Package 'OpenMx'

September 1, 2021

Encoding UTF-8

Date 2021-09-01

Title Extended Structural Equation Modelling

URL http://openmx.ssri.psu.edu, https://github.com/OpenMx/OpenMx

BugReports http://openmx.ssri.psu.edu/forums

Description Create structural equation models that can be manipulated programmatically.
Models may be specified with matrices or paths (LISREL or RAM)
Example models include confirmatory factor, multiple group, mixture
distribution, categorical threshold, modern test theory, differential
Fit functions include full information maximum likelihood, maximum likeli-
hood, and weighted least squares.
equations, state space, and many others.
Support and advanced package binaries available at <http://openmx.ssri.psu.edu>.
The software is described in Neale, Hunter, Pritikin, Zahery, Brick,
Kirkpatrick, Estabrook, Bates, Maes, & Boker (2016) <doi:10.1007/s11336-014-9435-8>.

SystemRequirements GNU make

ByteCompile yes

Language en-US

License Apache License (== 2.0)

LinkingTo
Rcpp, RcppEigen, RcppParallel, StanHeaders (>= 2.10.0.2), BH (>= 1.69.0-1), rpf (>= 0.45)

Imports digest,
MASS,
Matrix,
methods,
Rcpp, RcppParallel,
parallel, lifecycle

Depends R (>= 3.5.0)

Suggests mvtnorm,
numDeriv,
roxygen2 (>= 6.1),

rpf (>= 0.45),
snowfall,
lme4,
covr,
testthat,
umx,
ifaTools

LazyLoad yes

LazyData yes

Collate '0ClassUnion.R'
'cache.R'
'MxBaseNamed.R'
'MxData.R'
'MxDataWLS.R'
'DefinitionVars.R'
'MxReservedNames.R'
'MxNamespace.R'
'MxSearchReplace.R'
'MxFlatSearchReplace.R'
'MxUntitled.R'
'MxAlgebraFunctions.R'
'MxExponential.R'
'MxMatrix.R'
'DiagMatrix.R'
'FullMatrix.R'
'IdenMatrix.R'
'LowerMatrix.R'
'SdiagMatrix.R'
'StandMatrix.R'
'SymmMatrix.R'
'UnitMatrix.R'
'ZeroMatrix.R'
'MxMatrixFunctions.R'
'MxAlgebra.R'
'MxCycleDetection.R'
'MxDependencies.R'
'MxAlgebraConvert.R'
'MxSquareBracket.R'
'MxEval.R'
'MxRename.R'
'MxPath.R'
'MxObjectiveMetaData.R'
'MxExpectation.R'
'MxExpectationNormal.R'
'MxExpectationRAM.R'
'MxExpectationLISREL.R'
'MxFitFunction.R'
'MxFitFunctionAlgebra.R'

'MxFitFunctionML.R'
'MxFitFunctionMultigroup.R'
'MxFitFunctionRow.R'
'MxFitFunctionWLS.R'
'MxRAMObjective.R'
'MxLISRELObjective.R'
'MxFIMLObjective.R'
'MxMLObjective.R'
'MxRowObjective.R'
'MxAlgebraObjective.R'
'MxBounds.R'
'MxConstraint.R'
'MxInterval.R'
'MxTypes.R'
'MxCompute.R'
'MxModel.R'
'MxRAMModel.R'
'MxLISRELModel.R'
'MxModelDisplay.R'
'MxFlatModel.R'
'MxMultiModel.R'
'MxModelFunctions.R'
'MxModelParameters.R'
'MxUnitTesting.R'
'MxApply.R'
'MxRun.R'
'MxRunHelperFunctions.R'
'MxSummary.R'
'MxCompare.R'
'MxSwift.R'
'MxOptions.R'
'MxThreshold.R'
'OriginalMx.R'
'MxGraph.R'
'MxGraphviz.R'
'MxDeparse.R'
'MxCommunication.R'
'MxRestore.R'
'MxVersion.R'
'MxPPML.R'
'MxRAMtoML.R'
'MxErrorHandling.R'
'MxDetectCores.R'
'MxSaturatedModel.R'
'omxBrownie.R'
'omxConstrainThresholds.R'
'omxGetNPSOL.R'
'MxFitFunctionR.R'

'MxRObjective.R'
'MxExpectationHiddenMarkov.R'
'MxExpectationMixture.R'
'MxExpectationStateSpace.R'
'MxExpectationBA81.R'
'MxFitFunctionGREML.R'
'MxExpectationGREML.R'
'MxMI.R'
'MxFactorScores.R'
'MxRobustSE.R'
'MxAvailableOptimizers.R'
'MxTryHard.R'
'MxSE.R'
'MxAutoStart.R'
'MxRetro.R'
'MxMMI.R'
'omxReadGRMBin.R'
'MxPredict.R'
'zzz.R'

RdMacros lifecycle

Biarch true

Version 2.19.8

RoxygenNote 7.1.1

R topics documented:

`as.statusCode` *Convert a numeric or character vector into an optimizer status code factor*

Description

- 0,'OK': Optimization succeeded
- 1,'OK/green': Optimization succeeded, but the sequence of iterates has not yet converged (Mx status GREEN). This condition is only detected by NPSOL.
- 2,'infeasible linear constraint': The linear constraints and bounds could not be satisfied. The problem has no feasible solution.
- 3,'infeasible non-linear constraint': The nonlinear constraints and bounds could not be satisfied. The problem may have no feasible solution.
- 4,'iteration limit': Optimization was stopped prematurely because the iteration limit was reached (Mx status BLUE). You might want to rerun: `m1 = mxRun(m1)` or increase the iteration limit (see `mxOption`).
- 5,'not convex': The Hessian at the solution does not appear to be convex (Mx status RED). There may be more than one solution to the model. See `mxCheckIdentification`.
- 6,'nonzero gradient': The model does not satisfy the first-order optimality conditions to the required accuracy, and no improved point for the merit function could be found during the final linesearch (Mx status RED). To search nearby, see `mxTryHard`.
- 7,'bad deriv': You have provided analytic derivatives. However, your provided derivatives differ too much from numerically approximated derivatives. Double check your math.
- 9,'internal error': An input parameter was invalid. The most likely cause is a bug in the code. Please report occurrences to the OpenMx developers.
- 10,'infeasible start': Starting values were infeasible. Modify the start values for one or more parameters. For instance, set means to their measured value, or set variances and covariances to plausible values. See `mxAutoStart` and `mxTryHard`.

Usage

```
as.statusCode(code)
```

Arguments

`code`	a character or numeric vector of optimizer status code

See Also

`mxBootstrap` `summary.MxModel`

`BaseCompute-class` *BaseCompute*

Description

This is an internal class and should not be used directly.

See Also

mxComputeEM, mxComputeGradientDescent, mxComputeHessianQuality, mxComputeIterate, mxComputeNewtonRaphson, mxComputeNumericDeriv

---

`Bollen` *Bollen Data on Industrialization and Political Democracy*

Description

Data set used in some of OpenMx's examples, for instance WLS. The data were reported in Bollen (1989, p. 428, Table 9.4) This set includes data from 75 developing countries each assessed on four measures of democracy measured twice (1960 and 1965), and three measures of industrialization measured once (1960).

Usage

```
data("Bollen")
```

Format

A data frame with 75 observations on the following 11 numeric variables.

`y1`	Freedom of the press, 1960
`y2`	Freedom of political opposition, 1960
`y3`	Fairness of elections, 1960
`y4`	Effectiveness of elected legislature, 1960
`y5`	Freedom of the press, 1965
`y6`	Freedom of political opposition, 1965
`y7`	Fairness of elections, 1965

`y8`	Effectiveness of elected legislature, 1965
`x1`	GNP per capita, 1960
`x2`	Energy consumption per capita, 1960
`x3`	Percentage of labor force in industry, 1960

Details

Variables y1-y4 and y5-y8 are typically used as indicators of the latent trait of "political democracy" in 1960 and 1965 respectively. x1-x3 are used as indicators of industrialization (1960).

Source

The `sem` package (in turn, via personal communication Bollen to Fox)

References

Bollen, K. A. (1979). Political democracy and the timing of development. *American Sociological Review*, **44**, 572-587.

Bollen, K. A. (1980). Issues in the comparative measurement of political democracy. *American Sociological Review*, **45**, 370-390.

Bollen, K. A. (1989). *Structural equation models*. New York: Wiley-Interscience.

Examples

```
data(Bollen)
str(Bollen)
plot(y1 ~ y2, data = Bollen)
```

`cvectorize` *Vectorize By Column*

Description

This function returns the vectorization of an input matrix in a column by column traversal of the matrix. The output is returned as a column vector.

Usage

```
cvectorize(x)
```

Arguments

`x`	an input matrix.

See Also

`rvectorize`, `vech`, `vechs`

Examples

```
cvectorize(matrix(1:9, 3, 3))
cvectorize(matrix(1:12, 3, 4))
```

`demoOneFactor` *Demonstration data for a one factor model*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("demoOneFactor")
```

Format

A data frame with 500 observations on the following 5 numeric variables.

`x1`

`x2`

`x3`

`x4`

`x5`

Details

Variables x1-x5 are typically used as indicators of the latent trait.

Source

Simulated.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(demoOneFactor)
cov(demoOneFactor)
cor(demoOneFactor)
```

demoTwoFactor *Demonstration data for a two factor model*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("demoTwoFactor")
```

Format

A data frame with 500 observations on the following 10 numeric variables.

`x1`

`x2`

`x3`

`x4`

`x5`

`y1`

`y2`

`y3`

`y4`

`y5`

Details

Variables x1-x5 are typically used as indicators of one latent trait. Variables y1-y5 are typically used as indicators of another latent trait.

Source

Simulated.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(demoTwoFactor)
cov(demoTwoFactor)
cor(demoTwoFactor)
```

diag2vec *Extract Diagonal of a Matrix*

Description

Given an input matrix, `diag2vec` returns a column vector of the elements along the diagonal.

Usage

```
diag2vec(x)
```

Arguments

`x`	an input matrix.

Details

Similar to the function `diag`, except that the input argument is always treated as a matrix (i.e., it doesn't have diag()'s functions of returning an Identity matrix from an nrow specification, nor to return a matrix wrapped around a diagonal if provided with a vector). To get vector2matrix functionality, call `vec2diag`.

See Also

`vec2diag`

Examples

```
diag2vec(matrix(1:9, nrow=3))
#      [,1]
# [1,]    1
# [2,]    5
# [3,]    9

diag2vec(matrix(1:12, nrow=3, ncol=4))
#      [,1]
# [1,]    1
# [2,]    5
# [3,]    9
```

`DiscreteBase-class` *An S4 base class for discrete marginal distributions*

Description

An S4 base class for discrete marginal distributions

See Also

mxMarginalPoisson, mxMarginalNegativeBinomial

`dzfData` *Example twin extended kinship data: DZ female data*

Description

Data for extended twin example ETC88.R

Usage

```
data("dzfData")
```

Format

A data frame with 2007 observations on the following 37 variables.

`famid` a numeric vector

`e1` a numeric vector

`e2` a numeric vector

`e3` a numeric vector

`e4` a numeric vector

`e5` a numeric vector

`e6` a numeric vector

`e7` a numeric vector

`e8` a numeric vector

`e9` a numeric vector

`e10` a numeric vector

`e11` a numeric vector

`e12` a numeric vector

`e13` a numeric vector

`e14` a numeric vector

`e15` a numeric vector

`e16` a numeric vector

`e17` a numeric vector

`e18` a numeric vector

`a1` a numeric vector

`a2` a numeric vector

`a3` a numeric vector

`a4` a numeric vector

`a5` a numeric vector

`a6` a numeric vector

`a7` a numeric vector

`a8` a numeric vector

`a9` a numeric vector

`a10` a numeric vector

`a11` a numeric vector

`a12` a numeric vector

`a13` a numeric vector

`a14` a numeric vector

`a15` a numeric vector

`a16` a numeric vector

`a17` a numeric vector

`a18` a numeric vector

Examples

```
data(dzfData)
str(dzfData)
```

`dzmData` *Example twin extended kinship data: DZ Male data*

Description

Data for extended twin example ETC88.R

Usage

```
data("dzmData")
```

Format

A data frame with 1990 observations on the following 37 variables.

`famid`	a numeric vector
`e1`	a numeric vector
`e2`	a numeric vector
`e3`	a numeric vector
`e4`	a numeric vector
`e5`	a numeric vector
`e6`	a numeric vector
`e7`	a numeric vector
`e8`	a numeric vector
`e9`	a numeric vector
`e10`	a numeric vector
`e11`	a numeric vector
`e12`	a numeric vector
`e13`	a numeric vector
`e14`	a numeric vector
`e15`	a numeric vector
`e16`	a numeric vector
`e17`	a numeric vector
`e18`	a numeric vector
`a1`	a numeric vector
`a2`	a numeric vector
`a3`	a numeric vector
`a4`	a numeric vector
`a5`	a numeric vector
`a6`	a numeric vector
`a7`	a numeric vector
`a8`	a numeric vector
`a9`	a numeric vector
`a10`	a numeric vector
`a11`	a numeric vector
`a12`	a numeric vector
`a13`	a numeric vector
`a14`	a numeric vector
`a15`	a numeric vector
`a16`	a numeric vector
`a17`	a numeric vector
`a18`	a numeric vector

Examples

```
data(dzmData)
str(dzmData)
```

---

dzoData *Example twin extended kinship data: DZ opposite sex twins*

Description

Data for extended twin example ETC88.R

Usage

```
data("dzoData")
```

Format

A data frame with 3981 observations on the following 37 variables.

`famid` a numeric vector

`e1` a numeric vector

`e2` a numeric vector

`e3` a numeric vector

`e4` a numeric vector

`e5` a numeric vector

`e6` a numeric vector

`e7` a numeric vector

`e8` a numeric vector

`e9` a numeric vector

`e10` a numeric vector

`e11` a numeric vector

`e12` a numeric vector

`e13` a numeric vector

`e14` a numeric vector

`e15` a numeric vector

`e16` a numeric vector

`e17` a numeric vector

`e18` a numeric vector

`a1` a numeric vector

`a2` a numeric vector

`a3` a numeric vector

`a4`	a numeric vector
`a5`	a numeric vector
`a6`	a numeric vector
`a7`	a numeric vector
`a8`	a numeric vector
`a9`	a numeric vector
`a10`	a numeric vector
`a11`	a numeric vector
`a12`	a numeric vector
`a13`	a numeric vector
`a14`	a numeric vector
`a15`	a numeric vector
`a16`	a numeric vector
`a17`	a numeric vector
`a18`	a numeric vector

Examples

```
data(dzoData)
str(dzoData)
```

`eigenvec` *Eigenvector/Eigenvalue Decomposition*

Description

`eigenval` computes the real parts of the eigenvalues of a square matrix. `eigenvec` computes the real parts of the eigenvectors of a square matrix. `ieigenval` computes the imaginary parts of the eigenvalues of a square matrix. `ieigenvec` computes the imaginary parts of the eigenvectors of a square matrix. `eigenval` and `ieigenval` return nx1 matrices containing the real or imaginary parts of the eigenvalues, sorted in decreasing order of the modulus of the complex eigenvalue. For eigenvalues without an imaginary part, this is equivalent to sorting in decreasing order of the absolute value of the eigenvalue. (See `Mod` for more info.) `eigenvec` and `ieigenvec` return nxn matrices, where each column corresponds to an eigenvector. These are sorted in decreasing order of the modulus of their associated complex eigenvalue.

Usage

```
eigenval(x)
eigenvec(x)
ieigenval(x)
ieigenvec(x)
```

Arguments

`x`	the square matrix whose eigenvalues/vectors are to be calculated.

Details

Eigenvectors returned by `eigenvec` and `ieigenvec` are normalized to unit length.

See Also

`eigen`

Examples

```
A <- mxMatrix(values = runif(25), nrow = 5, ncol = 5, name = 'A')
G <- mxMatrix(values = c(0, -1, 1, -1), nrow=2, ncol=2, name='G')

model <- mxModel(A, G, name = 'model')

mxEval(eigenvec(A), model)
mxEval(eigenvec(G), model)
mxEval(eigenval(A), model)
mxEval(eigenval(G), model)
mxEval(ieigenvec(A), model)
mxEval(ieigenvec(G), model)
mxEval(ieigenval(A), model)
mxEval(ieigenval(G), model)
```

`example1` *Bivariate twin data, wide-format from Classic Mx Manual*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("example1")
```

Format

A data frame with 400 observations on the following variables.

`IDNum`	Twin pair ID
`Zygosity`	Zygosity of the twin pair
`X1`	X variable for twin 1

`Y1`	Y variable for twin 1
`X2`	X variable for twin 2
`Y2`	Y variable for twin 2

Details

Same as example2 but in wide format instead of tall.

Source

Classic Mx Manual.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(example1)
plot(X2 ~ X1, data = example1)
```

`example2` *Bivariate twin data, long-format from Classic Mx Manual*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("example2")
```

Format

A data frame with 800 observations on the following variables.

`IDNum`	ID number
`TwinNum`	Twin ID number
`Zygosity`	Zygosity of the twin
`X`	X variable for twins 1 and 2
`Y`	Y variable for twins 1 and 2

Details

Same as example1 but in tall format instead of wide.

Source

Classic Mx Manual.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation.

Examples

```
data(example2)
plot(Y ~ X, data = example2)
```

`expm` *Matrix exponential*

Description

Matrix exponential

Usage

```
expm(x)
```

Arguments

`x`	matrix

`factorExample1` *Example Factor Analysis Data*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("factorExample1")
```

Format

A data frame with 500 observations on the following variables.

`x1`

`x2`

`x3`

`x4`

`x5`

`x6`

`x7`

`x8`

`x9`

Details

This appears to be a three factor model, but perhaps with an odd loading structure.

Source

Simulated

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(factorExample1)
round(cor(factorExample1), 2)

factanal(covmat=cov(factorExample1), factors=3, rotation="promax")
```

`factorScaleExample1` *Example Factor Analysis Data for Scaling the Model*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("factorScaleExample1")
```

Format

A data frame with 200 observations on the following variables.

`X1`

`X2`

`X3`

`X4`

`X5`

`X6`

`X7`

`X8`

`X9`

`X10`

`X11`

`X12`

Details

This appears to be a three factor model with factor 1 loading on X1-X4, factor 2 on X5-X8, and factor 3 on X9-X12.

Source

Simulated

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
data(factorScaleExample1)
round(cor(factorScaleExample1), 2)
```

`factorScaleExample2` *Example Factor Analysis Data for Scaling the Model*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("factorScaleExample2")
```

Format

A data frame with 200 observations on the following variables.

`X1`

`X2`

`X3`

`X4`

`X5`

`X6`

`X7`

`X8`

`X9`

`X10`

`X11`

`X12`

Details

Three-factor data with factor 1 loading on X1-X4, factor 2 on X5-X8, and factor 3 on X9-X12. It differs from factorScaleExample1 in the scaling of the variables.

Source

Simulated

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(factorScaleExample2)
round(cor(factorScaleExample2), 2)

data(factorScaleExample2)
plot(sapply(factorScaleExample1, var), type='l', ylim=c(0, 6), lwd=3)
lines(1:12, sapply(factorScaleExample2, var), col='blue', lwd=3)
```

`genericFitDependencies,MxBaseFitFunction-method` *Add dependencies*

Description

If there is an expectation, then the fitfunction should always depend on it. Hence, subclasses that implement this method must ignore the passed-in dependencies and use "dependencies <- callNextMethod()" instead.

Usage

```
## S4 method for signature 'MxBaseFitFunction'
genericFitDependencies(.Object, flatModel, dependencies)
```

Arguments

`.Object`	fit function object
`flatModel`	flat model that lives with .Object
`dependencies`	accumulated dependency relationships

`HS.ability.data` *Holzinger & Swineford (1939) Ability in 301 children from 2 schools*

Description

This classic data set contains of intelligence-test scores from 301 children on 26 tests of cognitive ability.

The tests cover mental speed, memory, mathematical-ability, spatial, and verbal ability as listed below.

The data are also available in the MBESS package.

Usage

```
data("HS.ability.data")
```

Format

A data frame comprising 301 observations on 22 variables:

`id` student ID number (int)

`Gender` Sex (Factor w/ 2 levels “Female” and “Male”)

`grade` Grade in school (integer 7 or 8)

`agey` Age in years (integer)

`agem`	Age in months (integer)
`school`	School attended (Factor w/2 levels “Grant-White” and “Pasteur”)
`addition`	A speed test of addition (numeric)
`code`	A speed test (numeric)
`counting`	A speed test of counting groups of dots (numeric)
`straight`	A speed test discriminating straight and curved capitals (numeric)
`wordr`	A memory subtest of word recognition
`numberr`	A memory subtest of number recognition
`figurer`	A memory subtest of figure recognition
`object`	A memory subtest: object-number test
`numberf`	A memory subtest: number-figure test
`figurew`	A memory subtest: figure-word test
`deduct`	A mathematical subtest of deduction
`numeric`	A mathematical subtest of numerical puzzles
`problemr`	A mathematical subtest of problem reasoning
`series`	A mathematical subtest of series completion
`arithmet`	A mathematical subtest: Woody-McCall mixed fundamentals, form I
`visual`	A spatial subtest of visual perception
`cubes`	A spatial subtest
`paper`	A spatial subtest paper form board
`flags`	A spatial subtest (also known as lozenges)
`paperrev`	A spatial subtest additional paper form board test (can substitute for paper)
`flagssub`	A spatial subtest additional lozenges test (can substitute for flags)
`general`	A verbal subtest of general information
`paragrap`	A verbal subtest of paragraph comprehension
`sentence`	A verbal subtest of sentence completion
`wordc`	A verbal subtest of word classification
`wordm`	A verbal subtest of word meaning

Details

The data are from children who differ in grade (seventh- and eighth-grade) and are nested in one of two schools (Pasteur and Grant-White). You will see it in use elsewhere, both in R (`lavaan`, and `MBESS`), and in Joreskog (1969) reporting a CFA on the Grant-White school subject subset.

Some tests are alternate or substitute forms, e.g. `paperrev` (a paper form board test) can substitute for `paper` and `flagssub` for the lozenges test `flags`.

Source

Holzinger, K., and Swineford, F. (1939).

References

Holzinger, K., and Swineford, F. (1939). A study in factor analysis: The stability of a bifactor solution. *Supplementary Educational Monograph*, no. **48**. Chicago: University of Chicago Press.

Joreskog, K. G. (1969). A general approach to confirmatory maximum likelihood factor analysis. *Psychometrika*, **34**, 183-202.

Examples

```
data(HS.ability.data)
str(HS.ability.data)
levels(HS.ability.data$school)
plot(flags ~ flagssub, data = HS.ability.data)
```

`imxAddDependency` *Add a dependency*

Description

The dependency tracking system ensures that algebra and fitfunctions are not recomputed if their inputs have not changed. Dependency information is computed prior to handing the model off to the optimizer to reduce overhead during optimization.

Usage

```
imxAddDependency(source, sink, dependencies)
```

Arguments

`source`	a character vector of the names of the computation sources (inputs)
`sink`	the name of the computation sink (output)
`dependencies`	the dependency graph

Details

Each free parameter keeps track of all the objects that store that free parameter and the transitive closure of all algebras and fit functions that depend on that free parameter. Similarly, each definition variable keeps track of all the objects that store that free parameter and the transitive closure of all the algebras and fit functions that depend on that free parameter. At each iteration of the optimization, when the free parameter values are updated, all of the dependencies of that free parameter are marked as dirty (see `omxFitFunction.repopulateFun`). After an algebra or fit function is computed, `omxMarkClean()` is called to to indicate that the algebra or fit function is updated. Similarly, when definition variables are populated in FIML, all of the dependencies of the definition variables are marked as dirty. Particularly for FIML, the fact that non-definition-variable dependencies remain clean is a big performance gain.

`imxAutoOptionValue` *imxAutoOptionValue*

Description

Convert "Auto" placeholders in global mxOptions to actual default values.

Usage

```
imxAutoOptionValue(optionName, optionList = options()$mxOption)
```

Arguments

`optionName`	Character string naming the mxOption for which a numeric or integer value is wanted.
`optionList`	List of options; defaults to list of global mxOptions. imxAutoOptionValue

Details

This is an internal function exported for documentation purposes. Its primary purpose is to convert the on-load value of "Auto"to valid values for mxOptions ‘Gradient step size’, ‘Gradient iterations’, and ‘Function precision’–respectively, 1.0e-7, 1L, and 1e-14.

`imxCheckMatrices` *imxCheckMatrices*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxCheckMatrices(model)
```

Arguments

`model`	model

`imxCheckVariables` *imxCheckVariables*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxCheckVariables(flatModel, namespace)
```

Arguments

`flatModel`	flatModel
`namespace`	namespace

`imxConDecMatrixSlots` *Condense/de-condense slots of an MxMatrix*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxConDecMatrixSlots(object)
```

Arguments

`object`	of class MxMatrix

`imxConstraintRelations` *imxConstraintRelations*

Description

A string vector of valid constraint binary relations.

