R. S. Stepleman et al. (Eds.), North-Holland, Amsterdam, 1983, pp. 55-64.

Petzold, L. R. *Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations*. Siam J. Sci. Stat. Comput. 4 (1983), pp. 136-148.

Hairer, E., Norsett, S. P., and Wanner, G. *Solving ordinary differential equations I, nonstiff problems*. 2nd edition, Springer Series in Computational Mathematics, Springer-Verlag (1993).

Plevyak, J. dparser, https://dparser.sourceforge.net/. Web. 12 Oct. 2015.

See Also

`eventTable()`, `et()`, `add.sampling()`, `add.dosing()`

Examples

```
mod <- function() {
  ini({
    KA   <- .291
    CL   <- 18.6
    V2   <- 40.2
    Q    <- 10.5
    V3   <- 297.0
    Kin  <- 1.0
    Kout <- 1.0
    EC50 <- 200.0
  })
  model({
    # A 4-compartment model, 3 PK and a PD (effect) compartment
    # (notice state variable names 'depot', 'centr', 'peri', 'eff')
    C2 <- centr/V2
```

```
    C3 <- peri/V3
    d/dt(depot) <- -KA*depot;
    d/dt(centr) <- KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  <-                   Q*C2 - Q*C3;
    d/dt(eff)   <- Kin - Kout*(1-C2/(EC50+C2))*eff;
    eff(0)      <- 1
  })
}

m1 <- rxode2(mod)
print(m1)

# Step 2 - Create the model input as an EventTable,
# including dosing and observation (sampling) events

# QD (once daily) dosing for 5 days.

qd <- et(amountUnits = "ug", timeUnits = "hours") |>
  et(amt = 10000, addl = 4, ii = 24)

# Sample the system hourly during the first day, every 8 hours
# then after
qd <- qd |> et(0:24) |>
  et(from = 24 + 8, to = 5 * 24, by = 8)

# Step 3 - solve the system

qd.cp <- rxSolve(m1, qd)

head(qd.cp)
```

`rxode2<-` *Set the function body of an rxUi object while retaining other object information (like data)*

Description

Set the function body of an rxUi object while retaining other object information (like data)

Usage

```
rxode2(x, envir = environment(x)) <- value

## S3 replacement method for class '`function`'
rxode2(x, envir = environment(x)) <- value

## Default S3 replacement method:
```

```
rxode2(x, envir = environment(x)) <- value

rxode(x, envir = environment(x)) <- value

RxODE(x, envir = environment(x)) <- value
```

Arguments

x	The rxUi object
envir	environment where the assignment occurs
value	the value that will be assigned

Value

The rxode2 ui/function

Examples

```
one.compartment <- function() {
  ini({
    tka <- log(1.57); label("Ka")
    tcl <- log(2.72); label("Cl")
    tv <- log(31.5); label("V")
    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)
    d/dt(depot) = -ka * depot
    d/dt(center) = ka * depot - cl / v * center
    cp = center / v
    cp ~ add(add.sd)
  })
}

two.compartment <- function() {
  ini({
    lka <- 0.45 ; label("Absorption rate (Ka)")
    lcl <- 1 ; label("Clearance (CL)")
    lvc  <- 3 ; label("Central volume of distribution (V)")
    lvp  <- 5 ; label("Peripheral volume of distribution (Vp)")
    lq  <- 0.1 ; label("Intercompartmental clearance (Q)")
    propSd <- 0.5 ; label("Proportional residual error (fraction)")
  })
  model({
    ka <- exp(lka)
    cl <- exp(lcl)
```

```
    vc <- exp(lvc)
    vp <- exp(lvp)
    q  <- exp(lq)
    kel <- cl/vc
    k12 <- q/vc
    k21 <- q/vp
    d/dt(depot) <- -ka*depot
    d/dt(central) <-  ka*depot - kel*central - k12*central + k21*peripheral1
    d/dt(peripheral1) <- k12*central - k21*peripheral1
    cp <- central / vc
    cp ~ prop(propSd)
  })
}

ui <- rxode2(one.compartment)

rxode2(ui) <- two.compartment
```

`rxode2parse` *Internal translation to get model variables list*

Description

Internal translation to get model variables list

Usage

```
rxode2parse(
  model,
  linear = FALSE,
  linCmtSens = c("linCmtA", "linCmtB"),
  verbose = FALSE,
  code = NULL,
  envir = parent.frame()
)
```

Arguments

`model`	Model (either file name or string)
`linear`	boolean indicating if linear compartment model should be generated from `linCmt()` (default FALSE)
`linCmtSens`	Linear compartment model sensitivity type
`verbose`	is a boolean indicating the type of model detected with `linCmt()` parsing
`code`	is a file name where the c code is written to (for testing purposes mostly, it needs `rxode2` to do anything fancy)
`envir`	is the environment to look for R user functions (defaults to parent environment)

Value

A rxModelVars object that has the model variables of a rxode2 syntax expression

Examples

```
rxode2parse("a=3")
```

`rxode2parseAssignTranslation` *This assigns the c level linkages for a roxde2 model*

Description

This assigns the c level linkages for a roxde2 model

Usage

```
rxode2parseAssignTranslation(df)
```

Arguments

`df`	data frame containing the character column names rxFun, fun, type, package, packageFun and the integer column names argMin and argMax

Value

Nothing called for side effects

Author(s)

Matthew L. Fidler

Examples

```
rxode2parseAssignTranslation(rxode2parseGetTranslation())
```

`rxode2parseD` *This gives the derivative table for rxode2*

Description

This will help allow registration of functions in `rxode2`

Usage

```
rxode2parseD()
```

Details

This environment is a derivative table;

For example:

Derivative(f(a,b,c), a) = fa() Derivative(f(a,b,c), b) = fb() Derivative(f(a,b,c), c) = fc()

Then the derivative table for `f` would be:

assign("f", list(fa(a,b,c), fb(a,b,c), fc(a,b,c)), rxode2parseD())

fa translates the arguments to the derivative with respect to a fb translates the arguments to the derivative with respect to b

If any of the list is NULL then rxode2 won't know how to take a derivative with respect to the argument.

If the list is shorter than the length of the arguments then the argument then the derivative of arguments that are not specified cannot be taken.

Value

Derivative table environment for rxode2

Author(s)

Matthew L. Fidler

`rxode2parseGetPackagesToLoad` *Control the packages that are loaded when a* `rxode2` *model dll is loaded*

Description

Control the packages that are loaded when a `rxode2` model dll is loaded

Usage

```
rxode2parseGetPackagesToLoad()

rxode2parseAssignPackagesToLoad(pkgs = rxode2parseGetPackagesToLoad())
```

Arguments

pkgs	The packages to make sure are loaded every time you load an rxode2 model.

Value

List of packages to load

Author(s)

Matthew Fidler

Examples

```
rxode2parseGetPackagesToLoad()

rxode2parseAssignPackagesToLoad(rxode2parseGetPackagesToLoad())
```

---

rxode2parseGetPointerAssignment

This function gets the currently assigned function pointer assignments

---

Description

This function gets the currently assigned function pointer assignments

Usage

```
rxode2parseGetPointerAssignment()
```

Value

The currently assigned pointer assignments

Author(s)

Matthew L. Fidler

Examples

```
rxode2parseGetTranslation()
```

rxode2parseGetTranslation

This function gets the currently assigned translations

Description

This function gets the currently assigned translations

Usage

```
rxode2parseGetTranslation()
```

Value

The currently assigned translations

Author(s)

Matthew L. Fidler

Examples

```
rxode2parseGetTranslation()
```

rxOmegaVarCovDeriv

Variance-covariance (non-Cholesky) Omega parameterization derivatives

Description

A non-Cholesky `Omega` path that differentiates with respect to the variance-covariance entries directly (for reporting SEs on the natural scale or building an analytic covariance over the `Omega` elements). Returns `Omega^{-1}`, `log|Omega|`, and their first (and optionally second) derivatives with respect to each free lower-triangular element `omega_{ab}`, via

Usage

```
rxOmegaVarCovDeriv(omega, order = 2L)
```

Arguments

`omega`	symmetric positive-definite random-effects covariance matrix.
`order`	integer; `1` for first derivatives only, `2` (default) to also return the second derivatives needed for the covariance Hessian.

Details

$$\partial \Omega^{-1} / \partial \omega_{ab} = -\Omega^{-1} E_{ab} \Omega^{-1}$$

$$\partial \log |\Omega| / \partial \omega_{ab} = \mathrm{tr}(\Omega^{-1} E_{ab})$$

where E_{ab} is the symmetric single-entry basis matrix.

Value

a list with `omegaInv`, `logDet`, the free-element index matrix `elements` (each row `c(a, b)`, `a >= b`), first derivatives `dOmegaInv` / `dLogDet`, and (when `order = 2`) second derivatives `d2OmegaInv` / `d2LogDet`.

Author(s)

Hidde van de Beek

`rxOptExpr` *Optimize rxode2 for computer evaluation*

Description

This optimizes rxode2 code for computer evaluation by only calculating redundant expressions once.

Usage

```
rxOptExpr(x, msg = "model", chunkLines = 40L, parallel = 0L)
```

Arguments

`x`	rxode2 model that can be accessed by rxNorm
`msg`	This is the name of type of object that rxode2 is optimizing that will in the message when optimizing. For example "model" will produce the following message while optimizing the model: finding duplicate expressions in model...
`chunkLines`	Integer; when positive (the default is 40), a model longer than this many lines is optimized in contiguous cost-balanced chunks of roughly this many lines instead of in a single pass; `0` always optimizes the whole model at once. Chunking amortizes the strongly superlinear cost of normalizing a large machine-generated model, which is what dominates the optimization of a sensitivity- or Jacobian-augmented model. Subexpressions are then shared only within a chunk, so the result is an equivalent model – the same states, parameters, solution and errors – carrying more temporaries. A chunk is a fragment, so if any chunk fails to optimize the whole model is optimized instead.

parallel — Integer; number of `mirai` daemons used to optimize the chunks in parallel, used only when the model is chunked. It carries the same semantics as `rxControl(cores=)`: `0` (the default) means the rxode2 thread setting `rxCores()` (`setRxThreads()`, `OMP_THREAD_LIMIT`); `1` runs the chunks serially. It is capped by the number of chunks and by `rxCores()`, so it will not oversubscribe. An existing `mirai` daemon pool is used as-is and left running; otherwise a pool is started for the call and shut down when it returns, and only when the model splits into at least 4 chunks, where the parallel win covers the startup.

Value

Optimized rxode2 model text. The order and type lhs and state variables is maintained while the evaluation is sped up. While parameters names are maintained, their order may be modified.

Author(s)

Matthew L. Fidler

---

rxord *Simulate ordinal value*

Description

Simulate ordinal value

Usage

```
rxord(...)
```

Arguments

... — the probabilities to be simulated. These should sum up to a number below one.

Details

The values entered into the 'rxord' simulation will simulate the probability of falling each group. If it falls outside of the specified probabilities, it will simulate the group (number of probabilities specified + 1)

Value

A number from 1 to the (number of probabilities specified + 1)

Author(s)

Matthew L. Fidler

Examples

```
# This will give values 1, and 2
rxord(0.5)
rxord(0.5)
rxord(0.5)
rxord(0.5)

# This will give values 1, 2 and 3
rxord(0.3, 0.3)
rxord(0.3, 0.3)
rxord(0.3, 0.3)
```

---

rxParams *Parameters specified by the model*

Description

This returns the model's parameters that are required to solve the ODE system, and can be used to pipe parameters into an rxode2 solve

Usage

```
rxParams(obj, ...)

## S3 method for class 'rxode2'
rxParams(
  obj,
  constants = TRUE,
  ...,
  params = NULL,
  inits = NULL,
  iCov = NULL,
  keep = NULL,
  thetaMat = NULL,
  omega = NULL,
  dfSub = NULL,
  sigma = NULL,
  dfObs = NULL,
  nSub = NULL,
  nStud = NULL
)

## S3 method for class 'rxSolve'
rxParams(
  obj,
  constants = TRUE,
```

```
  ...,
  params = NULL,
  inits = NULL,
  iCov = NULL,
  keep = NULL,
  thetaMat = NULL,
  omega = NULL,
  dfSub = NULL,
  sigma = NULL,
  dfObs = NULL,
  nSub = NULL,
  nStud = NULL
)

## S3 method for class 'rxEt'
rxParams(
  obj,
  ...,
  params = NULL,
  inits = NULL,
  iCov = NULL,
  keep = NULL,
  thetaMat = NULL,
  omega = NULL,
  dfSub = NULL,
  sigma = NULL,
  dfObs = NULL,
  nSub = NULL,
  nStud = NULL
)

rxParam(obj, ...)
```

Arguments

obj	rxode2 family of objects
...	Other arguments including scaling factors for each compartment. This includes S# = numeric will scale a compartment # by a dividing the compartment amount by the scale factor, like NONMEM.
constants	is a boolean indicting if constants should be included in the list of parameters. Currently rxode2 parses constants into variables in case you wish to change them without recompiling the rxode2 model.
params	a numeric named vector with values for every parameter in the ODE system; the names must correspond to the parameter identifiers used in the ODE specification;
inits	a vector of initial values of the state variables (e.g., amounts in each compartment), and the order in this vector must be the same as the state variables (e.g., PK/PD compartments);

iCov	A data frame of individual non-time varying covariates to combine with the events dataset. The iCov dataset has one covariate per ID and should match the event table
keep	Columns to keep from either the input dataset or the iCov dataset. With the iCov dataset, the column is kept once per line. For the input dataset, if any records are added to the data LOCF (Last Observation Carried forward) imputation is performed.
thetaMat	Named theta matrix.
omega	Estimate of Covariance matrix. When omega is a list, assume it is a block matrix and convert it to a full matrix for simulations. When omega is NA and you are using it with a rxode2 ui model, the between subject variability described by the omega matrix are set to zero.
dfSub	Degrees of freedom to sample the between subject variability matrix from the inverse Wishart distribution (scaled) or scaled inverse chi squared distribution.
sigma	Named sigma covariance or Cholesky decomposition of a covariance matrix. The names of the columns indicate parameters that are simulated. These are simulated for every observation in the solved system. When sigma is NA and you are using it with a rxode2 ui model, the unexplained variability described by the sigma matrix are set to zero.
dfObs	Degrees of freedom to sample the unexplained variability matrix from the inverse Wishart distribution (scaled) or scaled inverse chi squared distribution.
nSub	Number between subject variabilities (ETAs) simulated for every realization of the parameters.
nStud	Number virtual studies to characterize uncertainty in estimated parameters.

Value

When extracting the parameters from an rxode2 model, a character vector listing the parameters in the model.

Author(s)

Matthew L.Fidler

See Also

Other Query model information: rxDfdy(), rxInits(), rxLhs(), rxModelVars(), rxState()

---

rxParLoader *Parameter-loader flag on a model*

Description

A model that needs an externally-owned parameter injector – a registered par-loader, e.g. a package's neural-network weight loader – flags that injector's `"<package>:<function>"` name here. At solve setup only the loader with that name runs, so an injector never overwrites an unrelated model's `par_ptr` merely because it happens to be registered. The flag is stored on the ui (a sticky item, so it survives model piping) like `rxForcedPars()`; set it to `NULL` to clear.

Usage

```
rxParLoader(ui)

rxParLoader(ui) <- value
```

Arguments

`ui`	an rxode2 ui / model function.
`value`	a single loader name (`"<package>:<function>"`), or `NULL`.

Value

the getter returns the name (or `NULL`); the setter returns the ui.

Author(s)

Matthew L. Fidler

`rxParseSuppressMsg` *Respect suppress messages*

Description

This turns on the silent REprintf in C when `suppressMessages()` is turned on. This makes the `REprintf` act like `messages` in R, they can be suppressed with `suppressMessages()`

Usage

```
rxParseSuppressMsg()
```

Value

Nothing

Author(s)

Matthew Fidler

Examples

```
# rxParseSuppressMsg() is called with rxode2()

# Note the errors are output to the console

try(rxode2parse("d/dt(matt)=/3"), silent = TRUE)

# When using suppressMessages, the output is suppressed

suppressMessages(try(rxode2parse("d/dt(matt)=/3"), silent = TRUE))

# In rxode2, we use REprintf so that interrupted threads do not crash R
# if there is a user interrupt. This isn't captured by R's messages, but
# This interface allows the `suppressMessages()` to suppress the C printing
# as well

# If you  want to suppress messages from rxode2 in other packages, you can use
# this function
```

---

rxPkg *Creates a package from compiled rxode2 models*

Description

Creates a package from compiled rxode2 models

Usage

```
rxPkg(
  ...,
  package,
  wd = getwd(),
  action = c("install", "build", "binary", "create"),
  license = c("gpl3", "lgpl", "mit", "agpl3"),
  name = "Firstname Lastname",
  fields = list()
)
```

Arguments

...	Models to build a package from
package	String of the package name to create
wd	character string with a working directory where to create a subdirectory according to modName. When specified, a subdirectory named after the "modName.d" will be created and populated with a C file, a dynamic loading library, plus various other working files. If missing, the files are created (and removed) in the temporary directory, and the rxode2 DLL for the model is created in the current directory named rx_????_platform, for example rx_129f8f97fb94a87ca49ca8dafe691e1e_i386.dll

action	Type of action to take after package is created
license	is the type of license for the package.
name	Full name of author
fields	A named list of fields to add to DESCRIPTION, potentially overriding default values. See use_description() for how you can set personalized defaults using package options.

Value

this function returns nothing and is used for its side effects

Author(s)

Matthew Fidler

---

rxpois *Simulate random Poisson variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the sitmo threefry, this currently generates one random deviate from the uniform distribution to seed the engine threefry and then run the code.

Usage

```
rxpois(lambda, n = 1L, ncores = 1L)
```

Arguments

lambda	vector of (non-negative) means.
n	number of random values to return.
ncores	Number of cores for the simulation rxnorm simulates using the threefry sitmo generator. rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

poission random number deviates

Examples

```
## Use threefry engine

rxpois(lambda = 3, n = 10) # with rxpois you have to explicitly state n
rxpois(lambda = 3, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxpois(4) ## The first arguments are the lambda parameter


## This example uses `rxpois` directly in the model

rx <- function() {
  model({
    a <- rxpois(3)
  })
}

et <- et(1, id = 1:2)


s <- rxSolve(rx, et)
```

---

rxPp *Simulate a from a Poisson process*

Description

Simulate a from a Poisson process

Usage

```
rxPp(
  n,
  lambda,
  gamma = 1,
  prob = NULL,
  t0 = 0,
  tmax = Inf,
  randomOrder = FALSE
)
```

Arguments

n	Number of time points to simulate in the Poisson process
lambda	Rate of Poisson process
gamma	Asymmetry rate of Poisson process. When gamma=1.0, this simulates a homogenous Poisson process. When gamma<1.0, the Poisson process has more events early, when gamma > 1.0, the Poisson process has more events late in the process. When gamma is non-zero, the tmax should not be infinite but indicate the end of the Poisson process to be simulated. In most pharamcometric cases, this will be the end of the study. Internally this uses a rate of: l(t) = lambda*gamma*(t/tmax)^(gamma-1)
prob	When specified, this is a probability function with one argument, time, that gives the probability that a Poisson time t is accepted as a rejection time.
t0	the starting time of the Poisson process
tmax	the maximum time of the Poisson process
randomOrder	when TRUE randomize the order of the Poisson events. By default (FALSE) it returns the Poisson process is in order of how the events occurred.

Value

This returns a vector of the Poisson process times; If the dropout is >= tmax, then all the rest of the times are = tmax to indicate the dropout is equal to or after tmax.

Author(s)

Matthew Fidler

Examples

```
## Sample homogenous Poisson process of rate 1/10
rxPp(10, 1 / 10)

## Sample inhomogenous Poisson rate of 1/10

rxPp(10, 1 / 10, gamma = 2, tmax = 100)
```

```
## Typically the Poisson process times are in a sequential order,
## using randomOrder gives the Poisson process in random order

rxPp(10, 1 / 10, gamma = 2, tmax = 10, randomOrder = TRUE)

## This uses an arbitrary function to sample a non-homogenous Poisson process

rxPp(10, 1 / 10, prob = function(x) {
  1/(1+abs(x))
})
```

`rxPreferredDistributionName` *Change distribution name to the preferred distribution name term*

Description

This is determined by the internal preferred condition name list `.errIdenticalDists`

Usage

```
rxPreferredDistributionName(dist)
```

Arguments

`dist`	This is the input distribution

Value

Preferred distribution term

Author(s)

Matthew Fidler

Examples

```
rxPreferredDistributionName("dt")

rxPreferredDistributionName("add")

# can be vectorized

rxPreferredDistributionName(c("add","dt"))
```

rxPriorBuildSpec *Build the C spec* rxPriorLogDensityEval() *(and the C API) evaluates*

Description

The one-time, R-only (main-thread) half of the kernel: parses the prior column and hands the result to src/priorDensity.cpp as an rx_prior_spec_t in R vectors owned by the returned R external pointer, which R frees with it. rxPriorLogDensityEval() (inst/include/rxode2prior.h) – reachable through rxode2's C function-pointer table – then evaluates it with no R/Rcpp call of any kind, safe to call from inside an OpenMP-parallel objective/gradient evaluation.

Usage

```
rxPriorBuildSpec(ui, method = c("general", "nwpri", "tnpri"))
```

Arguments

ui	rxode2 ui model
method	"general" (default), "nwpri" or "tnpri"; see this file's header comment for what differs

Value

external pointer to a rx_prior_spec_t

Author(s)

Matthew L. Fidler

See Also

Other Assertions: assertCompartmentExists(), assertCompartmentName(), assertCompartmentNew(), assertRxUi(), assertVariableExists(), assertVariableNew(), rxPriorLogDensity(), rxPriorOmegaToCholOmegaI rxUiPriors(), testIniDf(), testRxUnbounded()

rxPriorLogDensity *Value and gradient of a model's prior log density, on the natural scale*

Description

Every estimation method that implements a prior needs the same thing: turn the `prior` column on `$iniDf` (`rxode2::rxUiPriors()`) into a log-density and its gradient at the *current* parameter value – as opposed to `rxSolve()`'s use of the same priors, which draws study-level variability around the model's initial estimate. No Jacobian for an optimizer's own unconstrained-scale reparameterization is applied here; that is specific to the estimation method's own parameterization and is its caller's responsibility to add (chain-ruling `gradTheta`/`gradOmega` through `d(natural)/d(unconstrained)`), the same way `adviJacLogDet()` (`nlmixr2est/src/inner.cpp`) already does for the full-Bayes ADVI path.

Usage

```
rxPriorLogDensity(
  ui,
  theta = NULL,
  omega = NULL,
  method = c("general", "nwpri", "tnpri")
)
```

Arguments

`ui`	rxode2 ui model
`theta`	named numeric vector of current population-parameter values; only the ones a prior is on need to be present
`omega`	current omega matrix, needed only when a prior touches an omega element or block; `NULL` otherwise. Must carry every eta in the model (like `ui$omega` itself, in any name order) even when only one of its blocks has a prior – the underlying C kernel addresses omega positionally (the model's own eta numbering), not by submatrix, so a smaller matrix containing only the referenced block is not enough
`method`	`"general"` (default), `"nwpri"` or `"tnpri"`; see this file's header comment for what differs. `"nwpri"` implements NONMEM's own `$PRIOR NWPRI` omega parameterization; `"tnpri"` is the same raw-omega `om.<eta>` treatment as `"general"` (Monolix's Bayesian-estimation assumption is on the natural parameter, same as here – `chol(Omega^-1)` is FOCEI's own internal affair, not this kernel's). Neither `"nwpri"` nor `"tnpri"` has a Cauchy (or any other non-normal univariate) analogue, so both refuse one. Under `"general"`, every univariate continuous 'lotri' family except `wiener()` is evaluated, truncated to the parameter's own bounds.

Details

This is a thin R-level convenience shim: the actual math is the pure C++ `rxPriorLogDensityEval()` (`inst/include/rxode2prior.h`), exposed through rxode2's C function-pointer table so a downstream package's own C++ objective can call it directly (no R/Rcpp touch, safe from inside an OpenMP-parallel region) instead of round-tripping through R for every evaluation the way this function does.

Value

list with `value` (scalar log density, summed over every prior term), `gradTheta` (named numeric, d/dtheta) and `gradOmega` (a matrix the same dimension as `omega`, d/dOmega, or `NULL` when `omega` was not given). `gradOmega` is entrywise, treating `omega[i, j]` and `omega[j, i]` as independent, the way `-0i %*% Psi %*% 0i`-style matrix calculus is usually reported. A caller whose free parameter moves an *off-diagonal* pair together (a Cholesky or log-Cholesky parameterization, say) needs `gradOmega[i, j] + gradOmega[j, i]`, which is `2 * gradOmega[i, j]` for `i != j` since `gradOmega` is itself symmetric – but NOT for a diagonal entry, whose own free parameter is `gradOmega[i, i]` alone; doubling it as well would be wrong

Author(s)

Matthew L. Fidler

See Also

Other Assertions: `assertCompartmentExists()`, `assertCompartmentName()`, `assertCompartmentNew()`, `assertRxUi()`, `assertVariableExists()`, `assertVariableNew()`, `rxPriorBuildSpec()`, `rxPriorOmegaToCholOmegaIr` `rxUiPriors()`, `testIniDf()`, `testRxUnbounded()`

Examples

```
one.cmt <- function() {
 ini({
   tka <- 0.45
   tcl <- 1
   tv <- 3.45
   eta.ka ~ 0.6
   add.sd <- c(0, 0.7)
   prior(tka) ~ dnorm(0, 10)
   prior(add.sd) ~ dcauchy(0, 5)
 })
 model({
   ka <- exp(tka + eta.ka)
   cl <- exp(tcl)
   v <- exp(tv)
   d/dt(depot) <- -ka*depot
   d/dt(center) <- ka*depot - cl/v*center
   cp <- center/v
   cp ~ add(add.sd)
 })
}

# priors need 'lotri' >= 1.0.5
if (utils::packageVersion("lotri") >= "1.0.5") {
  rxPriorLogDensity(one.cmt, theta=c(tka=0.1, add.sd=0.5))
}
```

`rxPriorOmegaToCholOmegaInvGrad` *Chain-rule a natural-scale omega gradient into FOCEI's cholOmegaInv scale*

Description

`rxPriorLogDensity()`'s `gradOmega` is always on the raw omega scale – the prior itself is defined there, for every method (see this file's header comment). A caller whose optimizer varies `chol(Omega^-1)` internally instead (FOCEI's `op_focei.cholOmegaInv`, `nlmixr2est/src/inner.cpp`) still needs that gradient chain-ruled into its own parameterization; a method that estimates Omega directly (SAEM) uses `gradOmega` as-is and does not need this. This is a thin shim over the pure C++ `rxPriorOmegaToCholOmegaInvGrad()` (`inst/include/rxode2prior.h`), exposed the same way for a downstream package's own C++ objective.

Usage

```
rxPriorOmegaToCholOmegaInvGrad(omega, gradOmega)
```

Arguments

`omega`	a single omega block (p x p numeric matrix, positive definite)
`gradOmega`	the corresponding block of `rxPriorLogDensity()`'s `gradOmega`, i.e. `gradOmega[block, block]` (p x p); need not be symmetric on entry (`rxPriorLogDensity()`'s own `gradOmega` always is, but this treats it entrywise and symmetrizes internally either way, since only a symmetric perturbation of `omega` is ever reachable by varying `U`)

Value

p x p matrix, the gradient with respect to the upper-triangular free elements of `U = chol(solve(omega))` (`omega^-1 = t(U) %*% U`); its strict lower triangle is exactly 0 since those entries address no free parameter of `U`. `NULL` if `omega` is not positive definite.

Author(s)

Matthew L. Fidler

See Also

Other Assertions: `assertCompartmentExists()`, `assertCompartmentName()`, `assertCompartmentNew()`, `assertRxUi()`, `assertVariableExists()`, `assertVariableNew()`, `rxPriorBuildSpec()`, `rxPriorLogDensity()`, `rxUiPriors()`, `testIniDf()`, `testRxUnbounded()`

Examples

```
omega <- lotri::lotri(eta.ka + eta.cl ~ c(0.1, 0.01, 0.1))
gradOmega <- matrix(c(1, 0.2, 0.2, -0.5), 2, 2)
rxPriorOmegaToCholOmegaInvGrad(omega, gradOmega)
```

`rxProgress` *rxode2 progress bar functions*

Description

`rxProgress` sets up the progress bar

Usage

```
rxProgress(num, core = 0L)

rxTick()

rxProgressStop(clear = TRUE)

rxProgressAbort(error = "Aborted calculation")
```

Arguments

`num`	Tot number of operations to track
`core`	Number of cores to show. If below 1, don't show number of cores
`clear`	Boolean telling if you should clear the progress bar after completion (as if it wasn't displayed). By default this is TRUE
`error`	With rxProgressAbort this is the error that is displayed

Details

`rxTick` is a progress bar tick

`rxProgressStop` stop progress bar

`rxProgressAbort` shows an abort if `rxProgressStop` wasn't called.

Value

All return NULL invisibly.

Author(s)

Matthew L. Fidler

Examples

```
f <- function() {
  on.exit({
    rxProgressAbort()
  })
  rxProgress(100)
  for (i in 1:100) {
    rxTick()
    Sys.sleep(1 / 100)
  }
  rxProgressStop()
}

f()
```

`rxRateDur` *Creates a rxRateDur object*

Description

This is primarily to display information about rate

Usage

```
rxRateDur(x)

## S3 method for class 'rxRateDur'
x[...]

as.rxRateDur(x)

## S3 method for class 'rxRateDur'
as.character(x, ...)

## S3 method for class 'rxRateDur'
x[[...]]
```

Arguments

`x`	rxRateDur data
`...`	Other parameters

Value

rxRateDur object

`rxRegisterUiAssembled` *Register or remove a ui assembly hook*

Description

Package developers register a function called with a freshly assembled `rxUi` before it is compressed. Use it to attach parse-time state to the model – for example values that a user-defined function (see `rxUdfUi()`) generated while the model was being parsed, which the `rxUdfUi()` return list has no field to carry.

Usage

```
rxRegisterUiAssembled(name, fn)

rxRemoveUiAssembled(name)
```

Arguments

`name`	Unique hook name; re-registering the same name replaces it.
`fn`	Function of one argument (the freshly assembled ui environment). The return value is ignored; errors are downgraded to a warning so a buggy hook cannot break unrelated model builds.

Details

The hook receives the ui **environment** and may assign into it in place; pair that with the `sticky` mechanism (see `rxForcedPars()`) so the slot survives model piping and `saveRDS()`. A hook must not reorder `rxModelVars(ui)$params`, must be cheap, and must be a no-op for models it does not own.

Value

Invisibly, the hook name (register) or `NULL` (remove).

Author(s)

Matthew L. Fidler

rxRegisterUiPrep *Register or remove a solve-time ui preparation hook*

Description

Package developers register a function called with the ui at the start of a ui solve, before parameter loaders run. Use it to rehydrate transient C-side state from serializable ui slots (so a ui saved to disk and reloaded in a fresh session solves correctly). The hook must be cheap and a no-op for models it does not own.

Usage

```
rxRegisterUiPrep(name, fn)

rxRemoveUiPrep(name)
```

Arguments

name	Unique hook name; re-registering the same name replaces it.
fn	Function of one argument (the rxUi being solved). Return value is ignored; errors are downgraded to a warning so a buggy hook cannot break unrelated solves.

Value

Invisibly, the hook name (register) or NULL (remove).

rxRemoveControl *rxRemoveControl options for UI object*

Description

rxRemoveControl options for UI object

Usage

```
rxRemoveControl(ui)
```

Arguments

ui	rxode2 ui object

Value

Nothing, called for side effects

Author(s)

Matthew L. Fidler

---

rxRename *Rename items inside of a* rxode2 *ui model*

Description

rxRename() changes the names of individual variables, lhs, and ode states using new_name = old_name syntax

Usage

```
rxRename(.data, ..., envir = parent.frame())

.rxRename(.data, ..., envir = parent.frame())

rename.rxUi(.data, ...)

rename.function(.data, ...)

## S3 method for class 'rxUi'
rxRename(.data, ...)

## S3 method for class '`function`'
rxRename(.data, ...)

## Default S3 method:
rxRename(.data, ...)
```

Arguments

.data	rxode2 ui function, named data to be consistent with dplyr::rename()
...	rename items
envir	Environment for evaluation

Details

This is similar to dplyr's rename() function. When dplyr is loaded, the s3 methods work for the ui objects.

Note that the .rxRename() is the internal function that is called when renaming and is likely not what you need to call unless you are writing your own extension of the function

Value

New model with items renamed

Author(s)

Matthew L. Fidler

Examples

```
ocmt <- function() {
  ini({
    tka <- exp(0.45) # Ka
    tcl <- exp(1) # Cl
    ## This works with interactive models
    ## You may also label the preceding line with label("label text")
    tv <- exp(3.45) # log V
    ## the label("Label name") works with all models
    add.sd <- 0.7
  })
  model({
    ka <- tka
    cl <- tcl
    v <- tv
    d/dt(depot) = -ka * depot
    d/dt(center) = ka * depot - cl / v * center
    cp = center / v
    cp ~ add(add.sd)
  })
}

ocmt |> rxRename(cpParent=cp)
```

---

rxReservedKeywords *A list and description of rxode2 supported reserved keywords*

Description

A list and description of rxode2 supported reserved keywords

Usage

```
rxReservedKeywords
```

Format

A data frame with 3 columns and 31 rows

Reserved Name Reserved Keyword Name

Meaning Reserved Keyword Meaning

Alias Keyword Alias

rxResidualError *A description of Rode2 supported residual errors*

Description

A description of Rode2 supported residual errors

Usage

```
rxResidualError
```

Format

A data frame with 6 columns and 183 rows

Error model A description of the type of residual error

Functional Form For additive and proportional what functional form is used

Transformation The type of transformation that is done on the DV and the prediction

code Example code for the residual error type

addProp The type of add+prop residual error default that would be equivalent

lhs what the left handed side of the specification represents, either a response variable, or a compartment specification

References

The transformation-based and autocorrelated (`ar()`, AR(1)) residual error models follow Karlsson MO, Beal SL, Sheiner LB. Three new residual error models for population PK/PD analyses. J Pharmacokinet Biopharm. 1995;23(6):651-672. doi:10.1007/BF02353466.

rxRmvn *Simulate from a (truncated) multivariate normal*

Description

This is simulated with the fast, thread-safe threefry simulator and can use multiple cores to generate the random deviates.

Usage

```
rxRmvn(
  n,
  mu = NULL,
  sigma,
  lower = -Inf,
  upper = Inf,
  ncores = 1,
  isChol = FALSE,
  keepNames = TRUE,
  a = 0.4,
  tol = 2.05,
  nlTol = 1e-10,
  nlMaxiter = 100L
)
```

Arguments

n	Number of random row vectors to be simulated OR the matrix to use for simulation (faster).
mu	mean vector
sigma	Covariance matrix for multivariate normal or a list of covariance matrices. If a list of covariance matrix, each matrix will simulate n matrices and combine them to a full matrix
lower	is a vector of the lower bound for the truncated multivariate norm
upper	is a vector of the upper bound for the truncated multivariate norm
ncores	Number of cores used in the simulation
isChol	A boolean indicating if sigma is a cholesky decomposition of the covariance matrix.
keepNames	Keep the names from either the mean or covariance matrix.
a	threshold for switching between methods; They can be tuned for maximum speed; There are three cases that are considered: case 1: a < l < u case 2: l < u < -a case 3: otherwise where l=lower and u = upper
tol	When case 3 is used from the above possibilities, the tol value controls the acceptance rejection and inverse-transformation; When abs(u-l)>tol, uses accept-reject from randn
nlTol	Tolerance for newton line-search
nlMaxiter	Maximum iterations for newton line-search

Value

If n==integer (default) the output is an (n x d) matrix where the i-th row is the i-th simulated vector.

If is.matrix(n) then the random vector are store in n, which is provided by the user, and the function returns NULL invisibly.

Author(s)

Matthew Fidler, Zdravko Botev and some from Matteo Fasiolo

References

John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw (2011). Parallel Random Numbers: As Easy as 1, 2, 3. D. E. Shaw Research, New York, NY 10036, USA.

The thread safe multivariate normal was inspired from the mvnfast package by Matteo Fasiolo https://CRAN.R-project.org/package=mvnfast

The concept of the truncated multivariate normal was taken from Zdravko Botev Botev (2017) doi:10.1111/rssb.12162 and Botev and L'Ecuyer (2015) doi:10.1109/WSC.2015.7408180 and converted to thread safe simulation;

Examples

```
## From mvnfast
## Unlike mvnfast, uses threefry simulation

d <- 5
mu <- 1:d

# Creating covariance matrix
tmp <- matrix(rnorm(d^2), d, d)
mcov <- tcrossprod(tmp, tmp)


set.seed(414)
rxRmvn(4, 1:d, mcov)

set.seed(414)
rxRmvn(4, 1:d, mcov)

set.seed(414)
rxRmvn(4, 1:d, mcov, ncores = 2) # r.v. generated on the second core are different

###### Here we create the matrix that will hold the simulated
#  random variables upfront.
A <- matrix(NA, 4, d)
class(A) <- "numeric" # This is important. We need the elements of A to be of class "numeric".

set.seed(414)
rxRmvn(A, 1:d, mcov, ncores = 2) # This returns NULL ...
A # ... but the result is here
```

```
## You can also simulate from a truncated normal:

rxRmvn(10, 1:d, mcov, lower = 1:d - 1, upper = 1:d + 1)


# You can also simulate from different matrices (if they match
# dimensions) by using a list of matrices.

matL <- lapply(1:4, function(...) {
  tmp <- matrix(rnorm(d^2), d, d)
  tcrossprod(tmp, tmp)
})


rxRmvn(4, setNames(1:d, paste0("a", 1:d)), matL)
```

---

rxS *Load a model into a symengine environment*

Description

Load a model into a symengine environment

Usage

```
rxS(
  x,
  doConst = TRUE,
  promoteLinSens = FALSE,
  envir = parent.frame(),
  pkTime = FALSE
)
```

Arguments

x	rxode2 object
doConst	Load constants into the environment as well.
promoteLinSens	Promote solved linear compartment systems to sensitivity-based solutions.
envir	default is NULL; Environment to put symengine variables in.
pkTime	When TRUE, time in statements that do not depend on a state is loaded as rx_time_pk, defined by rx_time_pk~t in ..lhs0, so it stays separate from the time in d/dt() after those statements are inlined; rxSolve(..., nonmem = TRUE) then reads it as the record time. Only for callers that write ..lhs0 into the model they build.

Value

rxode2/symengine environment

Author(s)

Matthew Fidler

`rxSensMatExp` *Differentiate and expand a matrix exponential model with forward sensitivities*

Description

The system is split the way `indLin()` splits it: a rate matrix that is constant in the states, expressed as `k_from_to` micro-constants, plus an `indLin()` forcing carrying everything else. Each sensitivity compartment gets the same rate matrix, the `(dA/dp).X` cross terms as non-depleting transfers, and its own forcing `d(f)/dp + (df/dy).S^p`.

Usage

```
rxSensMatExp(
  model,
  calcSens,
  calcSens2 = NULL,
  calcSens3 = NULL,
  doConst = FALSE,
  env = NULL
)
```

Arguments

`model`	rxode2 model, text, or function
`calcSens`	A character vector of parameter names for which sensitivities should be calculated.
`calcSens2`	character vector (or `NULL`) requesting second-order sensitivities `rx__sens_<x>_BY_<p>_BY_<q>__` (`p` over `calcSens`, `q` over `calcSens2`; every `calcSens2` element must also be in `calcSens`). Ignored for `linCmt()` states (those use Stan forward-AD).
`calcSens3`	character vector (or `NULL`) requesting third-order sensitivities `rx__sens_<x>_BY_<p>_BY_<q>_BY_<r>__` (`r` over `calcSens3`). Requires `calcSens2`; every `calcSens3` element must also be in `calcSens2`.
`doConst`	Replace constants with values; By default this is `FALSE`.
`env`	A pre-loaded symengine environment (from `.rxLoadPrune()`) to reuse instead of reloading `model`; when `NULL` it is built internally.

Value

A character string representing the matrix exponential sensitivity-expanded model code

Author(s)

Matthew L. Fidler

`rxSetActiveParLoader` *Activate a registered parameter loader for solves*

Description

`rxParLoader()` flags the loader a MODEL owns, which `rxSolve()` applies for that model's solve. These activate a loader directly, for solves that do not go through `rxSolve.rxUi()` – an estimation method's internal solves, for example, where the model being solved is assembled by the method rather than handed to it.

Usage

```
rxSetActiveParLoader(name)

rxClearActiveParLoader()
```

Arguments

`name`	Loader name, `"<package>:<function>"`, as registered from C with `rxRegisterParLoaderNamed()`.

Details

Only the registered loader whose name is active runs, so a package can inject its own parameters without touching an unrelated model's `par_ptr`. Activate for as short a span as possible and clear afterwards (`on.exit()`), because the flag is global to the session, not attached to a model.

Value

Invisibly `NULL`; called for the side effect.

Author(s)

Matthew L. Fidler

See Also

`rxParLoader()`, `rxForcedPars()`

rxSetControl *rxSetControl options for UI object*

Description

rxSetControl options for UI object

Usage

```
rxSetControl(ui, control)
```

Arguments

ui	rxode2 ui object
control	Default value

Value

Nothing, called for side effects

Author(s)

Matthew L. Fidler

rxSetCovariateNamesForPiping *Assign covariates for piping*

Description

Assign covariates for piping

Usage

```
rxSetCovariateNamesForPiping(covariates = NULL)
```

Arguments

covariates	NULL (for no covariates), or the list of covariates. nlmixr uses this function to set covariates if you pipe from a nlmixr fit.

Value

Nothing, called for side effects

Author(s)

Matthew L. Fidler

Examples

```
# First set the name of known covariates
# Note this is case sensitive

rxSetCovariateNamesForPiping(c("WT","HT", "TC"))

one.compartment <- function() {
 ini({
   tka <- 0.45 ; label("Log Ka")
   tcl <- 1 ; label("Log Cl")
   tv <- 3.45 ; label("Log V")
   eta.ka ~ 0.6
   eta.cl ~ 0.3
   eta.v ~ 0.1
   add.err <- 0.7
 })
 model({
   ka <- exp(tka + eta.ka)
   cl <- exp(tcl + eta.cl)
   v <- exp(tv + eta.v)
   d / dt(depot) <- -ka * depot
   d/dt(depot) <- -ka * depot
   d / dt(center) <- ka * depot - cl / v * center
   cp <- center / v
   cp ~ add(add.err)
 })
}

# now TC is detected as a covariate instead of a population parameter

one.compartment |>
  model({ka <- exp(tka + eta.ka + TC * cov_C)})

# You can turn it off by simply adding it back

rxSetCovariateNamesForPiping()

one.compartment |>
  model({ka <- exp(tka + eta.ka + TC * cov_C)})

# The covariates you set with `rxSetCovariateNamesForPiping()`
# are turned off every time you solve (or fit in nlmixr)
```

`rxSetIni0` *Set Initial conditions to time zero instead of the first observed/dosed time*

Description

Set Initial conditions to time zero instead of the first observed/dosed time

Usage

```
rxSetIni0(ini0 = TRUE)
```

Arguments

ini0	When TRUE (default), set initial conditions to time zero. Otherwise the initial conditions are the first observed time.

Value

the boolean ini0, though this is called for its side effects

---

rxSetPipingAuto *Set the variables for the model piping automatic covarite selection*

Description

Set the variables for the model piping automatic covarite selection

Usage

```
rxSetPipingAuto(
 thetamodelVars = rex::rex(or("tv", "t", "pop", "POP", "Pop", "TV", "T", "cov", "err",
    "eff")),
  covariateExceptions = rex::rex(start, or("wt", "sex", "crcl", "kout"), end),
 etaParts = c("eta", "ETA", "Eta", "ppv", "PPV", "Ppv", "iiv", "Iiv", "bsv", "Bsv",
    "BSV", "bpv", "Bpv", "BPV", "psv", "PSV", "Psv")
)
```

Arguments

thetamodelVars	This is the prefixes for the theta model variables in a regular expression
covariateExceptions	This is a regular expression of covariates that should always be covariates
etaParts	This is the list of eta prefixes/post-fixes that identify a variable as a between subject variability

Details

This is called once at startup to set the defaults, though you can change this if you wish so that piping can work differently for your individual setup

Value

Nothing, called for side effects

Author(s)

Matthew L. Fidler

`rxSetProd` *Defunct setting of product*

Description

Defunct setting of product

Usage

```
rxSetProd(type = c("long double", "double", "logify"))
```

Arguments

`type`	used to be type of product

Value

nothing

`rxSetProgressBar` *Set timing for progress bar*

Description

Set timing for progress bar

Usage

```
rxSetProgressBar(seconds = 1)
```

Arguments

`seconds`	This sets the number of seconds that need to elapse before drawing the next segment of the progress bar. When this is zero or below this turns off the progress bar.

Value

nothing, used for side effects

Author(s)

Matthew Fidler

`rxSetSeed` *Set the parallel seed for rxode2 random number generation*

Description

This sets the seed for the rxode2 parallel random number generation. If set, then whenever a seed is set for the threefry or vandercorput simulation engine, it will use this seed, increment for the number of seeds and continue with the sequence the next time the random number generator is called.

Usage

```
rxSetSeed(seed)
```

Arguments

`seed`	An integer that represents the rxode2 parallel and internal random number generator seed. When positive, use this seed for random number generation and increment and reseed any parallel or new engines that are being called. When negative, turn off the rxode2 seed and generate a seed from the R's uniform random number generator. Best practice is to set this seed.

Details

In contrast, when this is not called, the time that the vandercorput or threefry simulation engines are seeded it comes from a uniform random number generated from the standard R random seed. This may cause a duplicate seed based on the R seed state. This means that there could be correlations between simulations that do not exist This will avoid the birthday problem picking exactly the same seed using the seed state of the R random number generator. The more times the seed is called, the more likely this becomes.

Value

Nothing, called for its side effects

Author(s)

Matthew Fidler

References

JD Cook. (2016). Random number generator seed mistakes. https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/

See Also

rxGetSeed, rxWithSeed, rxWithPreserveSeed

Examples

```
rxSetSeed(42)

# seed with generator 42
rxnorm()

# Use R's random number generator
rnorm(1)

rxSetSeed(42)

# reproduces the same number
rxnorm()

# But R's random number is not the same

rnorm(1)

# If we reset this to use the R's seed
# (internally rxode2 uses a uniform random number to span seeds)
# This can lead to duplicate sequences and seeds

rxSetSeed(-1)

# Now set seed works for both.

# This is not recommended, but illustrates the different types of
# seeds that can be generated.

set.seed(42)

rxnorm()

rnorm(1)

set.seed(42)

rxnorm()

rnorm(1)
```

---

rxSetSum *Defunct setting of sum*

Description

Defunct setting of sum

Usage

```
rxSetSum(type = c("pairwise", "fsum", "kahan", "neumaier", "c"))
```

Arguments

type	used to be type of product

Value

nothing

---

rxShiny *Use Shiny to help develop an rxode2 model*

Description

Use Shiny to help develop an rxode2 model

Usage

```
rxShiny(
  object,
  params = NULL,
  events = NULL,
  inits = NULL,
  ...,
  data = data.frame()
)

## S3 method for class 'rxSolve'
rxShiny(
  object,
  params = NULL,
  events = NULL,
  inits = NULL,
  ...,
  data = data.frame()
)

## Default S3 method:
rxShiny(
  object = NULL,
  params = NULL,
```

```
  events = NULL,
  inits = NULL,
  ...,
  data = data.frame()
)
```

Arguments

object	A rxode2 family of objects. If not supplied a 2-compartment indirect effect model is used. If it is supplied, use the model associated with the rxode2 object for the model exploration.
params	Initial parameters for model
events	Event information (currently ignored)
inits	Initial estimates for model
...	Other arguments passed to rxShiny. Currently doesn't do anything.
data	Any data that you would like to plot. If the data has a `time` variable as well as a compartment or calculated variable that matches the rxode2 model, the data will be added to the plot of a specific compartment or calculated variable.

Value

Nothing; Starts a shiny server

Author(s)

Zufar Mulyukov and Matthew L. Fidler

---

rxSimThetaOmega *Simulate Parameters from a Theta/Omega specification*

Description

Simulate Parameters from a Theta/Omega specification

Usage

```
rxSimThetaOmega(
  params = NULL,
  omega = NULL,
  omegaDf = NULL,
  omegaLower = as.numeric(c(R_NegInf)),
  omegaUpper = as.numeric(c(R_PosInf)),
  omegaIsChol = FALSE,
  omegaSeparation = "auto",
  omegaXform = 1L,
  nSub = 1L,
```

```
  thetaMat = NULL,
  thetaLower = as.numeric(c(R_NegInf)),
  thetaUpper = as.numeric(c(R_PosInf)),
  thetaDf = NULL,
  thetaIsChol = FALSE,
  nStud = 1L,
  sigma = NULL,
  sigmaLower = as.numeric(c(R_NegInf)),
  sigmaUpper = as.numeric(c(R_PosInf)),
  sigmaDf = NULL,
  sigmaIsChol = FALSE,
  sigmaSeparation = "auto",
  sigmaXform = 1L,
  nCoresRV = 1L,
  nObs = 1L,
  dfSub = 0,
  dfObs = 0,
  simSubjects = TRUE,
  simVariability = as.logical(c(NA_LOGICAL))
)
```

Arguments

params	Named Vector of rxode2 model parameters
omega	Estimate of Covariance matrix. When omega is a list, assume it is a block matrix and convert it to a full matrix for simulations. When omega is NA and you are using it with a rxode2 ui model, the between subject variability described by the omega matrix are set to zero.
omegaDf	The degrees of freedom of a t-distribution for simulation. By default this is NULL which is equivalent to Inf degrees, or to simulate from a normal distribution instead of a t-distribution.
omegaLower	Lower bounds for simulated ETAs (by default -Inf)
omegaUpper	Upper bounds for simulated ETAs (by default Inf)
omegaIsChol	Indicates if the omega supplied is a Cholesky decomposed matrix instead of the traditional symmetric matrix.
omegaSeparation	Omega separation strategy Tells the type of separation strategy when simulating covariance with parameter uncertainty with standard deviations modeled in the thetaMat matrix. • "lkj" simulates the correlation matrix from the rLKJ1 matrix with the distribution parameter eta equal to the degrees of freedom nu by (nu-1)/2 • "separation" simulates from the identity inverse Wishart covariance matrix with nu degrees of freedom. This is then converted to a covariance matrix and augmented with the modeled standard deviations. While computationally more complex than the "lkj" prior, it performs better when the covariance matrix size is greater or equal to 10

- "auto" chooses "lkj" when the dimension of the matrix is less than 10 and "separation" when greater than equal to 10.
- "tnpri" does not use a separation strategy at all: the omega entries themselves are carried in the thetaMat and are drawn from it jointly with the thetas, which is what a covariance step from NONMEM or 'nlmixr2' gives and what NONMEM calls TNPRI. This keeps the off diagonal entries and the covariances between the thetas and the omega entries, which the strategies above redraw or discard. omega has to be a matrix, since the draws are added to it, and the thetaMat columns are matched to entries by name – om.eta.cl or eta.cl for a diagonal, and cov.eta.cl.eta.v or omega2.1 for an off diagonal. A drawn matrix that is not positive definite is redrawn, see priorPdRetry. dfSub is not used: an entry the thetaMat gives no variance stays at its estimate.

omegaXform When taking omega values from the thetaMat simulations (using the separation strategy for covariance simulation), how should the thetaMat values be turned int standard deviation values:

- identity This is when standard deviation values are directly modeled by the params and thetaMat matrix
- variance This is when the params and thetaMat simulates the variance that are directly modeled by the thetaMat matrix
- log This is when the params and thetaMat simulates log(sd)
- nlmixrSqrt This is when the params and thetaMat simulates the inverse cholesky decomposed matrix with the x\^2 modeled along the diagonal. This only works with a diagonal matrix.
- nlmixrLog This is when the params and thetaMat simulates the inverse cholesky decomposed matrix with the exp(x\^2) along the diagonal. This only works with a diagonal matrix.
- nlmixrIdentity This is when the params and thetaMat simulates the inverse cholesky decomposed matrix. This only works with a diagonal matrix.

nSub Number between subject variabilities (ETAs) simulated for every realization of the parameters.

thetaMat Named theta matrix.

thetaLower Lower bounds for simulated population parameter variability (by default -Inf)

thetaUpper Upper bounds for simulated population unexplained variability (by default Inf)

thetaDf The degrees of freedom of a t-distribution for simulation. By default this is NULL which is equivalent to Inf degrees, or to simulate from a normal distribution instead of a t-distribution.

thetaIsChol Indicates if the theta supplied is a Cholesky decomposed matrix instead of the traditional symmetric matrix.

nStud Number virtual studies to characterize uncertainty in estimated parameters.

sigma Named sigma covariance or Cholesky decomposition of a covariance matrix. The names of the columns indicate parameters that are simulated. These are simulated for every observation in the solved system. When sigma is NA and you are using it with a rxode2 ui model, the unexplained variability described by the sigma matrix are set to zero.

`sigmaLower` Lower bounds for simulated unexplained variability (by default -Inf)

`sigmaUpper` Upper bounds for simulated unexplained variability (by default Inf)

`sigmaDf` Degrees of freedom of the sigma t-distribution. By default it is equivalent to `Inf`, or a normal distribution.

`sigmaIsChol` Boolean indicating if the sigma is in the Cholesky decomposition instead of a symmetric covariance

`sigmaSeparation`

separation strategy for sigma;

Tells the type of separation strategy when simulating covariance with parameter uncertainty with standard deviations modeled in the `thetaMat` matrix.