Usage

```
imxConstraintRelations
```

Format

An object of class `character` of length 3.

`imxConvertIdentifier` *imxConvertIdentifier*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxConvertIdentifier(identifiers, modelname, namespace, strict = FALSE)
```

Arguments

`identifiers`	identifiers
`modelname`	modelname
`namespace`	namespace
`strict`	strict

`imxConvertLabel` *imxConvertLabel*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxConvertLabel(label, modelname, dataname, namespace)
```

Arguments

`label`	label
`modelname`	modelname
`dataname`	dataname
`namespace`	namespace

imxConvertSubstitution *imxConvertSubstitution*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxConvertSubstitution(substitution, modelname, namespace)
```

Arguments

`substitution`	substitution
`modelname`	modelname
`namespace`	namespace

imxCreateMatrix *Create a matrix*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxCreateMatrix(
  .Object,
  labels,
  values,
  free,
  lbound,
  ubound,
  nrow,
  ncol,
  byrow,
  name,
  condenseSlots,
  joinKey,
  joinModel
)
```

Arguments

`.Object`	the matrix
`labels`	labels
`values`	values
`free`	free
`lbound`	lbound
`ubound`	ubound
`nrow`	nrow
`ncol`	ncol
`byrow`	byrow
`name`	name
`condenseSlots`	condenseSlots
`joinKey`	joinKey
`joinModel`	joinModel

`imxDataTypes` *Valid types of data that can be contained by MxData*

Description

Valid types of data that can be contained by MxData

Usage

```
imxDataTypes
```

Format

An object of class `character` of length 4.

imxDefaultGetSlotDisplayNames *imxDefaultGetSlotDisplayNames*

Description

Returns a list of display-friendly object slot names This is an internal function exported for those people who know what they are doing.

Usage

```
imxDefaultGetSlotDisplayNames(x, pattern = ".*")
```

Arguments

x	The object from which to get slot names
pattern	Initial pattern to match (default of '.*' matches any)

imxDeparse *Deparse for MxObjects*

Description

Deparse for MxObjects

Usage

```
imxDeparse(object, indent = "   ")
```

Arguments

object	object
indent	indent

imxDependentModels *Are submodels dependence?*

Description

Are submodels dependence?

Usage

```
imxDependentModels(model)
```

Arguments

`model`	model

imxDetermineDefaultOptimizer *imxDetermineDefaultOptimizer*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxDetermineDefaultOptimizer()
```

Details

Returns a character, the default optimizer

imxDmvnorm *A C implementation of dmvnorm*

Description

This API is visible to permit testing. Please do not use.

Usage

```
imxDmvnorm(loc, mean, sigma)
```

Arguments

`loc`	loc
`mean`	mean
`sigma`	sigma

imxEvalByName *imxEvalByName*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxEvalByName(name, model, compute = FALSE, show = FALSE)
```

Arguments

name	name
model	model
compute	compute
show	show

imxExtractMethod *imxExtractMethod*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxExtractMethod(model, index)
```

Arguments

model	model
index	index

`imxExtractNames` *imxExtractNames*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxExtractNames(lst)
```

Arguments

`lst`	lst

`imxExtractReferences` *imxExtractReferences*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxExtractReferences(lst)
```

Arguments

`lst`	lst

`imxExtractSlot` *imxExtractSlot*

Description

Checks for and extracts a slot from the object This is an internal function exported for those people who know what they are doing.

Usage

```
imxExtractSlot(x, name)
```

Arguments

`x`	The object
`name`	the name of the slot

`imxFlattenModel` *Remove hierarchical structure from model*

Description

Remove hierarchical structure from model

Usage

```
imxFlattenModel(model, namespace, unsafe = FALSE)
```

Arguments

`model`	model
`namespace`	namespace
`unsafe`	whether to skip sanity checks

`imxFreezeModel` *Freeze model*

Description

Remove free parameters and fit function from model.

Usage

```
imxFreezeModel(model)
```

Arguments

`model`	model

`imxGenerateLabels` *imxGenerateLabels*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxGenerateLabels(model)
```

Arguments

`model`	model

imxGenerateNamespace *imxGenerateNamespace*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxGenerateNamespace(model)
```

Arguments

model	model

imxGenericModelBuilder *imxGenericModelBuilder*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxGenericModelBuilder(
  model,
  lst,
  name,
  manifestVars,
  latentVars,
  productVars,
  submodels,
  remove,
  independent
)
```

Arguments

model	model
lst	lst
name	name
manifestVars	manifestVars
latentVars	latentVars

`productVars`	productVars
`submodels`	submodels
`remove`	remove
`independent`	independent

`imxGenSwift` *imxGenSwift*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxGenSwift(tc, sites, sfile)
```

Arguments

`tc`	tc
`sites`	sites
`sfile`	sfile

`imxGentleResize` *Resize an MxMatrix while preserving entries*

Description

Resize an MxMatrix while preserving entries

Usage

```
imxGentleResize(matrix, dimnames)
```

Arguments

`matrix`	the MxMatrix to resize
`dimnames`	desired dimnames for the new matrix

Value

a resized MxMatrix

Examples

```
m1 <- mxMatrix(values=1:9, nrow=3, ncol=3,
               dimnames=list(paste0('r',1:3), paste0('c',1:3)))

imxGentleResize(m1, dimnames=list(paste0('r',c(1,3,5)),
                                  paste0('c',c(2,4,6))))
```

---

imxGetNumThreads *imxGetNumThreads*

Description

This is an internal function exported for those people who know what they are doing.

This function hard codes responses to a set of environments, like detecting snowfall, or running on a cluster where "OMP_NUM_THREADS" is set or otherwise returning 1 or 2 cores to avoid consuming all the resources on CRAN's test machines during release cycles.

This makes it *not* suitable for getting the number of available threads.

To get the number of cores available locally you want `omxDetectCores` or perhaps the `detectCores` function in the parallel package.

Usage

```
imxGetNumThreads()
```

---

imxGetSlotDisplayNames *imxGetSlotDisplayNames*

Description

Returns a list of display-friendly object slot names This is an internal function exported for those people who know what they are doing.

Usage

```
imxGetSlotDisplayNames(
  object,
  pattern = ".*",
  slotList = slotNames(object),
  showDots = FALSE,
  showEmpty = FALSE
)
```

Arguments

`object`	The object from which to get slot names
`pattern`	Initial pattern to match (default of '.*' matches any)
`slotList`	List of slots for which toget display names (default = slotNames(object), i.e., all)
`showDots`	Include slots whose names start with '.' (default FALSE)
`showEmpty`	Include slots with length-zero contents (default FALSE)

`imxHasConstraint` *imxHasConstraint*

Description

This is an internal function exported for those people who know what they are doing. This function checks if a model (or its submodels) has at least one MxConstraint.

Usage

```
imxHasConstraint(model)
```

Arguments

`model`	model

`imxHasDefinitionVariable` *imxHasDefinitionVariable*

Description

This is an internal function exported for those people who know what they are doing. This function checks if a model (or its submodels) has at least one definition variable.

Usage

```
imxHasDefinitionVariable(model)
```

Arguments

`model`	model

`imxHasNPSOL` *imxHasNPSOL*

Description

imxHasNPSOL

Usage

```
imxHasNPSOL()
```

Value

Returns TRUE if the NPSOL proprietary optimizer is compiled and linked with OpenMx. Otherwise FALSE.

`imxHasOpenMP` *imxHasOpenMP*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxHasOpenMP()
```

`imxHasThresholds` *imxHasThresholds*

Description

This is an internal function exported for those people who know what they are doing. This function checks if a model (or its submodels) has any thresholds.

Usage

```
imxHasThresholds(model)
```

Arguments

`model`	model

`imxHasWLS` *imxHasWLS*

Description

This is an internal function exported for those people who know what they are doing. This function checks if a model uses a fitfunction with WLS units.

Usage

```
imxHasWLS(model)
```

Arguments

`model`	model

`imxIdentifier` *imxIdentifier*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxIdentifier(namespace, name)
```

Arguments

`namespace`	namespace
`name`	name

`imxIndependentModels` *Are submodels independent?*

Description

Are submodels independent?

Usage

```
imxIndependentModels(model)
```

Arguments

`model`	model

imxInitModel *imxInitModel*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxInitModel(model)
```

Arguments

`model`	model

imxIsDefinitionVariable *imxIsDefinitionVariable*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxIsDefinitionVariable(name)
```

Arguments

`name`	name

imxIsMultilevel *imxIsMultilevel*

Description

This is an internal function exported for those people who know what they are doing. If you don't know what you're doing, but want to, here's a brief description of the function. You give this function an MxModel. It returns TRUE if the model is multilevel and FALSE otherwise.

Usage

```
imxIsMultilevel(model)
```

Arguments

`model`	model

`imxIsPath` *imxIsPath*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxIsPath(value)
```

Arguments

`value`	value

`imxLocateFunction` *imxLocateFunction*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxLocateFunction(function_name)
```

Arguments

`function_name`	function_name

`imxLocateIndex` *imxLocateIndex*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxLocateIndex(model, name, referant)
```

Arguments

`model`	model
`name`	name
`referant`	referant

`imxLocateLabel` *imxLocateLabel*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxLocateLabel(label, model, parameter)
```

Arguments

`label`	label
`model`	model
`parameter`	parameter

`imxLog` *Test thread-safe output code*

Description

This is the code that the backend uses to write diagnostic information to standard error. This function should not be called from R. We make it available only for testing.

Usage

```
imxLog(str)
```

Arguments

`str`	the character string to output

`imxLookupSymbolTable` *imxLookupSymbolTable*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxLookupSymbolTable(name)
```

Arguments

`name`	name

`imxModelBuilder`	*imxModelBuilder*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxModelBuilder(
  model,
  lst,
  name,
  manifestVars,
  latentVars,
  productVars,
  submodels,
  remove,
  independent
)
```

Arguments

`model`	model
`lst`	lst
`name`	name
`manifestVars`	manifestVars
`latentVars`	latentVars
`productVars`	productVars
`submodels`	submodels
`remove`	remove
`independent`	independent

Details

TODO: It probably makes sense to split this into separate methods. For example, modelAddVariables and modelRemoveVariables could be their own methods. This would reduce some cut&paste duplication.

`imxModelTypes` *imxModelTypes*

Description

A list of supported model types

Usage

```
imxModelTypes
```

Format

An object of class `list` of length 3.

`imxMpiWrap` *imxMpiWrap*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxMpiWrap(fun)
```

Arguments

`fun`	fun

`imxOriginalMx` *Run an classic mx script*

Description

For this to work, classic mx must be installed, and callable from the command line.

Usage

```
imxOriginalMx(mx.filename, output.directory)
```

Arguments

`mx.filename`	Name of file containing the mx script.
`output.directory`	Where to write mxo output from the script

Value

processed matrix output.

Examples

```
## Not run:
output = imxOriginalMx(mx.filename = "power1.mx", "~/Desktop")

## End(Not run)
```

`imxPPML` *imxPPML*

Description

Potentially enable the PPML optimization for the given model.

Usage

```
imxPPML(model, flag = TRUE)
```

Arguments

`model`	the MxModel to evaluate
`flag`	whether to potentially enable PPML

`imxPPML.Test.Battery` *imxPPML.Test.Battery*

Description

PPML can be applied to a number of special cases. This function will test the given model for all of these special cases.

Usage

```
imxPPML.Test.Battery(
  model,
  verbose = FALSE,
  testMissingness = TRUE,
  testPermutations = TRUE,
  testEstimates = TRUE,
  testFakeLatents = TRUE,
  tolerances = c(0.001, 0.001, 0.001)
)
```

Arguments

`model`	the model to test
`verbose`	whether to print diagnostics
`testMissingness`	try with missingness
`testPermutations`	try with permutations
`testEstimates`	examine estimates
`testFakeLatents`	try with fake latents
`tolerances`	a vector of tolerances

Details

Requirements for model passed to this function: - Path-specified - Means vector must be present - Covariance data (with data means vector) - (Recommended) All error variances should be specified on the diagonal of the S matrix, and not as a latent with a loading only on to that manifest

Function will test across all permutations of: - Covariance vs Raw data - Means vector present vs Means vector absent - Path versus Matrix specification - All orders of permutations of latents with manifests

---

`imxPPML.Test.Test` *imxPPML.Test.Test*

Description

Test that PPML solutions match non-PPML solutions.

Usage

```
imxPPML.Test.Test(
  model,
  checkLL = TRUE,
  checkByName = FALSE,
  tolerance = 0.5,
  testEstimates = TRUE
)
```

Arguments

`model`	the MxModel to evaluate
`checkLL`	whether to check log likelihood
`checkByName`	check values using their names
`tolerance`	closeness tolerance for check
`testEstimates`	whether to test for the same parameter estimates

Details

This is an internal function used for comparing PPML and non-PPML solutions. Generally, non-developers will not use this function.

`imxPreprocessModel` *imxPreprocessModel*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxPreprocessModel(model)
```

Arguments

`model`	model

`imxReplaceMethod` *imxReplaceMethod*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxReplaceMethod(x, name, value)
```

Arguments

`x`	the thing
`name`	name
`value`	value

`imxReplaceModels` *Replace parts of a model*

Description

Replace parts of a model

Usage

```
imxReplaceModels(model, replacements)
```

Arguments

`model`	model
`replacements`	replacements

imxReplaceSlot *imxReplaceSlot*

Description

Checks for and replaces a slot from the object This is an internal function exported for those people who know what they are doing.

Usage

```
imxReplaceSlot(x, name, value, check = TRUE)
```

Arguments

`x`	object
`name`	the name of the slot
`value`	replacement value
`check`	Check replacement value for validity (default TRUE)

imxReportProgress *Report backend progress*

Description

Prints a show status string to the console without emitting a newline.

Usage

```
imxReportProgress(info, eraseLen)
```

Arguments

`info`	the character string to print
`eraseLen`	the number of characters to erase

Examples

```
library(OpenMx)

previousLen <<- 0

easyReportProcess <- function(msg) {
imxReportProgress(msg, previousLen)
previousLen <<- nchar(msg)
}
```

```
demo <- function() {
easyReportProcess("abc123")
Sys.sleep(1)
easyReportProcess("this is much longer")
Sys.sleep(1)
easyReportProcess("this is short")
Sys.sleep(1)
easyReportProcess("almost done")
Sys.sleep(1)
easyReportProcess("")
cat("DONE!", fill=TRUE)
}

demo()
```

`imxReservedNames` *imxReservedNames*

Description

Vector of reserved names

Usage

```
imxReservedNames
```

Format

An object of class `character` of length 7.

`imxReverseIdentifier` *imxReverseIdentifier*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxReverseIdentifier(model, name)
```

Arguments

`model`	model
`name`	name

`imxRobustSE` *imxRobustSE*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxRobustSE(model, details = FALSE)
```

Arguments

`model`	An OpenMx model object that has been run.
`details`	Logical. whether to return the full parameter covariance matrix.

Details

This function computes robust standard errors via a sandwich estimator. The "bread" of the sandwich is the numerically computed inverse Hessian of the likelihood function. This is what is typically used for standard errors throughout OpenMx. The "meat" of the sandwich is proportional to the covariance matrix of the numerically computed row derivatives of the likelihood function (i.e. row gradients).

When `details=FALSE`, only the standard errors are returned.

When `details=TRUE`, a list with five named elements is returned. Element `SE` is the vector of standard errors that is also returned when `details=FALSE`. Element `cov` is the full robust covariance matrix of the parameter estimates; the square root of the diagonal of `cov` gives the standard errors. Element `bread` is the aforementioned "bread"–the naive (non-robust) covariance matrix of the parameter estimates. Element `meat` is the aforementioned "meat," proportional to the covariance matrix of the row gradients. Element `TIC` is the model's Takeuchi Information Criterion, which is a generalization of AIC calculated from the "bread," the "meat," and the loglikelihood at the maximum-likelihood solution.

This function does not work correctly with multigroup models in which the groups themselves contain subgroups, or in which groups contain references to objects in other groups. This function also does not correctly handle multilevel data.

`imxRowGradients` *imxRowGradients*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxRowGradients(model, robustSE = FALSE)
```

Arguments

`model`	An OpenMx model object that has been run
`robustSE`	Logical; are the row gradients being requested to calculate robust standard errors?

Details

This function computes the gradient for each row of data. The returned object is a matrix with the same number of rows as the data, and the same number of columns as there are free parameters.

`imxSameType` *imxSameType*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxSameType(a, b)
```

Arguments

`a`	a
`b`	b

`imxSeparatorChar` *imxSeparatorChar*

Description

The character between the model name and the named entity inside the model.

Usage

```
imxSeparatorChar
```

Format

An object of class `character` of length 1.

`imxSfClient` *imxSfClient*

Description

As of snowfall 1.84, the snowfall supervisor process stores an internal state information in a variable named ".sfOption" that is located in the "snowfall" namespace. The snowfall client processes store internal state information in a variable named ".sfOption" that is located in the global namespace.

Usage

```
imxSfClient()
```

Details

As long as the previous statement is true, then the current process is a snowfall client if-and-only-if exists(".sfOption").

`imxSimpleRAMPredicate` *imxSimpleRAMPredicate*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxSimpleRAMPredicate(model)
```

Arguments

`model`	model

`imxSparseInvert` *Sparse symmetric matrix invert*

Description

This API is visible to permit testing. Please do not use.

Usage

```
imxSparseInvert(mat)
```

Arguments

`mat`	the matrix to invert

`imxSquareMatrix` *imxSquareMatrix*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxSquareMatrix(.Object)
```

Arguments

`.Object`	.Object

`imxSymmetricMatrix` *imxSymmetricMatrix*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxSymmetricMatrix(.Object)
```

Arguments

`.Object`	.Object

`imxTypeName` *imxTypeName*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxTypeName(model)
```

Arguments

`model`	model

`imxUntitledName` *imxUntitledName*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxUntitledName()
```

Details

Returns a character, the name of the next untitled entity

`imxUntitledNumber` *imxUntitledNumber*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxUntitledNumber()
```

Details

Increments the untitled number counter and returns its value

`imxUntitledNumberReset` *imxUntitledNumberReset*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxUntitledNumberReset()
```

Details

Resets the imxUntitledNumber counter

`imxUpdateModelValues` *imxUpdateModelValues*

Description

Deprecated. This function does not handle parameters with equality constraints. Do not use.

Usage

```
imxUpdateModelValues(model, flatModel, values)
```

Arguments

`model`	model
`flatModel`	flat model
`values`	values to update

`imxVariableTypes` *imxVariableTypes*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxVariableTypes
```

Format

An object of class `character` of length 2.

Details

The acceptable variable types

`imxVerifyMatrix` *imxVerifyMatrix*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxVerifyMatrix(.Object)
```

Arguments

`.Object`	.Object

`imxVerifyModel` *imxVerifyModel*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxVerifyModel(model)
```

Arguments

`model`	model

`imxVerifyName` *imxVerifyName*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxVerifyName(name, stackNumber)
```

Arguments

`name`	name
`stackNumber`	stackNumber

`imxVerifyReference` *imxVerifyReference*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
imxVerifyReference(reference, stackNumber)
```

Arguments

`reference`	reference
`stackNumber`	stackNumber

`imxWlsChiSquare` *Calculate Chi Square for a WLS Model*

Description

This is an internal function used to calculate the Chi Square distributed fit statistic for weighted least squares models.