- `"lkj"` simulates the correlation matrix from the `rLKJ1` matrix with the distribution parameter `eta` equal to the degrees of freedom `nu` by `(nu-1)/2`
- `"separation"` simulates from the identity inverse Wishart covariance matrix with `nu` degrees of freedom. This is then converted to a covariance matrix and augmented with the modeled standard deviations. While computationally more complex than the `"lkj"` prior, it performs better when the covariance matrix size is greater or equal to 10
- `"auto"` chooses `"lkj"` when the dimension of the matrix is less than 10 and `"separation"` when greater than equal to 10.
- `"tnpri"` draws the sigma entries jointly from the `thetaMat`, the same way `omegaSeparation="tnpri"` does for the omega; `dfObs` is not used.

`sigmaXform` When taking `sigma` values from the `thetaMat` simulations (using the separation strategy for covariance simulation), how should the `thetaMat` values be turned int standard deviation values:

- `identity` This is when standard deviation values are directly modeled by the `params` and `thetaMat` matrix
- `variance` This is when the `params` and `thetaMat` simulates the variance that are directly modeled by the `thetaMat` matrix
- `log` This is when the `params` and `thetaMat` simulates `log(sd)`
- `nlmixrSqrt` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix with the `x\^2` modeled along the diagonal. This only works with a diagonal matrix.
- `nlmixrLog` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix with the `exp(x\^2)` along the diagonal. This only works with a diagonal matrix.
- `nlmixrIdentity` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix. This only works with a diagonal matrix.

`nCoresRV` Number of cores used for the simulation of the sigma variables. By default this is 1. To reproduce the results you need to run on the same platform with the same number of cores. This is the reason this is set to be one, regardless of what the number of cores are used in threaded ODE solving.

`nObs` Number of observations to simulate (with `sigma` matrix)

`dfSub` Degrees of freedom to sample the between subject variability matrix from the inverse Wishart distribution (scaled) or scaled inverse chi squared distribution.

dfObs	Degrees of freedom to sample the unexplained variability matrix from the inverse Wishart distribution (scaled) or scaled inverse chi squared distribution.
simSubjects	boolean indicated rxode2 should simulate subjects in studies (TRUE, default) or studies (FALSE)
simVariability	determines if the variability is simulated. When NA (default) this is determined by the solver.

Value

a data frame with the simulated subjects

Author(s)

Matthew L.Fidler

---

rxSolve *Options, Solving & Simulation of an ODE/solved system*

Description

This uses rxode2 family of objects, file, or model specification to solve a ODE system. There are many options for a solved rxode2 model, the first are the required object, and events with the some-times optional params and inits.

Usage

```
rxSolve(
  object,
  params = NULL,
  events = NULL,
  inits = NULL,
  scale = NULL,
 method = c("liblsoda", "lsoda", "dop853", "indLin", "f78", "rk4", "ck54", "ab", "abm",
  "dop5", "bs", "ros4", "iem", "sem", "sb3a", "sb3am4", "vv", "mm", "em", "cvode",
  "trapz", "ssp3", "f32", "rk43", "dop54", "vern65", "vern76", "dop87", "vern98",
  "ros43", "ros6", "backwardEuler", "gauss6", "iiic6", "radauiia5", "geng5", "sdirk43",
  "euler", "midpoint", "heun", "ssp22", "rk3", "ssp53", "s4", "r4", "ls44", "ls54",
   "ssp54", "s5", "rk5", "c5", "l5", "lk5a", "lk5b", "b6", "s7", "s8_10", "cv8",
    "s8_12", "s10",
     "z10", "o10", "h10", "dp54", "v65e", "v76e", "dp87", "v98e",
  "ssp33", "bs32", "ssp43", "f45", "t54", "s54", "pp54", "pp54b", "bs54", "ss54",
  "dp65", "c65", "tp64", "v65r", "v65", "dverk65", "tf65", "tp75", "tmy7", "tmy7s",
  "v76r", "ss76", "v78", "dverk78", "dp85", "tp86", "v87e", "v87r", "ev87", "k87",
  "f89", "v89", "t98a", "v98r", "s98", "f108", "c108", "b109", "s1110a", "f1210",
  "o129", "f1412", "lsode", "bdf", "rk4s", "eulers", "midpoints", "heuns", "dop5s",
   "dop853s", "ck54s", "bs32s", "vern65s",
    "vern76s", "dop87s", "f78s", "ros4s",
```

```
 "radauiia5s", "backwardEulers", "gauss6s", "sdirk43s", "iiic6s", "ros43s", "ros6s",
 "geng5s", "rk3s", "rk43s", "cvodesadj", "liblsodaadj", "abs", "dop54s", "dp54s",
 "vern98s", "f45s", "t54s", "pp54s", "pp54bs", "bs54s", "ss54s", "dp65s", "c65s",
 "tp64s", "v65rs", "dverk65s", "tf65s", "tp75s", "tmy7sadj", "tmy7adj", "v76rs",
 "ss76s", "v78s", "dverk78s", "dp85s", "tp86s", "v87es", "v87rs", "ev87s", "k87s",
  "v89s", "t98as", "v98rs", "s98s", "c108s", "b109s",
   "s1110as", "o129s"),
sigdig = NULL,
atol = 1e-08,
rtol = 1e-06,
maxsteps = 70000L,
hmin = 0,
hmax = NA_real_,
hmaxSd = 0,
hini = 0,
maxordn = 12L,
maxords = 5L,
order = 5L,
...,
cores,
covsInterpolation = c("locf", "linear", "nocb", "midpoint"),
naInterpolation = c("locf", "nocb"),
keepInterpolation = c("na", "locf", "nocb"),
addCov = TRUE,
sigma = NULL,
sigmaDf = NULL,
sigmaLower = -Inf,
sigmaUpper = Inf,
nCoresRV = 1L,
sigmaIsChol = FALSE,
sigmaSeparation = c("auto", "lkj", "separation", "tnpri"),
sigmaXform = c("identity", "variance", "log", "nlmixrSqrt", "nlmixrLog",
  "nlmixrIdentity"),
nDisplayProgress = 10000L,
amountUnits = NA_character_,
timeUnits = "hours",
theta = NULL,
thetaLower = -Inf,
thetaUpper = Inf,
eta = NULL,
addDosing = FALSE,
stateTrim = Inf,
updateObject = FALSE,
omega = NULL,
omegaDf = NULL,
omegaIsChol = FALSE,
omegaSeparation = c("auto", "lkj", "separation", "tnpri"),
omegaXform = c("variance", "identity", "log", "nlmixrSqrt", "nlmixrLog",
```

```
  "nlmixrIdentity"),
 omegaLower = -Inf,
 omegaUpper = Inf,
 nSub = 1L,
 thetaMat = NULL,
 thetaDf = NULL,
 thetaIsChol = FALSE,
 nStud = 1L,
 dfSub = 0,
 dfObs = 0,
returnType = c("rxSolve", "matrix", "data.frame", "data.frame.TBS", "data.table",
  "tbl", "tibble"),
 seed = NULL,
 nsim = NULL,
 minSS = 10L,
 maxSS = 10000L,
 infSSstep = 12,
 strictSS = TRUE,
 istateReset = TRUE,
 subsetNonmem = TRUE,
 maxAtolRtolFactor = 0.1,
 from = NULL,
 to = NULL,
 by = NULL,
 length.out = NULL,
 iCov = NULL,
 keep = NULL,
 indLinPhiTol = 1e-07,
 indLinPhiM = 0L,
 indLinMatExpType = c("Al-Mohy", "expokit", "arma", "taylor"),
 indLinMatExpOrder = 6L,
 drop = NULL,
 idFactor = TRUE,
 mxhnil = 0,
 hmxi = 0,
 warnIdSort = TRUE,
 warnDrop = TRUE,
 ssAtol = 1e-08,
 ssRtol = 1e-06,
 safeZero = TRUE,
 safeLog = TRUE,
 safePow = TRUE,
 sumType = c("pairwise", "fsum", "kahan", "neumaier", "c"),
 prodType = c("long double", "double", "logify"),
 resample = NULL,
 resampleID = TRUE,
 maxwhile = 1e+05,
 atolSens = 1e-08,
```

```
      rtolSens = 1e-06,
      ssAtolSens = 1e-08,
      ssRtolSens = 1e-06,
      simVariability = NA,
      nLlikAlloc = NULL,
      useStdPow = FALSE,
      naTimeHandle = c("ignore", "warn", "error"),
      addlKeepsCov = FALSE,
      addlDropSs = TRUE,
      ssAtDoseTime = TRUE,
      ss2cancelAllPending = FALSE,
      ssSolved = TRUE,
      linCmtSensType = c("auto", "endpoint5", "endpoint5G", "forward3", "forward3G", "AD",
        "ADm", "ADr", "central", "forward", "forwardG", "forwardH", "centralH", "forward3H",
          "endpointH5", "forwardG"),
      linCmtSensH = 1e-04,
      linCmtSensPhi = TRUE,
      nonmem = FALSE,
      linCmtGillFtol = 0,
      linCmtGillK = 20L,
      linCmtGillStep = 4,
      linCmtGillRtol = sqrt(.Machine$double.eps),
      linCmtShiErr = sqrt(.Machine$double.eps),
      linCmtShiMax = 20L,
      linCmtScale = FALSE,
      linCmtHcmt = NULL,
      linCmtHmeanI = c("geometric", "arithmetic", "harmonic"),
      linCmtHmeanO = c("geometric", "arithmetic", "harmonic"),
      linCmtSuspect = 1e-06,
      linCmtForwardMax = 2L,
      indOwnAlloc = NA,
      maxExtra = 1000L,
      tolFactor = NULL,
      serializeFile = NULL,
      dense = FALSE,
      cvodeLinSolver = c("dense", "band", "gmres", "bicgstab", "tfqmr"),
      autoSwitchNonstifftol = 9/10,
      autoSwitchStifftol = 9/10,
      autoSwitchDtfac = 2,
      autoSwitchMaxStiff = 10L,
      autoSwitchMaxNonstiff = 3L,
      autoSwitchStiffFirst = FALSE,
      autoSwitchSwitchMax = 5L,
      stiff2 = 0L,
      useLinCmt = getOption("rxode2.useLinCmt", FALSE),
      file = NULL,
      chunkSize = NULL,
      parallel = 0L,
```

```
  zeroVarParamHandle = c("warn", "ignore", "keep"),
  indLinStepSearch = c("secant", "exact", "none"),
  indLinMaxIter = 20L,
  indLinRichardson = c("auto", "always", "never", "always4", "always5"),
  indLinIteration = c("auto", "picard", "newton", "exprb", "exprb32"),
  indLinJac = c("auto", "symbolic", "fd"),
  indLinForcing = c("ramp", "constant"),
  usePrior = NA,
  priorPdRetry = 10L,
  priorOmega = NULL,
  priorOmegaEl = NULL,
  priorSigmaEl = NULL,
  envir = parent.frame()
)

## S3 method for class '`function`'
rxSolve(
  object,
  params = NULL,
  events = NULL,
  inits = NULL,
  ...,
  theta = NULL,
  eta = NULL,
  envir = parent.frame()
)

## S3 method for class 'rxUi'
rxSolve(
  object,
  params = NULL,
  events = NULL,
  inits = NULL,
  ...,
  useLinCmt = getOption("rxode2.useLinCmt", FALSE),
  theta = NULL,
  eta = NULL,
  envir = parent.frame()
)

## S3 method for class 'rxode2tos'
rxSolve(
  object,
  params = NULL,
  events = NULL,
  inits = NULL,
  ...,
  useLinCmt = getOption("rxode2.useLinCmt", FALSE),
```

```
  theta = NULL,
  eta = NULL,
  envir = parent.frame()
)

## S3 method for class 'nlmixr2FitData'
rxSolve(
  object,
  params = NULL,
  events = NULL,
  inits = NULL,
  ...,
  theta = NULL,
  eta = NULL,
  envir = parent.frame()
)

## S3 method for class 'nlmixr2FitCore'
rxSolve(
  object,
  params = NULL,
  events = NULL,
  inits = NULL,
  ...,
  theta = NULL,
  eta = NULL,
  envir = parent.frame()
)

## Default S3 method:
rxSolve(
  object,
  params = NULL,
  events = NULL,
  inits = NULL,
  ...,
  indOwnAlloc = TRUE,
  theta = NULL,
  eta = NULL,
  envir = parent.frame()
)

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

## S3 method for class 'rxSolve'
rxSolve(
  object,
```

```
  params = NULL,
  events = NULL,
  inits = NULL,
  ...,
  theta = NULL,
  eta = NULL,
  envir = parent.frame()
)

## S3 method for class 'rxode2'
predict(object, ...)

## S3 method for class '`function`'
predict(object, ...)

## S3 method for class 'rxUi'
predict(object, ...)

## S3 method for class 'rxSolve'
predict(object, ...)

## S3 method for class 'rxEt'
predict(object, ...)

## S3 method for class 'rxParams'
predict(object, ...)

## S3 method for class 'rxode2'
simulate(object, nsim = 1L, seed = NULL, ...)

## S3 method for class 'rxSolve'
simulate(object, nsim = 1L, seed = NULL, ...)

## S3 method for class 'rxParams'
simulate(object, nsim = 1L, seed = NULL, ...)

## S3 method for class 'rxSolve'
solve(a, b, ...)

## S3 method for class 'rxUi'
solve(a, b, ...)

## S3 method for class '`function`'
solve(a, b, ...)

## S3 method for class 'rxode2'
solve(a, b, ...)
```

```
## S3 method for class 'rxParams'
solve(a, b, ...)

## S3 method for class 'rxEt'
solve(a, b, ...)

rxControl(
  ...,
  params = NULL,
  events = NULL,
  inits = NULL,
  envir = parent.frame()
)
```

Arguments

object	is a either a rxode2 family of objects, or a file-name with a rxode2 model specification, or a string with a rxode2 model specification.
params	a numeric named vector with values for every parameter in the ODE system; the names must correspond to the parameter identifiers used in the ODE specification;
events	an eventTable object describing the input (e.g., doses) to the dynamic system and observation sampling time points (see eventTable());
inits	a vector of initial values of the state variables (e.g., amounts in each compartment), and the order in this vector must be the same as the state variables (e.g., PK/PD compartments);
scale	a numeric named vector with scaling for ode parameters of the system. The names must correspond to the parameter identifiers in the ODE specification. Each of the ODE variables will be divided by the scaling factor. For example scale=c(center=2) will divide the center ODE variable by 2.
method	The method for solving ODEs. Currently this supports: • "liblsoda" thread safe lsoda. This supports parallel thread-based solving, and ignores user Jacobian specification. • "lsoda" – LSODA solver. Does not support parallel thread-based solving, but allows user Jacobian specification. • "dop853" – DOP853 solver. Supports parallel thread-based solving; does not support user Jacobian specification. • "indLin" – Solving through inductive linearization. The rxode2 dll must be setup specially to use this solving routine. Supports parallel thread-based solving and honors cores. • "f78" – Runge-Kutta Fehlberg 78 solver using Boost's odeint library. • "rk4" – Runge-Kutta 4 solver using Boost's odeint library. • "ck54" – Cash-Karp 5(4) solver using Boost's odeint library. • "ab" – Adams-Bashforth multi-step solver with a direct implementation (order 1-8, default 5). Uses RK4 to initialize the derivative history and then applies the Adams-Bashforth extrapolation formula. Is a fixed-step method (step size controlled by hmin).

- "abm" – Adams-Bashforth-Moulton solver using Boost's odeint library.
- "dop5" – DOPRI5 solver using Boost's odeint library (supports dense output).
- "bs" – Bulirsch-Stoer solver using Boost's odeint library (supports dense output).
- "ros4" **(implicit)** – Rosenbrock 4 solver using Boost's odeint library (supports dense output). Requires an analytical Jacobian (auto-computed from the model when not provided); falls back to liblsoda if Jacobian generation fails.
- "ros43" **(implicit)** – Kaps-Rentrop 4th-order A-stable Rosenbrock method (GRK4A) from libode. Adaptive step size via embedded 3rd-order error estimate. Requires an analytical Jacobian (auto-computed from the model when not provided); falls back to liblsoda if Jacobian generation fails.
- "ros6" **(implicit)** – Kaps-Wanner 6th-order A-stable Rosenbrock method (ROW6A) from libode. Fixed step size (set with hmin). Requires Jacobian; falls back to liblsoda if unavailable.
- "backwardEuler" **(implicit)** – Backward Euler (BDF-1), 1st-order L-stable fully implicit method from libode. Fixed step size (set with hmin). Requires Jacobian; falls back to liblsoda if unavailable.
- "gauss6" **(implicit)** – Gauss-Legendre 6th-order A-stable fully-implicit method (3 stages) from libode. Fixed step size. Requires Jacobian; falls back to liblsoda if unavailable.
- "iiic6" **(implicit)** – Lobatto IIIC 6th-order L-stable fully-implicit method (4 stages) from libode. Fixed step size. Requires Jacobian; falls back to liblsoda if unavailable.
- "radauiia5" **(implicit)** – Radau IIA 5th-order L-stable fully-implicit method (3 stages) from libode. Fixed step size (set with hmin). Requires Jacobian; falls back to liblsoda if unavailable.
- "geng5" **(implicit)** – Geng 5th-order symplectic fully-implicit method (3 stages) from libode. Fixed step size. Requires Jacobian; falls back to liblsoda if unavailable.
- "sdirk43" **(implicit)** – L-stable SDIRK 4(3) pair from libode. Adaptive step size via embedded 3rd-order error estimate. Requires Jacobian; falls back to liblsoda if unavailable.
- "iem" **(implicit)** – Implicit Euler solver using Boost's odeint library. Requires an explicit Jacobian (auto-generated via calcJac=TRUE when not provided). Uses Boost.uBLAS vectors and is a fixed-step method (step size controlled by hmin).
- "sem" – Symplectic Euler solver using Boost's odeint library. Requires splitting the state into coordinate-momentum pairs (and automatically pads odd-dimensional systems). Is a fixed-step method (step size controlled by hmin).
- "sb3a" – Symplectic Runge-Kutta-Nystrom SB3A McLachlan stepper using Boost's odeint library. Requires splitting the state into coordinate-momentum pairs (and automatically pads odd-dimensional systems). Is a fixed-step method (step size controlled by hmin).

- "sb3am4" – Symplectic Runge-Kutta-Nystrom SB3A M4 McLachlan stepper using Boost's odeint library. Requires splitting the state into coordinate-momentum pairs (and automatically pads odd-dimensional systems). Is a fixed-step method (step size controlled by hmin).
- "vv" – Velocity Verlet stepper using Boost's odeint library. Requires splitting the state into coordinate-momentum pairs (and automatically pads odd-dimensional systems). Is a fixed-step method (step size controlled by hmin).
- "mm" – Modified Midpoint stepper using Boost's odeint library. Is a fixed-step method (step size controlled by hmin) and uses the order parameter to configure the number of intermediate steps.
- "em" – Explicit Euler stepper using Boost's odeint library. Is a fixed-step method (step size controlled by hmin).
- "cvode" – CVODE BDF stiff solver from the SUNDIALS library (sources and headers vendored into rxode2). Supports thread-parallel solving and per-compartment absolute tolerances.
- "trapz" – Explicit trapezoidal rule (Heun's method), a 2nd-order fixed-step Runge-Kutta method implemented via the libode library. Each inter-event interval is integrated with fixed steps of size hmin (default 0.01 when hmin=0). If hmin would require more than maxsteps steps to span the interval, the step size is automatically enlarged to stay within that limit. When an inter-event interval is shorter than the nominal step size (e.g., two doses very close together), the step is silently clamped to the interval length so that short intervals are always handled correctly. Supports parallel thread-based solving and steady-state (ss=1) dosing. The method does not use atol or rtol (fixed-step; no error control). Steady-state convergence is governed by ssAtol, ssRtol, minSS, maxSS, and strictSS.
- "ssp3" – Strong Stability-Preserving Runge-Kutta of order 3 (SSP-RK3, Shu-Osher method), implemented via the libode library. A 3-stage, 3rd-order explicit fixed-step method with superior non-oscillatory properties for problems with discontinuities or sharp fronts. Uses the Butcher tableau c2=1, a21=1; c3=1/2, a31=1/4, a32=1/4; b1=1/6, b2=1/6, b3=2/3. The step-size behavior, clamping, and steady-state options (hmin, maxsteps, ssAtol, ssRtol, minSS, maxSS, strictSS) are identical to "trapz". Does not use atol or rtol (fixed-step; no error control). Supports parallel thread-based solving and steady-state (ss=1) dosing.
- "f32" – Fehlberg 3(2) embedded pair from libode (class OdeRKF32). A 3-stage, 3rd-order adaptive method with a built-in 2nd-order error estimate for automatic step-size control. Tableau: c2=1, a21=1; c3=1/2, a31=1/4, a32=1/4; primary (3rd-order) weights d1=1/6, d2=1/6, d3=4/6; embedded (2nd-order) weights b1=1/2, b2=1/2. Uses atol and rtol for error control. The hmin parameter sets the initial step size (default 0.01); subsequent steps are chosen adaptively. The total number of accepted and rejected steps is bounded by maxsteps. Supports parallel thread-based solving and steady-state (ss=1) dosing with convergence governed by ssAtol, ssRtol, minSS, maxSS, and strictSS.
- "rk43" – Runge-Kutta 4(3) pair from libode. A 5-stage, 4th-order adaptive method with a built-in 3rd-order embedded error estimate for automatic step-size control. Tableau: c2=1/3, a21=1/3; c3=2/3, a31=-1/3, a32=1;

c4=1, a41=1, a42=-1, a43=1; a5j=bj (FSAL). Primary (4th-order) weights b1=b4=1/8, b2=b3=3/8; embedded (3rd-order) weights d1=1/12, d2=1/2, d3=1/4, d5=1/6. Since a5j=bj the 5th stage evaluation is reused as the 1st stage of the next step (FSAL). Uses `atol` and `rtol` for error control. The `hmin` parameter sets the initial step size (default `0.01`); subsequent steps are chosen adaptively. The total number of steps is bounded by `maxsteps`. Supports parallel thread-based solving and steady-state (`ss=1`) dosing with convergence governed by `ssAtol`, `ssRtol`, `minSS`, `maxSS`, and `strictSS`.

- `"dop54"` – Dormand-Prince 5(4) FSAL method (Dormand & Prince 1980), implemented via the libode library. The same algorithm as MATLAB's `ode45`. A 7-stage, 5th-order adaptive method with an embedded 4th-order error estimate for automatic step-size control (FSAL: the 7th stage evaluation is reused as the 1st stage of the next step, giving effectively 6 function evaluations per accepted step). Uses `atol` and `rtol` for error control. The `hmin` parameter sets the initial step size (default `0.01`); subsequent steps are chosen adaptively. The total number of steps is bounded by `maxsteps`. Supports parallel thread-based solving and steady-state (`ss=1`) dosing with convergence governed by `ssAtol`, `ssRtol`, `minSS`, `maxSS`, and `strictSS`. NaN/Inf in derivatives (e.g. from NA parameters) is detected immediately and the solve exits with NA output for the affected subject.
- `"vern65"` – Jim Verner's "most efficient" 6(5) FSAL pair (9 stages), implemented via the libode library using coefficients from Verner's own website (RKV65.IIIXb.Efficient). A 6th-order adaptive method with an embedded 5th-order error estimate. The sparse tableau (many zero a-coefficients) reduces function evaluations; the FSAL property means the 9th stage evaluation is reused as the 1st stage of the next step, giving effectively 8 function evaluations per accepted step. Uses `atol` and `rtol` for error control. The `hmin` parameter sets the initial step size (default `0.01`); subsequent steps are chosen adaptively. The total number of steps is bounded by `maxsteps`. Supports parallel thread-based solving and steady-state (`ss=1`) dosing with convergence governed by `ssAtol`, `ssRtol`, `minSS`, `maxSS`, and `strictSS`. NaN/Inf in derivatives is detected immediately and the solve exits with NA output.
- `"vern76"` – Jim Verner's "most efficient" 7(6) pair (10 stages), implemented via the libode library using coefficients from Verner's own website (RKV76.IIa.Efficient). A 7th-order adaptive method with an embedded 6th-order error estimate. The sparse tableau reduces function evaluations per step. Uses `atol` and `rtol` for error control. The `hmin` parameter sets the initial step size (default `0.01`); subsequent steps are chosen adaptively. The total number of steps is bounded by `maxsteps`. Supports parallel thread-based solving and steady-state (`ss=1`) dosing with convergence governed by `ssAtol`, `ssRtol`, `minSS`, `maxSS`, and `strictSS`. NaN/Inf in derivatives is detected immediately and the solve exits with NA output.
- `"dop87"` – Dormand-Prince 8(7) pair (13 stages), implemented via the libode library using coefficients from Hairer, Norsett and Wanner (1993) "Solving ODEs I" (2nd ed.). An 8th-order adaptive method with an embedded 7th-order error estimate. Both solutions are computed from the original state in the final step loop. Uses `atol` and `rtol` for error control. The

hmin parameter sets the initial step size (default 0.01); subsequent steps are chosen adaptively. The total number of steps is bounded by maxsteps. Supports parallel thread-based solving and steady-state (ss=1) dosing with convergence governed by ssAtol, ssRtol, minSS, maxSS, and strictSS. NaN/Inf in derivatives is detected immediately and the solve exits with NA output.

- "vern98" – Jim Verner's "most efficient" 9(8) pair (16 stages), implemented via the libode library using coefficients from Verner's own website (RKV98.IIa.Efficient). A 9th-order adaptive method with an embedded 8th-order error estimate. The highly sparse tableau (stages 8-16 use only k_0 and $k_{5..}$) minimizes function evaluations per step. Both solutions are computed from the original state in the final step loop. Uses atol and rtol for error control. The hmin parameter sets the initial step size (default 0.01); subsequent steps are chosen adaptively. The total number of steps is bounded by maxsteps. Supports parallel thread-based solving and steady-state (ss=1) dosing with convergence governed by ssAtol, ssRtol, minSS, maxSS, and strictSS. NaN/Inf in derivatives is detected immediately and the solve exits with NA output.

Aliases for existing methods (no additional C++ code; hmin, atol, rtol, and maxsteps follow the aliased method):

- "dp54" – alias for "dop54" (Dormand-Prince 5(4) FSAL, libode).
- "v65e" – alias for "vern65" (Verner 6(5) efficient, libode).
- "v76e" – alias for "vern76" (Verner 7(6) efficient, libode).
- "dp87" – alias for "dop87" (Dormand-Prince 8(7), libode).
- "v98e" – alias for "vern98" (Verner 9(8) efficient, libode).
- "ssp33" – alias for "ssp3" (Strong Stability-Preserving RK3, libode).

libode fixed-step explicit methods. All use hmin as the fixed step size (default 0.01 when hmin=0); atol and rtol are ignored (no error control); maxsteps bounds the total number of steps; NaN/Inf in derivatives sets ind$err and the subject exits with NA output. All support parallel thread-based solving.

- "euler" – Forward (explicit) Euler, 1st-order, 1 stage. Requires a very small step size for accuracy (e.g., hmin=1e-4); intended for pedagogical use or method-comparison baselines.
- "midpoint" – Explicit midpoint rule, 2nd-order, 2 stages.
- "heun" – Heun's method (explicit trapezoid), 2nd-order, 2 stages. Identical in structure to "trapz" but uses the libode fixed-step driver.
- "ssp22" – Strong Stability-Preserving 2-stage 2nd-order method (SSP-RK22), 2 stages. Superior non-oscillatory properties near discontinuities.
- "rk3" – Classical 3rd-order Runge-Kutta (Kutta 1901), 3 stages.
- "ssp53" – Strong Stability-Preserving 5-stage 3rd-order method (SSP-RK53), 5 stages. High SSP coefficient for hyperbolic PDEs or event-heavy ODE systems.
- "s4" – Shanks 4th-order method, 4 stages.
- "r4" – Ralston's 4th-order method, 4 stages. Minimizes local truncation error among classical 4-stage 4th-order methods.

- "ls44" – Low-storage 4th-order method, 4 stages. Uses a 2-register update scheme that minimizes memory bandwidth at the cost of a less general tableau structure.
- "ls54" – Low-storage 4th-order method, 5 stages. Five-stage variant of the 2-register low-storage scheme.
- "ssp54" – Strong Stability-Preserving 5-stage 4th-order method (SSP-RK54), 5 stages.
- "s5" – Shanks 5th-order method, 5 stages.
- "rk5" – Classical 5th-order Runge-Kutta, 6 stages.
- "c5" – Cassity 5th-order method, 6 stages.
- "l5" – Lawson 5th-order method, 6 stages.
- "lk5a" – Luther-Konen 5th-order method, variant A, 6 stages.
- "lk5b" – Luther-Konen 5th-order method, variant B, 6 stages.
- "b6" – Butcher 6th-order method, 7 stages.
- "s7" – Shanks 7th-order method, 9 stages.
- "s8_10" – Shanks 8th-order method, 10 stages.
- "cv8" – Cooper-Verner 8th-order method, 11 stages.
- "s8_12" – Shanks 8th-order method, 12 stages.
- "s10" – Stepanov 10th-order method, 15 stages. Requires a moderate step size (e.g., hmin=1.0) because a single step is accurate to very high order.
- "z10" – Zhang 10th-order method, 16 stages.
- "o10" – Ono 10th-order method, 17 stages.
- "h10" – Hairer 10th-order method, 17 stages.

libode adaptive (variable-step) explicit methods. All use atol and rtol for error control; hmin sets the initial step size (default 0.01 when hmin=0); hmax sets the maximum step size; maxsteps bounds total steps; NaN/Inf in derivatives sets ind$err and the subject exits with NA output. All support parallel thread-based solving and steady-state (ss=1) dosing (convergence governed by ssAtol, ssRtol, minSS, maxSS, strictSS).

- "bs32" – Bogacki-Shampine 3(2) FSAL pair, 4 stages (Bogacki & Shampine 1989). 3rd-order primary with 2nd-order embedded error estimate. FSAL: the 4th-stage evaluation is reused as the 1st stage of the next step. The same algorithm as Julia BS3() and MATLAB ode23.
- "ssp43" – Strong Stability-Preserving 4(3) pair, 4 stages. Adaptive SSP method with a 3rd-order embedded error estimate.
- "f45" – Fehlberg 4(5) pair, 6 stages (Fehlberg 1970). 4th-order primary solution with a 5th-order embedded estimate used for error control.
- "t54" – Tsitouras 5(4) FSAL pair, 7 stages (Tsitouras 2011). 5th-order primary with 4th-order embedded error estimate. FSAL: the 7th stage is reused as the 1st stage of the next step. The same Butcher tableau as Julia's Tsit5().
- "s54" – Stepanov 5(4) FSAL pair, 7 stages. 5th-order primary with 4th-order embedded error estimate.
- "pp54" – Papakostas-Papageorgiou 5(4) FSAL pair, 7 stages.

- "pp54b" – Papakostas-Papageorgiou 5(4) variant B FSAL pair, 7 stages.
- "bs54" – Bogacki-Shampine 5(4) pair, 8 stages. 5th-order primary with 4th-order embedded error estimate (non-FSAL).
- "ss54" – Sharp-Smart 5(4) pair, 7 stages.
- "dp65" – Dormand-Prince 6(5) pair, 8 stages. 6th-order primary with 5th-order embedded error estimate.
- "c65" – Calvo 6(5) pair, 9 stages.
- "tp64" – Tsitouras-Papakostas 6(4) pair, 7 stages. 6th-order primary with 4th-order embedded error estimate.
- "v65r" – Verner "robust" 6(5) FSAL pair, 9 stages. Robust variant of Verner's 6(5) family with wider stability region.
- "v65" – Verner 6(5) pair, 8 stages (non-FSAL).
- "dverk65" – Verner DVERK 6(5) pair, 8 stages. Coefficients from the classic DVERK Fortran code distributed by Hull and Enright.
- "tf65" – Tsitouras-Famelis 6(5) FSAL pair, 9 stages.
- "tp75" – Tsitouras-Papakostas 7(5) pair, 9 stages. 7th-order primary with 5th-order embedded error estimate.
- "tmy7" – Tanaka-Muramatsu-Yamashita 7th-order pair, 10 stages. The same family as Julia's `TanYam7()`.
- "tmy7s" – Tanaka-Muramatsu-Yamashita 7th-order stable variant, 10 stages. Alternative coefficient set with wider stability region.
- "v76r" – Verner "robust" 7(6) pair, 10 stages.
- "ss76" – Sharp-Smart 7(6) pair, 11 stages.
- "v78" – Verner 7(8) pair, 13 stages. 7th-order primary with 8th-order embedded error estimate. Closest rxode2 analog to Julia `Vern8()`.
- "dverk78" – Verner DVERK 7(8) pair, 13 stages. Coefficients from the classic DVERK Fortran code; companion to "dverk65". Also close to Julia `Vern8()`.
- "dp85" – Dormand-Prince 8(5) pair, 12 stages. 8th-order primary with 5th-order embedded error estimate.
- "tp86" – Tsitouras-Papakostas 8(6) pair, 12 stages. The same family as Julia's `TsitPap8()`.
- "v87e" – Verner "efficient" 8(7) pair, 13 stages.
- "v87r" – Verner "robust" 8(7) pair, 13 stages.
- "ev87" – Enright-Verner 8(7) pair, 13 stages.
- "k87" – Kovalnogov-Fedorov-Karpukhina-Simos 8(7) pair, 13 stages.
- "f89" – Fehlberg 8(9) pair, 17 stages. 8th-order primary with 9th-order embedded estimate. Same family as MATLAB `ode89`.
- "v89" – Verner 8(9) pair, 16 stages. Alternative to "f89" in the same order bracket. Also close to MATLAB `ode89`.
- "t98a" – Tsitouras 9(8) variant A pair, 16 stages.
- "v98r" – Verner "robust" 9(8) pair, 16 stages.
- "s98" – Sharp 9(8) pair, 16 stages.

- "f108" – Feagin 10(8) pair, 17 stages (Feagin 2007). 10th-order primary with 8th-order embedded error estimate. The same method as Julia's Feagin10(). The large number of stages carries elevated transcription-error risk; verified against Williams' libode canonical order test.
- "c108" – Curtis 10(8) pair, 21 stages.
- "b109" – Baker 10(9) pair, 21 stages.
- "s1110a" – Stone 11(10) variant A pair, 26 stages. 11th-order primary; very few published references.
- "f1210" – Feagin 12(10) pair, 25 stages (Feagin 2007). 12th-order primary with 10th-order embedded error estimate. The same method as Julia's Feagin12(). Elevated transcription-error risk due to many stages; verified against Williams' libode canonical order test.
- "o129" – Ono 12(9) pair, 29 stages.
- "f1412" – Feagin 14(12) pair, 35 stages (Feagin 2007). 14th-order primary with 12th-order embedded error estimate. The same method as Julia's Feagin14(). The highest-order method in rxode2; elevated transcription-error risk due to 35 stages; verified against Williams' libode canonical order test.

deSolve-derived Fortran solvers (from the deSolve R package). Both use non-reentrant Fortran COMMON blocks and therefore run single-threaded only, regardless of the cores setting. They are BDF (Backward Differentiation Formula) stiff solvers and are most useful when you want behavior comparable to deSolve's lsode() or bdf()/vode() functions.

- "lsode" – DLSODE (Hindmarsh 1983, ODEPACK) in Adams (non-stiff) mode, MF=10. Variable-order Adams method, order 1-12; no Jacobian evaluation. Corresponds to deSolve's lsode(method="adams"). Adaptive step-size; error controlled by atol and rtol. **Not thread-safe** – always runs on a single core.
- "bdf" – DLSODE (Hindmarsh 1983, ODEPACK) in BDF (stiff) mode, MF=22. Variable-order BDF method, order 1-5; internally generated dense Jacobian. Corresponds to deSolve's bdf() / vode(method="bdf"). Adaptive step-size; error controlled by atol and rtol. **Not thread-safe** – always runs on a single core.

sigdig

Specifies the "significant digits" that the ODE solving requests. This is NULL by default, and while it is NULL it has no effect at all: rxSolve() uses the standard atol/rtol (and the standard sensitivity and steady-state tolerances). sigdig only changes a tolerance when you ask for it explicitly.

When it is supplied, the tolerances are derived with one solver-independent formula – the same for stiff, non-stiff and auto-switching solvers. The rtol exponent IS sigdig and atol sits three orders below it:

- rtol = 10^(-sigdig), atol = 10^(-sigdig-3)
- the sensitivity tolerances match the main solve, so rtolSens = rtol and atolSens = atol (gradients and covariances are built from them)
- the steady-state tolerances run one order looser than the corresponding main tolerance, so ssRtol = ssRtolSens = 10*rtol and ssAtol = ssAtolSens = 10*atol

Each of these is set only when you did not pass that tolerance yourself; a tolerance you supply always wins. Because they are resolved independently, an explicit atol/rtol overrides the main solve but does *not* propagate to the sensitivity or steady-state tolerances – set those directly if you need them changed too.

This mapping matches how nlmixr2est derives solver tolerances from its optimization sigdig, so a sigdig used for estimation and the same sigdig used for a plain rxSolve() mean the same thing. Note it is keyed to sigdig as a request for that many significant digits, and is looser than the atol/rtol defaults for small sigdig – at sigdig = 4 it gives rtol = 1e-4 against a default rtol = 1e-6. Raise sigdig, or set atol/rtol directly, when you want a tighter solve.

atol — a numeric absolute tolerance (1e-8 by default) used by the ODE solver to determine if a good solution has been achieved; This is also used in the solved linear model to check if prior doses do not add anything to the solution.

rtol — a numeric relative tolerance (1e-6 by default) used by the ODE solver to determine if a good solution has been achieved. This is also used in the solved linear model to check if prior doses do not add anything to the solution.

maxsteps — maximum number of (internally defined) steps allowed during one call to the solver. (5000 by default)

hmin — The minimum absolute step size allowed. The default value is 0.

For the fixed-step Boost methods `"rk4"`, `"trapz"`, `"ssp3"`, `"ab"`, `"abm"`, `"sem"`, `"sb3a"`, `"sb3am4"`, `"vv"`, `"mm"`, `"em"`, `"ros6"`, `"backwardEuler"`, `"gauss6"`, `"iiic6"`, `"radauiia5"`, `"geng5"`, and the libode fixed-step family (`"euler"`, `"midpoint"`, `"heun"`, `"ssp22"`, `"rk3"`, `"ssp53"`, `"s4"`, `"r4"`, `"ls44"`, `"ls54"`, `"ssp54"`, `"s5"`, `"rk5"`, `"c5"`, `"l5"`, `"lk5a"`, `"lk5b"`, `"b6"`, `"s7"`, `"s8_10"`, `"cv8"`, `"s8_12"`, `"s10"`, `"z10"`, `"o10"`, `"h10"`), this specifies the fixed step size.
If `hmin=0` (the default), it uses a default of `0.01` for `"rk4"`, `"trapz"`, `"ssp3"`, `"ros6"`, `"backwardEuler"`, `"gauss6"`, `"iiic6"`, `"radauiia5"`, `"geng5"`, and all libode fixed-step methods; `0.0001` for `"ab"`, `"abm"`, `"sem"`, `"sb3a"`, `"sb3am4"`, `"vv"`, `"mm"`, and `"em"`.
If the requested step size would cause the number of steps to exceed `maxsteps`, the step size is automatically increased to ensure the integration completes within the `maxsteps` limit. For `"trapz"`, the step is also silently clamped to the interval length when the inter-event interval is shorter than the nominal step size, so short intervals (e.g., between closely spaced doses) are always handled correctly.

For the adaptive methods `"f78"`, `"ck54"`, `"dop5"`, `"bs"`, `"f32"`, `"rk43"`, `"dop54"`, `"vern65"`, `"vern76"`, `"dop87"`, `"vern98"`, `"ros43"`, `"sdirk43"`, and all libode adaptive methods (`"bs32"`, `"ssp43"`, `"f45"`, `"t54"`, `"s54"`, `"pp54"`, `"pp54b"`, `"bs54"`, `"ss54"`, `"dp65"`, `"c65"`, `"tp64"`, `"v65r"`, `"v65"`, `"dverk65"`, `"tf65"`, `"tp75"`, `"tmy7"`, `"tmy7s"`, `"v76r"`, `"ss76"`, `"v78"`, `"dverk78"`, `"dp85"`, `"tp86"`, `"v87e"`, `"v87r"`, `"ev87"`, `"k87"`, `"f89"`, `"v89"`,

`"t98a"`, `"v98r"`, `"s98"`, `"f108"`, `"c108"`, `"b109"`, `"s1110a"`, `"f1210"`, `"o129"`, `"f1412"`, the libode aliases `"dp54"`, `"v65e"`, `"v76e"`, `"dp87"`, `"v98e"`, `"ssp33"`, and the deSolve-derived methods `"lsode"`, `"bdf"`), this specifies the initial step size (default `0.01` when `hmin=0`); subsequent steps are chosen adaptively using `atol`, `rtol`, and `maxsteps`.

hmax The maximum absolute step size allowed. When hmax=NA (default), uses the average difference + hmaxSd*sd in times and sampling events. The hmaxSd is a user specified parameter and which defaults to zero. When hmax=NULL rxode2 uses the maximum difference in times in your sampling and events. The value 0 is equivalent to infinite maximum absolute step size.

Note that for dense output methods ("dop853", "dop5", "bs", "ros4"), hmax defaults to NULL to allow the solvers to determine the step size when dense=TRUE

For method="indLin", hmax caps how long an interval is treated as having a CONSTANT matrix-exponential term before re-linearizing. A model with a state-dependent indLin() forcing now chooses its own relinearization step from a local error estimate, so atol/rtol (and maxsteps) are what control its accuracy and hmax only sets an upper bound on the step. A model whose forcing reads no state, or which has none, has a rate matrix that is constant in the states, so there is no error to control and hmax only changes how many (mathematically equivalent) matrix exponentials are taken.

hmaxSd The number of standard deviations of the time difference to add to hmax. The default is 0

hini The step size to be attempted on the first step. The default value is determined by the solver (when hini = 0)

maxordn The maximum order to be allowed for the nonstiff (Adams) method. The default is 12. It can be between 1 and 12.

maxords The maximum order to be allowed for the stiff (BDF) method. The default value is 5. This can be between 1 and 5.

order The order for the "ab", "abm", and "mm" methods. For "ab" and "abm", the default is 5, and it can be between 1 and 8. For "mm" (Modified Midpoint), it represents the number of intermediate steps, the default is 5, and it must be a positive integer (>= 1).

... Other arguments including scaling factors for each compartment. This includes S# = numeric will scale a compartment # by a dividing the compartment amount by the scale factor, like NONMEM.

cores Number of cores used in parallel ODE solving. This is equivalent to calling setRxThreads()

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.

- "locf" – 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.
  For time-varying covariates where a missing value is present, the interpolation method will use either "locf" or "nocb" throughout if they are the type of covariate interpolation that is selected.
  When using the linear or midpoint interpolation, the lower point in the interpolation will use locf to interpolate missing covariates and the upper point will use the nocb to interpolate missing covariates.

naInterpolation

specifies the interpolation method for time-varying covariates when the instantaneous value is NA (not during an explicit interpolation) and the covsInterpolation is either "midpoint" or "linear". This can be:

- "locf" – last observation carried forward (default)
- "nocb" – next observation carried backward.

This will look for the prior value (backwards/locf) when instantaneously missing, or the next value when instantaneously missing. If all the covariates are missing and you find the end/beginning of the individual record, switch direction. If all are really missing, then return missing.

keepInterpolation

specifies the interpolation method for variables in the keep column. When nlmixr2 creates mtime, addl doses etc, these items were not originally in the dataset. The interpolation methods you can choose are:

- "locf" – last observation carried forward (default)
- "nocb" – next observation carried backward.
- "na" – no interpolation, simply put NA for the interpolated keep covariates.

addCov A boolean indicating if covariates should be added to the output matrix or data frame. By default this is disabled.

sigma Named sigma covariance or Cholesky decomposition of a covariance matrix. The names of the columns indicate parameters that are simulated. These are simulated for every observation in the solved system. When sigma is NA and you are using it with a rxode2 ui model, the unexplained variability described by the sigma matrix are set to zero.

sigmaDf Degrees of freedom of the sigma t-distribution. By default it is equivalent to Inf, or a normal distribution.

sigmaLower Lower bounds for simulated unexplained variability (by default -Inf)

sigmaUpper Upper bounds for simulated unexplained variability (by default Inf)

nCoresRV Number of cores used for the simulation of the sigma variables. By default this is 1. To reproduce the results you need to run on the same platform with the same number of cores. This is the reason this is set to be one, regardless of what the number of cores are used in threaded ODE solving.

sigmaIsChol Boolean indicating if the sigma is in the Cholesky decomposition instead of a symmetric covariance

sigmaSeparation

separation strategy for sigma;

Tells the type of separation strategy when simulating covariance with parameter uncertainty with standard deviations modeled in the thetaMat matrix.

- "lkj" simulates the correlation matrix from the rLKJ1 matrix with the distribution parameter eta equal to the degrees of freedom nu by (nu-1)/2
- "separation" simulates from the identity inverse Wishart covariance matrix with nu degrees of freedom. This is then converted to a covariance matrix and augmented with the modeled standard deviations. While computationally more complex than the "lkj" prior, it performs better when the covariance matrix size is greater or equal to 10
- "auto" chooses "lkj" when the dimension of the matrix is less than 10 and "separation" when greater than equal to 10.
- "tnpri" draws the sigma entries jointly from the thetaMat, the same way omegaSeparation="tnpri" does for the omega; dfObs is not used.

sigmaXform When taking sigma values from the thetaMat simulations (using the separation strategy for covariance simulation), how should the thetaMat values be turned int standard deviation values:

- identity This is when standard deviation values are directly modeled by the params and thetaMat matrix
- variance This is when the params and thetaMat simulates the variance that are directly modeled by the thetaMat matrix
- log This is when the params and thetaMat simulates log(sd)
- nlmixrSqrt This is when the params and thetaMat simulates the inverse cholesky decomposed matrix with the x\^2 modeled along the diagonal. This only works with a diagonal matrix.
- nlmixrLog This is when the params and thetaMat simulates the inverse cholesky decomposed matrix with the exp(x\^2) along the diagonal. This only works with a diagonal matrix.
- nlmixrIdentity This is when the params and thetaMat simulates the inverse cholesky decomposed matrix. This only works with a diagonal matrix.

nDisplayProgress

An integer indicating the minimum number of c-based solves before a progress bar is shown. By default this is 10,000.

amountUnits This supplies the dose units of a data frame supplied instead of an event table. This is for importing the data as an rxode2 event table.

timeUnits This supplies the time units of a data frame supplied instead of an event table. This is for importing the data as an rxode2 event table.

theta A vector of parameters that will be named THETA\[#\] and added to parameters

thetaLower Lower bounds for simulated population parameter variability (by default -Inf)

thetaUpper Upper bounds for simulated population unexplained variability (by default Inf)

eta A vector of parameters that will be named ETA\[#\] and added to parameters

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.

stateTrim — When amounts/concentrations in one of the states are above this value, trim them to be this value. By default Inf. Also trims to -stateTrim for large negative amounts/concentrations. If you want to trim between a range say c(0, 2000000) you may specify 2 values with a lower and upper range to make sure all state values are in the reasonable range.

updateObject — This is an internally used flag to update the rxode2 solved object (when supplying an rxode2 solved object) as well as returning a new object. You probably should not modify it's FALSE default unless you are willing to have unexpected results.

omega — Estimate of Covariance matrix. When omega is a list, assume it is a block matrix and convert it to a full matrix for simulations. When omega is NA and you are using it with a rxode2 ui model, the between subject variability described by the omega matrix are set to zero.

omegaDf — The degrees of freedom of a t-distribution for simulation. By default this is NULL which is equivalent to Inf degrees, or to simulate from a normal distribution instead of a t-distribution.

omegaIsChol — Indicates if the omega supplied is a Cholesky decomposed matrix instead of the traditional symmetric matrix.

omegaSeparation

Omega separation strategy

Tells the type of separation strategy when simulating covariance with parameter uncertainty with standard deviations modeled in the thetaMat matrix.

- "lkj" simulates the correlation matrix from the rLKJ1 matrix with the distribution parameter eta equal to the degrees of freedom nu by (nu-1)/2
- "separation" simulates from the identity inverse Wishart covariance matrix with nu degrees of freedom. This is then converted to a covariance matrix and augmented with the modeled standard deviations. While computationally more complex than the "lkj" prior, it performs better when the covariance matrix size is greater or equal to 10
- "auto" chooses "lkj" when the dimension of the matrix is less than 10 and "separation" when greater than equal to 10.

- "tnpri" does not use a separation strategy at all: the omega entries themselves are carried in the thetaMat and are drawn from it jointly with the thetas, which is what a covariance step from NONMEM or 'nlmixr2' gives and what NONMEM calls TNPRI. This keeps the off diagonal entries and the covariances between the thetas and the omega entries, which the strategies above redraw or discard. omega has to be a matrix, since the draws are added to it, and the thetaMat columns are matched to entries by name – om.eta.cl or eta.cl for a diagonal, and cov.eta.cl.eta.v or omega2.1 for an off diagonal. A drawn matrix that is not positive definite is redrawn, see priorPdRetry. dfSub is not used: an entry the thetaMat gives no variance stays at its estimate.

omegaXform When taking omega values from the thetaMat simulations (using the separation strategy for covariance simulation), how should the thetaMat values be turned int standard deviation values:

- identity This is when standard deviation values are directly modeled by the params and thetaMat matrix
- variance This is when the params and thetaMat simulates the variance that are directly modeled by the thetaMat matrix
- log This is when the params and thetaMat simulates log(sd)
- nlmixrSqrt This is when the params and thetaMat simulates the inverse cholesky decomposed matrix with the x\^2 modeled along the diagonal. This only works with a diagonal matrix.
- nlmixrLog This is when the params and thetaMat simulates the inverse cholesky decomposed matrix with the exp(x\^2) along the diagonal. This only works with a diagonal matrix.
- nlmixrIdentity This is when the params and thetaMat simulates the inverse cholesky decomposed matrix. This only works with a diagonal matrix.

omegaLower Lower bounds for simulated ETAs (by default -Inf)

omegaUpper Upper bounds for simulated ETAs (by default Inf)

nSub Number between subject variabilities (ETAs) simulated for every realization of the parameters.

thetaMat Named theta matrix.

thetaDf The degrees of freedom of a t-distribution for simulation. By default this is NULL which is equivalent to Inf degrees, or to simulate from a normal distribution instead of a t-distribution.

thetaIsChol Indicates if the theta supplied is a Cholesky decomposed matrix instead of the traditional symmetric matrix.

nStud Number virtual studies to characterize uncertainty in estimated parameters.

dfSub Degrees of freedom to sample the between subject variability matrix from the inverse Wishart distribution (scaled) or scaled inverse chi squared distribution.

dfObs Degrees of freedom to sample the unexplained variability matrix from the inverse Wishart distribution (scaled) or scaled inverse chi squared distribution.

returnType This tells what type of object is returned. The currently supported types are:

- "rxSolve" (default) will return a reactive data frame that can change easily change different pieces of the solve and update the data frame. This is the currently standard solving method in rxode2, is used for rxSolve(object, ...), solve(object,...),
- "data.frame" – returns a plain, non-reactive data frame; Currently very slightly faster than returnType="matrix"
- "matrix" – returns a plain matrix with column names attached to the solved object. This is what is used object$run as well as object$solve
- "data.table" – returns a data.table; The data.table is created by reference (ie setDt()), which should be fast.
- "tbl" or "tibble" returns a tibble format.