Usage

```
imxWlsChiSquare(model, J=NA)
```

Arguments

`model`	An MxModel object with acov (WLS) data
`J`	Optional pre-computed Jacobian matrix

Details

The Chi Square fit statistic for models fit with maximum likelihood depends on the difference in model fit in minus two log likelihood units between the saturated model and the more restricted model under investigation. For models fit with weighted least squares a different expression is required. If J is the first derivative (Jacobian) of the mapping from the free parameters to the unique elements of the expected covariance, means, and threholds, J_c is the orthogonal complement of J, W is the inverse of the full weight matrix, and e is the difference between the sample-estimated and model-implied covariance, means, and thresholds, then the Chi Square fit statistic is

$$\chi^2 = e' J_c (J_c' W J_c)^- 1 J_c' e$$

with e' indicating the transpose of e. This Equation 2.20a from Browne (1984) where he showed that this statistic is chi-square distributed with the conventional degrees of freedom.

Mean and variance adjusted Chi Square statistics are also computed following Asparouhov and Muthen (2006).

Value

A named list with components

Chi numeric value of the Chi Square fit statistic.

ChiDoF degrees of freedom for the Chi Square fit statistic.

ChiM numeric value of the mean adjusted Chi Square fit statistic

ChiMV numeric value of the mean and variance adjusted Chi Square fit statistic

mAdjust numeric value of the mean adjustment

mvAdjust numeric value of the mean and variance adjustment

dstar adjusted degrees of freedom for the mean and variance adjusted Chi Square fit statistic

References

M. W. Browne. (1984). Asymptotically Distribution-Free Methods for the Analysis of Covariance Structures. *British Journal of Mathematical and Statistical Psychology*, **37**, 62-83.

T. Asparouhov and B. O. Muthen. (2006). Robust Chi Square Difference Testing with Mean and Variance Adjusted Test Statistics. *Mplus Web Notes: No. 10*.

`imxWlsStandardErrors` *Calculate Standard Errors for a WLS Model*

Description

This is an internal function used to calculate standard errors for weighted least squares models.

Usage

```
imxWlsStandardErrors(model)
```

Arguments

`model`	An MxModel object with acov (WLS) data

Details

The standard errors for models fit with maximum likelihood are related to the second derivative (Hessian) of the likelihood function with respect to the free parameters. For models fit with weighted least squares a different expression is required. If J is the first derivative (Jacobian) of the mapping from the free parameters to the unique elements of the expected covariance, means, and thresholds, V is the weight matrix used, W is the inverse of the full weight matrix, and $U = VJ(J'VJ)^{-1}$, then the asymptotic covariance matrix of the free parameters is

$$Acov(\theta) = U'WU$$

with U' indicating the transpose of U.

Value

A named list with components

SE The standard errors of the free parameters

Cov The full covariance matrix of the free parameters. The square root of the diagonal elements of Cov equals SE.

Jac The Jacobian computed to obtain the standard errors.

References

M. W. Browne. (1984). Asymptotically Distribution-Free Methods for the Analysis of Covariance Structures. *British Journal of Mathematical and Statistical Psychology*, **37**, 62-83.

F. Yang-Wallentin, K. G. Jöreskog, & H. Luo. (2010). Confirmatory Factor Analysis of Ordinal Variables with Misspecified Models. *Structural Equation Modeling*, **17**, 392-423.

`jointdata` *Joint Ordinal and continuous variables to be modeled together*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("jointdata")
```

Format

A data frame with 250 observations on the following variables.

`z1` Continuous variable

`z2` Ordinal variable with 2 levels (0, 1)

`z3` Continuous variable

`z4` Ordinal variable with 4 levels (0, 1, 2, 3)

`z5` Ordinal variable with 3 levels (0, 1, 3)

Details

Data generated to test the joint ML algorithm thoroughly.

Source

Simulated.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(jointdata)
head(jointdata)
```

`latentMultipleRegExample1`
Example data for multiple regression among latent variables

Description

Data set used in some of OpenMx's examples.

Usage

```
data("latentMultipleRegExample1")
```

Format

A data frame with 200 observations on the following variables.

`X1`	Factor 1 indicator
`X2`	Factor 1 indicator
`X3`	Factor 1 indicator
`X4`	Factor 1 indicator
`X5`	Factor 2 indicator
`X6`	Factor 2 indicator
`X7`	Factor 2 indicator
`X8`	Factor 2 indicator
`X9`	Factor 3 indicator
`X10`	Factor 3 indicator
`X11`	Factor 3 indicator
`X12`	Factor 3 indicator

Details

Factor 1 strongly predicts factor 3. Factor 2 weakly predicts factor 3.

Source

Simulated.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(latentMultipleRegExample1)
round(cor(latentMultipleRegExample1), 2)
```

---

latentMultipleRegExample2

Example data for multiple regression among latent variables

Description

Data set used in some of OpenMx's examples.

Usage

```
data("latentMultipleRegExample2")
```

Format

A data frame with 200 observations on the following variables.

`X1`	Factor 1 indicator
`X2`	Factor 1 indicator
`X3`	Factor 1 indicator
`X4`	Factor 1 indicator
`X5`	Factor 2 indicator
`X6`	Factor 2 indicator
`X7`	Factor 2 indicator
`X8`	Factor 2 indicator
`X9`	Factor 3 indicator
`X10`	Factor 3 indicator
`X11`	Factor 3 indicator
`X12`	Factor 3 indicator

Details

Factor 1 strongly predicts factor 3. Factor 2 weakly predicts factor 3. Very similar to latentMultipleRegExample1.

Source

Simulated.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(latentMultipleRegExample2)
round(cor(latentMultipleRegExample2), 2)
```

`lazarsfeld` *Respondent-soldiers on four dichotomous items*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("lazarsfeld")
```

Format

A data frame with 1000 observations on four dichotomous items.

`armyrun` In general how do you feel the Army is run?

`favatt` Do you think when you are discharged you will [have] a favorable attitude toward the Army?

`squaredeal` In general do you feel you yourself have gotten a square deal from the Army?

`welfare` Do you feel that the Army is trying its best to look out for the welfare of enlisted men?

`frequency` Frequency of response pattern.

Details

A straightforward descriptive analysis of these data shows that negative responses are more numerous except on item 1; and that there is a positive association between each pair of items. A soldier who responds positively to any one item is more likely to respond positively to a second item. Lazarsfeld's analysis is based on the assumption that each soldier can be thought of as belong to one of two latent classes. The probability of positive response to an item is different in one group than in the other. Most importantly, he is willing to assume that for an individual respondent the responses to items are statistically independent.

Source

Lazarsfeld, Paul F. (1950b) "Some Latent Structures", Chapter 11 in Stouffer (1950).

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

http://www.people.vcu.edu/~nhenry/LSA50.htm

Examples

```
data(lazarsfeld)
```

`logm` *Matrix logarithm*

Description

Matrix logarithm

Usage

```
logm(x, tol = .Machine$double.eps)
```

Arguments

`x`	matrix
`tol`	tolerance

`LongitudinalOverdispersedCounts` *Longitudinal, Overdispersed Count Data*

Description

Four-timepoint longitudinal data generated from an arbitrary Monte Carlo simulation, for 1000 simulees. The response variable is a discrete count variable. There are three time-invariant covariates. The data are available in both "wide" and "long" format.

Usage

```
data("LongitudinalOverdispersedCounts")
```

Format

The "long" format dataframe, `longData`, has 4000 rows and the following variables (columns):

1. `id`: Factor; simulee ID code.
2. `tiem`: Numeric; represents the time metric, wave of assessment.
3. `x1`: Numeric; time-invariant covariate.
4. `x2`: Numeric; time-invariant covariate.
5. `x3`: Numeric; time-invariant covariate.
6. `y`: Numeric; the response ("dependent") variable.

The "wide" format dataset, `wideData`, is a numeric 1000x12 matrix containing the following variables (columns):

1. `id`: Simulee ID code.
2. `x1`: Time-invariant covariate.
3. `x3`: Time-invariant covariate.
4. `x3`: Time-invariant covariate.
5. `y0`: Response at initial wave of assessment.
6. `y1`: Response at first follow-up.
7. `y2`: Response at second follow-up.
8. `y3`: Response at third follow-up.
9. `t0`: Time variable at initial wave of assessment (in this case, 0).
10. `t1`: Time variable at first follow-up (in this case, 1).
11. `t2`: Time variable at second follow-up (in this case, 2).
12. `t3`: Time variable at third follow-up (in this case, 3).

Examples

```
data(LongitudinalOverdispersedCounts)
head(wideData)
str(longData)
#Let's try ordinary least-squares (OLS) regression:
olsmod <- lm(y~tiem+x1+x2+x3, data=longData)
#We will see in the diagnostic plots that the residuals are poorly approximated by normality,
#and are heteroskedastic.  We also know that the residuals are not independent of one another,
#because we have repeated-measures data:
plot(olsmod)
#In the summary, it looks like all of the regression coefficients are significantly different
#from zero, but we know that because the assumptions of OLS regression are violated that
#we should not trust its results:
summary(olsmod)

#Let's try a generalized linear model (GLM).  We'll use the quasi-Poisson quasilikelihood
#function to see how well the y variable is approximated by a Poisson distribution
#(conditional on time and covariates):
glm.mod <- glm(y~tiem+x1+x2+x3, data=longData, family="quasipoisson")
#The estimate of the dispersion parameter should be about 1.0 if the data are
#conditionally Poisson.  We can see that it is actually greater than 2,
#indicating overdispersion:
summary(glm.mod)
```

`multiData1` *Data for multiple regression*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("multiData1")
```

Format

A data frame with 500 observations on the following variables.

`x1`

`x2`

`x3`

`x4`

`y`

Details

x1-x4 are predictor variables, and y is the outcome.

Source

Simulated.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
data(multiData1)
summary(lm(y ~ ., data=multiData1))
#results can be replicated in OpenMx.
```

mxAlgebra *Create MxAlgebra Object*

Description

This function creates a new MxAlgebra. The common use is to compute a value in a model: for instance a standardized value of a parameter, or a parameter which is a function of other values. It is also used in models with an mxFitFunctionAlgebra objective function.

note: Unless needed in the model objective, algebras are only computed twice: once at the beginning and once at the end of running a model, so adding them doesn't often add a lot of overhead.

Usage

```
mxAlgebra(expression, name = NA, dimnames = NA, ..., fixed = FALSE,
          joinKey=as.character(NA), joinModel=as.character(NA),
verbose=0L, initial=matrix(as.numeric(NA),1,1),
        recompute=c('always','onDemand'))
```

Arguments

`expression`	An R expression of OpenMx-supported matrix operators and matrix functions.
`name`	An optional character string indicating the name of the object.
`dimnames`	list. The dimnames attribute for the algebra: a list of length 2 giving the row and column names respectively. An empty list is treated as NULL, and a list of length one as row names. The list can be named, and the list names will be used as names for the dimensions.
`...`	Not used. Forces other arguments to be specified by name.
`fixed`	Deprecated. Use the 'recompute' argument instead.
`joinKey`	The name of the column in current model's raw data that is used as a foreign key to match against the primary key in the joinModel's raw data.
`joinModel`	The name of the model that this matrix joins against.
`verbose`	For values greater than zero, enable runtime diagnostics.
`initial`	a matrix. When `recompute='onDemand'`, you must provide this initial algebra result.
`recompute`	If 'onDemand', this algebra will not be recomputed automatically when things it depends on change. mxComputeOnce can be used to force it to recompute.

Details

The mxAlgebra function is used to create algebraic expressions that operate on one or more MxMatrix objects. To evaluate an MxAlgebra object, it must be placed in an MxModel object, along with all referenced `MxMatrix` objects and the `mxFitFunctionAlgebra` function. The `mxFitFunctionAlgebra` function must reference by name the `MxAlgebra` object to be evaluated.

Note: f the result for an MxAlgebra depends upon one or more "definition variables" (see `mxMatrix()`), then the value returned after the call to `mxRun()` will be computed using the values of those definition variables in the first (i.e., first before any automated sorting is done) row of the raw dataset.

The following operators and functions are supported in mxAlgebra:

Operators

`solve()` Inversion

`t()` Transposition

`^` Elementwise powering

`%^%` Kronecker powering

`+` Addition

`-` Subtraction

`%*%` Matrix Multiplication

`*` Elementwise product

`/` Elementwise division

`%x%` Kronecker product

`%&%` Quadratic product: pre- and post-multiply B by A and its transpose t(A), i.e: A `%&%` B == A `%*%` B `%*%` t(A)

Functions

`cov2cor` Convert covariance matrix to correlation matrix

`chol` Cholesky Decomposition

`cbind` Horizontal adhesion

`rbind` Vertical adhesion

`colSums` Matrix column sums as a column vector

`rowSums` Matrix row sums as a column vector

`det` Determinant

`tr` Trace

`sum` Sum

`mean` Arithmetic mean

`prod` Product

`max` Maximum

`min` Min

`abs` Absolute value

`sin` Sine

`sinh` Hyperbolic sine

`asin` Arcsine

`asinh` Inverse hyperbolic sine

`cos` Cosine

`cosh`	Hyperbolic cosine
`acos`	Arccosine
`acosh`	Inverse hyperbolic cosine
`tan`	Tangent
`tanh`	Hyperbolic tangent
`atan`	Arctangent
`atanh`	Inverse hyperbolic tangent
`exp`	Exponent
`log`	Natural Logarithm
`mxRobustLog`	Robust natural logarithm
`sqrt`	Square root
`p2z`	*Standard*-normal quantile
`logp2z`	*Standard*-normal quantile from log probabilities
`lgamma`	Log-gamma function
`lgamma1p`	Compute log(gamma(x+1)) accurately for small x
`eigenval`	Eigenvalues of a square matrix. Usage: eigenval(x); eigenvec(x); ieigenval(x); ieigenvec(x)
`rvectorize`	Vectorize by row
`cvectorize`	Vectorize by column
`vech`	Half-vectorization
`vechs`	Strict half-vectorization
`vech2full`	Inverse half-vectorization
`vechs2full`	Inverse strict half-vectorization
`vec2diag`	Create matrix from a diagonal vector (similar to diag)
`diag2vec`	Extract diagonal from matrix (similar to diag)
`expm`	Matrix Exponential
`logm`	Matrix Logarithm
`omxExponential`	Matrix Exponential
`omxMnor`	Multivariate Normal Integration
`omxAllInt`	All cells Multivariate Normal Integration
`omxNot`	Perform unary negation on a matrix
`omxAnd`	Perform binary and on two matrices
`omxOr`	Perform binary or on two matrices
`omxGreaterThan`	Perform binary greater on two matrices
`omxLessThan`	Perform binary less than on two matrices
`omxApproxEquals`	Perform binary equals to (within a specified epsilon) on two matrices
`omxSelectRows`	Filter rows from a matrix

omxSelectCols Filter columns from a matrix

omxSelectRowsAndCols Filter rows and columns from a matrix

mxEvaluateOnGrid Evaluate an algebra on an abscissa grid and collect column results

mpinv Moore-Penrose Inverse

If solve is used on an uninvertible square matrix in R, via mxEval(), it will fail with an error will; if solve is used on an uninvertible square matrix during runtime, it will fail silently.

mxRobustLog is the same as log except that it returns -745 instead of -Inf for an argument of 0. The value -745 is less than log(4.94066e-324), a good approximation of negative infinity because the log of any number represented as a double will be of smaller absolute magnitude.

There are also several multi-argument functions usable in MxAlgebras, which apply themselves elementwise to the matrix provided as their first argument. These functions have slightly different usage from their R counterparts. Their result is always a matrix with the same dimensions as that provided for their first argument. Values must be provided for ALL arguments of these functions, in order. Provide zeroes as logical values of FALSE, and non-zero numerical values as logical values of TRUE. For most of these functions, OpenMx cycles over values of arguments other than the first, by column (i.e., in column-major order), to the length of the first argument. Notable exceptions are the log, log.p, and lower.tail arguments to probability-distribution-related functions, for which only the [1,1] element is used. It is recommended that all arguments after the first be either (1) scalars, or (2) matrices with the same dimensions as the first argument.

Function	Arguments	Notes
besselI & besselK	x,nu,expon.scaled	Note that OpenMx *does* cycle over the elements of exp
besselJ & besselY	x,nu	
dbeta	x,shape1,shape2,ncp,log	The algorithm for the non-central beta distribution is u
pbeta	q,shape1,shape2,ncp,lower.tail,log.p	Values of ncp are handled as with dbeta().
dbinom	x,size,prob,log	
pbinom	q,size,prob,lower.tail,log.p	
dcauchy	x,location,scale,log	
pcauchy	q,location,scale,lower.tail,log.p	
dchisq	x,df,ncp,log	The algorithm for the non-central chi-square distributio
pchisq	q,df,ncp,lower.tail,log.p	Values of ncp are handled as with dchisq().
omxDnbinom	x,size,prob,mu,log	Exactly one of arguments size, prob, and mu should b
omxPnbinom	q,size,prob,mu,lower.tail,log.p	Arguments are handled as with omxDnbinom().
dpois	x,lambda,log	
ppois	q,lambda,lower.tail,log.p	

Value

Returns a new MxAlgebra object.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

MxAlgebra for the S4 class created by mxAlgebra. mxFitFunctionAlgebra for an objective function which takes an MxAlgebra or MxMatrix object as the function to be minimized. MxMatrix and mxMatrix for objects which may be entered in the expression argument and the function that creates them. More information about the OpenMx package may be found here.

Examples

```
A <- mxMatrix("Full", nrow = 3, ncol = 3, values=2, name = "A")

# Simple example: algebra B simply evaluates to the matrix A
B <- mxAlgebra(A, name = "B")

# Compute A + B
C <- mxAlgebra(A + B, name = "C")

# Compute sin(C)
D <- mxAlgebra(sin(C), name = "D")

# Make a model and evaluate the mxAlgebra object 'D'
A <- mxMatrix("Full", nrow = 3, ncol = 3, values=2, name = "A")
model <- mxModel(model="AlgebraExample", A, B, C, D )
fit   <- mxRun(model)
mxEval(D, fit)


# Numbers in mxAlgebras are upgraded to 1x1 matrices
# Example of Kronecker powering (%^%) and multiplication (%*%)
A  <- mxMatrix(type="Full", nrow=3, ncol=3, value=c(1:9), name="A")
m1 <- mxModel(model="kron", A, mxAlgebra(A %^% 2, name="KroneckerPower"))
mxRun(m1)$KroneckerPower

# Running kron
# mxAlgebra 'KroneckerPower'
# $formula:  A %^% 2
# $result:
#      [,1] [,2] [,3]
# [1,]    1   16   49
# [2,]    4   25   64
# [3,]    9   36   81
```

MxAlgebra-class *MxAlgebra Class*

Description

MxAlgebra is an S4 class. An MxAlgebra object is a named entity. New instances of this class can be created using the function mxAlgebra.

Details

The MxAlgebra class has the following slots:

name	-	The name of the object
formula	-	The R expression to be evaluated
result	-	a matrix with the computation result

The 'name' slot is the name of the MxAlgebra object. Use of MxAlgebra objects in the `mxConstraint` function or an objective function requires reference by name.

The 'formula' slot is an expression containing the expression to be evaluated. These objects are operated on or related to one another using one or more operations detailed in the `mxAlgebra` help file.

The 'result' slot is used to hold the results of computing the expression in the 'formula' slot. If the containing model has not been executed, then the 'result' slot will hold a 0 x 0 matrix. Otherwise the slot will store the computed value of the algebra using the final estimates of the free parameters.

Slots may be referenced with the $ symbol. See the documentation for Classes and the examples in the `mxAlgebra` document for more information.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

`mxAlgebra`, `mxMatrix`, `MxMatrix`

`MxAlgebraFormula-class` *MxAlgebraFormula*

Description

This is an internal class for the formulas used in `mxAlgebra` calls.

mxAlgebraFromString *Create MxAlgebra object from a string*

Description

Create MxAlgebra object from a string

Usage

```
mxAlgebraFromString(algString, name = NA, dimnames = NA, ...)
```

Arguments

`algString`	the character string to convert into an R expression
`name`	An optional character string indicating the name of the object.
`dimnames`	list. The dimnames attribute for the algebra: a list of length 2 giving the row and column names respectively. An empty list is treated as NULL, and a list of length one as row names. The list can be named, and the list names will be used as names for the dimensions.
`...`	Forwarded verbatim to mxAlgebra

See Also

mxAlgebra

Examples