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

nsim — represents the number of simulations. For rxode2, if you supply single subject event tables (created with [eventTable()])

minSS — Minimum number of iterations for a steady-state dose

maxSS — Maximum number of iterations for a steady-state dose

infSSstep — Step size for determining if a constant infusion has reached steady state. By default this is large value, 12.

strictSS — Boolean indicating if a strict steady-state is required. If a strict steady-state is (TRUE) required then at least minSS doses are administered and the total number of steady states doses will continue until maxSS is reached, or atol and rtol for every compartment have been reached. However, if ODE solving problems occur after the minSS has been reached the whole subject is considered an invalid solve. If strictSS is FALSE then as long as minSS has been reached the last good solve before ODE solving problems occur is considered the steady state, even though either atol, rtol or maxSS have not been achieved.

istateReset — When TRUE, reset the ISTATE variable to 1 for lsoda and liblsoda with doses, like deSolve; When FALSE, do not reset the ISTATE variable with doses.

subsetNonmem — subset to NONMEM compatible EVIDs only. By default TRUE.

maxAtolRtolFactor — The maximum atol/rtol that FOCEi and other routines may adjust to. By default 0.1

from — When there is no observations in the event table, start observations at this value. By default this is zero.

to — When there is no observations in the event table, end observations at this value. By default this is 24 + maximum dose time.

by — When there are no observations in the event table, this is the amount to increment for the observations between from and to.

length.out — The number of observations to create if there isn't any observations in the event table. By default this is 200.

iCov — A data frame of individual non-time varying covariates to combine with the events dataset. The iCov dataset has one covariate per ID and should match the event table

keep — Columns to keep from either the input dataset or the iCov dataset. With the iCov dataset, the column is kept once per line. For the input dataset, if any records are added to the data LOCF (Last Observation Carried forward) imputation is performed.

indLinPhiTol — the requested accuracy tolerance on exponential matrix.

indLinPhiM — the maximum size for the Krylov basis

indLinMatExpType — This is them matrix exponential type that is use for rxode2. Currently the following are supported:

- Al-Mohy Uses the exponential matrix method of Al-Mohy Higham (2009); the default. It is the fastest on the models where the exponential is a measurable share of the solve, and all four agree to solver tolerance.
- arma Use the exponential matrix from RcppArmadillo
- expokit Use the exponential matrix from Roger B. Sidje (1998)
- taylor Taylor scaling and squaring, needing no linear solve

indLinMatExpOrder — an integer, the order of approximation to be used, for the expokit value. The best value for this depends on machine precision (and slightly on the matrix). We use 6 as a default. It no longer applies to Al-Mohy, whose degree is not free to choose: each accuracy threshold in the Al-Mohy-Higham table belongs to one specific degree, so the degree and the scaling are selected together from the matrix norm. taylor has always chosen its degree the same way.

drop — Columns to drop from the output

idFactor — This boolean indicates if original ID values should be maintained. This changes the default sequentially ordered ID to a factor with the original ID values in the original dataset. By default this is enabled.

mxhnil — maximum number of messages printed (per problem) warning that T + H = T on a step (H = step size). This must be positive to result in a non-default value. The default value is 0 (or infinite).

hmxi — inverse of the maximum absolute value of H to are used. hmxi = 0.0 is allowed and corresponds to an infinite hmax1 (default). hminandhmximay be changed at any time, but wi

warnIdSort — Warn if the ID is not present and rxode2 assumes the order of the parameters/iCov are the same as the order of the parameters in the input dataset.

warnDrop — Warn if column(s) were supposed to be dropped, but were not present.

ssAtol — Steady state atol convergence factor. Can be a vector based on each state.

ssRtol — Steady state rtol convergence factor. Can be a vector based on each state.

safeZero — Use safe zero divide. By default this is turned on but you may turn it off if you wish.

safeLog — Use safe log. When enabled (TRUE, the default) if your value that you are taking log() of is negative or zero, this will return log(machine epsilon). With FALSE both return the usual NaN/-Inf. With 2 only zero is floored to log(machine epsilon); a *negative* argument is treated as a domain error and returns NaN. Use 2 when a hand-written likelihood takes log() of a parameter that must stay positive, so an invalid value propagates as NaN instead of a large finite number the optimizer could mistake for a good fit.

safePow — Use safe powers. When enabled if your power is negative and your base is zero, this will return the machine epsilon^(negative power). By default this is turned on.

sumType — Sum type to use for sum() in rxode2 code blocks.

pairwise uses the pairwise sum (fast, default)

fsum uses the PreciseSum package's fsum function (most accurate)

kahan uses Kahan correction

neumaier uses Neumaier correction

c uses no correction: default/native summing

prodType — Product to use for prod() in rxode2 blocks

long double converts to long double, performs the multiplication and then converts back.

double uses the standard double scale for multiplication.

resample — A character vector of model variables to resample from the input dataset; This sampling is done with replacement. When NULL or FALSE no resampling is done. When TRUE resampling is done on all covariates in the input dataset

resampleID — boolean representing if the resampling should be done on an individual basis TRUE (ie. a whole patient is selected) or each covariate is resampled independent of the subject identifier FALSE. When resampleID=TRUE correlations of parameters are retained, where as when resampleID=FALSE ignores patient covariate correaltions. Hence the default is resampleID=TRUE.

maxwhile — represents the maximum times a while loop is evaluated before exiting. By default this is 100000

atolSens — Sensitivity atol, can be different than atol with liblsoda. This allows a less accurate solve for gradients (if desired)

rtolSens — Sensitivity rtol, can be different than rtol with liblsoda. This allows a less accurate solve for gradients (if desired)

ssAtolSens — Sensitivity absolute tolerance (atol) for calculating if steady state has been achieved for sensitivity compartments.

ssRtolSens — Sensitivity relative tolerance (rtol) for calculating if steady state has been achieved for sensitivity compartments.

simVariability — determines if the variability is simulated. When NA (default) this is determined by the solver.

nLlikAlloc — The number of log likelihood endpoints that are used in the model. This allows independent log likelihood per endpoint in focei for nlmixr2. It likely shouldn't be set, though it won't hurt anything if you do (just may take up more memory for larger allocations).

useStdPow — This uses C's pow for exponentiation instead of R's R_pow or R_pow_di. By default this is FALSE

naTimeHandle — Determines what time of handling happens when the time becomes NA: current options are:

- ignore this ignores the NA time input and passes it through.
- warn (default) this will produce a warning at the end of the solve, but continues solving passing through the NA time

- `error` this will stop this solve if this is not a parallel solved ODE (otherwise stopping can crash R)

`addlKeepsCov` This determines if the additional dosing items repeats the dose only (`FALSE`) or keeps the covariates at the record of the dose (`TRUE`)

`addlDropSs` When there are steady state doses with an `addl` specification the steady state flag is dropped with repeated doses (when `TRUE`) or retained (when `FALSE`)

`ssAtDoseTime` Boolean that when `TRUE` back calculates the steady concentration at the actual time of dose, otherwise when `FALSE` the doses are shifted

`ss2cancelAllPending`
When `TRUE` the `SS=2` event type cancels all pending doses like `SS=1`. When `FALSE` the pending doses not canceled with `SS=2` (the infusions started before `SS=2` occurred are canceled, though).

`ssSolved` When `TRUE` this will return the solved steady state solutions for the linear compartment model. When `FALSE` this will solve to steady state using the linear solutions instead. This is only used when the method only has `linCmt()` and does not mix ODEs with the solution. The default is `TRUE`.

`linCmtSensType` The type of linear compartment sensitivity/gradients to use. The current options are:

- `auto` (the default) – forward-mode automatic differentiation (`AD`). On an optimized build forward mode is at least as fast as reverse mode for every number of requested sensitivity directions on every configuration (reverse is 2-4x slower on a three compartment oral model: its per-row tape build costs more than the extra forward passes), so `auto` resolves to `AD` on every solve path.
- `ADr` – reverse-mode automatic differentiation (`stan::math::jacobian`): each row is differentiated on its own nested tape with one adjoint sweep per compartment, independent of how many directions are requested. The tape is per thread, so this solves across threads like `AD`. Slower than `AD` on an optimized build; kept as an explicit option for validation.
- `AD` – forward-mode automatic differentiation (`stan::math::fvar`), differentiating the same analytic solution on the C++ stack with one pass per requested direction; matches `ADr` to round-off.
- `ADm` – the same forward-mode differentiation carrying every requested direction through a single pass, so the solution itself (each exponential, division and eigen-decomposition) is evaluated once instead of once per direction. Bitwise identical to `AD`, and unlike the amortized ordinary-row path it also shares the work on steady-state rows.
- `forward` – forward sensitivity where the step size is determined by shi 2021 optimization (only once per problem)
- `forwardG` – forward sensitivity where the step size is determined by the Gill 1983 optimization for forward differences (only once per problem).
- `central` – central sensitivity where the step size is determined by shi 2021 optimization (only once per problem)
- `forward3` – three point central difference where step size is determined by shi 2021 optimization for central differences (only once per problem)

- endpoint5 – five point endpoint difference where step size is determined by the shi 2021 optimization for central differences (only once per problem)
- fowardH – forward sensitivity where the step size is fixed
- centralH – central sensitivity where the step size is fixed
- forward3H – three point central difference where step size is fixed
- endpoint5H – five point endpoint difference where step size is fixed

linCmtSensH The step size for the forward and central differences when using the option centralH, forwardH, foward3H or endpoint5H options.

linCmtSensPhi Which state-transition matrix a linCmt() sensitivity row propagates through, if any. TRUE (the default) or 2 assembles the matrix and its parameter derivatives from their closed form in the constants the theta window already holds; that costs about one kernel evaluation, so it is assembled for any interval and cached where the interval repeats, and it carries the depot and infusion terms. 1 assembles it instead by probing the kernel with unit-basis prior states, which costs one evaluation per direction per column and so is only worth paying on an interval that repeats; rate-bearing rows are excluded. FALSE or 0 evaluates the closed form row by row in the order earlier versions used. All three are the same exact closed-form solution evaluated in a different order, so they can differ in the last few digits; use FALSE to reproduce earlier results digit for digit.

nonmem Boolean that when TRUE evaluates statements that do not depend on a state (like NONMEM's $PK) with time set to the time of the data record that ends the interval being integrated, the way NONMEM calls $PK at its records. d/dt() and state-dependent statements keep the continuous time (like $DES). addl repeats, infusion ends, lagged doses and doses added while solving do not end an interval; mtime() times do. Covariate interpolation is still set by covsInterpolation. By default FALSE.

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

linCmtGillK 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.

linCmtGillStep 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

linCmtGillRtol The relative tolerance used for Gill 1983 optimal step size determination.

linCmtShiErr Shi difference error

linCmtShiMax The maximum number of steps for the optimization of the forward-difference step size in linear compartment numeric difference.

linCmtScale The scale of the linear compartment model. This is applied to sensitivity approximation using numeric differences. When TRUE or NULL use default scaling, when FALSE use no scaling. If it is one element numeric, the value is duplicated 7 times and applies to all the parameters. Otherwise this is a seven element numeric vector implying the scaling for each of the linear compartmental model parameters.

`linCmtHcmt` This represents the compartments considered when optimizing the forward difference step size. When a character vector it can be any of the following (multiple allowed):

- `"depot"` – depot compartment
- `"central"` – central compartment
- `"peripheral1"` – peripheral compartment
- `"peripheral2"` – second peripheral compartment
- `"concentration"` – concentration value (i.e. central compartment/volume)

`linCmtHmeanI` This represents the type of sum done for each time-point of the linear solved systems (as defined by `"linCmtHcmt"`).

- `"arithmetic"` – gives the arithmetic mean
- `"geometric"` – gives the geometric mean
- `"harmonic"` – gives the harmonic mean

`linCmtHmeanO` This represents the type of sum done for the overall problem of the linear solved systems (first each time point mean is calculated with `linCmtHmeanI`).

- `"arithmetic"` – gives the arithmetic mean
- `"geometric"` – gives the geometric mean
- `"harmonic"` – gives the harmonic mean

`linCmtSuspect` The tolerance for gradients in linear compartment solutions to re-compute when gradients seem to be zero.

`linCmtForwardMax`
The maximum number of points in a forward difference to take while calculating the gradients. This is an integer from 1 to 3. There is at least 1 extra point taken for gradient calculation, if the gradient is suspect another is taken (if this value is 2), and finally a third is calculated if the gradient is still suspect.

`indOwnAlloc` Logical; when `TRUE` each individual's `dose`, `ii`, `all_times`, and `solve` arrays are allocated independently via `malloc`/`calloc` rather than as pointers into a single global buffer. This enables per-individual reallocation for dose and observation-pushing with `evid_()` and related functions. When `FALSE` these arrays are allocated as a single global buffer, which is slightly faster and slightly more memory efficient but does not allow for dynamic dosing/observations. By default this is `NA` which automatically decides based on if there is any dosing or observation pushing in the model.

`maxExtra` Integer; maximum number of events (doses and observations) that `evid_()` may push per individual per solve. When an individual exceeds this limit the solve is aborted for that individual (output filled with `NA`) and an error is raised after the full parallel solve completes. Set to `0L` to allow unlimited pushes (use with care – cascading `evid_()` calls can grow without bound). Default is `100L`.

`tolFactor` A per-individual tolerance multiplier (>= 1.0, or `NULL` for no effect). When supplied, each individual's `atol`, `rtol`, `ssAtol`, and `ssRtol` are multiplied by this factor before the ODE solver is called, loosening the tolerances for that individual. This is useful when a small number of subjects are numerically stiff and would otherwise cause solve failures: pass a larger factor for the problematic subjects and leave the rest at 1.0 (or omit them). The effective tolerance is always capped at `maxAtolRtolFactor`.

`tolFactor` may be:
* `NULL` (default) -- no adjustment applied.
* A single numeric value -- the same factor is applied to
  the first `length(tolFactor)` subjects in order.
* A numeric vector -- applied element-wise to subjects in
  the order they appear.
* A **named** numeric vector -- names are matched to subject
  IDs; unmatched subjects retain `tolFactor = 1.0`.

The per-subject factors used (after matching) are stored back
in the solved object and accessible via `$tolFactor`.

`serializeFile` Controls serialization-driven solves. When this is a file path, `rxSolve()` writes the pre-integration serialized state to that file and returns the file path invisibly without running the solve. When this is `TRUE`, `rxSolve()` writes the serialized state to a temporary `.rxbin` file, solves by reloading from that file, returns the solve result, and then removes the temporary file (mostly for testing). When solving from an existing serialization file via `rxSolve(model, "state.rxbin")`, only the model and serialization file are allowed because the file already stores the solve inputs and controls.

`dense` Logical; when `TRUE` and the method supports dense output, enables continuous interpolation so the solver can take large internal steps and reconstruct the solution cheaply at each observation time. Dense-capable single methods are `"dop853"`, `"dop5"`, `"bs"`, and `"ros4"`. `"dop853+ros4"` is the only dense AutoSwitch composite: the secondary must be dense too (`"ros4"` is the only stiff method that is), and handing a dense segment from the primary to the secondary mid-segment is implemented for a `"dop853"` primary. Other composites solve densely only if you drop the secondary. A warning is issued and `dense` is set to `FALSE` when a composite stiff secondary does not support dense output. Silently ignored for non-dense single methods. Not yet supported for `linCmt()` models (a warning is emitted and the standard path is used instead).

`cvodeLinSolver` Character; selects the linear solver used by the CVODE integrator when `method = "cvode"`. Ignored for all other methods. Available choices and when to use them:

- `"dense"` (default) – Direct LU factorization of the full Jacobian. Best for small to medium systems (typically fewer than ~50 compartments). Requires `O(n^2)` storage and `O(n^3)` work per Newton iteration.
- `"band"` – Currently aliases to `"dense"`. Kept for compatibility until rxode2 can determine model bandwidth safely during solve setup.
- `"gmres"` – Generalized Minimal Residual iterative Krylov solver. Avoids explicit Jacobian formation, making it practical for large stiff systems (tens to hundreds of compartments) where LU factorization would dominate runtime. Generally the first iterative solver to try.
- `"bicgstab"` – Bi-Conjugate Gradient Stabilized iterative Krylov solver. Similar use case to `"gmres"` but can converge faster on some non-symmetric problems. Try if `"gmres"` is slow or fails to converge.
- `"tfqmr"` – Transpose-Free Quasi-Minimal Residual iterative Krylov solver.

Another alternative for large systems; avoids the matrix transpose required by classical QMR.

For most pharmacometric models with fewer than ~50 compartments the default "dense" solver is fastest. Switch to an iterative solver ("gmres" is a good first choice) only for large QSP or PBPK models where Jacobian factorization becomes the bottleneck.

autoSwitchNonstifftol

Numeric in (0, 1]; the stiffness ratio at which an interval is called stiff. Each accepted step of a dop853 primary estimates the dominant eigenvalue and forms rho * h / S(dop853) with S(dop853) = 6.1; a step above autoSwitchNonstifftol is a stiff verdict, and 15 verdicts hand the rest of the interval to the stiff secondary. Only a dop853 primary carries this estimate; any other primary switches when it fails. Default 9/10.

autoSwitchStifftol

Numeric in (0, 1]; the same ratio, but applied once the subject has already switched at least once – that is, on the optimistic re-probe of the primary rather than on the first attempt. Lowering it below autoSwitchNonstifftol makes the re-probe give up sooner, so stiff mode is stickier. Default 9/10, equal to autoSwitchNonstifftol, i.e. a re-probe behaves like a first probe.

autoSwitchDtfac

Accepted for backwards compatibility and currently inert. It scaled a suggested step size across a switch, for a predictive switching scheme that has been replaced by the reactive one described above. Default 2.0.

autoSwitchMaxStiff

Integer; number of consecutive intervals in which the primary reported stiffness before the solve stops probing it and stays on the stiff secondary. Default 10L.

autoSwitchMaxNonstiff

Integer; intervals to spend on the stiff secondary before optimistically probing the primary again. Each probe that fails doubles the wait (capped at 64x), so a persistently stiff subject stops paying for probes. Default 3L.

autoSwitchStiffFirst

Logical; when TRUE, start each subject solve with the stiff solver instead of the non-stiff primary. Default FALSE.

autoSwitchSwitchMax

Non-negative integer; minimum number of integration intervals that must elapse after a switch before the solver is allowed to switch back in the opposite direction. Acts as an oscillation guard, and raises autoSwitchMaxNonstiff when it is larger. Set to 0L to leave the wait to autoSwitchMaxNonstiff alone. Default 5L.

stiff2

Integer method code for the stiff secondary solver used in AutoSwitch composite methods. Normally set automatically when method is a composite string of the form "primary+stiff" (e.g. "dop853+ros4"); the "+" notation is the preferred way to configure composite solving and stiff2 need not be supplied directly. When supplied as a raw integer it must be a stiff method code as returned by odeMethodToInt(); 0L (the default) disables the stiff secondary and causes the solver to run as a plain non-composite method. Dense output (dense = TRUE) is silently disabled when stiff2 names a stiff method that does not support dense output; "ros4" (code 13L) is the only stiff secondary that does support it.

useLinCmt
: Logical; when TRUE and a function-style (ini()/model()) model contains linear-compartment ODEs that can be solved analytically, automatically convert them to a linCmt() call before solving. The detection and conversion use odeToLin(); the converted model is cached so the compilation cost is paid only once. A model whose ODEs carry a term linCmt() cannot represent (an exogenous input such as transit() absorption, a zero-order or endogenous production rate, or a dose carried in a covariate column), or whose rates linCmt() would not rebuild exactly from the parameter names, or whose compartments it would renumber under event data addressing them by index, is solved with its original ODEs. Set to FALSE to keep the original ODE solver. This flag is also stored in the returned rxControl() object so that downstream hooks (e.g. in nlmixr2) can read and apply it. The default is a useLinCmt set in the model's meta block, else the rxode2.useLinCmt option, or FALSE when it is unset.

file
: Character string giving a file path prefix for out-of-memory chunk solving. When set, rxSolve() splits subjects into chunks, writes each chunk's result to <file>_chunk_NNNNN.parquet (or .rds if arrow is unavailable), and returns an rxSolveOom object. The manifest is saved to <file>_manifest.rds. Set to NULL (default) to use the normal in-memory path.

  A chunked solve draws the parameters once in the parent so that every chunk shares them, which is what keeps subjects in different chunks in the same study. Four things that draw cannot express are a clear error rather than a quietly different answer: a joint (TNPRI) draw of the omega/sigma entries a thetaMat carries, sigma uncertainty (dfObs > 0), a per-subject parameter data.frame/matrix alongside an omega/thetaMat, and nSub greater than the number of subjects the event table has.

chunkSize
: Integer; number of subjects per chunk when file is set. If NULL (default), the chunk size is auto-computed from available free RAM using rxMemoryEstimate().

parallel
: Integer; number of mirai daemons to use for parallel chunk solving when file is set. 0L (default) uses serial solving. Requires the mirai package.

zeroVarParamHandle
: Determines what happens when params supplies a value for an omega/sigma item whose variance is zero (say eta.base ~ fix(0)). Such an item is dropped from the matrix that is simulated from and given to the model as a literal zero instead; the options are:

  - warn (default) replace the supplied value with zero and warn
  - ignore replace the supplied value with zero silently
  - keep use the supplied value instead of zero

indLinStepSearch
: how method="indLin" picks the relaxation factor for its fixed-point iteration. "secant" (the default) estimates the iteration's contraction ratio from the last two residuals and costs nothing extra; "exact" spends one more matrix exponential per iteration to locate the residual-minimizing factor in closed form; "none" is plain, undamped Picard iteration. All three converge to the same answer – the relaxation does not move the fixed point – so this trades iterations against work per iteration.

`indLinMaxIter` the maximum number of `method="indLin"` fixed-point iterations per relinearization step. Running out is not an error: the iteration contracts in proportion to the step, so the solver reads it as a step that is too long and retries with a shorter one.

`indLinRichardson`

how far `method="indLin"` extrapolates each relinearization step. The base step is second order; running it also at half length and combining removes the leading error term for third order at three fixed-point solves, and adding a quarter-length run gives a three-entry Romberg column for fourth order at seven. Each level pays for itself only once the tolerance is tight enough that taking far fewer steps outweighs the extra solves per step: a second-order step needing `N` steps becomes a p-th order one needing about `N^(2/p)`, so third order wins once `3*N^(2/3) < N` (`N > 27`) and fourth once `7*N^(1/2) < 3*N^(2/3)` (`N > 161`). `"auto"` (the default) starts each interval second order and raises the level as the step the controller settles on crosses those thresholds. `"never"` (or `FALSE`), `"always"` (or `TRUE`, third order) and `"always4"` (fourth order) force the choice.

`indLinIteration`

how `method="indLin"` solves each relinearization step. `"picard"` is the relaxed fixed-point iteration; `"newton"` replaces it with a Newton iteration, which removes the contraction condition so convergence stops limiting the step; `"exprb"` uses an exponential Rosenbrock step, which does not iterate at all – it puts the full linearization into the exponential, so nothing can fail to converge. Their costs differ sharply by problem: on a non-stiff model the iteration never limits the step and Picard is cheapest, while on a stiff one it is the only thing limiting it. `"auto"` (the default) therefore starts on Picard and switches only once steps are actually being cut for non-convergence, so a model that never needs a Jacobian never forms one. `"exprb32"` is the Luan-Ostermann third-order exponential Rosenbrock pair, which carries its own embedded error estimate and so does not need the extrapolation column that `"exprb"` takes its estimate from.

`indLinJac` where the forcing Jacobian comes from when `method="indLin"` needs one, which is only under the `"newton"`, `"exprb"` and `"exprb32"` iteration schemes – Picard needs none, so a non-stiff model under the default `"auto"` scheme never forms one at all. `"symbolic"` uses the model's own analytic Jacobian, available because the `matExp()` conversion emits `df()/dy()` lines, and costs no extra forcing evaluations; `"fd"` central-differences the forcing, at `2n` evaluations per Jacobian. `"auto"` (the default) takes the symbolic one when the model carries it and falls back to finite differences otherwise – which is what happens above `getOption("rxode2.indLinJacMaxStates")` states, where the symbolic emission is skipped to bound the symengine work at model build, and on a sensitivity model, whose `df()/dy()` lines cover the physical states only.

`indLinForcing` how `method="indLin"` carries the `indLin()` forcing across one relinearization step. `"ramp"` (the default) evaluates it at both ends of the step and integrates the line between them exactly, with the rate matrix taken at the step midpoint; `"constant"` holds it fixed across the step and averages a start-linearized and an end-linearized answer to reach the same second order. The ramp step is symmetric, so its error expands in even powers of the step alone and `indLinRichardson` removes two orders per extrapolation level from it rather than one – which is where most of the difference between the two shows up, since the default

indLinRichardson="auto" extrapolates. It applies to the "picard" and "newton" schemes; the exponential Rosenbrock ones never freeze the forcing.

usePrior — Whether the prior distributions specified in the model's ini({}) block drive the uncertainty simulation. NA (or "auto", the default) uses them whenever the model has them and variability is being simulated (nStud > 1, or whatever simVariability forces); TRUE requires them, and is an error when the model has none or when no variability would be simulated; FALSE ignores them and falls back to a supplied thetaMat/dfSub. The priors take precedence over a thetaMat/dfSub carried in the model's meta block, with a warning; one given at the call site wins over the priors instead.

priorPdRetry — How many times a prior draw of a covariance matrix is retried when it is not positive definite. After that many tries the nearest positive definite matrix of the kept draws is used and a warning is given. Only prior draws retry; cvPost() is unchanged.

priorOmega — Internal. The per-block prior degrees of freedom of the omega, built from the ini({}) block; not meant to be set directly.

priorOmegaEl — Internal. Which thetaMat columns are omega elements of a joint prior; not meant to be set directly – use omegaSeparation="tnpri".

priorSigmaEl — Internal. Which thetaMat columns are sigma elements of a joint prior; not meant to be set directly – use sigmaSeparation="tnpri".

envir — is the environment to look for R user functions (defaults to parent environment)

a — when using solve(), this is equivalent to the object argument. If you specify object later in the argument list it overwrites this parameter.

b — when using solve(), this is equivalent to the params argument. If you specify params as a named argument, this overwrites the output

Details

The rest of the document focus on the different ODE solving methods, followed by the core solving method's options, rxode2 event handling options, rxode2's numerical stability options, rxode2's output options, and finally internal rxode2 options or compatibility options.

Value

An "rxSolve" solve object that stores the solved value in a special data.frame or other type as determined by returnType. By default this has as many rows as there are sampled time points and as many columns as system variables (as defined by the ODEs and additional assignments in the rxode2 model code). It also stores information about the call to allow dynamic updating of the solved object.

The operations for the object are similar to a data-frame, but expand the $ and [[""]] access operators and assignment operators to resolve based on different parameter values, initial conditions, solver parameters, or events (by updating the time variable).

You can call the eventTable() methods on the solved object to update the event table and resolve the system of equations.

Author(s)

Matthew Fidler, Melissa Hallow and Wenping Wang

References

"New Scaling and Squaring Algorithm for the Matrix Exponential", by Awad H. Al-Mohy and Nicholas J. Higham, August 2009

Roger B. Sidje (1998). EXPOKIT: Software package for computing matrix exponentials. ACM - Transactions on Mathematical Software *24*(1), 130-156.

Hindmarsh, A. C. *ODEPACK, A Systematized Collection of ODE Solvers*. Scientific Computing, R. S. Stepleman et al. (Eds.), North-Holland, Amsterdam, 1983, pp. 55-64.

Petzold, L. R. *Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations*. Siam J. Sci. Stat. Comput. 4 (1983), pp. 136-148.

Hairer, E., Norsett, S. P., and Wanner, G. *Solving ordinary differential equations I, nonstiff problems*. 2nd edition, Springer Series in Computational Mathematics, Springer-Verlag (1993).

See Also

`rxode2()`

`rxSolveAdjoint` *Solve a model with adjoint (backward) sensitivities, drop-in for forward sens*

Description

Solves `object` as `rxSolve()` would, then appends `rx__sens_<state>_BY_<param>__` columns computed via adjoint (backward) sensitivity analysis. Same column names and output structure as `rxSolve(object, ..., calcSens=)`. Prefer `.rxAdjointGrad()` when only a scalar objective gradient is needed.

Usage

```
rxSolveAdjoint(
  object,
  params,
  events,
  calcSens,
  adjStates = NULL,
  denseBy = 0.01,
  atol = 1e-08,
  rtol = 1e-06,
  ...
)
```

Arguments

object	model definition accepted by rxode2().
params	named numeric vector of parameter values.
events	event table / data used to define dosing and sampling times; sampling (evid==0) times become the sensitivity output times.
calcSens	character vector of parameter names to differentiate with respect to.
adjStates	character vector of output states of interest; defaults to all ODE states (full forward-sensitivity parity).
denseBy, atol, rtol	passed to the adjoint checkpoint solve; see .rxAdjointSolveEvalC().
...	additional arguments passed to the primal rxSolve() call.

Value

the standard rxSolve() output (as returnType="data.frame") with rx__sens_<state>_BY_<param>__ columns appended.

Author(s)

Matthew L. Fidler

---

rxSolveAdjointRk4 *Solve a model with an in-engine discrete-adjoint (s) method*

Description

Applies the (cached) adjoint expansion, solves with the requested adjoint method, and drops the internal rx__adj* lhs, returning the full-trajectory rx__sens_<state>_BY_<param>__ columns. The stiff analytic Jacobian is emitted into the expansion automatically, inferred from method.