```
A <- mxMatrix(values = runif(25), nrow = 5, ncol = 5, name = 'A')
B <- mxMatrix(values = runif(25), nrow = 5, ncol = 5, name = 'B')
model <- mxModel(A, B, name = 'model',
  mxAlgebraFromString("A * (B + A)", name = 'test'))
model <- mxRun(model)
model[['test']]$result
A$values * (B$values + A$values)
```

mxAlgebraObjective *DEPRECATED: Create MxAlgebraObjective Object*

Description

WARNING: Objective functions have been deprecated as of OpenMx 2.0.

Please use MxFitFunctionAlgebra() instead. As a temporary workaround, MxAlgebraObjective returns a list containing a NULL MxExpectation object and an MxFitFunctionAlgebra object.

All occurrences of

mxAlgebraObjective(algebra, numObs = NA, numStats = NA)

Should be changed to

mxFitFunctionAlgebra(algebra, numObs = NA, numStats = NA)

Arguments

`algebra`	A character string indicating the name of an MxAlgebra or MxMatrix object to use for optimization.
`numObs`	(optional) An adjustment to the total number of observations in the model.
`numStats`	(optional) An adjustment to the total number of observed statistics in the model.

Details

NOTE: THIS DESCRIPTION IS DEPRECATED. Please change to using mxFitFunctionAlgebra as shown in the example below.

Fit functions are functions for which free parameter values are chosen such that the value of the objective function is minimized. While the other fit functions in OpenMx require an expectation function for the model, the `mxAlgebraObjective` function uses the referenced `MxAlgebra` or `MxMatrix` object as the function to be minimized.

If a model's primary objective function is a `mxAlgebraObjective` objective function, then the referenced algebra in the objective function must return a 1 x 1 matrix (when using OpenMx's default optimizer). There is no restriction on the dimensions of an objective function that is not the primary, or 'topmost', objective function.

To evaluate an algebra objective function, place the following objects in a `MxModel` object: a `MxAlgebraObjective`, `MxAlgebra` and `MxMatrix` entities referenced by the `MxAlgebraObjective`, and optional `MxBounds` and `MxConstraint` entities. This model may then be evaluated using the `mxRun` function. The results of the optimization may be obtained using the `mxEval` function on the name of the `MxAlgebra`, after the model has been run.

Value

Returns a list containing a NULL MxExpectation object and an MxFitFunctionAlgebra object. MxFitFunctionAlgebra objects should be included with models with referenced `MxAlgebra` and `MxMatrix` objects.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

mxAlgebra to create an algebra suitable as a reference function to be minimized. More information about the OpenMx package may be found here.

Examples

```
# Create and fit a very simple model that adds two numbers using mxFitFunctionAlgebra

library(OpenMx)

# Create a matrix 'A' with no free parameters
A <- mxMatrix('Full', nrow = 1, ncol = 1, values = 1, name = 'A')

# Create an algebra 'B', which defines the expression A + A
B <- mxAlgebra(A + A, name = 'B')

# Define the objective function for algebra 'B'
objective <- mxFitFunctionAlgebra('B')

# Place the algebra, its associated matrix and
# its objective function in a model
tmpModel <- mxModel(model="Addition", A, B, objective)

# Evalulate the algebra
tmpModelOut <- mxRun(tmpModel)

# View the results
tmpModelOut$output$minimum
```

`mxAutoStart` *Automatically set starting values for an MxModel*

Description

Automatically set starting values for an MxModel

Usage

```
mxAutoStart(model, type = c("ULS", "DWLS"))
```

Arguments

`model`	The MxModel for which starting values are desired
`type`	The type of starting values to obtain, currently unweighted or diagonally weighted least squares, ULS or DWLS

Details

This function automatically picks very good starting values for many models (RAM, LISREL, Normal), including multiple group versions of these. It works for models with algebras. Models of continuous, ordinal, and joint ordinal-continuous variables are also acceptable. It works for models with covariance or raw data. However, it does not currently work for models with definition variables, state space models, and item factor analysis models.

The method used to obtain new starting values is quite simple. The user's model is changed to an unweighted least squares (ULS) model. The ULS model is estimated and its final point estimates are returned as the new starting values. Optionally, diagonally weighted least squares (DWLS) can be used instead with the `type` argument.

Please note that ULS is sensitive to the scales of your variables. For example, if you have variables with means of 20 and variances of 0.001, then ULS will "weight" the means 20,000 times more than the variances and might result in zero variance estimates. Likewise if one variable has a variance of 20 and another has a variance of 0.001, the same problem may arise. To avoid this, make sure your variables are scaled accordingly. You could also use `type='DWLS'` to have the function use diagonally weighted least squares to obtain starting values. Of course, using diagonally weighted least squares will take much much longer and will usually not provide better starting values than unweighted least squares.

Also note that if `model` contains a GREML expectation, argument `type` is ignored, and the function always uses a form of ULS.

Value

an MxModel with new free parameter values

Examples

```
# Use the frontpage model with negative variances to show better
# starting values
library(OpenMx)
data(demoOneFactor)

latents  = c("G") # the latent factor
manifests = names(demoOneFactor) # manifest variables to be modeled

m1 <- mxModel("One Factor", type = "RAM",
manifestVars = manifests, latentVars = latents,
mxPath(from = latents, to = manifests),
mxPath(from = manifests, arrows = 2, values=-.2),
mxPath(from = latents, arrows = 2, free = FALSE, values = 1.0),
mxPath(from = "one", to = manifests),
mxData(demoOneFactor, type = "raw")
)

# Starting values imply negative variances!
mxGetExpected(m1, 'covariance')

# Use mxAutoStart to get much better starting values
m1s <- mxAutoStart(m1)
```

```
mxGetExpected(m1s, 'covariance')
```

---

mxAvailableOptimizers *mxAvailableOptimizers*

Description

List the Optimizers available in this version, e.g. "SLSQP" "CSOLNP"

Usage

```
mxAvailableOptimizers()
```

Details

note for advanced users: Special-purpose optimizers like Newton-Raphson or EM are not included in this list.

Value

- list of valid Optimizer names

See Also

- `mxOption`(model, "Default optimizer")

Examples

```
mxAvailableOptimizers()
```

MxBaseExpectation-class *MxBaseExpectation*

Description

The virtual base class for all expectations. Expectations contain enough information to generate simulated data. This is an internal class and should not be used directly.

See Also

mxExpectationNormal, mxExpectationRAM, mxExpectationLISREL, mxExpectationStateSpace, mxExpectationBA81

`MxBaseFitFunction-class` *MxBaseFitFunction*

Description

The virtual base class for all fit functions. This is an internal class and should not be used directly.

See Also

mxFitFunctionAlgebra, mxFitFunctionML, mxFitFunctionMultigroup, mxFitFunctionR, mxFitFunctionWLS, mxFitFunctionRow, mxFitFunctionGREML

`MxBaseNamed-class` *MxBaseNamed*

Description

This is an internal class and should not be used directly. It is the base class for named entities. Fit functions, expectations, and computes contain this class.

`MxBaseObjectiveMetaData-class` *MxBaseObjectiveMetaData*

Description

This is an internal class and should not be used directly. It is the virtual base class for all objective functions meta-data

mxBootstrap *Repeatedly estimate model using resampling with replacement*

Description

Bootstrapping is used to quantify the variability of parameter estimates. A new sample is drawn from the model data (uniformly sampling the original data with replacement). The model is re-fitted to this new sample. This process is repeated many times. This yields a series of estimates from these replications which can be used to assess the variability of the parameters.

note: `mxBootstrap` only bootstraps free model parameters:

To bootstrap algebras, see `mxBootstrapEval`

To report bootstrapped standardized paths in RAM models, `mxBootstrap` the model, and then run through `mxBootstrapStdizeRAMpaths`

Usage

```
mxBootstrap(model, replications=200, ...,
                        data=NULL, plan=NULL, verbose=0L,
                        parallel=TRUE, only=as.integer(NA),
OK=mxOption(model, "Status OK"), checkHess=FALSE)
```

Arguments

`model`	The MxModel to be run.
`replications`	The number of resampling replications. If available, replications from prior mxBootstrap invocations will be reused.
`...`	Not used. Forces remaining arguments to be specified by name.
`data`	A character vector of data or model names
`plan`	Deprecated
`verbose`	For levels greater than 0, enables runtime diagnostics
`parallel`	Whether to process the replications in parallel (not yet implemented!)
`only`	When provided, only the given replication from a prior run of `mxBootstrap` will be performed. See details.
`OK`	The set of status code that are considered successful
`checkHess`	Whether to approximate the Hessian in each replication

Details

By default, all datasets in the given model are resampled independently. If resampling is desired from only some of the datasets then the models containing them can be listed in the ‘data’ parameter.

The `frequency` column in the `mxData` object is used represent a resampled dataset. When resampling, the original row proportions, as given by the original `frequency` column, are respected.

When the model has a default compute plan and ‘checkHess’ is kept at FALSE then the Hessian will not be approximated or checked. On the other hand, ‘checkHess’ is TRUE then the Hessian will be approximated by finite differences. This procedure is of some value because it can be informative to check whether the Hessian is positive definite (see `mxComputeHessianQuality`). However, approximating the Hessian is often costly in terms of CPU time. For bootstrapping, the parameter estimates derived from the resampled data are typically of primary interest.

On occasion, replications will fail. Sometimes it can be helpful to exactly reproduce a failed replication to attempt to pinpoint the cause of failure. The ‘only’ option facilitates this kind of investigation. In normal operation, mxBootstrap uses the regular R random number generator to generate a seed for each replication. This seed is used to seed an internal pseudorandom number generator (currently the Mersenne Twister algorithm). These per-replication seeds are stored as part of the bootstrap output. When ‘only’ is specified, the associated stored seed is used to seed the internal random number generator so that identical weights can be regenerated.

‘mxBootstrap’ does not currently offer special support for nested or multilevel data. Rows are assumed model-wise independent.

Value

The given model is returned with the compute plan modified to consist of `mxComputeBootstrap`. Results of the bootstrap replications are stored inside the compute plan. `mxSummary` can be used to obtain per-parameter quantiles and standard errors.

See Also

`mxBootstrapEval`, `mxComputeBootstrap`, `mxSummary`, `mxBootstrapStdizeRAMpaths`, `as.statusCode`

Examples

```
library(OpenMx)

data(multiData1)

manifests <- c("x1", "x2", "y")

biRegModelRaw <- mxModel(
  "Regression of y on x1 and x2",
  type="RAM",
  manifestVars=manifests,
  mxPath(from=c("x1","x2"), to="y",
         arrows=1,
         free=TRUE, values=.2, labels=c("b1", "b2")),
  mxPath(from=manifests,
         arrows=2,
         free=TRUE, values=.8,
         labels=c("VarX1", "VarX2", "VarE")),
  mxPath(from="x1", to="x2",
         arrows=2,
         free=TRUE, values=.2,
         labels=c("CovX1X2")),
  mxPath(from="one", to=manifests,
         arrows=1, free=TRUE, values=.1,
         labels=c("MeanX1", "MeanX2", "MeanY")),
  mxData(observed=multiData1, type="raw"))

biRegModelRawOut <- mxRun(biRegModelRaw)

boot <- mxBootstrap(biRegModelRawOut, 10)   # start with 10
summary(boot)

# Looks good, now do the rest
boot <- mxBootstrap(boot)
summary(boot)

# examine replication 3
boot3 <- mxBootstrap(boot, only=3)

print(coef(boot3))
print(boot$compute$output$raw[3,names(coef(boot3))])
```

`mxBootstrapEval` *Evaluate Values in a bootstrapped MxModel*

Description

This function can be used to evaluate an arbitrary R expression that includes named entities from a MxModel object, or labels from a MxMatrix object.

Usage

```
mxBootstrapEval(expression, model, defvar.row = 1, ...,
 bq=c(.25,.75), method=c('bcbci','quantile'))

mxBootstrapEvalByName(name, model, defvar.row = 1, ...,
 bq=c(.25,.75), method=c('bcbci','quantile'))

omxBootstrapEval(expression, model, defvar.row = 1L, ...)

omxBootstrapEvalCov(expression, model, defvar.row = 1L, ...)

omxBootstrapEvalByName(name, model, defvar.row=1L, ...)
```

Arguments

`expression`	An arbitrary R expression.
`name`	The character name of an object to evaluate.
`model`	The model in which to evaluate the expression.

`defvar.row`	The row to use for definition variables when compute=TRUE (defaults to 1). When compute=FALSE, values for definition variables are always taken from the first (i.e., first before any automated sorting is done) row of the raw data.
`...`	Not used. Forces remaining arguments to be specified by name.
`bq`	numeric. A vector of bootstrap quantiles at which to summarize the bootstrap replication.
`method`	character. One of 'quantile' or 'bcbci'.

Details

The argument 'expression' is an arbitrary R expression. Any named entities that are used within the R expression are translated into their current value from the model. Any labels from the matrices within the model are translated into their current value from the model. Finally the expression is evaluated and the result is returned. To enable debugging, the 'show' argument has been provided. The most common mistake when using this function is to include named entities in the model that are identical to R function names. For example, if a model contains a named entity named 'c', then the following mxEval call will return an error: `mxEval(c(A,B,C),model)`.

The `mxEvalByName` function is a wrapper around `mxEval` that takes a character instead of an R expression.

nb: 'bcbci' stands for 'bias-corrected bootstrap confidence interval'

The default behavior is to use the 'bcbci' `method`, due to its superior theoretical properties.

Value

`omxBootstrapEval` and `omxBootstrapEvalByName` return the raw matrix of `cvectorize`'d results. `omxBootstrapEvalCov` returns the covariance matrix of the `cvectorize`'d results. `mxBootstrapEval` and `mxBootstrapEvalByName` return the `cvectorize`'d results summarized by `method` at quantiles `bq`.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

mxAlgebra to create algebraic expressions inside your model and mxModel for the model object mxEval looks inside when evaluating. mxBootstrap to create bootstrap data.

Examples

```
library(OpenMx)
# make a unit-weighted 10-row data set of values 1 thru 10
myData = mxData(data.frame(weight=1.0, value=1:10), "raw", weight = "weight")
sum(1:10)

# Model sums data$value (sum(1:10)= 55), subtracts "A", squares the result,
# and tries to minimize this (achieved by setting A=55)
testModel = mxModel(model = "testModel1", myData,
mxMatrix(name  = "A", "Full", nrow = 1, ncol = 1, values = 1, free=TRUE),
# nb: filteredDataRow is an auto-generated matrix of
# non-missing data from the present row.
# This is placed into the "rowResults" matrix (also auto-generated)
mxAlgebra(name = "rowAlg", data.weight * filteredDataRow),
# Algebra to turn the rowResults into a single number
mxAlgebra(name = "reduceAlg", (sum(rowResults) - A)^2),
mxFitFunctionRow(
rowAlgebra    = "rowAlg",
reduceAlgebra = "reduceAlg",
dimnames      = "value"
)
# no need for an MxExpectation object when using mxFitFunctionRow
)

testModel = mxRun(testModel) # A is estimated at 55, with SE= 1
testBoot = mxBootstrap(testModel)
summary(testBoot) # A is estimated at 55, with SE= 0

# Let's compute A^2 (55^2 = 3025)
mxBootstrapEval(A^2, testBoot)
#       SE 25.0% 75.0%
# [1,]  0  3025  3025
```

`mxBootstrapStdizeRAMpaths` *Bootstrap distribution of standardized RAM path coefficients*

Description

Uses the distribution of a bootstrapped RAM model's raw parameters to create a bootstrapped estimate of its standardized path coefficients.

note: Model must have already been run through `mxBootstrap`.

Usage

```
mxBootstrapStdizeRAMpaths(model, bq= c(.25, .75),
method= c('bcbci','quantile'), returnRaw= FALSE)
```

Arguments

`model`	An MxModel that uses RAM expectation and has already been run through `mxBootstrap`.
`bq`	vector of 2 bootstrap quantiles corresponding to the lower and upper limits of the desired confidence interval.
`method`	One of 'bcbci' or 'quantile'.
`returnRaw`	Whether or not to return the raw bootstrapping results (Defaults to `FALSE`: returning a dataframe summarizing the results).

Details

`mxBootstrapStdizeRAMpaths` applies `mxStandardizeRAMpaths` to each bootstrap replication, thus creating a distribution of standardized estimates for each nonzero path coefficient.

The default bq (bootstrap quantiles) of c(.25, .75) correspond to a 50% CI. This default is chosen as many more bootstraps are required to accurately estimate more extreme quantiles. For a 95% CI, use `bq=c(.025,.0975)`.

nb: 'bcbci' stands for 'bias-corrected bootstrap confidence interval' To learn more about bcbci and quantile methods, see Efron (1982) and Efron and Tibshirani (1994).

note 1: It is possible (though unlikely) that the number of nonzero paths (elements of the `A` and `S` RAM matrices) may vary among bootstrap replications. This precludes a simple summary of the standardized paths' bootstrapping results. In this rare case, if `returnRaw=TRUE`, a raw list of bootstrapping results is returned, with a warning. Otherwise an error is thrown.

note 2: `mxBootstrapStdizeRAMpaths` ignores sub-models. To standardize bootstrapped sub-models, run it on the sub-models directly.

Value

If `returnRaw=FALSE` (default), it returns a dataframe containing, among other things, the standardized path coefficients as estimated from the real data, their bootstrap SEs, and the lower and upper limits of a bootstrap confidence interval. If `returnRaw=TRUE`, typically, a matrix containing the raw bootstrap results is returned; this matrix has one column per non-zero path coefficient, and one row for each successfully converged bootstrap replication or, if the number of paths varies between bootstraps, a raw list of results is returned.

References

Efron B. (1982). *The Jackknife, the Bootstrap, and Other Resampling Plans*. Philadelphia: Society for Industrial and Applied Mathematics.

Efron B, Tibshirani RJ. (1994). *An Introduction to the Bootstrap*. Boca Raton: Chapman & Hall/CRC.

See Also

`mxBootstrap()`, `mxStandardizeRAMpaths()`, `mxBootstrapEval`, `mxSummary`

Examples

```
require(OpenMx)
data(myFADataRaw)
manifests = c("x1","x2","x3","x4","x5","x6")

# Build and run 1-factor raw-data CFA
m1 = mxModel("CFA", type="RAM", manifestVars=manifests, latentVars="F1",
# Factor loadings
mxPath("F1", to = manifests, values=1),

# Means and variances of F1 and manifests
mxPath(from="F1", arrows=2, free=FALSE, values=1), # fix var  F1 @1
mxPath("one", to= "F1", free= FALSE, values = 0),  # fix mean F1 @0

# Freely-estimate means and residual variances of manifests
mxPath(from = manifests, arrows=2, free=TRUE, values=1),
mxPath("one", to= manifests, values = 1),

mxData(myFADataRaw, type="raw")
)
m1 = mxRun(m1)
set.seed(170505) # Desirable for reproducibility

# ==========================
# = 1. Bootstrap the model =
# ==========================

m1_booted = mxBootstrap(m1)

# ====================================================
# = 2. Estimate and accumulate a distribution of  =
# =    standardized values from each bootstrap.   =
# ====================================================

tmp = mxBootstrapStdizeRAMpaths(m1_booted)
#          name label matrix row col Std.Value    Boot.SE     25.0%     75.0%
# 1  CFA.A[1,7]    NA      A  x1  F1 0.8049842 0.01583737 0.7899938 0.8124311
# 2  CFA.A[2,7]    NA      A  x2  F1 0.7935255 0.01373320 0.7865666 0.8045558
# 3  CFA.A[3,7]    NA      A  x3  F1 0.7772050 0.01629684 0.7698374 0.7907878
# 4  CFA.A[4,7]    NA      A  x4  F1 0.8248493 0.01315534 0.8150299 0.8351416
# 5  CFA.A[5,7]    NA      A  x5  F1 0.7995083 0.01479210 0.7869158 0.8057788
# 6  CFA.A[6,7]    NA      A  x6  F1 0.8126734 0.01527586 0.8012809 0.8218805
# 7  CFA.S[1,1]    NA      S  x1  x1 0.3520004 0.02546392 0.3399556 0.3759097
# 8  CFA.S[2,2]    NA      S  x2  x2 0.3703173 0.02171159 0.3526899 0.3813130
# 9  CFA.S[3,3]    NA      S  x3  x3 0.3959524 0.02529583 0.3746547 0.4073505
# 10 CFA.S[4,4]    NA      S  x4  x4 0.3196237 0.02163979 0.3025384 0.3357263
# 11 CFA.S[5,5]    NA      S  x5  x5 0.3607865 0.02364008 0.3507206 0.3807635
# 12 CFA.S[6,6]    NA      S  x6  x6 0.3395619 0.02476480 0.3245124 0.3579489
# 13 CFA.S[7,7]    NA      S  F1  F1 1.0000000 0.00000000 1.0000000 1.0000000
# 14 CFA.M[1,1]    NA      M   1  x1 2.9950397 0.08745209 2.9368758 3.0430917
# 15 CFA.M[1,2]    NA      M   1  x2 2.9775235 0.07719970 2.9109289 3.0197492
# 16 CFA.M[1,3]    NA      M   1  x3 3.0133665 0.08645522 2.9598062 3.0779683
# 17 CFA.M[1,4]    NA      M   1  x4 3.0505604 0.08210810 2.9952130 3.1103674
# 18 CFA.M[1,5]    NA      M   1  x5 2.9776983 0.07973619 2.9362410 3.0311999
# 19 CFA.M[1,6]    NA      M   1  x6 2.9830050 0.07632118 2.9360469 3.0416504
```

mxBounds *Create MxBounds Object*

Description

This function creates a new MxBounds object.

Usage

```
mxBounds(parameters, min = NA, max = NA)
```

Arguments

`parameters`	A character vector indicating the names of the parameters on which to apply bounds.
`min`	A numeric value for the lower bound. NA means use default value.
`max`	A numeric value for the upper bound. NA means use default value.

Details

Creates a set of boundaries or limits for a parameter or set of parameters. Parameters may be any free parameter or parameters from an MxMatrix object. Parameters may be referenced either by name or by referring to their position in the 'spec' matrix of an `MxMatrix` object.

Minima and maxima may be specified as scalar numeric values.

Value

Returns a new MxBounds object. If used as an argument in an MxModel object, the parameters referenced in the 'parameters' argument must also be included prior to optimization.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

MxBounds for the S4 class created by mxBounds. MxMatrix and mxMatrix for free parameter specification. More information about the OpenMx package may be found here.

Examples

```
#Create lower and upper bounds for parameters 'A' and 'B'
bounds <- mxBounds(c('A', 'B'), 3, 5)