Usage

```
rxSolveAdjointRk4(
  object,
  events,
  params = NULL,
  calcSens,
  method = "rk4s",
  ...
)
```

Arguments

object	model definition accepted by rxode2().
events	an rxode2 event table / data set.
params	optional named parameter vector or data frame (per subject).
calcSens	character vector of parameters to differentiate w.r.t.
method	adjoint solve method (default "rk4s"). May be a composite ("dop853s+ros4s").
...	passed to rxSolve().

Value

the solved data frame with rx__sens_<state>_BY_<param>__ columns.

Author(s)

Matthew L. Fidler

---

rxSolveChunked *Solve an ODE model in memory-safe chunks*

Description

Splits subjects into chunks sized to fit in available RAM, solves each chunk, and writes output to parquet (or rds) files. Returns an rxSolveOom object that lazily reads chunks on demand.

Usage

```
rxSolveChunked(
  object,
  params = NULL,
  events = NULL,
  inits = NULL,
  ...,
  chunkSize,
  seed = NULL,
  parallel = 0L
)
```

Arguments

object	rxode2 model
params	model parameters
events	event table or rxEtFile
inits	initial conditions
...	additional arguments passed to rxControl()
chunkSize	number of subjects per chunk (auto-computed from free RAM if omitted)
seed	random seed (sets before solving if not NULL)
parallel	number of mirai daemons for parallel chunk solving (0 = serial)

Details

The storage/query engine used by the resulting rxSolveOom object is controlled by the rxode2.oom.backend option:

"auto" (default) DuckDB if installed, else arrow, else rds.

"duckdb" lazy DuckDB SQL queries over arrow-written parquet.

"arrow" arrow parquet reads/writes, no DuckDB.

"rds" plain rds files, no arrow or DuckDB.

A requested engine that is not installed silently degrades (duckdb → arrow → rds).

Value

An rxSolveOom object

---

rxStack *Stack a solved object for things like default ggplot2 plot*

Description

Stack a solved object for things like default ggplot2 plot

Usage

```
rxStack(data, vars = NULL, doSim = TRUE, doIpredSim = TRUE)
```

Arguments

data	is a rxode2 object to be stacked.
vars	Variables to include in stacked data; By default this is all the variables when vars is NULL. When vars is sim and comes from a rxode2 ui simulation with multiple endpoints (ie it has a CMT in the simulation), it will rework the data as if it was stacked based the value based on the compartments in the multiple endpoint model. When the vars is sim.endpoint1 it will subset the stack to endpoint1, you can also have 'c("sim.endpoint1", "sim.endpoint2") and the "stack" will subset to endpoint1 and endpoint2. When you specify the sim type variables they have to be all prefixed with sim otherwise, the stack will not treat them differently.
doSim	boolean that determines if the "sim" variable in a rxSolve dataset is actually "stacking" based on the endpoint (TRUE) or simply treating sim as a variable.
doIpredSim	boolean that determines if the "ipredSim" variable in a rxSolve dataset is actually "stacking" based on the endpoint (TRUE) or simply treating ipredSim as a variable.

Value

Stacked data with `value` and `trt`, where value is the values and `trt` is the state and `lhs` variables.

Author(s)

Matthew Fidler

`rxState` *State variables*

Description

This returns the model's compartments or states.

Usage

```
rxState(obj = NULL, state = NULL)
```

Arguments

`obj`	rxode2 family of objects
`state`	is a string indicating the state or compartment that you would like to lookup.

Value

If state is missing, return a character vector of all the states.

If state is a string, return the compartment number of the named state.

Author(s)

Matthew L.Fidler

See Also

`rxode2()`

Other Query model information: `rxDfdy()`, `rxInits()`, `rxLhs()`, `rxModelVars()`, `rxParams()`

`rxStateOde` *Get the ODE states only*

Description

Get the ODE states only

Usage

```
rxStateOde(obj)
```

Arguments

`obj`	rxode2 object

Value

ODE states only

Author(s)

Matthew L. Fidler

Examples

```
mod <- rxode2({
  Cp <- linCmt(Cl, V, Q2, V2, Q3, V3)
  ke0 <- log(2)/(50)
  d/dt(Ce) <- (Cp-Ce)*ke0
})

rxStateOde(mod)

rxState(mod)

mod <- rxode2({
  Cp <- linCmt(Cl, V, Q2, V2, Q3, V3, ka)
  ke0 <- log(2)/(50)
  d/dt(Ce) <- (Cp-Ce)*ke0
}, linCmtSens="linCmtB")

rxStateOde(mod)

rxState(mod)
```

rxSumProdModel *Recast model in terms of sum/prod*

Description

Recast model in terms of sum/prod

Usage

```
rxSumProdModel(model, expand = FALSE, sum = TRUE, prod = TRUE)
```

Arguments

model	rxode2 model
expand	Boolean indicating if the expression is expanded.
sum	Use sum(...)
prod	Use prod(...)

Value

model string with prod(.) and sum(.) for all these operations.

Author(s)

Matthew L. Fidler

rxSupportedFuns *Get list of supported functions*

Description

Get list of supported functions

Usage

```
rxSupportedFuns()
```

Value

list of supported functions in rxode2

Examples

```
rxSupportedFuns()
```

rxSuppressMsg *Respect suppress messages*

Description

This turns on the silent REprintf in C when suppressMessages() is turned on. This makes the REprintf act like messages in R, they can be suppressed with suppressMessages()

Usage

```
rxSuppressMsg()
```

Value

Nothing

Author(s)

Matthew Fidler

Examples

```
# rxSupressMsg() is called with rxode2()

# Note the errors are output to the console

try(rxode2("d/dt(matt)=/3"), silent = TRUE)

# When using suppressMessages, the output is suppressed

suppressMessages(try(rxode2("d/dt(matt)=/3"), silent = TRUE))

# In rxode2, we use REprintf so that interrupted threads do not crash R
# if there is a user interrupt. This isn't captured by R's messages, but
# This interface allows the `suppressMessages()` to suppress the C printing
# as well

# If you  want to suppress messages from rxode2 in other packages, you can use
# this function
```

`rxSymInvChol` *Get Omega^-1 and derivatives*

Description

Get Omega^-1 and derivatives

Usage

```
rxSymInvChol(
  invObjOrMatrix,
  theta = NULL,
  type = "cholOmegaInv",
  thetaNumber = 0L
)
```

Arguments

`invObjOrMatrix`	Object for inverse-type calculations. If this is a matrix, setup the object for inversion `rxSymInvCholCreate()` with the default arguments and return a reactive s3 object. Otherwise, use the inversion object to calculate the requested derivative/inverse.
`theta`	Thetas to be used for calculation. If missing (`NULL`), a special s3 class is created and returned to access `Omega^1` objects as needed and cache them based on the theta that is used.
`type`	The type of object. Currently the following types are supported: • `cholOmegaInv` gives the Cholesky decomposition of the Omega Inverse matrix. • `omegaInv` gives the Omega Inverse matrix. • `d(omegaInv)` gives the `d(Omega^-1)` with respect to the theta parameter specified in `thetaNumber`. • `d(D)` gives the `d(diagonal(Omega^-1))` with respect to the theta parameter specified in the `thetaNumber` parameter
`thetaNumber`	For types `d(omegaInv)` and `d(D)`, the theta number that the derivative is taken against. This must be positive from 1 to the number of thetas defining the Omega matrix.

Value

Matrix based on parameters or environment with all the matrixes calculated in variables `omega`, `omegaInv`, `dOmega`, `dOmegaInv`.

Author(s)

Matthew L. Fidler

rxSyncOptions *Sync options with rxode2 variables*

Description

Sync options with rxode2 variables

Usage

```
rxSyncOptions(setDefaults = c("none", "permissive", "strict"))
```

Arguments

setDefaults	This will setup rxode2's default solving options with the following options: • "none" leave the options alone • "permissive" This is a permissive option set similar to R language specifications. • "strict" This is a strict option set similar to the original rxode2(). It requires semicolons at the end of lines and equals for assignment

Value

nothing; called for side effects

Author(s)

Matthew L. Fidler

rxSyntaxFunctions *A list and description of Rode supported syntax functions*

Description

A list and description of Rode supported syntax functions

Usage

```
rxSyntaxFunctions
```

Format

A data frame with 3 columns and 119 rows

Function Reserved function Name

Description Description of function

Aliases Function Aliases

rxt *Simulate student t variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the sitmo threefry, this currently generates one random deviate from the uniform distribution to seed the engine threefry and then run the code.

Usage

```
rxt(df, n = 1L, ncores = 1L)
```

Arguments

df	degrees of freedom (> 0, maybe non-integer). df = Inf is allowed.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
ncores	Number of cores for the simulation rxnorm simulates using the threefry sitmo generator. rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the rxode2 environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with rxSetSeed()

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

t-distribution random numbers

Examples

```
## Use threefry engine
rxt(df = 3, n = 10) # with rxt you have to explicitly state n
rxt(df = 3, n = 10, ncores = 2) # You can parallelize the simulation using openMP
```

```
rxt(4) ## The first argument is the df parameter


## This example uses `rxt` directly in the model

rx <- function() {
   model({
    a <- rxt(3)
   })
}

et <- et(1, id = 1:2)

s <- rxSolve(rx, et)
```

---

rxTempDir *Get the rxode2 temporary directory*

Description

Get the rxode2 temporary directory

Usage

```
rxTempDir()
```

Value

rxode2 temporary directory.

---

rxTheme *rxTheme is the ggplot2 theme for rxode2 plots*

Description

rxTheme is the ggplot2 theme for rxode2 plots

Usage

```
rxTheme(
  base_size = 11,
  base_family = "",
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  grid = TRUE
)
```

Arguments

base_size	base font size, given in pts.
base_family	base font family
base_line_size	base size for line elements
base_rect_size	base size for rect elements
grid	a Boolean indicating if the grid is on (TRUE) or off (FALSE). This could also be a character indicating x or y.

Value

ggplot2 theme used in rxode2

See Also

Other rxode2 plotting: plot.rxSolve()

---

rxToSE *rxode2 to symengine environment*

Description

rxode2 to symengine environment

Usage

```
rxToSE(
  x,
  envir = NULL,
  progress = FALSE,
  promoteLinSens = TRUE,
  parent = parent.frame()
)

.rxToSE(x, envir = NULL, progress = FALSE)

rxFromSE(
  x,
  unknownDerivatives = c("forward", "central", "error"),
  parent = parent.frame()
)

.rxFromSE(x)
```

Arguments

x	expression
envir	default is NULL; Environment to put symengine variables in.
progress	shows progress bar if true.
promoteLinSens	Promote solved linear compartment systems to sensitivity-based solutions.
parent	is the parent environment to look for R-based user functions
unknownDerivatives	When handling derivatives from unknown functions, the translator will translate into different types of numeric derivatives. The currently supported methods are: - `forward` for forward differences - `central` for central differences - `error` for throwing an error for unknown derivatives

Value

An rxode2 symengine environment

Author(s)

Matthew L. Fidler

---

rxTrans *Translate the model to C code if needed*

Description

This function translates the model to C code, if needed

Usage

```
rxTrans(
  model,
  modelPrefix = "",
  md5 = "",
  modName = NULL,
  modVars = FALSE,
  ...
)

## Default S3 method:
rxTrans(
  model,
  modelPrefix = "",
  md5 = "",
```

```
  modName = NULL,
  modVars = FALSE,
  ...
)

## S3 method for class 'character'
rxTrans(
  model,
  modelPrefix = "",
  md5 = "",
  modName = NULL,
  modVars = FALSE,
  eventSensKey = .rxState$eventSensCacheKey,
  ...
)
```

Arguments

model	This is the ODE model specification. It can be: • a string containing the set of ordinary differential equations (ODE) and other expressions defining the changes in the dynamic system. • a file name where the ODE system equation is contained An ODE expression enclosed in \{\} (see also the filename argument). For details, see the sections "Details" and rxode2 Syntax below.
modelPrefix	Prefix of the model functions that will be compiled to make sure that multiple rxode2 objects can coexist in the same R session.
md5	Is the md5 of the model before parsing, and is used to embed the md5 into DLL, and then provide for functions like rxModelVars().
modName	a string to be used as the model name. This string is used for naming various aspects of the computations, including generating C symbol names, dynamic libraries, etc. Therefore, it is necessary that modName consists of simple ASCII alphanumeric characters starting with a letter.
modVars	returns the model variables instead of the named vector of translated properties.
...	Ignored parameters.
eventSensKey	event ("jump") sensitivity mode key folded into the parsed md5 cache key so the same model text translated in different modes (e.g. "fd" vs "jump") compiles to distinct DLLs instead of colliding on a cached translation. Defaults to the session global .rxState$eventSensCacheKey (set by rxode2() before parsing); an empty string leaves the md5 unchanged so the legacy "fd" path keeps existing caches valid. It is a formal argument (rather than read from the global in the body) because rxTrans.character is memoised and memoise keys on default-valued formals, so the cache distinguishes the modes.

Value

a named vector of translated model properties including what type of jacobian is specified, the C function prefixes, as well as the C functions names to be called through the compiled model.

Author(s)

Matthew L.Fidler

See Also

rxode2(), rxCompile().

`rxUdfUiControl` *Return the control that is being processed or setup control for processing*

Description

Return the control that is being processed or setup control for processing

Usage

```
rxUdfUiControl(value)
```

Arguments

value	when specified, this assigns the control to be processed, or resets it by assigning it to be NULL.

Value

value of the data.frame being processed or NULL.

Author(s)

Matthew L. Fidler

See Also

Other User functions: linMod(), rxUdfUiData(), rxUdfUiEst(), rxUdfUiIniLhs(), rxUdfUiMv(), rxUdfUiNum(), rxUdfUiParsing()

Examples

```
rxUdfUiControl()
```

rxUdfUiData — *Return the data.frame that is being processed or setup data.frame for processing*

Description

Return the data.frame that is being processed or setup data.frame for processing

Usage

```
rxUdfUiData(value)
```

Arguments

value	when specified, this assigns the data.frame to be processed, or resets it by assigning it to be NULL.

Value

value of the data.frame being processed or NULL.

Author(s)

Matthew L. Fidler

See Also

Other User functions: linMod(), rxUdfUiControl(), rxUdfUiEst(), rxUdfUiIniLhs(), rxUdfUiMv(), rxUdfUiNum(), rxUdfUiParsing()

Examples

```
rxUdfUiData()
```

rxUdfUiEst — *Return the current estimation method for the UI processing*

Description

Return the current estimation method for the UI processing

Usage

```
rxUdfUiEst(value)
```

Arguments

value	when specified, this assigns the character value of the estimation method or NULL if there is nothing being estimated

Value

value of the estimation method being processed or NULL

Author(s)

Matthew L. Fidler

See Also

Other User functions: linMod(), rxUdfUiControl(), rxUdfUiData(), rxUdfUiIniLhs(), rxUdfUiMv(), rxUdfUiNum(), rxUdfUiParsing()

Examples

```
rxUdfUiEst()
```

---

rxUdfUiExpr *Give the expression as a compressed model or expression*

Description

Here it means that it evaluates to a variable or a number.

Usage

```
rxUdfUiExpr(expr, env = parent.frame())
```

Arguments

expr	the R language expression to evaluate
env	the environment in which to evaluate the expression

Value

expression

Author(s)

Matthew L. Fidler

rxUdfUiFlag *Is the expression actually a flag that can be used in the rxUdfUi functions?*

Description

This is useful when writing replacement UI functions

Usage

```
rxUdfUiFlag(expr, arg = "arg", funName = "fun", env = baseenv())
```

Arguments

expr	expression to evaluate
arg	argument name for error messages for the argument name in the rxUdfUi extension when the expression is not logical
funName	function name for the error message for the argument name in the rxUdfUi extension when the expression is not logical.
env	the environment in which to evaluate the expression (in case it is numeric)

Value

logical value of the expression if it can be evaluated to a scalar TRUE/FALSE value, otherwise an error is thrown.

Author(s)

Matthew L. Fidler

rxUdfUiIniDf *Get the rxode2 iniDf of the current UI being processed (or return NULL)*

Description

Get the rxode2 iniDf of the current UI being processed (or return NULL)

Usage

```
rxUdfUiIniDf()
```

Value

Initial data.frame being processed or NULL for nothing.

Author(s)

Matthew L. Fidler

Examples

```
rxUdfUiIniDf()
```

---

rxUdfUiIniLhs *Return the lhs parsed language expression*

Description

Return the lhs parsed language expression

Usage

```
rxUdfUiIniLhs()
```

Value

lhs language expression or NULL

Author(s)

Matthew L. Fidler

See Also

Other User functions: `linMod()`, `rxUdfUiControl()`, `rxUdfUiData()`, `rxUdfUiEst()`, `rxUdfUiMv()`, `rxUdfUiNum()`, `rxUdfUiParsing()`

Examples

```
rxUdfUiIniLhs()
```

rxUdfUiIsValue *Is the expression actually equal to a value?*

Description

This is used to determine if the location and scale parameters are actually equal to their default values, which allows for more efficient translation to expit expressions when possible.

Usage

```
rxUdfUiIsValue(expr, value, env = baseenv())
```

Arguments

expr	the R language expression to evaluate
value	the value to compare against
env	the environment in which to evaluate the expression

Value

TRUE if the expression evaluates to a scalar value equal to the specified value, FALSE otherwise

Author(s)

Matthew L. Fidler

rxUdfUiLhs.dist *Declare a non-Gaussian random effect distribution in the model block*

Description

The `model({})` form of `ini({})`'s `dist()` line:

Usage

```
## S3 method for class 'dist'
rxUdfUiLhs(fun, rhs)
```

Arguments

fun	the `dist(<eta>)` call
rhs	the declared distribution, eg `dgamma(shape=a, rate=b)`

Details

```
dist(eta.cl) ~ dgamma(shape = 1/exp(lclrv), rate = 1/(exp(lclrv)*aCl))
```

Reached through rxUdfUiLhs(), a user-function dispatch on the LEFT of a model line – `dist(eta.cl)` is not rxode2 grammar and ~ is already overloaded, so the UI claims the whole line before rxode2 ever sees it.

The point of the model-block form is that a distribution parameter can be any expression the model has already computed, including one built from covariates:

```
aCl <- exp(lclm + bWT*(WT - 70))
dist(eta.cl) ~ dgamma(shape = 1/exp(lclrv), rate = 1/(exp(lclrv)*aCl))
```

The `ini({})` form cannot express that – it is parsed before the model block exists, so its arguments can only name population parameters. This one emits the inverse-CDF line IN PLACE, at the declaration, so everything above it is in scope.

What it leaves for rxEtaDistExpand() is the part that needs the whole picture rather than one line: the latent normals, the Gaussian copula that correlates them, and the `rxCor.*` thetas. Those are prepended, so the `rxN.*` this line reads are defined above it.

Value

`rxUdfUiLhs()` list: the modified `iniDf` and the inverse-CDF line that replaces the declaration

Author(s)

Matthew L. Fidler

`rxUdfUiMv` *Return the model variables that is being processed or setup model variables for processing*

Description

Return the model variables that is being processed or setup model variables for processing

Usage

```
rxUdfUiMv(value)
```

Arguments

`value`	when specified, this assigns the model variables to be processed, or resets it by assigning it to be `NULL`.

Value

value of the `modelVariables` being processed or `NULL`.

Author(s)

Matthew L. Fidler

See Also

Other User functions: `linMod()`, `rxUdfUiControl()`, `rxUdfUiData()`, `rxUdfUiEst()`, `rxUdfUiIniLhs()`, `rxUdfUiNum()`, `rxUdfUiParsing()`

Examples

```
rxUdfUiMv()
```

`rxUdfUiNum` *This gives the current number in the ui of the particular function being called.*

Description

If this is called outside of function parsing or the input is unexpected this returns 1L. This is useful when writing replacement UI functions

Usage

```
rxUdfUiNum()
```

Value

integer greater than 1L

Author(s)

Matthew L. Fidler

See Also

Other User functions: `linMod()`, `rxUdfUiControl()`, `rxUdfUiData()`, `rxUdfUiEst()`, `rxUdfUiIniLhs()`, `rxUdfUiMv()`, `rxUdfUiParsing()`

Examples

```
rxUdfUiNum()
```

rxUdfUiParsing *Returns if the current ui function is being parsed*

Description

Returns if the current ui function is being parsed

Usage

```
rxUdfUiParsing()
```

Value

logical if the current ui function is being parsed

Author(s)

Matthew L. Fidler

See Also

Other User functions: `linMod()`, `rxUdfUiControl()`, `rxUdfUiData()`, `rxUdfUiEst()`, `rxUdfUiIniLhs()`, `rxUdfUiMv()`, `rxUdfUiNum()`

Examples

```
rxUdfUiParsing()
```

rxUiDecompress *Compress/Decompress* rxode2 *ui*

Description

Compress/Decompress rxode2 ui

Usage

```
rxUiDecompress(ui)

rxUiCompress(ui)
```

Arguments

ui	rxode2 ui object

Value

A compressed or decompressed rxui 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) | tmp
  })
}

f <- rxode2(one.cmt)
print(class(f))
print(is.environment(f))

f  <- rxUiDecompress(f)
print(class(f))
print(is.environment(f))

f  <- rxUiCompress(f)
print(class(f))
print(is.environment(f))
```

`rxUiDeparse` *This is a generic function for deparsing certain objects when printing out a rxode2 object. Currently it is used for any meta-information*

Description

This is a generic function for deparsing certain objects when printing out a rxode2 object. Currently it is used for any meta-information

rxUiDeparse.rxControl(rxControl(covsInterpolation="linear", method="dop853", naInterpolation="nocb", keepInterpolation="nocb", sigmaXform="variance", omegaXform="variance", returnType="data.frame", sumType="fsum", prodType="logify"), "ctl")

Usage

```
rxUiDeparse(object, var)

## S3 method for class 'lotriFix'
rxUiDeparse(object, var)

## Default S3 method:
rxUiDeparse(object, var)

## S3 method for class 'rxControl'
rxUiDeparse(object, var)
```

Arguments

`object`	object to be deparsed
`var`	variable name to be assigned

Value

parsed R expression that can be used for printing and `as.function()` calls.

Author(s)

Matthew L. Fidler

Examples

```
mat <- matrix(c(1, 0.1, 0.1, 1), 2, 2, dimnames=list(c("a", "b"), c("a", "b")))

rxUiDeparse(matrix(c(1, 0.1, 0.1, 1), 2, 2, dimnames=list(c("a", "b"), c("a", "b"))), "x")
```

`rxUiDevelop` *rxUiDevelop - Enable/Disable rxUi development. Here all $ completions are given*

Description

rxUiDevelop - Enable/Disable rxUi development. Here all $ completions are given

Usage

```
rxUiDevelop(enable = TRUE)
```

Arguments

enable	boolean to enable/disable rxUi development mode.

Value

nothing, called for side effects

Author(s)

Matthew L. Fidler

Examples

```
rxUiDevelop(TRUE)
rxUiDevelop(FALSE)
```

---

rxUiEtaDistAnchors *Where each declared distribution's arguments are computed in the model*

Description

`rxEtaDistExpand()` hoists every family argument onto its own model line, `rxEdA.<eta>.<role>`, so a declaration's arguments are computed per observation by the compiled model like any other model quantity – covariates included, through the ordinary covariate machinery, inside the ODE model pool. This says which model variable holds each argument, so an estimator can READ them from the solve instead of evaluating the argument expressions a second time in its own code.

Usage

```
rxUiEtaDistAnchors(ui, d = NULL)
```

Arguments

ui	rxode2 ui, model function, or solve-ready object – the EXPANDED model, since that is what carries the anchor lines
d	the declarations, as `rxUiEtaDists()` returns them (`name` and `etaDist` columns). Defaults to reading them off `ui`, which works before the expansion but NOT after: `rxEtaDistExpand()` removes `etaDist` from the `iniDf`, so an expanded ui reports no declarations and this would return an empty list. An estimator holds the declarations it stashed before expanding and should pass them.

Details

The order is the declaration's ARGUMENT order, which `rxEtaDistExpand()` normalizes at the storage point, so element `t` lines up with argument `t` of the family.

`NA` means that argument has no anchor: a family whose quantile template never references it gets no model line for it (a normal-based family collapses to the latent). An estimator must fall back to its own value there, and a covariate in such an argument is invisible to the model – so `NA` alongside a covariate is a gap, not a value to work around.

Value

named list, one character vector per declared random effect, in argument order; empty list when nothing is declared

Author(s)

Matthew L. Fidler

Examples

```
if (requireNamespace("lotri", quietly = TRUE) &&
      "lotriEtaDists" %in% getNamespaceExports("lotri")) {
mod <- function() {
  ini({
    lclm <- log(5)
    lclrv <- log(0.09)
    tv <- 3.45
    dist(eta.cl) ~ dgamma(shape = 1 / exp(lclrv),
                          rate = 1 / (exp(lclrv) * exp(lclm)))
    add.sd <- 0.7
  })
  model({
    cl <- eta.cl
    v <- exp(tv)
    linCmt() ~ add(add.sd)
  })
}
.d <- rxUiEtaDists(mod)
rxUiEtaDistAnchors(rxEtaDistExpand(mod(), param = "direct"), .d)
}
```

`rxUiEtaDists` *The random effects that declare a distribution*

Description

The `etaDist` column only exists when the installed 'lotri' supports declared random effect distributions AND the model uses one, so its absence means "no declarations" rather than an error.

Usage

```
rxUiEtaDists(ui)

testRxUiEtaDist(ui)

assertRxUiNoEtaDist(ui, extra = "")
```

Arguments

ui	rxode2 ui
extra	text appended to the error, naming what cannot use them

Value

data frame of the declaring random effects, with `name`, `etaDist`, `neta1` and `condition` columns; zero rows when there are none

Author(s)

Matthew L. Fidler

---

rxUiGet.cmtLines *S3 for getting information from UI model*

Description

S3 for getting information from UI model

Usage

```
## S3 method for class 'cmtLines'
rxUiGet(x, ...)

## S3 method for class 'dvidLine'
rxUiGet(x, ...)

## S3 method for class 'paramsLine'
rxUiGet(x, ...)

## S3 method for class 'interpLines'
rxUiGet(x, ...)

## S3 method for class 'splitDose'
rxUiGet(x, ...)

## S3 method for class 'splitDoseLines'
rxUiGet(x, ...)
```

```
## S3 method for class 'splitInfusion'
rxUiGet(x, ...)

## S3 method for class 'splitInfusionLines'
rxUiGet(x, ...)

## S3 method for class 'splitInfusionBolus'
rxUiGet(x, ...)

## S3 method for class 'splitInfusionBolusLines'
rxUiGet(x, ...)

## S3 method for class 'splitBolusInfusion'
rxUiGet(x, ...)

## S3 method for class 'splitBolusInfusionLines'
rxUiGet(x, ...)

## S3 method for class 'simulationSigma'
rxUiGet(x, ...)

## S3 method for class 'simulationModel'
rxUiGet(x, ...)

## S3 method for class 'symengineModelNoPrune'
rxUiGet(x, ...)

## S3 method for class 'symengineModelPrune'
rxUiGet(x, ...)

## S3 method for class 'simulationIniModel'
rxUiGet(x, ...)

rxUiGet(x, ...)