#Create a lower bound of zero for a set of variance parameters
varianceBounds <- mxBounds(c('Var1', 'Var2', 'Var3'), 0)
```

MxBounds-class *MxBounds Class*

Description

MxBounds is an S4 class. New instances of this class can be created using the function mxBounds.

Details

The MxBounds class has the following slots:

min	-	The lower bound
max	-	The upper bound
parameters	-	The vector of parameter names

The 'min' and 'max' slots hold scalar numeric values for the lower and upper bounds on the list of parameters, respectively.

Parameters may be any free parameter or parameters from an MxMatrix object. Parameters may be referenced either by name or by referring to their position in the 'spec' matrix of an `MxMatrix` object. To affect an estimation or optimization, an MxBounds object must be included in an MxModel object with all referenced MxAlgebra and MxMatrix objects.

Slots may be referenced with the $ symbol. See the documentation for Classes and the examples in the mxBounds document for more information.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

mxBounds for the function that creates MxBounds objects. MxMatrix and mxMatrix for free parameter specification. More information about the OpenMx package may be found here.

MxCharOrList-class *A character, list or NULL*

Description

A character, list or NULL

`MxCharOrLogical-class` *A character or logical*

Description

A character or logical

`MxCharOrNumber-class` *A character or integer*

Description

A character or integer

`mxCheckIdentification` *Check that a model is locally identified*

Description

Use the dimension of the null space of the Jacobian to determine whether or not a model is identified local to its current parameter values. The output is a list of the the identification status, the Jacobian, and which parameters are not identified.

Usage

```
mxCheckIdentification(model, details=TRUE)
```

Arguments

`model`	A MxModel object or list of MxModel objects.
`details`	logical.

Details

The mxCheckIdentification function is used to check that a model is identified. That is, the function will tell you if the model has a unique solution in parameter space. The function is most useful when applied to either (a) a model that has been run and had some NA standard errors, or (b) a model that has not been run but has reasonable starting values. In the former situation, mxCheckIdentification is used as a diagnostic after a problem was indicated. In the latter situation, mxCheckIdentification is used as a sanity check.

The method uses the Jacobian of the model expected means and the unique elements of the expected covariance matrix with respect to the free parameters. It is the first derivative of the mapping between the free parameters and the sufficient statistics for the Normal distribution. The method does not depend on data, but does depend on the current values of the free parameters. Thus, it only provides local identification, not global identification. Because the method does not depend on data, the model still could be empirically unidentified due to missing data.

The Jacobian is evaluated numerically and generally takes a few seconds, but much less than a minute.

The identification may not be accurate for models using definition variables. Currently, only the first row of the definition variable is evaluated.

When TRUE, the 'details' argument provides the names of the non-identified parameters. Otherwise, only the status and Jacobian are returned.

Value

A named list with components

status logical. TRUE if the model is locally identified; otherwise FALSE.

jacobian matrix. The numerically evaluated Jacobian.

non_identified_parameters vector. The free parameter names that are not identified

References

Bekker, P.A., Merckens, A., Wansbeek, T.J. (1994). Identification, Equivalent Models and Computer Algebra. Academic Press: Orlando, FL.

Bollen, K. A. & Bauldry, S. (2010). Model Identification and Computer Algebra. Sociological Methods & Research, 39, p. 127-156.

See Also

mxModel

Examples

```
require(OpenMx)

data(demoOneFactor)
manifests <- names(demoOneFactor)
latents <- "G1"
model2 <- mxModel(model="One Factor", type="RAM",
      manifestVars = manifests,
      latentVars = latents,
      mxPath(from = latents[1], to=manifests[1:5]),
      mxPath(from = manifests, arrows = 2, lbound=1e-6),
      mxPath(from = latents, arrows = 2, free = FALSE, values = 1.0),
      mxData(cov(demoOneFactor), type = "cov", numObs=500)
)
fit2 <- mxRun(model2)

id2 <- mxCheckIdentification(fit2)
id2$status
# The model is locally identified
# Build a model from the solution of the previous one
#  but now the factor variance is also free
model2n <- mxModel(fit2, name="Non Identified Two Factor",
      mxPath(from=latents[1], arrows=2, free=TRUE, values=1)
)

mid2 <- mxCheckIdentification(model2n)
mid2$non_identified_parameters
# The factor loadings and factor variance
#  are not identified.
```

`mxCI` *Create mxCI Object*

Description

This function creates a new MxCI object, which allows estimation of likelihood-based confidence intervals in a model (note: to estimate SEs around arbitrary objects, see `mxSE`)

Usage

```
mxCI(reference, interval = 0.95, type=c("both", "lower", "upper"), ..., boundAdj=TRUE)
```

Arguments

`reference`	A character vector of free parameters, mxMatrices, mxMatrix elements and mx-Algebras on which confidence intervals are to be estimated, listed by name.
`interval`	A scalar numeric value indicating the confidence interval to be estimated. Must be between 0 and 1. Defaults to 0.95.
`type`	A character string indicating whether the upper, lower or both confidence limits are returned. Defaults to "both".
`...`	Not used. Forces remaining arguments to be specified by name.
`boundAdj`	Whether to correct the likelihood-based confidence intervals for a lower or upper bound.

Details

The mxCI function creates MxCI objects, which can be used as arguments in MxModel objects. When models containing MxCI objects are optimized using mxRun with the ‘intervals’ argument set to TRUE, likelihood-based confidence intervals are returned. The likelihood-based confidence intervals calculated by MxCI objects are symmetric with respect to the change in likelihood in either direction, and are not necessarily symmetric around the parameter estimate. Estimation of confidence intervals requires both that an MxCI object be included in the model and that the ‘intervals’ argument of the mxRun function is set to TRUE. When estimated, confidence intervals can be accessed in the model output at `$output$confidenceIntervals` or by using summary on a fitted MxModel object.

A typical use case is when a model includes non-linear constraints, and hence, standard errors are not available. In all cases, a two-sided hypothesis test is assumed. Therefore, the upper bound will exclude 2.5% (for interval=0.95) even though only one bound is requested. To obtain a one-sided CI for a one-sided hypothesis test, interval=0.90 will obtain a 95% confidence interval.

When a confidence interval is requested for a free parameter (not an algebra) constrained by a lower bound or an upper bound (but not both) and `boundAdj=TRUE` then the Wu & Neale (2012) correction is used. This improves the accuracy of the confidence interval when the parameter is estimated close to the bound. For example, this correction will be activated when a variance with a lower bound of 10^{-6} and no upper bound that is estimated close to the bound. The sample size, or more precisely effective sample size for that particular parameter, will determine how close the variance needs to be to the bound at 10^{-6} to activate the correction.

The likelihood-based confidence intervals returned using MxCI are obtained by increasing or decreasing the value of each parameter until the -2 log likelihood of the model increases by an amount corresponding to the requested interval. The confidence limit specified by the 'interval' argument is transformed into a corresponding difference in the model -2 log likelihood based on the likelihood ratio test. Thus, a requested confidence interval for a parameter will first determine the corresponding quantile from the chi-squared distribution with one degree of freedom (a value of 3.841459 when a 95 percent confidence interval is requested). That quantile will be populated into either the 'lowerdelta' slot, the 'upperdelta' slot, or both in the output MxCI object.

Estimation of likelihood-based confidence intervals begins after optimization has been completed, with each parameter moved in the direction(s) specified in the 'type' argument until the specified increase in -2 log likelihood is reached. All other free parameters are left free for this stage of optimization. This process repeats until all confidence intervals have been calculated. The calculation of likelihood-based confidence intervals can be computationally intensive, and may add a significant amount of time to model estimation when many confidence intervals are requested.

Multiple parameters, MxMatrices and MxAlgebras may be listed in the 'reference' argument. Individual elements of MxMatrices and MxAlgebras may be listed as well, using the syntax "matrix[row,col]" (see Extract for more information). Only scalar numeric values for the 'interval' argument are supported. Users requesting different confidence ranges for different parameters must use separate mxCI statements. MxModel objects can hold multiple MxCI objects, but only one confidence interval may be requested per named-entity.

Confidence interval estimation may result in model non-convergence at the confidence limit. Separate optimizer messages may be passed for each confidence limit. This has no impact on the parameter estimates themselves, but may indicate a problem with the referenced confidence limit. Model non-convergence for a particular confidence limit may indicate parameter interdependence or the influence of a parameter boundary.

These error messages and their meanings are listed in the help for mxSummary

The validity of a confidence limit can be checked by running a model with the appropriate parameter fixed at the confidence limit in question. If the confidence limit is valid, the -2 log likelihoods of these two models should differ by the specified chi-squared criterion (as set using the 'lowerdelta' or 'upperdelta' slots in the MxCI object (you can choose which of these to set via the type parameter of mxCI).

Value

Returns a new MxCI object. If used as an argument in an MxModel object, the parameters, MxMatrices and MxAlgebras listed in the 'reference' argument must also be included prior to optimization.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`. Additional support for mxCI() can be found on the OpenMx wiki at http://openmx.ssri.psu.edu/wiki.

Neale, M. C. & Miller M. B. (1997). The use of likelihood based confidence intervals in genetic models. *Behavior Genetics, 27*(2), 113-120.

Pek, J. & Wu, H. (2015). Profile likelihood-based confidence intervals and regions for structural equation models. *Psychometrika, 80*(4), 1123-1145.

Wu, H. & Neale, M. C. (2012). Adjusted confidence intervals for a bounded parameter. *Behavior genetics, 42*(6), 886-898.

See Also

`mxSE` for computing SEs around arbitrary objects. `mxComputeConfidenceInterval` is the internal compute plan that implements the algorithm. MxMatrix and mxMatrix for free parameter specification. `MxCI` for the S4 class created by mxCI. More information about the OpenMx package may be found here.

Examples

```
library(OpenMx)

# generate data
covariance <- matrix(c(1.0, 0.5, 0.5, 1.0),
    nrow=2,
    dimnames=list(c("a", "b"), c("a", "b")))

data <- mxData(covariance, "cov", numObs=100)

# create an expected covariance matrix
expect <- mxMatrix("Symm", 2, 2,
    free=TRUE,
    values=c(1, .5, 1),
    labels=c("var1", "cov12", "var2"),
    name="expectedCov")

# request 95 percent confidence intervals
ci <- mxCI(c("var1", "cov12", "var2"))

# specify the model
model <- mxModel(model="Confidence Interval Example",
    data, expect, ci,
    mxExpectationNormal("expectedCov", dimnames=c("a", "b")),
    mxFitFunctionML())

# run the model
```

```
results <- mxRun(model, intervals=TRUE)

# view confidence intervals
print(summary(results)$CI)

# view all results
summary(results)

# remove a specific mxCI from a model
model <- mxModel(model, remove=TRUE, model$intervals[['cov12']])
model$intervals

# remove all mxCI from a model
model <- mxModel(model, remove=TRUE, model$intervals)
model$intervals
```

MxCI-class *MxCI Class*

Description

MxCI is an S4 class. An MxCI object is a named entity. New instances of this class can be created using the function mxCI. MxCI objects may be used as arguments in the mxModel function.

Details

The MxCI class has the following slots:

reference	-	The name of the object
lowerdelta	-	Either a matrix or a data frame
upperdelta	-	A vector for means, or NA if missing

The reference slot contains a character vector of named free parameters, MxMatrices and MxAlgebras on which confidence intervals are desired. Individual elements of MxMatrices and MxAlgebras may be listed as well, using the syntax “matrix[row,col]” (see Extract for more information).

The lowerdelta and upperdelta slots give the changes in likelihoods used to define the confidence interval. The upper bound of the likelihood-based confidence interval is estimated by increasing the parameter estimate, leaving all other parameters free, until the model -2 log likelihood increased by ‘upperdelta’. The lower bound of the confidence interval is estimated by decreasing the parameter estimate, leaving all other parameters free, until the model -2 log likelihood increased by ‘lowerdata’.

Likelihood-based confidence intervals may be specified by including one or more MxCI objects in an MxModel object. Estimation of confidence intervals requires model optimization using the mxRun function with the ‘intervals’ argument set to TRUE. The calculation of likelihood-based confidence intervals can be computationally intensive, and may add a significant amount of time to model estimation when many confidence intervals are requested.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

mxCI for creating MxCI objects. More information about the OpenMx package may be found here.

---

mxCompare *Likelihood ratio test*

Description

Compare the fit of one or more models to that of a reference (base) model or set of reference models.

Usage

```
mxCompare(base, comparison, ..., all = FALSE,
  boot=FALSE, replications=400, previousRun=NULL, checkHess=FALSE)
mxCompareMatrix(models,
    diag=c('minus2LL','ep','df','AIC'),
    stat=c('p', 'diffLL','diffdf'), ...,
  boot=FALSE, replications=400, previousRun=NULL,
 checkHess=FALSE, wholeTable=FALSE)
```

Arguments

`base`	A MxModel object or list of MxModel objects.
`comparison`	A MxModel object or list of MxModel objects.
`models`	A MxModel object or list of MxModel objects.
`diag`	statistic used for diagonal entries
`stat`	statistic used for off-diagonal entries
`...`	Not used.
`all`	Boolean. Whether to compare all base models with all comparison models. Defaults to FALSE.
`boot`	Whether to use the bootstrap distribution to compute the p-value.
`replications`	How many replications to use to approximate the bootstrap distribution.
`previousRun`	Results to re-use from a previous bootstrap.
`checkHess`	Whether to approximate the Hessian in each replication
`wholeTable`	Return the whole table instead of a matrix shaped summary

Details

`mxCompare` is used to compare the fit of one or more mxModels to one or more comparison models. `mxCompareMatrix` compares all the models provided against each other.

Model comparisons are made by subtracting the fit statistics for the comparison model from the fit statistics for the base model. Raw fit statistics of each 'base' model are also listed in the output table.

The fit statistics compared depend on the kinds of models compared. Models fit with maximum likelihood are compared based on their minus two log likelihood values. Under certain regularity conditions, the difference in minus two log likelihood values from nested models is chi-squared distributed and forms a likelihood ratio test statistic. Models fit with weighted least squares are compared based on their Satorra-Bentler (2001) scaled difference chi-squared test statistics. Under full weighted least squares, the Satorra-Bentler chi-squared value is equal to the difference in the model chi-squared values; however, for unweighted and diagonally weighted least squares, the two are no longer equal. Satorra and Bentler (2001) showed that that their test statistic behaved well under a variety of conditions, including small sample sizes. By contrast the much simpler difference in the chi-squared statistics only behaved well under large sample sizes (e.g., greater than or equal to 300 rows of data).

Specific to weighted least squares, researchers sometimes use mean-adjusted chi-squared statistics and mean-and-variance scaled chi-squared statistics. Some programs call these WLSM and WLSMV statistics. In some cases, it is fine to evaluate the total fit of a model using adjusted and scaled chi-squared statistics. However, never, ever, ever, ..., ever take differences in mean-adjusted chi-squared statistics, and use them for nested model comparisons. Similarly, never, ever, ever, ..., ever, ever take differences in mean-and-variance scaled chi-squared statistics, and use them for nested model comparisons. The differences in these adjusted and scaled chi-squared statistics are not chi-squared distributed and do not form a valid basis for model comparison. So, just don't do it.

Although not always checked by `mxCompare`, you should never compare models with different data sets or that use different variables from the same data set. `mxCompare` might not stop you from doing this, so be thoughtful when comparing models. Make sure your models are nested and use the same data. Weighted least squares models are one case of comparing different data sets that requires particular care. *When comparing WLS models, make sure you are using the same exogenous covariates for all compared models.* Because WLS is a multi-stage estimation approach, exogenous covariates residualize and change the data fitted in WLS. Consequently, WLS models with different exogenous covariates actually have different data. By contrast, maximum likelihood models with different exogenous covariates still use the same data and are valid to compare.

The `mxCompare` function makes an effort to only make valid comparisons. If a comparison is made where the `comparison` model has a higher minus 2 log likelihood (-2LL) than the `base` model, then the difference in their -2LLs will be negative. P-values for likelihood ratio tests will not be reported when either the -2LL or degrees of freedom for the comparison are negative. To ensure that the differences between models are positive and yield p-values for likelihood ratio tests, models listed in the 'base' argument must be more saturated (i.e., more estimated parameters and fewer degrees of freedom) than models listed in the 'comparison' argument. For `mxCompareMatrix` only the comparisons that make sense will be included.

When multiple models are included in both the 'base' and 'comparison' arguments, then comparisons are made between the two lists of models based on the value of the 'all' argument. If 'all' is set to FALSE (default), then the first model in the 'base' list is compared to the first model in the 'comparison' list, second with second, and so on. If there are an unequal number of 'base' and 'comparison' models, then the shorter list of models is repeated to match the length of the longer list. For example, comparing base models 'B1' and 'B2' with comparison models 'C1', 'C2' and 'C3' will yield three comparisons: 'B1' with 'C1', 'B2' with 'C2', and 'B1' with 'C3'. Each of those comparisons are prefaced by a comparison between the base model and a missing comparison model to present the fit of the base model.

If 'all' is set to TRUE, all possible comparisons between base and comparison models are made, and one entry is made for each base model. All comparisons involving the first model in 'base' are made first, followed by all comparisons with the second 'base' model, and so on. When there are multiple models in either the 'base' or 'comparison' arguments but not both, then the 'all' argument does not affect the set of comparisons made.

The following columns appear in the output for maximum likelihood comparisons:

base Name of the base model.

comparison Name of the comparison model. Is <NA> for the first

ep Estimated parameters of the comparison model.

minus2LL Minus 2*log-likelihood of the comparison model. If the comparison model is <NA>, then the minus 2*log-likelihood of the base model is given.

df Degrees in freedom of the comparison model. If the comparison model is <NA>, then the degrees of freedom of the base model is given.

AIC Akaike's Information Criterion for the comparison model. If the comparison model is <NA>, then the AIC of the base model is given.

diffLL Difference in minus 2*log-likelihoods of the base and comparison models. Will be positive when base model -2LL is higher than comparison model -2LL.

diffdf Difference in degrees of freedoms of the base and comparison models. Will be positive when base model DF is lower than comparison model DF (base model estimated parameters is higher than comparison model estimated parameters)

p P-value for likelihood ratio test based on diffLL and diffdf values.

Weighted least squares reports a similar set of columns with four substitutions:

chisq Replaces the `minus2LL` column. This is the comparison model's chi-squared statistic from Browne (1984, Equation 2.20a), accounting for some misspecification of the weight matrix.

AIC Although this has the same name as that in maximum likelihood, it is really a pseudo-AIC using the comparison model chi-squared and the number of estimated parameters. It is the chi-squared value plus two times the number of free parameters.

SBchisq Replaces the `diffLL` column. This is the Satorra-Bentler (2001, p. 511) scaled difference chi-squared statisic between the base model and the comparison model. If your models use full weighted least squares, then this will be the same as the difference between the individual model chi-squared statistics. However, for unweighted and diagonally weighted least square, the `SB chisq` will not be equal to the difference between the component model chi-squared statistics.

p p-value for the Satorra-Bentler chi-squared statistic.

In addition to the particular columns for maximum likelihood and weighted least squares, there are three general columns that are not printed but are accessible via the `$` and `[` extractors.

fit The individual model fit value: `m2ll` for maximum likelihood models, `chisq` for WLS models.

fitUnits The units of the fit function: `"-2LL"` for ML models, `"r'Wr"` for WLS models.

diffFit The difference in fit values between the base and comparison models: `diffLL` for ML models, `SBchisq` for WLS models.

`mxCompare` will give a p-value for any comparison in which both 'diffLL' and 'diffdf' are non-negative. However, this p-value is based on the assumptions of the likelihood ratio test, specifically that the two models being compared are nested. The likelihood ratio test and associated p-values are not valid when the comparison model is not nested in the referenced base model. For a more accurate p-value, the empirical bootstrap distribution can be computed ('boot=TRUE'). However, 'replications' must be set high enough for an accurate approximation. The Monte Carlo SE of a proportion for B replications is $\sqrt{(p*(1-p)/B)}$, but this will be zero if p is zero, which is nonsense. Note that a parametric-bootstrap p-value of zero must be interpreted as $p < 1/B$, which, depending on B and the desired Type I error rate, may not be "statistically significant."

When 'boot=TRUE', the model has a default compute plan, and 'checkHess' is kept at FALSE then the Hessian will not be approximated or checked. On the other hand, 'checkHess' is TRUE then the Hessian will be approximated by finite differences. This procedure is of some value because it can be informative to check whether the Hessian is positive definite (see `mxComputeHessianQuality`). However, approximating the Hessian is often costly in terms of CPU time. For bootstrapping, the parameter estimates derived from the resampled data are typically of primary interest.

note: The mxCompare function does not directly accept a digits argument, and depends on the value of the 'digits' option. To set the minimum number of significant digits printed, use options('digits' = N) (see example).

Value

Returns a new `MxCompare` object. If you want something more like a table of results, use `as.data.frame()` on the returned `MxCompare` object.

See Also

`mxPowerSearch`; `mxModel`; `options` (use options('mxOptions') to see all the OpenMx-specific options)

Examples

```
data(demoOneFactor)
manifests <- names(demoOneFactor)
latents <- "G1"
model1 <- mxModel(model="One Factor", type="RAM",
      manifestVars = manifests,
      latentVars = latents,
      mxPath(from = latents, to=manifests),
      mxPath(from = manifests, arrows = 2),
      mxPath(from = latents, arrows = 2, free = FALSE, values = 1.0),
      mxData(cov(demoOneFactor), type = "cov", numObs = 500)
)

fit1 <- mxRun(model1)
```

```
latents <- c("G1", "G2")
model2 <- mxModel(model="One factor Rasch equated", type="RAM",
      manifestVars = manifests,
      latentVars = latents,
      mxPath(from = latents[1], to=manifests[1:5], labels='raschEquated'),

      mxPath(from = manifests, arrows = 2),
      mxPath(from = latents, arrows = 2, free = FALSE, values = 1.0),
      mxData(cov(demoOneFactor), type = "cov", numObs=500)
)
fit2 <- mxRun(model2)

mxCompare(fit1, fit2) # Rasch equated is significantly worse

# Vary precision (rounding) of the table
oldPrecision = as.numeric(options('digits'))
options('digits' = 1)
mxCompare(fit1, fit2)
options('digits' = oldPrecision)
```

`MxCompare-class` *The MxCompare Class*

Description

The MxCompare Class

Details

This is an internal class structure. You should not use it directly. Use `mxCompare` instead.

`MxCompute-class` *MxCompute*

Description

This is an internal class and should not be used directly.

mxComputeBootstrap *Repeatedly estimate model using resampling with replacement*

Description

This is a low-level compute plan object to perform resampling with replacement.

Usage

```
mxComputeBootstrap(data, plan, replications=200, ...,
                        verbose=0L, parallel=TRUE, freeSet=NA_character_,
OK=c("OK", "OK/green"), only=NA_integer_)
```

Arguments

`data`	A vector of dataset or model names.
`plan`	The compute plan used to optimize the model for each data set.
`replications`	The number of resampling replications. If available, replications from prior mxBootstrap invocations will be reused.
`...`	Not used. Forces remaining arguments to be specified by name.
`verbose`	For levels greater than 0, enables runtime diagnostics
`parallel`	Whether to process the replications in parallel
`freeSet`	names of matrices containing free variables
`OK`	The set of status code that are considered successful
`only`	When provided, only the given replication from a prior run of `mxBootstrap` will be performed. See details.

Details

The ‘only’ option facilitates investigation of a single replication attempt.

Value

Output is stored in the compute object’s `output` slot. Specifically, `model$compute$output$raw` contains a data frame with parameters in columns and replications in rows. In addition to parameters, the `seed`, `fit`, and `statusCode` of the replication is also included.

When ‘only’ is set to a particular replications, the weight vectors (one per dataset) are also returned in the compute object’s `output` slot. `model$compute$output$weight` is a character vector (by dataset name) of numeric vectors (the weights). These weights can be used to recreate a model identical to the model used in the given replication.

See Also

`mxBootstrap`, `as.statusCode`

mxComputeCheckpoint *Log parameters and state to disk or memory*

Description

Captures the current state of the backend. When `path` is set, the state is written to disk in a single row. When `toReturn` is set, the state is recorded in memory and returned after `mxRun`.

Usage

```
mxComputeCheckpoint(
  what = NULL,
  ...,
  path = NULL,
  append = FALSE,
  header = TRUE,
  toReturn = FALSE,
  parameters = TRUE,
  loopIndices = TRUE,
  fit = TRUE,
  counters = TRUE,
  status = TRUE,
  standardErrors = FALSE,
  gradient = FALSE,
  vcov = FALSE,
  vcovFilter = c(),
  sampleSize = FALSE,
  vcovWLS = FALSE,
  useVcovFilter = FALSE
)
```

Arguments

`what`	a character vector of algebra names to include in each checkpoint
`...`	Not used. Forces remaining arguments to be specified by name
`path`	a character vector of where to write the checkpoint file
`append`	if FALSE, truncates the checkpoint file upon open. If TRUE, existing data is preserved and checkpoints are appended.
`header`	whether to write the header that describes the content of each column
`toReturn`	logical. Whether to store the checkpoint in memory and return it after the model is run
`parameters`	logical. Whether to include the parameter vector
`loopIndices`	logical. Whether to include the loop indices
`fit`	logical. Whether to include the fit value

`counters`	logical. Whether to include counters (number of evaluations and iterations)
`status`	logical. Whether to include the status code
`standardErrors`	logical. Whether to include the standard errors
`gradient`	logical. Whether to include the gradients
`vcov`	logical. Whether to include the vcov in half-vectorized order
`vcovFilter`	character vector. Vector of parameters indicating which parameter covariances to include. Only the variance is included for those parameters not mentioned.
`sampleSize`	logical. Whether to include the sample size of the mxData. **[Experimental]**
`vcovWLS`	logical. Whether to include the vcov from WLS residualizing regressions in half-vectorized order
`useVcovFilter`	logical. Whether to use the vcovFilter (TRUE) or include all entries (FALSE)

See Also

`mxComputeLoadData`, `mxComputeLoadMatrix`, `mxComputeLoadContext`, `mxComputeLoop`

Other model state: `mxRestore()`, `mxSave()`

Examples

```
library(OpenMx)

m1 <- mxModel(
  "poly22", # Eqn 22 from Tsallis & Stariolo (1996)
  mxMatrix(type='Full', values=runif(4, min=-1e6, max=1e6),
           ncol=1, nrow=4, free=TRUE, name='x'),
  mxAlgebra(sum((x*x-8)^2) + 5*sum(x) + 57.3276, name="fit"),
  mxFitFunctionAlgebra('fit'))

plan <- mxComputeLoop(list(
  mxComputeSetOriginalStarts(),
    mxComputeSimAnnealing(method="tsallis1996",
                          control=list(tempEnd=1)),
    mxComputeCheckpoint(path = "result.log")),
  i=1:4)

m1 <- mxRun(mxModel(m1, plan)) # see the file 'result.log'
```

---

mxComputeConfidenceInterval *Find likelihood-based confidence intervals*

Description

There are various equivalent ways to pose the optimization problems required to estimate confidence intervals. Most accurate solutions are achieved when the problem is posed using non-linear constraints. However, the available optimizers (CSOLNP, SLSQP, and NPSOL) often have difficulty with non-linear constraints.

Usage

```
mxComputeConfidenceInterval(
  plan,
  ...,
  freeSet = NA_character_,
  verbose = 0L,
  engine = NULL,
  fitfunction = "fitfunction",
  tolerance = NA_real_,
  constraintType = "none"
)
```

Arguments

`plan`	compute plan to optimize the model
`...`	Not used. Forces remaining arguments to be specified by name.
`freeSet`	names of matrices containing free variables
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable
`engine`	deprecated
`fitfunction`	the name of the deviance function
`tolerance`	deprecated
`constraintType`	one of c('ineq', 'none')

References

Neale, M. C. & Miller M. B. (1997). The use of likelihood based confidence intervals in genetic models. *Behavior Genetics, 27*(2), 113-120.

Pek, J. & Wu, H. (2015). Profile likelihood-based confidence intervals and regions for structural equation models. *Psychometrika, 80*(4), 1123-1145.

Wu, H. & Neale, M. C. (2012). Adjusted confidence intervals for a bounded parameter. *Behavior genetics, 42*(6), 886-898.

`mxComputeDefault` *Default compute plan*

Description

This is an empty placeholder for the default compute plan. To create an actual plan, use omxDefaultComputePlan.

Usage

```
mxComputeDefault(freeSet = NA_character_)
```

Arguments

`freeSet`	names of matrices containing free variables

`mxComputeEM` *Fit a model using DLR's (1977) Expectation-Maximization (EM) algorithm*

Description

The EM algorithm constitutes the following steps: Start with an initial parameter vector. Predict the missing data to form a completed data model. Optimize the completed data model to obtain a new parameter vector. Repeat these steps until convergence criteria are met.

Usage

```
mxComputeEM(
  expectation = NULL,
  predict = NA_character_,
  mstep,
  observedFit = "fitfunction",
  ...,
  maxIter = 500L,
  tolerance = 1e-09,
  verbose = 0L,
  freeSet = NA_character_,
  accel = "varadhan2008",
  information = NA_character_,
  infoArgs = list(),
  estep = NULL
)
```

Arguments

`expectation`	a vector of expectation names **[Deprecated]**
`predict`	what to predict from the observed data **[Deprecated]**
`mstep`	a compute plan to optimize the completed data model
`observedFit`	the name of the observed data fit function (defaults to "fitfunction")
`...`	Not used. Forces remaining arguments to be specified by name.
`maxIter`	maximum number of iterations
`tolerance`	optimization is considered converged when the maximum relative change in fit is less than tolerance
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable
`freeSet`	names of matrices containing free variables
`accel`	name of acceleration method ("varadhan2008" or "ramsay1975")

`information`	name of information matrix approximation method
`infoArgs`	arguments to control the information matrix method
`estep`	a compute plan to perform the expectation step

Details

The arguments to this function have evolved. The old style `mxComputeEM(e,p,mstep=m)` is equivalent to the new style `mxComputeEM(estep=mxComputeOnce(e,p),mstep=m)`. This change allows the API to more closely match the literature on the E-M method. You might use `mxAlgebra(...,recompute='onDemand')` to contain the results of the E-step and then cause this algebra to be recomputed using `mxComputeOnce`.

This compute plan does not work with any and all expectations. It requires a special kind of expectation that can predict its missing data to create a completed data model.

The EM algorithm does not produce a parameter covariance matrix for standard errors. The Oakes (1999) direct method and S-EM, an implementation of Meng & Rubin (1991), are included.

Ramsay (1975) was recommended in Bock, Gibbons, & Muraki (1988).

References

Bock, R. D., Gibbons, R., & Muraki, E. (1988). Full-information item factor analysis. *Applied Psychological Measurement, 6*(4), 431-444.

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 1-38.

Meng, X.-L. & Rubin, D. B. (1991). Using EM to obtain asymptotic variance-covariance matrices: The SEM algorithm. *Journal of the American Statistical Association, 86* (416), 899-909.

Oakes, D. (1999). Direct calculation of the information matrix via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Statistical Methodology), 61*(2), 479-482.

Ramsay, J. O. (1975). Solving implicit equations in psychometric data analysis. *Psychometrika, 40* (3), 337-360.

Varadhan, R. & Roland, C. (2008). Simple and globally convergent methods for accelerating the convergence of any EM algorithm. *Scandinavian Journal of Statistics, 35*, 335-353.

See Also

MxAlgebra, mxComputeOnce

Examples

```
library(OpenMx)
set.seed(190127)

N <- 200
x <- matrix(c(rnorm(N/2,0,1),
              rnorm(N/2,3,1)),ncol=1,dimnames=list(NULL,"x"))
data4mx <- mxData(observed=x,type="raw")

class1 <- mxModel("Class1",
mxMatrix(type="Full",nrow=1,ncol=1,free=TRUE,values=0,name="Mu"),
mxMatrix(type="Full",nrow=1,ncol=1,free=TRUE,values=4,name="Sigma"),
mxExpectationNormal(covariance="Sigma",means="Mu",dimnames="x"),
mxFitFunctionML(vector=TRUE))

class2 <- mxRename(class1, "Class2")

mm <- mxModel(
"Mixture", data4mx, class1, class2,
mxAlgebra((1-Posteriors) * Class1.fitfunction, name="PL1"),
mxAlgebra(Posteriors * Class2.fitfunction, name="PL2"),
mxAlgebra(PL1 + PL2, name="PL"),
mxAlgebra(PL2 / PL,  recompute='onDemand',
          initial=matrix(runif(N,.4,.6), nrow=N, ncol = 1), name="Posteriors"),
mxAlgebra(-2*sum(log(PL)), name="FF"),
mxFitFunctionAlgebra(algebra="FF"),
mxComputeEM(
  estep=mxComputeOnce("Mixture.Posteriors"),
  mstep=mxComputeGradientDescent(fitfunction="Mixture.fitfunction")))

mm <- mxOption(mm, "Max minutes", 1/20)  # remove this line to find optimum
mmfit <- mxRun(mm)
summary(mmfit)
```

`mxComputeGenerateData` *Generate data*

Description

Generate data specified by the model expectations.

Usage

```
mxComputeGenerateData(expectation = "expectation")
```

Arguments

`expectation`	a character vector of expectations to generate data for

`mxComputeGradientDescent`

Optimize parameters using a gradient descent optimizer

Description

This optimizer does not require analytic derivatives of the fit function. The fully open-source CRAN version of OpenMx offers 2 choices, CSOLNP and SLSQP (from the NLOPT collection). The OpenMx Team's version of OpenMx offers the choice of three optimizers: CSOLNP, SLSQP, and NPSOL.

Usage

```
mxComputeGradientDescent(
  freeSet = NA_character_,
  ...,
  engine = NULL,
  fitfunction = "fitfunction",
  verbose = 0L,
  tolerance = NA_real_,
  useGradient = deprecated(),
  warmStart = NULL,
  nudgeZeroStarts = mxOption(NULL, "Nudge zero starts"),
  maxMajorIter = NULL,
  gradientAlgo = deprecated(),
  gradientIterations = deprecated(),
  gradientStepSize = deprecated()
)
```

Arguments

`freeSet`	names of matrices containing free parameters.
`...`	Not used. Forces remaining arguments to be specified by name.
`engine`	specific 'CSOLNP', 'SLSQP', or 'NPSOL'
`fitfunction`	name of the fitfunction (defaults to 'fitfunction')
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable
`tolerance`	how close to the optimum is close enough (also known as the optimality tolerance)
`useGradient`	**[Soft-deprecated]**
`warmStart`	a Cholesky factored Hessian to use as the NPSOL Hessian starting value (preconditioner)
`nudgeZeroStarts`	whether to nudge any zero starting values prior to optimization (default TRUE)
`maxMajorIter`	maximum number of major iterations
`gradientAlgo`	**[Soft-deprecated]**
`gradientIterations`	**[Soft-deprecated]**
`gradientStepSize`	**[Soft-deprecated]**

Details

All three optimizers can use analytic gradients, and only NPSOL uses `warmStart`. To customize more options, see mxOption.

References

Luenberger, D. G. & Ye, Y. (2008). *Linear and nonlinear programming.* Springer.

Examples

```
data(demoOneFactor)
factorModel <- mxModel(name ="One Factor",
  mxMatrix(type="Full", nrow=5, ncol=1, free=FALSE, values=0.2, name="A"),
    mxMatrix(type="Symm", nrow=1, ncol=1, free=FALSE, values=1, name="L"),
    mxMatrix(type="Diag", nrow=5, ncol=5, free=TRUE, values=1, name="U"),
    mxAlgebra(expression=A %*% L %*% t(A) + U, name="R"),
  mxExpectationNormal(covariance="R", dimnames=names(demoOneFactor)),
  mxFitFunctionML(),
    mxData(observed=cov(demoOneFactor), type="cov", numObs=500),
     mxComputeSequence(steps=list(
     mxComputeGradientDescent(),
     mxComputeNumericDeriv(),
     mxComputeStandardError(),
     mxComputeHessianQuality()
    )))
factorModelFit <- mxRun(factorModel)
factorModelFit$output$conditionNumber # 29.5
```

---

mxComputeHessianQuality
Compute the quality of the Hessian

Description

Tests whether the Hessian is positive definite (model$output$infoDefinite) and, if so, computes the approximate condition number (model$output$conditionNumber). See Luenberger & Ye (2008) Second Order Test (p. 190) and Condition Number (p. 239).

Usage

```
mxComputeHessianQuality(freeSet = NA_character_, ..., verbose = 0L)
```

Arguments

freeSet	names of matrices containing free variables
...	Not used. Forces remaining arguments to be specified by name.
verbose	integer. Level of run-time diagnostic output. Set to zero to disable

Details

The condition number is approximated by $\mathrm{norm}(H) * \mathrm{norm}(H^{-1})$ where H is the Hessian. The norm is either the 1- or infinity-norm (both obtain the same result due to symmetry).

References

Luenberger, D. G. & Ye, Y. (2008). Linear and nonlinear programming. Springer.

mxComputeIterate *Repeatedly invoke a series of compute objects until change is less than tolerance*

Description

One step (typically the last) must compute the fit or maxAbsChange.

Usage

```
mxComputeIterate(
  steps,
  ...,
  maxIter = 500L,
  tolerance = 1e-09,
  verbose = 0L,
  freeSet = NA_character_,
  maxDuration = as.numeric(NA)
)
```

Arguments

`steps`	a list of compute objects
`...`	Not used. Forces remaining arguments to be specified by name.
`maxIter`	the maximum number of iterations
`tolerance`	iterates until maximum relative change is less than tolerance
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable
`freeSet`	Names of matrices containing free variables.
`maxDuration`	the maximum amount of time (in seconds) to iterate

mxComputeJacobian *Numerically estimate the Jacobian with respect to free parameters*

Description

When algebra names are given, all algebras must belong to the same model.

When expectations are given, the Jacobian is taken with respect to the manifest model. The manifest model excludes any latent variables or processes. For RAM and LISREL models, the manifest model contains only the manifest variables with free means, covariance, and thresholds. Ordinal manifest variables are standardized.

Usage

```
mxComputeJacobian(freeSet=NA_character_, ..., of = "expectation",
 defvar.row=as.integer(NA), data='data')
```

Arguments

`freeSet`	names of matrices containing free variables
`...`	Not used. Forces remaining arguments to be specified by name.
`of`	a character vector of expectations or algebra names
`defvar.row`	A row index. Which row to load for definition variables.
`data`	From which data to load definition variables.

See Also

omxManifestModelByParameterJacobian, mxGetExpected

`mxComputeLoadContext` *Load contextual data to supplement checkpoint*

Description

[Experimental]

Usage

```
mxComputeLoadContext(
  method = c("csv"),
  path = c(),
  column,
  ...,
  sep = " ",
  verbose = 0L,
  header = TRUE,
  col.names = NULL
)
```

Arguments

`method`	name of the conduit used to load the columns.
`path`	the path to the file containing the data
`column`	a character vector. The column names to log.
`...`	Not used. Forces remaining arguments to be specified by name.
`sep`	the field separator character. Values on each line of the file are separated by this character.
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable
`header`	logical. Whether the first row contains column headers.
`col.names`	character vector. Column names

Details

Currently, this only supports comma separated value format and no row names. If `header=TRUE` and `col.names` are provided, the `col.names` take precedence. If `header=FALSE` and no `col.names` are provided then the column names consist of the file name and column offset.

An `originalDataIsIndexOne` option is not offered. You'll need to add an extra line at the start on your file if you wish to make use of `originalDataIsIndexOne` in `mxComputeLoad*`.

See Also

mxComputeCheckpoint, mxComputeLoadData, mxComputeLoadMatrix

`mxComputeLoadData` *Load columns into an MxData object*

Description

[Experimental]

Usage

```
mxComputeLoadData(
  dest,
  column,
  method = c("csv", "data.frame"),
  ...,
  path = c(),
  originalDataIsIndexOne = FALSE,
  byrow = TRUE,
  row.names = c(),
  col.names = c(),
  skip.rows = 0,
  skip.cols = 0,
  verbose = 0L,
  cacheSize = 100L,
  checkpointMetadata = TRUE,
  na.strings = c("NA"),
  observed = NULL,
  rowFilter = c()
)
```

Arguments

`dest`	the name of the model where the columns will be loaded
`column`	a character vector. The column names to replace.
`method`	name of the conduit used to load the columns.
`...`	Not used. Forces remaining arguments to be specified by name.

`path`	the path to the file containing the data
`originalDataIsIndexOne`	logical. Whether to use the initial data for index 1
`byrow`	logical. Whether the data columns are stored in rows.
`row.names`	optional integer. Column containing the row names.
`col.names`	optional integer. Row containing the column names.
`skip.rows`	integer. Number of rows to skip before reading data.
`skip.cols`	integer. Number of columns to skip before reading data.
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable
`cacheSize`	integer. How many columns to cache per scan through the data. Only used when byrow=FALSE.
`checkpointMetadata`	logical. Whether to add per record metadata to the checkpoint
`na.strings`	character vector. A vector of strings that denote a missing value.
`observed`	data frame. The reservoir of data for `method='data.frame'`.
`rowFilter`	logical vector. Whether to skip the source row.

Details

The purpose of this compute step is to help quickly perform many similar analyses. For example, if we are given a sample of people with a few million SNPs (single-nucleotide polymorphism) per person then we could fit a separate model for each SNP by iterating over the SNP data.

The column names given in the `column` parameter must already exist in the model's MxData object. Pre-existing data is assumed to be a placeholder and is not used unless `originalDataIsIndexOne` is set to TRUE.

For `method='csv'`, the highest performance arrangement is `byrow=TRUE` because entire columns are stored in single chunks (rows) on the disk and can be easily loaded. For `byrow=FALSE`, the data requires transposition. To load a single column of observed data, it is necessary to read through the whole file. This can be slow for large files. To amortize the cost of transposition, `cacheSize` columns are loaded on every pass through the file.

After `mxRun` returns, the `dest` mxData object will contain the most recently loaded data. Hence, any single analysis of a series can be reproduced by issuing `mxComputeLoadData` with the single index associated with a particular dataset, replacing the compute plan with something like `omxDefaultComputePlan`, and then passing the model back through `mxRun`. This can be a helpful approach when investigating unexpected results.

See Also

mxComputeLoadMatrix, mxComputeCheckpoint, mxRun, omxDefaultComputePlan

mxComputeLoadMatrix *Load data from CSV files directly into the backend*

Description

THIS INTERFACE IS EXPERIMENTAL AND SUBJECT TO CHANGE.

For method='csv', the file must be formatted in a specific way. The number of columns must match the number of entries available in the mxMatrix. Matrix types (e.g., symmetric or diagonal) are respected (see mxMatrix). For example, a *Full* 2x2 matrix will require 4 entries, but a diagonal matrix of the same size will only require 2 entries. CSV data must be stored space separated and without row or column names. The destination `mxMatrix` can have free parameters, but cannot have square bracket populated entries.

If `originalDataIsIndexOne` is TRUE then this compute step does nothing when the loop index is 1. The purpose of `originalDataIsIndexOne` is to permit usage of the dataset that was initially included with the model.

Usage

```
mxComputeLoadMatrix(dest, method=c('csv','data.frame'), ...,
 path=NULL, originalDataIsIndexOne=FALSE,
 row.names=FALSE, col.names=FALSE, observed=NULL)
```

Arguments

`dest`	a character vector of matrix names
`method`	name of the conduit used to load the data.
`...`	Not used. Forces remaining arguments to be specified by name.
`path`	a character vector of paths
`originalDataIsIndexOne`	logical. Whether to use the initial data for index 1
`row.names`	logical. Whether row names are present
`col.names`	logical. Whether column names are present
`observed`	data frame. The reservoir of data for `method='data.frame'`

See Also

mxComputeLoadData, mxComputeCheckpoint

Examples