## S3 method for class 'levels'
rxUiGet(x, ...)

## S3 method for class 'state'
rxUiGet(x, ...)

## S3 method for class 'stateDf'
rxUiGet(x, ...)

## S3 method for class 'statePropDf'
rxUiGet(x, ...)
```

```
## S3 method for class 'props'
rxUiGet(x, ...)

## S3 method for class 'theta'
rxUiGet(x, ...)

## S3 method for class 'lstChr'
rxUiGet(x, ...)

## S3 method for class 'omega'
rxUiGet(x, ...)

## S3 method for class 'omegaSameMap'
rxUiGet(x, ...)

## S3 method for class 'funTxt'
rxUiGet(x, ...)

## S3 method for class 'allCovs'
rxUiGet(x, ...)

## S3 method for class 'muRefTable'
rxUiGet(x, ...)

## S3 method for class 'multipleEndpoint'
rxUiGet(x, ...)

## S3 method for class 'funPrint'
rxUiGet(x, ...)

## S3 method for class 'fun'
rxUiGet(x, ...)

## S3 method for class 'funPartsDigest'
rxUiGet(x, ...)

## S3 method for class 'md5'
rxUiGet(x, ...)

## S3 method for class 'sha1'
rxUiGet(x, ...)

## S3 method for class 'ini'
rxUiGet(x, ...)

## S3 method for class 'iniFun'
rxUiGet(x, ...)
```

```
## S3 method for class 'modelFun'
rxUiGet(x, ...)

## S3 method for class 'model'
rxUiGet(x, ...)

## S3 method for class 'modelDesc'
rxUiGet(x, ...)

## S3 method for class 'thetaLower'
rxUiGet(x, ...)

## S3 method for class 'thetaUpper'
rxUiGet(x, ...)

## S3 method for class 'lhsVar'
rxUiGet(x, ...)

## S3 method for class 'varLhs'
rxUiGet(x, ...)

## S3 method for class 'lhsEta'
rxUiGet(x, ...)

## S3 method for class 'lhsTheta'
rxUiGet(x, ...)

## S3 method for class 'lhsCov'
rxUiGet(x, ...)

## S3 method for class 'etaLhs'
rxUiGet(x, ...)

## S3 method for class 'thetaLhs'
rxUiGet(x, ...)

## S3 method for class 'covLhs'
rxUiGet(x, ...)

## S3 method for class 'levelLhs'
rxUiGet(x, ...)

## Default S3 method:
rxUiGet(x, ...)

## S3 method for class 'modelName'
rxUiGet(x, ...)
```

Arguments

x	list of (UIenvironment, exact). UI environment is the parsed function for rxode2. exact is a boolean that says if an exact match is required.
...	Other arguments

Value

value that was requested from the UI object

Author(s)

Matthew Fidler

---

rxUiPriors *Priors specified in a model*

Description

This is the accessor an estimation method uses to *implement* priors, as opposed to the assertRxUi* functions which reject the ones it cannot implement.

Usage

```
rxUiPriors(ui)
```

Arguments

ui	rxode2 ui model

Value

data frame of the parameters that carry a prior, with the columns name, prior (the prior as written, ie "invWishart(4)"), neta1/neta2 (NA for a population parameter) and lower/upper from the parameter, which is what gives a truncated prior its bounds. Zero rows when the model has no priors, and also when the iniDf has no prior column at all.

Author(s)

Matthew L. Fidler

See Also

Other Assertions: assertCompartmentExists(), assertCompartmentName(), assertCompartmentNew(), assertRxUi(), assertVariableExists(), assertVariableNew(), rxPriorBuildSpec(), rxPriorLogDensity(), rxPriorOmegaToCholOmegaInvGrad(), testIniDf(), testRxUnbounded()

Examples

```
one.cmt <- function() {
 ini({
   tka <- 0.45; label("Ka")
   tcl <- log(c(0, 2.7, 100)); label("Cl")
   tv <- 3.45; label("V")
   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)
 })
}

# a model with no priors gives a zero row data frame; a prior is added
# with `prior(tka) ~ dnorm(0, 10)` in the `ini({})` block, which needs a
# 'lotri' new enough to support them
rxUiPriors(one.cmt)
```

`rxunif` *Simulate uniform variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxunif(min = 0, max = 1, n = 1L, ncores = 1L)
```

Arguments

`min, max`	lower and upper limits of the distribution. Must be finite.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator.

rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the rxode2 environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with rxSetSeed()

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

uniform random numbers

Examples

```
## Use threefry engine

rxunif(min = 0, max = 4, n = 10) # with rxunif you have to explicitly state n
rxunif(min = 0, max = 4, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxunif()


## This example uses `rxunif` directly in the model

rx <- function() {
  model({
    a <- rxunif(0, 3)
  })
}

et <- et(1, id = 1:2)


s <- rxSolve(rx, et)
```

rxUnloadAll *Unloads all rxode2 compiled DLLs*

Description

Unloads all rxode2 compiled DLLs

Usage

```
rxUnloadAll(set = TRUE)
```

Arguments

set	If specified and TRUE, unloads all rxode2 dlls. If specified and FALSE, block a simple rxUnloadAll() will not actually unload all dlls (helps with CRAN ASAN check)

Value

boolean telling if rxUnloadAll() completed the unloading procedure

Examples

```
# print(rxUnloadAll())
```

rxUse *Use model object in your package*

Description

Use model object in your package

Usage

```
rxUse(obj, overwrite = TRUE, compress = "bzip2", internal = FALSE)
```

Arguments

obj	model to save.
overwrite	By default, use_data() will not overwrite existing files. If you really want to do so, set this to TRUE.
compress	Choose the type of compression used by save(). Should be one of "gzip", "bzip2", or "xz".
internal	If this is run internally. By default this is FALSE

Value

Nothing; This is used for its side effects and shouldn't be called by a user

rxValidate *Validate rxode2 This allows easy validation/qualification of nlmixr by running the testing suite on your system.*

Description

Validate rxode2 This allows easy validation/qualification of nlmixr by running the testing suite on your system.

Usage

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

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

Arguments

type	Type of test or filter of test type, When this is an expression, evaluate the contents, respecting skipOnCran. In the expression form, stray progress messages are muffled unless options(rxode2.test.verbose = TRUE) is set.
skipOnCran	when TRUE skip the test on CRAN.

Value

nothing

Author(s)

Matthew L. Fidler

rxweibull *Simulate Weibull variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the sitmo threefry, this currently generates one random deviate from the uniform distribution to seed the engine threefry and then run the code.

Usage

```
rxweibull(shape, scale = 1, n = 1L, ncores = 1L)
```

Arguments

shape, scale	shape and scale parameters, the latter defaulting to 1.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
ncores	Number of cores for the simulation rxnorm simulates using the threefry sitmo generator. rxnormV used to simulate with the vandercorput simulator, but since it didn't satisfy the normal properties it was changed to simple be an alias of rxnorm. It is no longer supported in rxode2({}) blocks

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the rxode2 environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with rxSetSeed()

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

Weibull random deviates

Examples

```
## Use threefry engine

# with rxweibull you have to explicitly state n
rxweibull(shape = 1, scale = 4, n = 10)

# You can parallelize the simulation using openMP
rxweibull(shape = 1, scale = 4, n = 10, ncores = 2)

rxweibull(3)


## This example uses `rxweibull` directly in the model

rx <- function() {
  model({
    a <- rxweibull(1, 3)
  })
}

et <- et(1, id = 1:2)
```

```
s <- rxSolve(rx, et)
```

`rxWithSeed` *Preserved seed and possibly set the seed*

Description

Preserved seed and possibly set the seed

Usage

```
rxWithSeed(
  seed,
  code,
  rxseed = rxGetSeed(),
  kind = "default",
  normal.kind = "default",
  sample.kind = "default"
)

rxWithPreserveSeed(code)
```

Arguments

`seed`	R seed to use for the session
`code`	Is the code to evaluate
`rxseed`	is the rxode2 seed that is being preserved
`kind`	character or `NULL`. If `kind` is a character string, set R's RNG to the kind desired. Use `"default"` to return to the R default. See 'Details' for the interpretation of `NULL`.
`normal.kind`	character string or `NULL`. If it is a character string, set the method of Normal generation. Use `"default"` to return to the R default. `NULL` makes no change.
`sample.kind`	character string or `NULL`. If it is a character string, set the method of discrete uniform generation (used in `sample`, for instance). Use `"default"` to return to the R default. `NULL` makes no change.

Value

returns whatever the code is returning

See Also

rxGetSeed, rxSetSeed

Examples

```
rxGetSeed()
rxWithSeed(1, {
   print(rxGetSeed())
   rxnorm()
   print(rxGetSeed())
   rxnorm()
}, rxseed=3)
```

SELU *Scaled Exponential Linear Unit (SELU) Activation Function*

Description

Scaled Exponential Linear Unit (SELU) Activation Function

Usage

```
SELU(x)
```

Arguments

x	A numeric vector. All elements must be finite and non-missing.

Value

A numeric vector where the ReLU function has been applied to each element of x.

Author(s)

Matthew Fidler

See Also

Other Activation Functions: ELU(), GELU(), PReLU(), ReLU(), Swish(), dELU(), dGELU(), dPReLU(), dReLU(), dSELU(), dSwish(), dlReLU(), dsoftplus(), lReLU(), softplus()

Examples

```
SELU(c(-1, 0, 1, 2))


# Can also be used in rxode2:
x <- rxode2({
   r=SELU(time)
})

e <- et(c(-1, 0, 1, 2))
```

```
rxSolve(x, e)
```

`softplus` *Softplus Activation Function*

Description

Softplus Activation Function

Usage

```
softplus(x)
```

Arguments

`x`	numeric vector

Value

numeric vector

Author(s)

Matthew L. Fidler

See Also

Other Activation Functions: `ELU()`, `GELU()`, `PReLU()`, `ReLU()`, `SELU()`, `Swish()`, `dELU()`, `dGELU()`, `dPReLU()`, `dReLU()`, `dSELU()`, `dSwish()`, `dlReLU()`, `dsoftplus()`, `lReLU()`

Examples

```
softplus(c(-1, 0, 1, 2))


# You can use rxode2 too:

r <- rxode2({
 s <- softplus(time)
})

e <- et(c(-1, 0, 1, 2))

rxSolve(r, e)
```

splitBolus *Split bolus doses across compartments inside an rxode2 model*

Description

`splitBolus()` is a model-only directive that rewrites bolus doses aimed at `cmt` into parallel bolus doses to the target compartments. Each target compartment receives the full original amount. The source-compartment bolus is replaced, not retained.

This rewrite applies both to event tables translated by `etTrans()` and to future doses scheduled during solving with `evid_()`.

The source compartment may also appear in the target list, but the target compartments in `...` must be unique.

Usage

```
splitBolus(cmt, ...)
```

Arguments

`cmt`	Source compartment name.
`...`	Target compartment names. Provide at least one target compartment.

Value

This function is only meaningful inside an rxode2 model; it errors when called directly from R.

splitBolusInfusion-directive *Split bolus doses into bolus and infusion paths (model directive)*

Description

`splitBolusInfusion()` is a model-only directive that rewrites doses aimed at `cmt` into parallel doses to the target compartments, regardless of whether the dose record is a bolus or an infusion. Each target compartment receives the full original amount; use `f()` to scale the split.

A plain bolus record is split so the LAST target compartment receives a modeled infusion start/stop pair (it must declare a modeled `dur()` or `rate()` property, otherwise its copies stay boluses) while every other target receives a bolus copy. Use `splitInfusionBolus()` to give the FIRST target the infusion instead. This feeds one bolus dose record into both a bolus path and an infusion path (Monolix-style double absorption with mixed first- and zero-order routes). Steady-state bolus records are not promoted; they are copied as boluses with a warning when the designated infusion target declares `dur()`/`rate()`.

Infusion records (data `RATE`/`DUR` or modeled `rate()`/`dur()`) are split preserving their type, exactly like `splitInfusion()`.

This rewrite applies at `etTrans()` translation time only; doses pushed while solving with `evid_()` are not split. Only one splitting directive (`splitBolus()`, `splitInfusion()`, `splitInfusionBolus()` or `splitBolusInfusion()`) is allowed per model.

There is no R function named `splitBolusInfusion()` exported by rxode2; the directive is parsed from the model text directly, as shown below.

Usage inside a model

```
model({
  splitBolusInfusion(depot, depot2, central)
  dur(central) <- tk0
  f(depot2) <- 1 - f1
  f(central) <- f1
  d/dt(depot2) <- -ka2 * depot2
  d/dt(central) <- ka2 * depot2 - cl / v * central
  ...
})
```

A bolus dose recorded against `depot` is split so `depot2` receives the bolus (first-order input scaled by `1 - f1`) and `central` (last target) receives a modeled-duration infusion (zero-order input over `tk0`, scaled by `f1`).

---

splitInfusion *Split infusion doses across compartments inside an rxode2 model*

Description

`splitInfusion()` is a model-only directive that rewrites infusion doses aimed at `cmt` into parallel infusion doses to the target compartments. Each target compartment receives the full original amount (or rate); use compartment properties like `f()` to scale the split (for example a Monolix-style double-absorption `F1`/`1-F1` dose apportionment). The source-compartment infusion is replaced, not retained.

Both data-driven infusions (`RATE` or `DUR` records) and modeled `rate()`/`dur()` infusions are split. The source and every start or stop record of the infusion is re-targeted, so the pairings the solver relies on are preserved.

Unlike `splitBolus()`, this rewrite applies at `etTrans()` translation time only; doses pushed while solving with `evid_()` are not split.

The source compartment may also appear in the target list, but the target compartments in `...` must be unique.

Usage

```
splitInfusion(cmt, ...)
```

Arguments

`cmt`	Source compartment name.
`...`	Target compartment names. Provide at least one target compartment.

Value

This function is only meaningful inside an rxode2 model; it errors when called directly from R.

`splitInfusionBolus-directive`

Split infusion doses into infusion and bolus paths (model directive)

Description

`splitInfusionBolus()` is a model-only directive that rewrites doses aimed at `cmt` into parallel doses to the target compartments, regardless of whether the dose record is a bolus or an infusion. Each target compartment receives the full original amount; use `f()` to scale the split.

A plain bolus record is split so the FIRST target compartment receives a modeled infusion start/stop pair (it must declare a modeled `dur()` or `rate()` property, otherwise its copies stay boluses) while every other target receives a bolus copy. Use `splitBolusInfusion()` to give the LAST target the infusion instead. This feeds one bolus dose record into both an infusion path and a bolus path (Monolix-style double absorption with mixed zero- and first-order routes). Steady-state bolus records are not promoted; they are copied as boluses with a warning when the designated infusion target declares `dur()`/`rate()`.

Infusion records (data `RATE`/`DUR` or modeled `rate()`/`dur()`) are split preserving their type, exactly like `splitInfusion()`.

This rewrite applies at `etTrans()` translation time only; doses pushed while solving with `evid_()` are not split. Only one splitting directive (`splitBolus()`, `splitInfusion()`, `splitInfusionBolus()` or `splitBolusInfusion()`) is allowed per model.

There is no R function named `splitInfusionBolus()` exported by rxode2; the directive is parsed from the model text directly, as shown in `?splitBolusInfusion-directive`.

Usage inside a model

```
model({
  splitInfusionBolus(depot, central, depot2)
  dur(central) <- tk0
  f(central) <- f1
  f(depot2) <- 1 - f1
  d/dt(depot2) <- -ka2 * depot2
  d/dt(central) <- ka2 * depot2 - cl / v * central
  ...
})
```

A bolus dose recorded against `depot` is split so `central` (first target) receives a modeled-duration infusion (zero-order input over `tk0`, scaled by `f1`) and `depot2` receives the bolus (first-order input scaled by `1 - f1`).

stat_amt *Dosing/Amt geom/stat*

Description

This is a dosing geom that shows the vertical lines where a dose occurs

Usage

```
stat_amt(
  mapping = NULL,
  data = NULL,
  position = "identity",
  show.legend = NA,
  inherit.aes = TRUE,
  ...
)

geom_amt(
  mapping = NULL,
  data = NULL,
  position = "identity",
  show.legend = NA,
  inherit.aes = TRUE,
  ...
)
```

Arguments

mapping	Set of aesthetic mappings created by aes(). If specified and inherit.aes = TRUE (the default), it is combined with the default mapping at the top level of the plot. You must supply mapping if there is no plot mapping.
data	The data to be displayed in this layer. There are three options: If NULL, the default, the data is inherited from the plot data as specified in the call to ggplot(). A data.frame, or other object, will override the plot data. All objects will be fortified to produce a data frame. See fortify() for which variables will be created. A function will be called with a single argument, the plot data. The return value must be a data.frame, and will be used as the layer data. A function can be created from a formula (e.g. ~ head(.x, 10)).
position	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The position argument accepts the following: • The result of calling a position function, such as position_jitter(). This method allows for passing extra arguments to the position.

- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`show.legend` logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.

`inherit.aes` If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn't inherit behaviour from the default plot specification, e.g. `annotation_borders()`.

`...` Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `....` Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `....` This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

Details

Requires the following aesthetics:

- x representing the x values, usually time
- amt representing the dosing values; They are missing or zero when no dose is given

Value

This returns a stat_amt in context of a ggplot2 plot

Examples

```
library(rxode2)
library(units)


## Model from RxODE tutorial
mod1 <- function() {
  ini({
    KA <- 2.94E-01
    CL <- 1.86E+01
    V2 <- 4.02E+01
    Q  <- 1.05E+01
    V3 <- 2.97E+02
    Kin <- 1
    Kout <- 1
    EC50 <- 200
  })
  model({
    C2 <- centr/V2
    C3 <- peri/V3
    d/dt(depot) <- -KA*depot
    d/dt(centr) <-  KA*depot - CL*C2 - Q*C2 + Q*C3
    d/dt(peri)  <-                     Q*C2 - Q*C3
    d/dt(eff)   <- Kin - Kout*(1-C2/(EC50+C2))*eff
  })
}

## These are making the more complex regimens of the rxode2 tutorial

## bid for 5 days
bid <- et(timeUnits="hr") |>
  et(amt=10000,ii=12,until=set_units(5, "days"))

## qd for 5 days
qd <- et(timeUnits="hr") |>
      et(amt=20000,ii=24,until=set_units(5, "days"))

## bid for 5 days followed by qd for 5 days

et <- seq(bid,qd) |>
      et(seq(0,11*24,length.out=100))

bidQd <- rxSolve(mod1, et, addDosing=TRUE)

# by default dotted and under-stated
plot(bidQd, C2) + geom_amt(aes(amt=amt))

# of course you can make it a bit more visible

plot(bidQd, C2) + geom_amt(aes(amt=amt), col="red", lty=1, linewidth=1.2)
```

stat_cens *Censoring geom/stat*

Description

This is a censoring geom that shows the left or right censoring specified in the nlmixr input data-set or fit

Usage

```
stat_cens(
  mapping = NULL,
  data = NULL,
  position = "identity",
  show.legend = NA,
  inherit.aes = TRUE,
  width = 0.01,
  ...
)

geom_cens(
  mapping = NULL,
  data = NULL,
  position = "identity",
  show.legend = NA,
  inherit.aes = TRUE,
  width = 0.01,
  ...
)
```

Arguments

mapping	Set of aesthetic mappings created by aes(). If specified and inherit.aes = TRUE (the default), it is combined with the default mapping at the top level of the plot. You must supply mapping if there is no plot mapping.
data	The data to be displayed in this layer. There are three options: If NULL, the default, the data is inherited from the plot data as specified in the call to ggplot(). A data.frame, or other object, will override the plot data. All objects will be fortified to produce a data frame. See fortify() for which variables will be created. A function will be called with a single argument, the plot data. The return value must be a data.frame, and will be used as the layer data. A function can be created from a formula (e.g. ~ head(.x, 10)).

position — A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The position argument accepts the following:

- The result of calling a position function, such as position_jitter(). This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the position_ prefix. For example, to use position_jitter(), give the position as "jitter".
- For more information and other ways to specify the position, see the layer position documentation.

show.legend — logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use TRUE. If NA, all levels are shown in legend, but unobserved levels are omitted.

inherit.aes — If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn't inherit behaviour from the default plot specification, e.g. annotation_borders().

width — represents the width (in \ censoring box

... — Other arguments passed on to layer()'s params argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the position argument, or aesthetics that are required can *not* be passed through .... Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, colour = "red" or linewidth = 3. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the params. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a stat_*() function, the ... argument can be used to pass on parameters to the geom part of the layer. An example of this is stat_density(geom = "area", outline.type = "both"). The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a geom_*() function, the ... argument can be used to pass on parameters to the stat part of the layer. An example of this is geom_area(stat = "density", adjust = 0.5). The stat's documentation lists which parameters it can accept.
- The key_glyph argument of layer() may also be passed on through .... This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

Details

Requires the following aesthetics:

- x represents the independent variable, often the time scale

- y represents the dependent variable

Plus exactly one of these aesthetic forms (the two forms cannot be mixed):

- lower and upper give the lower and upper bounds of the censoring interval directly. Both must be supplied. Use -Inf/Inf for an unbounded side; use NA on both for observed (non-censored) rows.
- cens (with optional limit) gives the censoring indicator (-1 right censored, 0 not censored, 1 left censored) and, when limit is also supplied, the associated limit value. Internally converted to lower/upper.

Will add boxes representing the areas of the fit that were censored.

Value

This returns a ggplot2 stat

---

summary.rxode2 *Print expanded information about the rxode2 object.*

Description

This prints the expanded information about the rxode2 object.

Usage

```
## S3 method for class 'rxode2'
summary(object, ...)
```

Arguments

object	rxode2 object
...	Ignored parameters

Value

object is returned

Author(s)

Matthew L.Fidler

`swapMatListWithCube` *Swaps the matrix list with a cube*

Description

Swaps the matrix list with a cube

Usage

```
swapMatListWithCube(matrixListOrCube)
```

Arguments

`matrixListOrCube`	Either a list of 2-dimensional matrices or a cube of matrices

Value

A list or a cube (opposite format as input)

Author(s)

Matthew L. Fidler

Examples

```
# Create matrix list
matLst <- cvPost(10, lotri::lotri(a+b~c(1, 0.25, 1)), 3)
print(matLst)

# Convert to cube
matCube <- swapMatListWithCube(matLst)
print(matCube)

# Convert back to list
matLst2 <- swapMatListWithCube(matCube)
print(matLst2)
```

`Swish` *Switch Activation Function*

Description

The switch activation function is defined as:

Usage

```
Swish(x)
```

Arguments

`x`	A numeric vector. All elements must be finite and non-missing.

Details

$$f(x) = x \cdot \text{sigmoid}(x)$$

Value

A numeric vector where the ReLU function has been applied to each element of `x`.

Author(s)

Matthew Fidler

See Also

Other Activation Functions: `ELU()`, `GELU()`, `PReLU()`, `ReLU()`, `SELU()`, `dELU()`, `dGELU()`, `dPReLU()`, `dReLU()`, `dSELU()`, `dSwish()`, `dlReLU()`, `dsoftplus()`, `lReLU()`, `softplus()`

Examples

```
Swish(c(-1, 0, 1, 2))


# Can also be used in rxode2:
x <- rxode2({
   r<- Swish(time)
})

e <- et(c(-1, 0, 1, 2))

rxSolve(x, e)
```

testIniDf *This function tests if this object is a iniDf as needed by the UI*

Description

This function tests if this object is a iniDf as needed by the UI

Usage

```
testIniDf(iniDf)

assertIniDf(iniDf, extra = "", .var.name = .vname(iniDf), null.ok = FALSE)
```

Arguments

iniDf	the object to test if it is a rxode2 ui iniDf data.frame
extra	information to append to the error message
.var.name	[character(1)] Name of the checked object to print in assertions. Defaults to the heuristic implemented in vname.
null.ok	[logical(1)] If set to TRUE, x may also be NULL. In this case only a type check of x is performed, all additional checks are disabled.

Value

boolean, indicating if the object is a valid initialization data frame

Functions

- assertIniDf(): Assert that the object is a valid rxode2 ui initialization data frame

Author(s)

Matthew L. Fidler

See Also

Other Assertions: assertCompartmentExists(), assertCompartmentName(), assertCompartmentNew(), assertRxUi(), assertVariableExists(), assertVariableNew(), rxPriorBuildSpec(), rxPriorLogDensity(), rxPriorOmegaToCholOmegaInvGrad(), rxUiPriors(), testRxUnbounded()

Examples

```
testIniDf(TRUE)
```

`testRxLinCmt` *Test if rxode2 uses linear solved systems*

Description

Test if rxode2 uses linear solved systems

Usage

```
testRxLinCmt(ui, extra = "", .var.name = .vname(ui))

assertRxLinCmt(ui, extra = "", .var.name = .vname(ui))
```

Arguments

`ui`	rxode2 model
`extra`	Extra text to append to the error message (like "for focei")
`.var.name`	[`character(1)`] Name of the checked object to print in assertions. Defaults to the heuristic implemented in `vname`.