```
library(OpenMx)

dir <-tempdir()  # safe place to create files

Cov <- rWishart(4, 20, toeplitz(c(2,1)/20))
```

```
write.table(t(apply(Cov, 3, vech)),
            file=file.path(dir, "cov.csv"),
            col.names=FALSE, row.names=FALSE)
Mean <- matrix(rnorm(8),4,2)
write.table(Mean, file=file.path(dir, "mean.csv"),
            col.names=FALSE, row.names=FALSE)

m1 <- mxModel(
  "test1",
  mxMatrix("Full", 1,2, values=0,       name="mean"),
  mxMatrix("Symm", 2,2, values=diag(2), name="cov"),
  mxMatrix("Full", 1,2, values=-1,      name="lbound"),
  mxMatrix("Full", 1,2, values=1,       name="ubound"),
  mxAlgebra(omxMnor(cov,mean,lbound,ubound), name="area"),
  mxFitFunctionAlgebra("area"),
  mxComputeLoop(list(
    mxComputeLoadMatrix(c('mean', 'cov'),
                        path=file.path(dir, c('mean.csv', 'cov.csv'))),
    mxComputeOnce('fitfunction', 'fit'),
    mxComputeCheckpoint(path=file.path(dir, "loadMatrix.csv"))
  ), i=1:4))

m1 <- mxRun(m1)
```

`mxComputeLoop` *Repeatedly invoke a series of compute objects*

Description

When `i` is given then these values are iterated over instead of the sequence 1 to the number of iterations.

Usage

```
mxComputeLoop(
  steps,
  ...,
  i = NULL,
  maxIter = as.integer(NA),
  freeSet = NA_character_,
  maxDuration = as.numeric(NA),
  verbose = 0L,
  startFrom = 1L
)
```

Arguments

`steps`	a list of compute objects
`...`	Not used. Forces remaining arguments to be specified by name.

`i`	the values to iterate over
`maxIter`	the maximum number of iterations
`freeSet`	Names of matrices containing free variables.
`maxDuration`	the maximum amount of time (in seconds) to iterate
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable
`startFrom`	When `i=NULL`, permits starting from an index greater than 1.

`mxComputeNelderMead` *Optimize parameters using a variation of the Nelder-Mead algorithm.*

Description

OpenMx includes a flexible, options-rich implementation of the Nelder-Mead algorithm.

Usage

```
mxComputeNelderMead(
freeSet=NA_character_, fitfunction="fitfunction", verbose=0L,
nudgeZeroStarts=mxOption(NULL,"Nudge zero starts"),
maxIter=NULL,...,
alpha=1, betao=0.5, betai=0.5, gamma=2, sigma=0.5, bignum=1e35,
iniSimplexType=c("regular","right","smartRight","random"),
iniSimplexEdge=1, iniSimplexMat=NULL, greedyMinimize=FALSE,
altContraction=FALSE, degenLimit=0, stagnCtrl=c(-1L,-1L),
validationRestart=TRUE,
xTolProx=1e-8, fTolProx=1e-8,
doPseudoHessian=TRUE,
ineqConstraintMthd=c("soft","eqMthd"),
eqConstraintMthd=c("GDsearch","soft","backtrack","l1p"),
backtrackCtrl=c(0.5,5),
centerIniSimplex=FALSE)
```

Arguments

`freeSet`	Character-string names of MxMatrices containing free parameters.
`fitfunction`	Character-string name of the fitfunction; defaults to 'fitfunction'.
`verbose`	Integer level of reporting printed to terminal at runtime; defaults to 0.
`nudgeZeroStarts`	Should free parameters with start values of zero be "nudged" to 0.1 at runtime? Defaults to the current global value of mxOption "Nudge zero starts". May be a logical value, or one of character strings "Yes" or "No".
`maxIter`	Integer maximum number of iterations. Value of `NULL` is accepted, in which case the value used at runtime will be 10 times the number of iterations specified by the effective value of mxOption "Major iterations".

`...`	Not used. Forces remaining arguments to be specified by name.
`alpha`	Numeric reflection coefficient. Must be positive. Defaults to 1.0.
`betao, betai`	Numeric outside- and inside-contraction coefficients, respectively. Both must be within unit interval (0,1). Both default to 0.5.
`gamma`	Numeric expansion coefficient. If positive, must be greater than `alpha`. If non-positive, expansion transformations will not be carried out. Defaults to 2.0.
`sigma`	Numeric shrink coefficient. Cannot exceed 1.0. If non-positive, shrink transformations will not be carried out, and failed contractions will instead be followed by a simplex restart. Defaults to 0.5.
`bignum`	Numeric value with which the fitfunction value is to be replaced if the fit is non-finite or is evaluated at infeasible parameter values. Defaults to 1e35.
`iniSimplexType`	Character string naming the method by which to construct the initial simplex from the free-parameter start values. Defaults to "regular".
`iniSimplexEdge`	Numeric edge-length of the initial simplex. Defaults to 1.0.
`iniSimplexMat`	Optional numeric matrix providing the vertices of the initial simplex. The matrix must have as many columns as there are free parameters in the MxModel. The matrix's number of rows must be no less than the number of free parameters minus the number of degrees-of-freedom gained from equality MxConstraints, if any. If a non-`NULL` value is provided, argument `iniSimplexEdge` is ignored, and argument `iniSimplexType` is only used in the case of a restart.
`greedyMinimize`	Logical; should the optimizer use "greedy minimization?" Defaults to `FALSE`. See below for details.
`altContraction`	Logical; should the optimizer use an "alternate contraction" transformation? Defaults to `FALSE`. See below for details.
`degenLimit`	Numeric "degeneracy limit;" defaults to 0. If positive, the simplex will be restarted if the measure of the angle between any two of its edges is within 0 or pi by less than `degenLimit`.
`stagnCtrl`	"Stagnation control;" integer vector of length 2; defaults to `c(-1L,-1L)`. See below for details.
`validationRestart`	Logical; defaults to TRUE.
`xTolProx`	Numeric "domain-convergence" criterion; defaults to 1e-8. See below for details.
`fTolProx`	Numeric "range-convergence" criterion; defaults to 1e-8. See below for details.
`doPseudoHessian`	Logical; defaults to `TRUE`.
`ineqConstraintMthd`	"Inequality constraint method;" character string. Defaults to "soft".
`eqConstraintMthd`	"Equality constraint method;" character string. Defaults to "GDsearch".
`backtrackCtrl`	Numeric vector of length two. See below for details.

`centerIniSimplex`
: Logical. If `FALSE` (default), the MxModel's start values are used as the "first" vertex of the initial simplex. If `TRUE`, the initial simplex is re-centered so that the MxModel's start values are its eucentroid. However, if `iniSimplexMat` is non-`NULL` or if `iniSimplexType="smartRight"`, a value of `TRUE` is treated as `FALSE`.

Details

The state of a Nelder-Mead optimization problem is represented by a simplex (polytope) of $n + 1$ vertices in the space of the free parameters, where n is the number of free parameters minus the number of degrees-of-freedom gained from equality MxConstraints. An iteration of the algorithm first sorts the $n + 1$ vertices by their corresponding fitfunction values (i.e., the values of the fitfunction when evaluated at each vertex), in ascending order (i.e., from "best" fit to "worst" fit). Then, the "subcentroid," which is the centroid of the "best" n vertices, is calculated. Then, the algorithm attempts to improve upon the worst fit by transforming the simplex; see Singer & Nelder (2009) for details.

Argument `iniSimplexType` dictates how the initial simplex will be constructed from the start values if argument `iniSimplexMat` is `NULL`, and how the simplex will be re-initialized in the case of a restart. In all four cases, the vector of start values constitutes the "starting vertex" of the initial simplex. If `iniSimplexType="regular"`, the initial simplex is merely a regular simplex with edge length equal to `iniSimplexEdge`. A `"right"` simplex is constructed by incrementing each free parameter by `iniSimplexEdge` from its starting value; thus, all the edges that intersect at the starting vertex do so at right angles. A `"smartRight"` simplex is constructed similarly, except that each free parameter is both incremented *and* decremented by `iniSimplexEdge`, and of those two points the one with the smaller fitfunction value is retained as a vertex. A `"random"` simplex is constructed by randomly perturbing the start values, in a manner similar to the default for `mxTryHard()`, to generate the coordinates of the other vertices. The user is advised that bounds on the free parameters may keep the initial simplex from having the requested regularity or edge-length, and that `iniSimplexType` is at best a *suggestion* in the presence of equality MxConstraints.

Note that if argument `iniSimplexMat` has nonzero length, the actual start values of the MxModel's free parameters are not used as a vertex of the initial simplex (unless one of the rows of `iniSimplexMat` happens to contain those start values).

If the simplex is restarted, a new simplex is constructed per argument `iniSimplexType`, with edge length equal to the distance between the current best and second-best vertices, and with the current best vertex used as the "first" vertex.

If `greedyMinimize=FALSE`, "greedy expansion" (Singer & Singer, 2004) is used: if the expansion point and reflection point both have smaller fitfunction values than the best vertex, the expansion point is accepted. If `greedyMinimize=TRUE`, "greedy minimization" (Singer & Singer, 2004) is used: if the expansion point and the reflection point both have smaller fitfunction values than the best vertex, the better of the two new points is accepted.

If argument `altContraction=TRUE`, the "modified contraction step" of Gill et al. (1982, Chapter 4) is used, and the candidate point is contracted toward the best vertex instead of toward the subcentroid.

If positive, the first element of argument `stagnCtrl` sets a threshold for the number of successive iterations in which the best vertex of the simplex does not change, after which the algorithm is said to be "stagnant" (in a sense similar to that of Kelley, 1999). To attempt to remedy the stagnation, the simplex is restarted. If positive, the second element of argument `stagnCtrl` sets threshold for the number of restarts conducted, beyond which stagnation no longer triggers a restart. The rationale for the second element is that the best vertex may not change for many iterations when the optimizer is close to convergence, under which circumstances restarting would be counterproductive, and in any event would require additional fitfunction evaluations.

If argument `validationRestart=TRUE`, then when the optimizer has successfully converged, it will restart the simplex and attempt to improve upon the tentative solution it already found. This validation restart (Gill et al., 1982, Chapter 4) always re-initializes the simplex as a regular simplex, centered on the best vertex of the tentative solution, with edge-length equal to the distance between the best and worst vertices of the tentative solution. Optimization proceeds until convergence to a solution with a better fit value, or $2n$ iterations have elapsed.

The Nelder-Mead optimizer is considered to have successfully converged if (1) the largest *l*-infinity norm of the vector-differences between the best vertex and the other vertices is less than argument `xTolProx`, or (2) if the largest absolute difference in fit value between the best vertex and the other vertices is less than `fTolProx`.

If argument `doPseudoHessian=TRUE`, there are no equality MxConstraints, and the "l1p" method (see below) is not in use for inequality MxConstraints, then OpenMx will attempt to calculate the "pseudo-Hessian" or "curvature" matrix as described in the appendix to Nelder & Mead (1965). If successful, this matrix will be stored in the 'output' slot of the post-run MxComputeNelderMead object. Although crude, its inverse can be used as an estimate of the repeated-sampling covariance matrix of the free parameters when the usual finite-differences Hessian is unreliable.

OpenMx's implementation of Nelder-Mead can handle nonlinear inequality MxConstraints reasonably well. Its default method for doing so, with argument `ineqConstraintMthd="soft"`, imposes a "soft" feasibility constraint by assigning a fitfunction value of `bignum` to points that violate the constraints by more than mxOption 'Feasibility tolerance'. Alternately, with argument `ineqConstraintMthd="eqMthd"`, inequality MxConstraints can be handled by the same method provided to argument `eqConstraintMthd`, whether or not equality MxConstraints are present.

OpenMx's implementation of Nelder-Mead respects equality MxConstraints, but does not handle them especially well. Its effectiveness at handling equalities may be improved by providing a matrix to argument `iniSimplexMat` that ensures *all* of the initial vertices are feasible. Users are warned that this Nelder-Mead implementation will not work correctly with MxModels containing redundant equality MxConstraints, and presently has no way of detecting whether any are present. If argument `eqConstraintMthd="GDsearch"` (the default), then whenever Nelder-Mead evaluates the fitfunction at an infeasible point, it initiates a subsidiary optimization that uses SLSQP to find the nearest (in squared Euclidean distance) feasible point, and replaces that feasible point for the infeasible one. The user should note that the function evaluations that occur during this subsidiary optimization are counted toward the total number of fitfunction evaluations during the call to `mxRun()`. The effectiveness of the 'GDsearch' method is often improved by setting mxOption 'Feasibility tolerance' to a stricter (smaller) value than the on-load default. The method specified by `eqConstraintMthd="soft"` is described in the preceding paragraph. If argument `eqConstraintMthd="backtrack"`, then the optimizer attempts to backtrack from an infeasible point to a feasible point in a manner similar to that of Ghiasi et al. (2008), except that it used with *all* new points, and not just those encountered via reflection, expansion and contraction. In this case, the displacement from the prior point to the candidate point is reduced by the proportion provided as the first element of argument `backtrackCtrl`, and thus a new candidate point is considered. This process is repeated until feasibility of the candidate point is restored, or the number of attempts exceeds the second element of argument `backtrackCtrl`. If argument `eqConstraintMthd="l1p"`, Nelder-Mead is used as part of an l_1-penalty algorithm. When using "l1p", the simplex gradient (Kelley, 1999) and "pseudo-Hessian" are never calculated.

Value

Returns an object of class 'MxComputeNelderMead'.

References

Ghiasi, H., Pasini, D., & Lessard, L. (2008). Constrained globalized Nelder-Mead method for simultaneous structural and manufacturing optimization of a composite bracket. *Journal of Composite Materials, 42*(7), p. 717-736. doi: 10.1177/0021998307088592

Gill, P. E., Murray, W., & Wright, M. H. (1982). *Practical Optimization*. Bingley, UK: Emerald Group Publishing Ltd.

Kelley, C. T. (1999). Detection and remediation of stagnation in the Nelder-Mead algorithm using a sufficient decrease condition. *SIAM Journal of Optimization 10*(1), p. 43-55.

Nelder, J. A., & Mead, R. (1965) . A simplex method for function minimization. *The Computer Journal, 7*, p. 308-313.

Singer, S., & Nelder, J. (2009). Nelder-Mead algorithm. *Scholarpedia, 4*(7):2928., revision #91557. http://www.scholarpedia.org/article/Nelder-Mead_algorithm .

Singer, S., & Singer, S. (2004). Efficient implementation of the Nelder-Mead search algorithm. *Applied Numerical Analysis & Computational Mathematics Journal, 1*(2), p. 524-534. doi: 10.1002/anac.200410015

Examples

```
foo <- mxComputeNelderMead()
str(foo)
```

`mxComputeNewtonRaphson` *Optimize parameters using the Newton-Raphson algorithm*

Description

This optimizer requires analytic 1st and 2nd derivatives of the fit function. Box constraints are supported. Parameters can approach box constraints but will not leave the feasible region (even by some small epsilon>0). Non-finite fit values are interpreted as soft feasibility constraints. That is, when a non-finite fit is encountered, line search is continued after the step size is multiplied by 10 available by increasing the verbose level.

Usage

```
mxComputeNewtonRaphson(
  freeSet = NA_character_,
  ...,
  fitfunction = "fitfunction",
  maxIter = 100L,
  tolerance = 1e-12,
  verbose = 0L
)
```

Arguments

`freeSet`	names of matrices containing free variables
`...`	Not used. Forces remaining arguments to be specified by name.
`fitfunction`	name of the fitfunction (defaults to 'fitfunction')
`maxIter`	maximum number of iterations
`tolerance`	optimization is considered converged when the maximum relative change in fit is less than tolerance
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable

References

Luenberger, D. G. & Ye, Y. (2008). *Linear and nonlinear programming.* Springer.

`mxComputeNothing` *Compute nothing*

Description

Note that this compute plan actually does nothing whereas `mxComputeOnce("expectation","nothing")` may remove the prediction of an expectation.

Usage

```
mxComputeNothing()
```

mxComputeNumericDeriv *Numerically estimate Hessian using Richardson extrapolation*

Description

For N free parameters, Richardson extrapolation requires (iterations * (N^2 + N)) function evaluations. The implementation is closely based on the numDeriv R package.

Usage

```
mxComputeNumericDeriv(
  freeSet = NA_character_,
  ...,
  fitfunction = "fitfunction",
  parallel = TRUE,
  stepSize = imxAutoOptionValue("Gradient step size"),
  iterations = 4L,
  verbose = 0L,
  knownHessian = NULL,
  checkGradient = TRUE,
  hessian = TRUE
)
```

Arguments

`freeSet`	names of matrices containing free variables
`...`	Not used. Forces remaining arguments to be specified by name.
`fitfunction`	name of the fitfunction (defaults to 'fitfunction')
`parallel`	whether to evaluate the fitfunction in parallel (defaults to TRUE)
`stepSize`	starting set size (defaults to 0.0001)
`iterations`	number of Richardson extrapolation iterations (defaults to 4L)
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable
`knownHessian`	an optional matrix of known Hessian entries
`checkGradient`	whether to check the first order convergence criterion (gradient is near zero)
`hessian`	whether to estimate the Hessian. If FALSE then only the gradient is estimated.

Details

In addition to an estimate of the Hessian, forward, central, and backward estimates of the gradient are made available in this compute plan's output slot.

When `checkGradient=TRUE`, the central difference estimate of the gradient is used to determine whether the first order convergence criterion is met. In addition, the forward and backward difference estimates of the gradient are compared for symmetry. When sufficient asymmetry is detected, the standard error is flagged. In the case, profile likelihood confidence intervals should be used for inference instead of standard errors (see `mxComputeConfidenceInterval`).

If provided, the square matrix `knownHessian` should have dimnames set to the names of some subset of the free parameters. Entries of the matrix set to NA will be estimated numerically while entries containing finite values will be copied to the Hessian result.

Examples

```
library(OpenMx)
data(demoOneFactor)
factorModel <- mxModel(name ="One Factor",
mxMatrix(type = "Full", nrow = 5, ncol = 1, free = FALSE, values = .2, name = "A"),
mxMatrix(type = "Symm", nrow = 1, ncol = 1, free = FALSE, values = 1 , name = "L"),
mxMatrix(type = "Diag", nrow = 5, ncol = 5, free = TRUE , values = 1 , name = "U"),
mxAlgebra(A %*% L %*% t(A) + U, name = "R"),
mxExpectationNormal(covariance = "R", dimnames = names(demoOneFactor)),
mxFitFunctionML(),
mxData(cov(demoOneFactor), type = "cov", numObs = 500),
mxComputeSequence(
list(mxComputeNumericDeriv(), mxComputeReportDeriv())
)
)
factorModelFit <- mxRun(factorModel)
factorModelFit$output$hessian
```

`mxComputeOnce` *Compute something once*

Description

Some models are optimized for a sparse Hessian. Therefore, it can be much more efficient to compute the inverse Hessian in comparison to computing the Hessian and then inverting it.

Usage

```
mxComputeOnce(
  from,
  what = NULL,
  how = NULL,
  ...,
  freeSet = NA_character_,
  verbose = 0L,
  .is.bestfit = FALSE
)
```

Arguments

`from`	the object to perform the computation (a vector of expectation or fit function names)
`what`	what to compute
`how`	to compute it (optional)
`...`	Not used. Forces remaining arguments to be specified by name.
`freeSet`	names of matrices containing free variables
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable
`.is.bestfit`	do not use; for backward compatibility

Details

The information matrix is only valid when parameters are at the maximum likelihood estimate. The information matrix is returned in model$output$hessian. You cannot request both the information matrix and the Hessian. The information matrix is invariant to the sign of the log likelihood scale whereas the Hessian is not. Use the `how` parameter to specify which approximation to use (one of "default", "hessian", "sandwich", "bread", and "meat").

Examples

```
data(demoOneFactor)
factorModel <- mxModel(name ="One Factor",
  mxMatrix(type="Full", nrow=5, ncol=1, free=TRUE, values=0.2, name="A"),
    mxMatrix(type="Symm", nrow=1, ncol=1, free=FALSE, values=1, name="L"),
    mxMatrix(type="Diag", nrow=5, ncol=5, free=TRUE, values=1, name="U"),
    mxAlgebra(expression=A %*% L %*% t(A) + U, name="R"),
   mxFitFunctionML(),mxExpectationNormal(covariance="R", dimnames=names(demoOneFactor)),
    mxData(observed=cov(demoOneFactor), type="cov", numObs=500),
    mxComputeOnce('fitfunction', 'fit'))
factorModelFit <- mxRun(factorModel)
factorModelFit$output$fit  # 972.15
```

`mxComputeReportDeriv` *Report derivatives*

Description

Copy the internal gradient and Hessian back to R.

Usage

```
mxComputeReportDeriv(freeSet = NA_character_)
```

Arguments

`freeSet`	names of matrices containing free variables

mxComputeReportExpectation *Report expectation*

Description

Copy the internal model expectations back to R.

Usage

```
mxComputeReportExpectation(freeSet = NA_character_)
```

Arguments

`freeSet`	names of matrices containing free variables

mxComputeSequence *Invoke a series of compute objects in sequence*

Description

Invoke a series of compute objects in sequence

Usage

```
mxComputeSequence(
  steps = list(),
  ...,
  freeSet = NA_character_,
  independent = FALSE
)
```

Arguments

`steps`	a list of compute objects
`...`	Not used; forces argument 'freeSet' to be specified by name.
`freeSet`	Names of matrices containing free parameters.
`independent`	Whether the steps could be executed out-of-order.

mxComputeSetOriginalStarts *Reset parameter starting values*

Description

Sets the current parameter vector back to the original starting values.

Usage

```
mxComputeSetOriginalStarts(freeSet = NA_character_)
```

Arguments

`freeSet`	names of matrices containing free variables

mxComputeSimAnnealing *Optimization using generalized simulated annealing*

Description

Performs simulated annealing to minimize the fit function. If the original starting values are outside of the feasible set, a few attempts are made to find viable starting values.

Usage