Value

TRUE if the model uses linear solved systems, FALSE otherwise

Functions

- `assertRxLinCmt()`: Assert that the rxode2 uses linear solved systems

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)
 })
}

testRxLinCmt(one.cmt)
```

`testRxUnbounded` *Test if the rxode2 model has any parameters with user defined boundaries*

Description

Test if the rxode2 model has any parameters with user defined boundaries

Usage

```
testRxUnbounded(ui)

assertRxUnbounded(ui, extra = "", .var.name = .vname(ui))

warnRxBounded(ui, extra = "", .var.name = .vname(ui))
```

Arguments

`ui`	rxode2 ui
`extra`	extra information to append to the error message
`.var.name`	variable name

Value

boolean indicating if any parameters have user defined boundaries

Functions

- `assertRxUnbounded()`: Assert that the rxode2 model has any parameters with user defined boundaries
- `warnRxBounded()`: Warn that the rxode2 model has any parameters with user defined boundaries

Author(s)

Matthew L. Fidler

See Also

Other Assertions: assertCompartmentExists(), assertCompartmentName(), assertCompartmentNew(), assertRxUi(), assertVariableExists(), assertVariableNew(), rxPriorBuildSpec(), rxPriorLogDensity(), rxPriorOmegaToCholOmegaInvGrad(), rxUiPriors(), testIniDf()

Examples

```
one.cmt <- function() {
  ini({
    tka <- 0.45; label("Ka")
    tcl <- log(c(0, 2.7, 100)); label("Cl")
    tv <- 3.45; label("V")
    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)
  })
}

testRxUnbounded(one.cmt)

try(assertRxUnbounded(one.cmt))

warnRxBounded(one.cmt)
```

---

toTrialDuration *Convert event data to trial duration data A helper function to create a custom event table. The observation time will start from the first event time (baseline) and end at trial duration. The interval is the spacing between each observation.*

Description

Convert event data to trial duration data A helper function to create a custom event table. The observation time will start from the first event time (baseline) and end at trial duration. The interval is the spacing between each observation.

Usage

```
toTrialDuration(ev, trialEnd, interval, writeDir = NULL)
```

Arguments

ev	event data
trialEnd	extend trial duration. Must be same time unit as event data
interval	observation interval. Must be same time unit as event data
writeDir	if not NULL, write the output to a csv file

Author(s)

Omar Elashkar

Examples

```
# Create event table with unique time for each ID
ev = et(data.frame(id = rep(1:10, 3),  time = runif(min = 10, max = 20, n = 30)))

# select the duration and spacing interval (assuming time is in years)
toTrialDuration(ev, trialEnd = 1.5, interval = 0.2)
```

---

update.rxUi *Update for rxUi*

Description

Update for rxUi

Usage

```
## S3 method for class 'rxUi'
update(object, ..., envir = parent.frame())
```

Arguments

object	rxode2 UI object
...	Lines to update
envir	Environment for evaluating ini() style calls

Value

a new rxode2 updated UI object

uppergamma *uppergamma: upper incomplete gamma function*

Description

This is the tgamma from the boost library

Usage

```
uppergamma(a, z)
```

Arguments

a	The numeric 'a' parameter in the upper incomplete gamma
z	The numeric 'z' parameter in the upper incomplete gamma

Details

The uppergamma function is given by:

$uppergamma(a, z) = \int_z^\infty t^{a-1} \cdot e^{-t} dt$

Value

uppergamma results

Author(s)

Matthew L. Fidler

Examples

```
uppergamma(1, 3)

uppergamma(1:3, 3)

uppergamma(1, 1:3)
```

zeroRe *Set random effects and residual error to zero*

Description

Set random effects and residual error to zero

Usage

```
zeroRe(object, which = c("omega", "sigma"), fix = TRUE)
```

Arguments

object	The model to modify
which	The types of parameters to set to zero
fix	Should the parameters be fixed to the zero value?

Value

The object with some parameters set to zero

Author(s)

Bill Denney

See Also

Other Initial conditions: ini.rxUi()

Examples

```
one.compartment <- function() {
  ini({
    tka <- log(1.57); label("Ka")
    tcl <- log(2.72); label("Cl")
    tv <- log(31.5); label("V")
    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)
    d/dt(depot) = -ka * depot
    d/dt(center) = ka * depot - cl / v * center
    cp = center / v
    cp ~ add(add.sd)
```

```
  })
}
zeroRe(one.compartment)
```

Index

∗ **Activation Functions**
dELU, 52
dGELU, 54
dlReLU, 55
dPReLU, 55
dReLU, 57
dSELU, 58
dsoftplus, 59
dSwish, 60
ELU, 61
GELU, 86
lReLU, 120
PReLU, 147
ReLU, 151
SELU, 359
softplus, 360
Swish, 371
∗ **Assertions**
assertCompartmentExists, 32
assertCompartmentName, 33
assertCompartmentNew, 34
assertRxUi, 34
assertVariableExists, 38
assertVariableNew, 39
rxPriorBuildSpec, 261
rxPriorLogDensity, 261
rxPriorOmegaToCholOmegaInvGrad, 264
rxUiPriors, 352
testIniDf, 372
testRxUnbounded, 374
∗ **Initial conditions**
ini.rxUi, 93
zeroRe, 378
∗ **Internal**
.matchesLangTemplate, 11
odeMethodToInt, 134
plot.rxSolve, 146
∗ **Model names**
rxModelName, 209
rxModelNameLhs, 210
∗ **Nonlinear regression**
eventTable, 77
rxode2, 215
∗ **ODE models**
rxode2, 215
∗ **Ordinary differential equations**
rxode2, 215
∗ **PK/PD**
genShinyApp.template, 87
∗ **Pharmacodynamics (PD)**
eventTable, 77
rxode2, 215
∗ **Pharmacokinetics (PK)**
eventTable, 77
rxode2, 215
∗ **Query model information**
rxDfdy, 171
rxLhs, 205
rxParams, 252
rxState, 324
∗ **User functions**
linMod, 98
rxUdfUiControl, 335
rxUdfUiData, 336
rxUdfUiEst, 336
rxUdfUiIniLhs, 339
rxUdfUiMv, 341
rxUdfUiNum, 342
rxUdfUiParsing, 343
∗ **datasets**
rxReservedKeywords, 270
rxResidualError, 271
rxSyntaxFunctions, 329
∗ **data**
eventTable, 77
∗ **models**
eventTable, 77

rxode2, 215
∗ **nonlinear**
genShinyApp.template, 87
rxode2, 215
∗ **ordinary differential equations**
eventTable, 77
∗ **pharmacometrics**
genShinyApp.template, 87
∗ **rxode2 plotting**
plot.rxSolve, 146
rxTheme, 331
∗ **simulation**
genShinyApp.template, 87
.C(), *166*
.Call(), *166*
.cbindOme, 9
.collectWarnings, 9
.copyUi, 10
.handleSingleErrTypeNormOrTFoceiBase, 10
.matchesLangTemplate, 11
.modelHandleModelLines, 12
.print.via.format, *149*
.quoteCallInfoLines, 13
.rxAdjointGrad(), *320*
.rxAdjointSolveEvalC(), *321*
.rxEventCall *(*rxEventListen*)*, 176
.rxEventEnter *(*rxEventListen*)*, 176
.rxEventExit *(*rxEventListen*)*, 176
.rxFromSE *(*rxToSE*)*, 332
.rxLinCmtGen, 14
.rxRename *(*rxRename*)*, 269
.rxSensStrippable, 15
.rxToSE *(*rxToSE*)*, 332
.rxWithOptions, 15
.rxWithWd, 16
.rxode2ptrs, 14
.toClassicEvid, 17
.vecDf, 18
[.rxEvid *(*rxEvid*)*, 178
[.rxRateDur *(*rxRateDur*)*, 266
[[.rxEvid *(*rxEvid*)*, 178
[[.rxRateDur *(*rxRateDur*)*, 266

add.dosing, 18, *19*, *22*, *65*, *69*, *72*, *75*
add.dosing(), *78*, *242*
add.sampling, *19*, 21, *22*, *65*, *69*, *72*, *75*
add.sampling(), *78*, *242*
aes(), *364*, *367*
annotation_borders(), *365*, *368*
as.arrow, 24
as.character.rxEvid *(*rxEvid*)*, 178
as.character.rxRateDur *(*rxRateDur*)*, 266
as.data.table.rxEt, 25
as.et, 25
as.ini, 26
as.model, 28
as.rxEvid *(*rxEvid*)*, 178
as.rxRateDur *(*rxRateDur*)*, 266
as.rxUi, 30
as_tibble.rxEt, 31
assertCompartmentExists, 32
assertCompartmentExists(), *33, 34*, *37*, *39, 40*, *261*, *263, 264*, *352*, *372*, *375*
assertCompartmentName, 33
assertCompartmentName(), *32*, *34*, *37*, *39, 40*, *261*, *263, 264*, *352*, *372*, *375*
assertCompartmentNew, 34
assertCompartmentNew(), *32, 33*, *37*, *39, 40*, *261*, *263, 264*, *352*, *372*, *375*
assertExists *(*assertCompartmentName*)*, 33
assertIniDf *(*testIniDf*)*, 372
assertParameterValue *(*assertCompartmentName*)*, 33
assertRxLinCmt *(*testRxLinCmt*)*, 373
assertRxUi, 34
assertRxUi(), *32–34*, *39, 40*, *261*, *263, 264*, *352*, *372*, *375*
assertRxUiAddProp *(*assertRxUi*)*, 34
assertRxUiErrType *(*assertRxUi*)*, 34
assertRxUiEstimatedResiduals *(*assertRxUi*)*, 34
assertRxUiIovNoCor *(*assertRxUi*)*, 34
assertRxUiMixedOnly *(*assertRxUi*)*, 34
assertRxUiMuRefOnly *(*assertRxUi*)*, 34
assertRxUiNoAutoregressive *(*assertRxUi*)*, 34
assertRxUiNoEtaDist *(*rxUiEtaDists*)*, 347
assertRxUiNoFixedOmega *(*assertRxUi*)*, 34
assertRxUiNoFixedResiduals *(*assertRxUi*)*, 34
assertRxUiNoMix *(*assertRxUi*)*, 34
assertRxUiNoOmegaDf *(*assertRxUi*)*, 34
assertRxUiNoOmegaNormalPriors *(*assertRxUi*)*, 34
assertRxUiNoPriors *(*assertRxUi*)*, 34
assertRxUiNormal *(*assertRxUi*)*, 34

assertRxUiNormalPriors *(assertRxUi)*, 34
assertRxUiPopulationOnly *(assertRxUi)*, 34
assertRxUiPrediction *(assertRxUi)*, 34
assertRxUiRandomOnIdOnly *(assertRxUi)*, 34
assertRxUiSingleEndpoint *(assertRxUi)*, 34
assertRxUiTransform *(assertRxUi)*, 34
assertRxUiTransformNormal *(assertRxUi)*, 34
assertRxUnbounded *(testRxUnbounded)*, 374
assertVariableExists, 38
assertVariableExists(), *32–34*, *37*, *40*, *261*, *263*, *264*, *352*, *372*, *375*
assertVariableName *(assertCompartmentName)*, 33
assertVariableNew, 39
assertVariableNew(), *32–34*, *37*, *39*, *261*, *263*, *264*, *352*, *372*, *375*

binomProbs, 40
binomProbs(), *46*
bolus, 42
bolus(), *81*
boxCox, 43
boxCoxInv *(boxCox)*, 43

c.rxEvid *(rxEvid)*, 178
c.rxRateDur *(rxEvid)*, 178
cat, *149*
class, *148*
coef.function *(coef.rxUi)*, 44
coef.rxUi, 44
confint.rxSolve, 45
cvPost, 47

d2aELU *(dELU)*, 52
d2ELU *(dELU)*, 52
d2ELUa *(dELU)*, 52
d2GELU *(dGELU)*, 54
d2softplus *(dsoftplus)*, 59
d3GELU *(dGELU)*, 54
d3softplus *(dsoftplus)*, 59
d4GELU *(dGELU)*, 54
d4softplus *(dsoftplus)*, 59
delay, 50
dELU, 52
dELU(), *54–61*, *86*, *120*, *148*, *151*, *359*, *360*, *371*
dELUa *(dELU)*, 52
dfWishart, 53
dGELU, 54
dGELU(), *52*, *55–61*, *86*, *120*, *148*, *151*, *359*, *360*, *371*
dlReLU, 55
dlReLU(), *52*, *54*, *56–61*, *86*, *120*, *148*, *151*, *359*, *360*, *371*
dmexpit *(mexpit)*, 123
dPReLU, 55
dPReLU(), *52*, *54*, *55*, *57–61*, *86*, *120*, *148*, *151*, *359*, *360*, *371*
dPReLUa *(dPReLU)*, 55
dPReLUa1 *(dPReLU)*, 55
dReLU, 57
dReLU(), *52*, *54–56*, *58–61*, *86*, *120*, *148*, *151*, *359*, *360*, *371*
dSELU, 58
dSELU(), *52*, *54–57*, *59–61*, *86*, *120*, *148*, *151*, *359*, *360*, *371*
dsoftplus, 59
dsoftplus(), *52*, *54–58*, *60*, *61*, *86*, *120*, *148*, *151*, *359*, *360*, *371*
dSwish, 60
dSwish(), *52*, *54–59*, *61*, *86*, *120*, *148*, *151*, *359*, *360*, *371*

ELU, 61
ELU(), *52*, *54–60*, *86*, *120*, *148*, *151*, *359*, *360*, *371*
environment, *63*
erf, 62
et, *19*, *22*, 62, *65*, *69*, *72*, *75*
et(), *19*, *78*, *242*
etExpand, 67
etRbind, *19*, *22*, *65*, 68, *69*, *72*, *75*
etRep, *19*, *22*, *65*, *69*, 71, *72*, *75*
etSeq, 74
etTrans(), *361–363*
eventTable, *19*, *22*, *65*, *69*, *72*, *75*, 77
eventTable(), *88*, *242*, *295*, *319*
evid_, 79
evid_(), *361–363*
expit *(logit)*, 118

format, *149*
format.rxEvid *(rxEvid)*, 178

format.rxRateDur (rxEvid), 178
fortify(), *364*, *367*

gammap, 82
gammapDer, 83
gammapDera (phiU), 145
gammapInv, 83
gammapInva (gammapInv), 83
gammaq, 84
gammaqInv, 85
gammaqInva (gammaqInv), 85
GELU, 86
GELU(), *52*, *54–61*, *120*, *148*, *151*, *359*, *360*, *371*
genShinyApp.template, 87
geom_amt (stat_amt), 364
geom_cens (stat_cens), 367
getRxThreads, 88
ggplot(), *364*, *367*

ibeta (phiU), 145
ibetaDer (phiU), 145
ibetaDera (phiU), 145
ibetaDerb (phiU), 145
ibetaInv (phiU), 145
indLin, 89
indLin(), *275*
infuse, 90
infuse(), *81*
infuseDur, 91
infuseDur(), *81*
ini (ini.rxUi), 93
ini.rxUi, 93
ini.rxUi(), *378*
ini<-, 95
invisible, *148*
is.rxEt, 96
is.rxStackData, 96

key glyphs, *365*, *368*

layer position, *365*, *368*
layer(), *365*, *368*
linCmtMicro, 97
linMod, 98
linMod(), *335–337*, *339*, *342*, *343*
linMod0 (linMod), 98
linModA (linMod), 98
linModA0 (linMod), 98
linModB (linMod), 98
linModB0 (linMod), 98
linModD (linMod), 98
linModD0 (linMod), 98
linModM (linMod), 98
linModM0 (linMod), 98
linToOde, 100, *142*
llikBeta, 101
llikBinom, 102
llikCauchy, 104
llikChisq, 105
llikExp, 106
llikF, 107
llikGamma, 108
llikGeom, 109
llikNbinom, 110
llikNbinomMu, 111
llikNorm, 112
llikPois, 114
llikT, 115
llikUnif, 116
llikWeibull, 117
logit, 118
logitNormInfo (logit), 118
lowergamma, 119
lReLU, 120
lReLU(), *52*, *54–61*, *86*, *148*, *151*, *359*, *360*, *371*

meanProbs, 121
meanProbs(), *46*
methods, *149*
mexpit, 123
mix, 124
mlogit, 125
model (model.function), 127
model.function, 127
model<-, 129
modelExtract, 129
multiply, 132

noquote, *149*

obs, 133
odeMethodToInt, 134, *207*
odeMethodToInt(), *199–203*, *316*
odeToLin, 142
odeToLin(), *317*
options, *149*

ordered, *149*

phantom, 143
phi, 144
phiU, 145
plot(), *46*
plot.rxSolve, 146
plot.rxSolve(), *332*
plot.rxSolveConfint1 *(*plot.rxSolve*)*, 146
plot.rxSolveConfint2 *(*plot.rxSolve*)*, 146
predict.function *(*rxSolve*)*, 288
predict.rxEt *(*rxSolve*)*, 288
predict.rxode2 *(*rxSolve*)*, 288
predict.rxParams *(*rxSolve*)*, 288
predict.rxSolve *(*rxSolve*)*, 288
predict.rxUi *(*rxSolve*)*, 288
PReLU, 147
PReLU(), *52*, *54–61*, *86*, *120*, *151*, *359, 360*, *371*
print.default, *149*
print.rxEvid *(*rxEvid*)*, 178
print.rxModelVars, 148
probit, 150
probitInv *(*probit*)*, 150
probitNormInfo *(*logit*)*, 118

rbind.rxEt *(*etRbind*)*, 68
ReLU, 151
ReLU(), *52*, *54–61*, *86*, *120*, *148*, *359, 360*, *371*
rename.function *(*rxRename*)*, 269
rename.rxUi *(*rxRename*)*, 269
rep.rxEt *(*etRep*)*, 71
replace, 152
reset, 152
reset(), *81*
rinvchisq, 153
rLKJ1(), *48*
rxAllowUnload, 154
rxAppendModel, 154
rxAssignControlValue, 156
rxAssignPtr, 156
rxbeta, 157
rxbinom, 158
rxcauchy, 159
rxCbindStudyIndividual, 161
rxchisq, 162
rxClean, 163
rxClearActiveParLoader *(*rxSetActiveParLoader*)*, 276
rxCompile, 164
rxCompile(), *335*
rxControl, *207*
rxControl *(*rxSolve*)*, 288
rxControl(), *317*
rxControlUpdateSens, 166
rxCores *(*getRxThreads*)*, 88
rxCreateCache, 167
rxD, 168
rxDelete, 169
rxDerived, 169
rxDfdy, 171
rxDfdy(), *205*, *254*, *324*
rxEtaDistExpand, 172
rxEtaDistExpand(), *341*, *347*
rxEtaDistMuRef, 173
rxEtDispatchSolve, 175
rxEventDepth *(*rxEventListen*)*, 176
rxEventEmit *(*rxEventListen*)*, 176
rxEventListen, 176
rxEventListeners *(*rxEventListen*)*, 176
rxEventScope *(*rxEventListen*)*, 176
rxEventSeq *(*rxEventListen*)*, 176
rxEventTableFile, 177
rxEventUnlisten *(*rxEventListen*)*, 176
rxEvid, 178
rxexp, 179
rxf, 180
rxFixPop, 182
rxFixRes, 183
rxForcedPars, 185
rxForcedPars(), *255*, *267*, *276*
rxForcedPars<- *(*rxForcedPars*)*, 185
rxFromSE *(*rxToSE*)*, 332
rxFun, 186
rxgamma, 188
rxgeom, 190
rxGetControl, 191
rxGetDefaultSerialize, 192
rxGetLin, 192
rxGetrxode2, 193
rxGetSeed, 193
rxHasAr, 194
rxHtml, 195
rxIndLinState, 195
rxIndLinStrategy, 196
rxIndLinStrategy(), *195*
rxInits(), *171*, *205*, *254*, *324*

rxInjectedPars, 197
rxIntToBase, 197
rxIntToLetter, 198
rxInv, 198
rxIsAutoSwitch, 199
rxIsCurrent, 199
rxIsDense, 200
rxIsImplicit, 201
rxIsImplicit(), *202, 203*
rxIsNonStiff, 202
rxIsNonStiff(), *199, 200*, *203*
rxIsStiff, 203
rxIsStiff(), *199, 200*, *202*
rxLastCompile, 204
rxLhs, 205
rxLhs(), *171*, *254*, *324*
rxLock, 205
rxMemoryEstimate, 206
rxMemSummary, *206*, 208
rxModelName, 209
rxModelName(), *211*
rxModelNameFromExpr(), *210, 211*
rxModelNameLhs, 210
rxModelNameLhs(), *210*
rxModelVars(), *171*, *205*, *254*, *324*, *334*
rxnbinom, 211
rxnbinomMu *(*rxnbinom*)*, 211
rxNorm, 213
rxnorm *(*rxnormV*)*, 214
rxnormV, 214
rxNumLoaded, 215
RxODE *(*rxode2*)*, 215
rxode *(*rxode2*)*, 215
rxode2, *19*, *22*, *65*, *69*, *72*, *75*, *205*, 215
rxode2(), *87, 88*, *166*, *320–322*, *324*, *334, 335*
rxode2<-, 243
rxode2parse, 245
rxode2parseAssignPackagesToLoad *(*rxode2parseGetPackagesToLoad*)*, 247
rxode2parseAssignTranslation, 246
rxode2parseD, 247
rxode2parseGetPackagesToLoad, 247
rxode2parseGetPointerAssignment, 248
rxode2parseGetTranslation, 249
RxODE<- *(*rxode2<-*)*, 243
rxode<- *(*rxode2<-*)*, 243
rxOdeToIndLin *(*indLin*)*, 89
rxOmegaVarCovDeriv, 249
rxOptExpr, 250
rxord, 251
rxParam *(*rxParams*)*, 252
rxParams, 252
rxParams(), *171*, *205*, *324*
rxParLoader, 254
rxParLoader(), *276*
rxParLoader<- *(*rxParLoader*)*, 254
rxParseSuppressMsg, 255
rxPkg, 256
rxpois, 257
rxPp, 258
rxPreferredDistributionName, 260
rxPriorBuildSpec, 261
rxPriorBuildSpec(), *32–34*, *37*, *39, 40*, *263*, *264*, *352*, *372*, *375*
rxPriorLogDensity, 261
rxPriorLogDensity(), *32–34*, *37*, *39, 40*, *261*, *264*, *352*, *372*, *375*
rxPriorOmegaToCholOmegaInvGrad, 264
rxPriorOmegaToCholOmegaInvGrad(), *32–34*, *37*, *39, 40*, *261*, *263*, *352*, *372*, *375*
rxProgress, 265
rxProgressAbort *(*rxProgress*)*, 265
rxProgressStop *(*rxProgress*)*, 265
rxRateDur, 266
rxRegisterUiAssembled, 267
rxRegisterUiPrep, 268
rxRemoveControl, 268
rxRemoveUiAssembled *(*rxRegisterUiAssembled*)*, 267
rxRemoveUiPrep *(*rxRegisterUiPrep*)*, 268
rxRename, 269
rxReservedKeywords, 270
rxResidualError, 271
rxRmFun *(*rxFun*)*, 186
rxRmvn, 271
rxS, 274
rxSensMatExp, 275
rxSetActiveParLoader, 276
rxSetControl, 277
rxSetCovariateNamesForPiping, 277
rxSetIni0, 278
rxSetPipingAuto, 279
rxSetProd, 280
rxSetProgressBar, 280

rxSetSeed, 281
rxSetSum, 282
rxShiny, 283
rxSimThetaOmega, 284
rxSolve, 288
rxSolve(), *43*, *51*, *81*, *87*, *91, 92*, *133*, *144*, *152, 153*, *320–322*
rxSolveAdjoint, 320
rxSolveAdjointRk4, 321
rxSolveChunked, 322
rxStack, 323
rxStack(), *46*
rxState, 324
rxState(), *171*, *205*, *254*
rxStateOde, 325
rxSumProdModel, 326
rxSupportedFuns, 326
rxSuppressMsg, 327
rxSymInvChol, 328
rxSymInvCholCreate(), *328*
rxSyncOptions, 329
rxSyntaxFunctions, 329
rxt, 330
rxTempDir, 331
rxTest *(*rxValidate*)*, 356
rxTheme, 331
rxTheme(), *147*
rxTick *(*rxProgress*)*, 265
rxToIndLin *(*indLin*)*, 89
rxToSE, 332
rxTrans, 333
rxTrans(), *166*
rxUdfUi(), *267*
rxUdfUiControl, 335
rxUdfUiControl(), *100*, *336, 337*, *339*, *342, 343*
rxUdfUiData, 336
rxUdfUiData(), *100*, *335*, *337*, *339*, *342, 343*
rxUdfUiEst, 336
rxUdfUiEst(), *100*, *335, 336*, *339*, *342, 343*
rxUdfUiExpr, 337
rxUdfUiFlag, 338
rxUdfUiIniDf, 338
rxUdfUiIniLhs, 339
rxUdfUiIniLhs(), *100*, *335–337*, *342, 343*
rxUdfUiIsValue, 340
rxUdfUiLhs(), *341*
rxUdfUiLhs.dist, 340
rxUdfUiMv, 341
rxUdfUiMv(), *100*, *335–337*, *339*, *342, 343*
rxUdfUiNum, 342
rxUdfUiNum(), *100*, *335–337*, *339*, *342, 343*
rxUdfUiParsing, 343
rxUdfUiParsing(), *100*, *335–337*, *339*, *342*
rxUiCompress *(*rxUiDecompress*)*, 343
rxUiDecompress, 343
rxUiDeparse, 344
rxUiDevelop, 345
rxUiEtaDistAnchors, 346
rxUiEtaDists, 347
rxUiEtaDists(), *346*
rxUiGet *(*rxUiGet.cmtLines*)*, 348
rxUiGet.cmtLines, 348
rxUiPriors, 352
rxUiPriors(), *32–34*, *37*, *39, 40*, *261*, *263, 264*, *372*, *375*
rxunif, 353
rxUnloadAll, 355
rxUnlock *(*rxLock*)*, 205
rxUse, 355
rxValidate, 356
rxweibull, 356
rxWithPreserveSeed *(*rxWithSeed*)*, 358
rxWithSeed, 358

sample, *358*
save(), *355*
SELU, 359
SELU(), *52*, *54–61*, *86*, *120*, *148*, *151*, *360*, *371*
seq.rxEt *(*etSeq*)*, 74
setRxThreads *(*getRxThreads*)*, 88
setRxThreads(), *304*
simulate.rxode2 *(*rxSolve*)*, 288
simulate.rxParams *(*rxSolve*)*, 288
simulate.rxSolve *(*rxSolve*)*, 288
softplus, 360
softplus(), *52*, *54–61*, *86*, *120*, *148*, *151*, *359*, *371*
solve.function *(*rxSolve*)*, 288
solve.rxEt *(*rxSolve*)*, 288
solve.rxode2 *(*rxSolve*)*, 288
solve.rxParams *(*rxSolve*)*, 288
solve.rxSolve *(*rxSolve*)*, 288
solve.rxUi *(*rxSolve*)*, 288
splitBolus, 361
splitBolusInfusion *(*splitBolusInfusion-directive*)*,

361
splitBolusInfusion(), *363*
splitBolusInfusion-directive, 361
splitInfusion, 362
splitInfusion(), *361*, *363*
splitInfusionBolus *(*splitInfusionBolus-directive*)*, 363
splitInfusionBolus(), *361*
splitInfusionBolus-directive, 363
stat_amt, 364
stat_cens, 367
studentTCdf *(*phiU*)*, 145
studentTCdfDnu *(*phiU*)*, 145
studentTDen *(*phiU*)*, 145
studentTInv *(*phiU*)*, 145
summary.rxode2, 369
swapMatListWithCube, 370
Swish, 371
Swish(), *52*, *54–61*, *86*, *120*, *148*, *151*, *359*, *360*
sys.call, *63*

table, *149*
testCompartmentExists *(*assertCompartmentExists*)*, 32
testExists *(*assertCompartmentName*)*, 33
testIniDf, 372
testIniDf(), *32–34*, *37*, *39*, *40*, *261*, *263*, *264*, *352*, *375*
testRxLinCmt, 373
testRxUiEtaDist *(*rxUiEtaDists*)*, 347
testRxUiNormalPriors *(*assertRxUi*)*, 34
testRxUiOmegaDf *(*assertRxUi*)*, 34
testRxUiOmegaNormalPriors *(*assertRxUi*)*, 34
testRxUiPriors *(*assertRxUi*)*, 34
testRxUnbounded, 374
testRxUnbounded(), *32–34*, *37*, *39*, *40*, *261*, *263*, *264*, *352*, *372*
testVariableExists *(*assertVariableExists*)*, 38
toTrialDuration, 375

units<-.rxEvid *(*rxEvid*)*, 178
update.rxSolve *(*rxSolve*)*, 288
update.rxUi, 376
uppergamma, 377
use_description(), *257*

vname, *36*, *372*, *373*

warnRxBounded *(*testRxUnbounded*)*, 374
write, *149*
write.template.server *(*genShinyApp.template*)*, 87
write.template.ui *(*genShinyApp.template*)*, 87
write.template.ui(), *87*

yeoJohnson *(*boxCox*)*, 43
yeoJohnsonInv *(*boxCox*)*, 43

zeroRe, 378
zeroRe(), *95*