```
mxComputeSimAnnealing(freeSet=NA_character_, ..., fitfunction='fitfunction',
 plan=mxComputeOnce('fitfunction','fit'),
 verbose=0L, method=c("tsallis1996", "ingber2012"), control=list(),
 defaultGradientStepSize=imxAutoOptionValue("Gradient step size"),
 defaultFunctionPrecision=imxAutoOptionValue("Function precision"))
```

Arguments

`freeSet`	names of matrices containing free variables
`...`	Not used. Forces remaining arguments to be specified by name.
`fitfunction`	name of the fitfunction (defaults to 'fitfunction')
`plan`	compute plan to optimize the model
`verbose`	level of debugging output
`method`	which algorithm to use
`control`	control parameters specific to the chosen method
`defaultGradientStepSize`	the default gradient step size
`defaultFunctionPrecision`	the default function precision

Details

For method ‘tsallis1996’, the number of function evaluations are determined by the `tempStart` and `tempEnd` parameters. There is no provision to stop early because there is no way to determine whether the algorithm has converged. The Markov step is implemented by cycling through each parameters in turn and considering a univariate jump (like a Gibbs sampler).

Control parameters include `qv` to control the shape of the visiting distribution, `qaInit` to control the shape of the initial acceptance distribution, `lambda` to reduce the probability of acceptance in time, `tempStart` to specify starting temperature, `tempEnd` to specify ending temperature, and `stepsPerTemp` to set the number of Markov steps per temperature step.

Non-linear constraints are accommodated by a penalty function. Inequality constraints work reasonably well, but equality constraints do not work very well. Constrained optimization will likely require increasing `stepsPerTemp`.

Classical simulated annealing (CSA) can be obtained with `qv=qa=1` and `lambda=0`. Fast simulated annealing (FSA) can be obtained with `qv=2`, `qa=1`, and `lambda=0`. FSA is faster than CSA, but GSA is faster than FSA. GenSA default parameters are set to those identified in Xiang, Sun, Fan & Gong (1997).

Method ‘ingber2012’ has spawned a cultural tradition over more than 30 years that is documented in Aguiar e Oliveira et al (2012). Options are specified using the traditional option names in the `control` list. However, there are a few option changes to make ASA fit better with OpenMx. Instead of option `Curvature_0`, use mxComputeNumericDeriv. ASA_PRINT output is directed to `/dev/null` by default. To direct ASA_PRINT output to console use `control=list('Asa_Out_File'= '/dev/fd/1')`. ASA’s option to control the finite differences gradient step size, `Delta_X`, defaults to mxOption’s ‘Gradient step size’ instead of ASA’s traditional 0.001. Similarly, `Cost_Precision` defaults to mxOption’s ‘Function Precision’ instead of ASA’s traditional 1e-18.

References

Aguiar e Oliveira, H., Ingber, L., Petraglia, A., Petraglia, M. R., & Machado, M. A. S. (2012). *Stochastic global optimization and its applications with fuzzy adaptive simulated annealing.* Springer Publishing Company, Incorporated.

Tsallis, C., & Stariolo, D. A. (1996). Generalized simulated annealing. *Physica A: Statistical Mechanics and its Applications, 233*(1-2), 395-406.

Xiang, Y., Sun, D. Y., Fan, W., & Gong, X. G. (1997). Generalized simulated annealing algorithm and its application to the Thomson model. *Physics Letters A, 233*(3), 216-220.

See Also

`mxComputeTryHard`

Examples

```
library(OpenMx)
m1 <- mxModel(
"poly22", # Eqn 22 from Tsallis & Stariolo (1996)
mxMatrix(type='Full', values=runif(4, min=-1e6, max=1e6),
ncol=1, nrow=4, free=TRUE, name='x'),
mxAlgebra(sum((x*x-8)^2) + 5*sum(x) + 57.3276, name="fit"),
mxFitFunctionAlgebra('fit'),
mxComputeSimAnnealing())

m1 <- mxRun(m1)
summary(m1)
```

`mxComputeStandardError` *Compute standard errors*

Description

When the fit is in -2 log likelihood units, the SEs are derived from the diagonal of the Hessian or inverse Hessian. The Hessian (in some form) must already be available.

Usage

```
mxComputeStandardError(freeSet = NA_character_, fitfunction = "fitfunction")
```

Arguments

`freeSet`	names of matrices containing free variables
`fitfunction`	name of the fitfunction (defaults to 'fitfunction')

Details

If there are active MxConstraints and the fit is in -2logL units, the SEs are derived from the Hessian and the Jacobian of the constraint functions (see references).

References

Moore T & Sadler B. (2006). *Maximum-Likelihood Estimation and Scoring Under Parametric Constraints*. Army Research Laboratory report ARL-TR-3805. Schoenberg R. (1997). Constrained maximum likelihood. *Computational Economics, 10*, p. 251-266.

`mxComputeTryCatch` *Execute a sub-compute plan, catching errors*

Description

[Experimental] Any error will be recorded in a subsequent checkpoint. After execution, the context will be reset to continue computation as if no errors has occurred.

Usage

```
mxComputeTryCatch(plan, ..., freeSet = NA_character_)
```

Arguments

`plan`	compute plan to optimize the model
`...`	Not used. Forces remaining arguments to be specified by name.
`freeSet`	names of matrices containing free variables

See Also

mxComputeCheckpoint

`mxComputeTryHard` *Repeatedly attempt a compute plan until successful*

Description

The provided compute plan is run until the status code indicates success (0 or 1). It gives up after a small number of retries.

Usage

```
mxComputeTryHard(
  plan,
  ...,
  freeSet = NA_character_,
  verbose = 0L,
  location = 1,
  scale = 0.25,
  maxRetries = 3L
)
```

Arguments

`plan`	compute plan to optimize the model
`...`	Not used. Forces remaining arguments to be specified by name.
`freeSet`	names of matrices containing free variables
`verbose`	integer. Level of run-time diagnostic output. Set to zero to disable
`location`	location of the perturbation distribution
`scale`	scale of the perturbation distribution
`maxRetries`	maximum number of plan evaluations per invocation (including the first evaluation)

Details

Upon failure, start values are randomly perturbed. Currently only the uniform distribution is implemented. The distribution is parameterized by arguments `location` and `scale`. The location parameter is the distribution's median. For the uniform distribution, `scale` is the absolute difference between its median and extrema (i.e., half the width of the rectangle). Each start value is multiplied by a random draw and then added to a random draw from a distribution with the same `scale` but with a median of zero.

References

Shanno, D. F. (1985). On Broyden-Fletcher-Goldfarb-Shanno method. *Journal of Optimization Theory and Applications, 46*(1), 87-94.

See Also

`mxTryHard`

`mxConstraint` *Create MxConstraint Object*

Description

This function creates a new MxConstraint object.

Usage

```
mxConstraint(expression, name=NA, ..., jac=character(0), verbose=0L, strict=TRUE)
```

Arguments

`expression`	The MxAlgebra-like expression representing the constraint function.
`name`	An optional character string indicating the name of the object.
`...`	Not used. Helps OpenMx catch bad input to argument `expression`, and requires argument `jac`–meant for advanced users–to be specified by name.
`jac`	An optional character string naming the MxAlgebra or MxMatrix representing the Jacobian for the constraint function.
`verbose`	For values greater than zero, enable runtime diagnostics.
`strict`	Whether to require that all Jacobian entries reference free parameters.

Details

The `mxConstraint()` function defines relationships between two MxAlgebra or MxMatrix objects. They are used to affect the estimation of free parameters in the referenced objects. The constraint relation is written identically to how a MxAlgebra expression would be written. The outermost operator in this relation must be either '<', '==' or '>'. To affect an estimation or optimization, an MxConstraint object must be included in an MxModel object with all referenced MxAlgebra and MxMatrix objects.

Usage Note: Use of `mxConstraint()` should be avoided where it is possible to achieve the constraint by equating free parameters by label or position in an MxMatrix or MxAlgebra object. Including mxConstraints in an mxModel will disable standard errors and the calculation of the final Hessian, and thus should be avoided when standard errors are of importance. Constraints also add computational overhead. If one labels two parameters the same, the optimizer has one fewer parameter to optimize. However, if one uses mxConstraint to do the same thing, both parameters remain estimated and a Lagrangian multiplier is added to maintain the constraint. This constraint also has to have its gradients computed and the order of the Hessian grows as well. So while both approaches should work, the mxConstraint() will take longer to do so.

Alternatives to mxConstraints include using labels, lbound or ubound arguments or algebras. Free parameters in the same MxModel may be constrained to equality by giving them the same name in their respective 'labels' matrices. Similarly, parameters may be fixed to an individual element in a MxModel object or the result of an MxAlgebra object through labeling. For example, assigning a label of "name[1,1]" fixes the value of a parameter at the value in first row and first column of the matrix or algebra "name". The mxConstraint function should be used to enforce inequalities that cannot be conveyed using other methods.

Note that constraints should not depend on definition variables. This mode of operation is not supported.

Argument `jac` is used to provide the name of an MxMatrix or MxAlgebra that equals the matrix of first derivatives–the Jacobian–of the constraint function with respect to the free parameters. Here, the "constraint function" refers to the constraint expression in canonical form: an arbitrary matrix expression on the left-hand side of the comparator, and a matrix of zeroes with the same dimensions on the right-hand side. The rows of the Jacobian correspond to elements of the matrix result of the right-hand side, in column-major order. Each row of the Jacobian is the vector of first partial derivatives, with respect to the free parameters of the MxModel, of its corresponding element. Each column of the Jacobian corresponds to a free parameter of the MxModel; each column must be named with the label of the corresponding free parameter. All the gradient-descent optimizers are able to take advantage of user-supplied Jacobians. To verify the analytic Jacobian against the same values estimated by finite differences, use 'verbose=3'.

In the past, OpenMx has relied on NPSOL's finite differences algorithm to fill in unknown Jacobian entries. When analytic Jacobians are used, OpenMx no longer relies on NPSOL's finite differences algorithm. Any missing entries are taken care of by OpenMx's finite differences algorithm. Whether NPSOL or OpenMx conducts finite differences, the results should be very similar.

Value

Returns an MxConstraint object.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

MxConstraint for the S4 class created by mxConstraint.

Examples

```
library(OpenMx)

#Create a constraint between MxMatrices 'A' and 'B'
constraint <- mxConstraint(A > B, name = 'AdominatesB')

# Constrain matrix 'K' to be equal to matrix 'limit'

model <- mxModel(model="con_test",
    mxMatrix(type="Full", nrow=2, ncol=2, free=TRUE, name="K"),
    mxMatrix(type="Full", nrow=2, ncol=2, free=FALSE, name="limit", values=1:4),
    mxConstraint(K == limit, name = "Klimit_equality"),
    mxAlgebra(min(K), name="minK"),
    mxFitFunctionAlgebra("minK")
)

fit <- mxRun(model)
fit$matrices$K$values

#      [,1] [,2]
# [1,]    1    3
# [2,]    2    4

# Constrain both free parameters of a matrix to equality using labels (both are set to "eq")
equal <- mxMatrix("Full", 2, 1, free=TRUE, values=1, labels="eq", name="D")

# Constrain a matrix element in to be equal to the result of an algebra
start <- mxMatrix("Full", 1, 1, free=TRUE,  values=1, labels="param", name="F")
alg   <- mxAlgebra(log(start), name="logP")

# Force the fixed parameter in matrix G to be the result of the algebra
end   <- mxMatrix("Full", 1, 1, free=FALSE, values=1, labels="logP[1,1]", name="G")
```

MxConstraint-class *Class* "MxConstraint"

Description

MxConstraint is an S4 class. An MxConstraint object is a named entity. New instances of this class can be created using the function `mxConstraint()`.

Details

Slots may be referenced with the $ symbol. See the documentation for Classes and the examples in the mxConstraint document for more information.

Slots

`name`**:** Character string; the name of the object.

`formula`**:** Object of class `"MxAlgebraFormula"`. The MxAlgebra-like expression representing the constraint function.

`alg1`**:** Object of class `"MxCharOrNumber"`. For internal use.

`alg2`**:** Object of class `"MxCharOrNumber"`. For internal use.

`relation`**:** Object of class `"MxCharOrNumber"`. For internal use.

`jac`**:** Object of class `"MxCharOrNumber"`. Identifies the MxAlgebra representing the Jacobian for the constraint function.

`linear`**:** Logical. For internal use.

`strict`**:** Logical. Whether to require that all Jacobian entries reference free parameters.

`verbose`**:** integer. For values greater than zero, enable runtime diagnostics.

Methods

$<- `signature(x = "MxConstraint")`

$ `signature(x = "MxConstraint")`

imxDeparse `signature(object = "MxConstraint")`

names `signature(x = "MxConstraint")`

print `signature(x = "MxConstraint")`

show `signature(object = "MxConstraint")`

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

`mxConstraint()` for the function that creates MxConstraint objects.

Examples

```
showClass("MxConstraint")
```

`mxData` *Create MxData Object*

Description

This function creates a new MxData object. This can be used all forms of analysis (including WLS: see mxFitFunctionWLS). It packages observed data (e.g. a dataframe, matrix, or cov or cor matrix) into an object with additional information allowing it to be processed in an mxModel.

Usage

```
mxData(observed=NULL, type="none", means = NA, numObs = NA, acov=NA, fullWeight=NA,
     thresholds=NA, ..., observedStats=NA, sort=NA, primaryKey = as.character(NA),
         weight = as.character(NA), frequency = as.character(NA),
         verbose = 0L, .parallel=TRUE, .noExoOptimize=TRUE,
   minVariance=sqrt(.Machine$double.eps), algebra=c(),
  warnNPDacov=TRUE, warnNPDuseWeight=TRUE, exoFree=NULL,
  naAction=c("pass","fail","omit","exclude"))
```

Arguments

`observed`	A matrix or data.frame which provides data to the MxData object. Can be NULL when summary data are provided via 'observedStats'.
`type`	A character string defining the type of data in the 'observed' argument. Must be one of "raw", "cov", "cor", or "acov". If no observed data are provided then use "none".
`means`	An optional vector of means for use when 'type' is "cov", or "cor".
`numObs`	The number of observations in the data supplied in the 'observed' argument. Required unless 'type' equals "raw".
`...`	Not used. Forces remaining arguments to be specified by name.
`observedStats`	A list containing observed statistics for weighted least squares estimation. See details for contents
`sort`	Whether to sort raw data prior to use (default NA).
`primaryKey`	The column name of the primary key used to uniquely identify rows (default NA)
`weight`	The column name containing row weights.
`frequency`	The column name containing row frequencies.
`verbose`	level of diagnostic output.
`.parallel`	logical. Whether to compute observed summary statistics in parallel.
`.noExoOptimize`	logical. Whether to use math short-cuts for the case of no exogenous predictors.
`minVariance`	numeric. The minimum acceptable variance for 'observedStats$cov'.
`acov`	Deprecated in favor of the acov element of observedStats.

`fullWeight`	Deprecated in favor of the fullWeight element of observedStats.
`thresholds`	Deprecated in favor of the thresholds element of observedStats.
`algebra`	character vector. Names of algebras used to fill in calculated columns of raw data. **[Experimental]**
`warnNPDacov`	**[Deprecated]**
`warnNPDuseWeight`	logical. Whether to warn when the asymptotic covariance matrix is non-positive definite.
`exoFree`	logical matrix of observed manifests by exogenous predictors. Defaults to all TRUE, but you can fix some regression coefficients in the `observedStats` `slope` matrix to zero by setting entries to FALSE. **[Experimental]**
`naAction`	Specify treatment of missing data. See details. **[Maturing]**

Details

The mxData function creates MxData objects used in mxModels. The 'observed' argument may take either a data frame or a matrix, which is then described with the 'type' argument. Data types describe compatibility and usage with expectation functions in MxModel objects. Three data types are supported (acov is deprecated).

raw The contents of the 'observed' argument are treated as raw data. Missing values are permitted and must be designated as the system missing value. The 'means' and 'numObs' arguments cannot be specified, as the 'means' argument is not relevant and the 'numObs' argument is automatically populated with the number of rows in the data. Data of this type may use fit functions such as mxFitFunctionML or mxFitFunctionWLS. mxFitFunctionML will automatically use use full-information maximum likelihood for raw data.

cov The contents of the 'observed' argument are treated as a covariance matrix. The 'means' argument is not required, but may be included for estimations involving means. The 'numObs' argument is required, which should reflect the number of observations or rows in the data described by the covariance matrix. Cov data typically use the mxFitFunctionML fit function, depending on the specified model.

acov This type was used for WLS data as created by mxDataWLS. Unless you are using summary data, its use is deprecated. Instead, use type ='raw' and an mxFitFunctionWLS. If type 'acov' is set, the 'observed' argument will (usually) contain raw data and the 'observedStats' slot contain a list of observed statistics.

cor The contents of the 'observed' argument are treated as a correlation matrix. The 'means' argument is not required, but may be included for estimations involving means. The 'numObs' argument is required, which should reflect the number of observations or rows in the data described by the covariance matrix. Models with cor data typically use the mxFitFunctionML fit function.

Note on data handling: OpenMx uses the names of variables to map them onto other elements of your model, such as expectation functions. Thus for data provided as a data.frame, ensure the columns have appropriate `names`. Covariance and correlation matrices need to have both the row and column names set and these must be identical, for instance by using `dimnames = list(varNames,varNames)`.

Correlation data

To obtain accurate parameter estimates and standard errors, it is necessary to constrain the model implied covariance matrix to have unit variances. This constraint is added automatically if you use an `mxModel` with `type='RAM'` or `type='LISREL'`. Otherwise, you will need to add this constraint yourself.

WLS data

The `observedStats` contains the following named objects: cov, slope, means, asymCov, useWeight, and thresholds.

'cov' The (polychoric) covariance matrix of raw data variables. An error is raised if any variance is smaller `minVariance`.

'slope' The regression coefficients from all exogenous predictors to all observed variables. Required for exogenous predictors.

'means' The means of the data variables. Required for estimations involving means.

'thresholds' Thresholds of ordinal variables. Required for models including ordinal variables.

'asymCov' The asymptotic covariance matrix (all entries non-zero). This matrix is sample size independent.

'useWeight' (optional) The weight matrix used in the mxFitFunctionWLS. Can be dense or diagonal for diagonally weighted least squares. This matrix is scaled by the sample size.

note: WLS data typically use the mxFitFunctionWLS function.

IMPORTANT: The WLS interface is under heavy development to support both very fast backend processing of raw data while continuing to support modeling applications which require direct access to the object in the front end. Some user-interface changes should be expected as we optimize both these workflows.

Missing values

For raw data, the 'naAction' option controls the treatment of missing values. When set to 'pass', the data is passed as-is. When set to 'fail', the presence of any missing value will trigger an error. When set to 'omit', missing data will be discarded row-wise. For example, a single missing value in a row will cause the whole row to be discarded. When set to 'exclude', rows with missing data are retained but their 'frequency' is set to zero.

Weights

In the case of raw data, the optional 'weight' argument names a column in the data that contains per-row weights. Similarly, the optional 'frequency' argument names a column in the 'observed' data that contains per-row frequencies. Frequencies must be integers but weights can be arbitrary real numbers. For data with many repeated response patterns, organizing the data into unique patterns and frequencies can reduce model evaluation time.

In some cases, the fit function can be evaluated more efficiently when data are sorted. When a primary key is provided, sorting is disabled. Otherwise, sort defaults to TRUE.

The mxData function does not currently place restrictions on the size, shape, or symmetry of matrices input into the 'observed' argument. While it is possible to specify MxData objects as covariance or correlation matrices that do not have the properties commonly associated with these matrices, failure to correctly specify these matrices will likely lead to problems in model estimation.

note: MxData objects may not be included in mxAlgebras nor in the mxFitFunctionAlgebra function. To reference data in these functions, use a mxMatrix or a definition variable (data.var) label.

Also, while column names are stored in the 'observed' slot of MxData objects, these names are not automatically recognized as variable names in mxPaths in RAM models. These models use the 'manifestVars' of the mxModel function to explicitly identify used variables used in the model.

Value

Returns a new MxData object.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

To generate data, see `mxGenerateData`; For objects which may be entered as arguments in the 'observed' slot, see matrix and data.frame. See MxData for the S4 class created by mxData. For WLS data, see mxDataWLS (deprecated). More information about the OpenMx package may be found here.

Examples

```
library(OpenMx)

# Simple covariance model. See other mxFitFunctions for examples with different data types

# 1. Create a covariance matrix x and y
covMatrix <- matrix(nrow = 2, ncol = 2, byrow = TRUE,
c(0.77642931, 0.39590663,
      0.39590663, 0.49115615)
)
covNames <- c("x", "y")
dimList <- list(covNames, covNames)
dimnames(covMatrix) <- dimList

# 2. Create an MxData object from covMatrix
testData <- mxData(observed=covMatrix, type="cov", numObs = 100)

testModel <- mxModel(model="testModel2",
mxMatrix(name="expCov", type="Symm", nrow=2, ncol=2,
                 values=c(.2,.1,.2), free=TRUE, dimnames=dimList),
    mxExpectationNormal("expCov", dimnames=covNames),
    mxFitFunctionML(),
testData
)

outModel <- mxRun(testModel)

summary(outModel)
```

`MxData-class` *MxData Class*

Description

MxData is an S4 class. An MxData object is a named entity. New instances of this class can be created using the function mxData. MxData is an S4 class union. An MxData object is either NULL or a MxNonNullData object.

Details

The MxNonNullData class has the following slots:

name	-	The name of the object
observed	-	Either a matrix or a data frame
vector	-	A vector for means, or NA if missing
type	-	Either 'raw', 'cov', or 'cor'
numObs	-	The number of observations

The 'name' slot is the name of the MxData object.

The 'observed' slot is used to contain data, either as a matrix or as a data frame. Use of the data in this slot by other functions depends on the value of the 'type' slot. When 'type' is equal to 'cov' or 'cor', the data input into the 'matrix' slot should be a symmetric matrix or data frame.

The 'vector' slot is used to contain a vector of numeric values, which is used as a vector of means for MxData objects with 'type' equal to 'cov' or 'cor'. This slot may be used in estimation using the mxFitFunctionML function.

The 'type' slot may take one of four supported values:

raw The contents of the 'observed' slot are treated as raw data. Missing values are permitted and must be designated as the system missing value. The 'vector' and 'numObs' slots cannot be specified, as the 'vector' argument is not relevant and the 'numObs' argument is automatically populated with the number of rows in the data. Data of this type may use the mxFitFunctionML function as its fit function in MxModel objects, which can deal with covariance estimation under full-information maximum likelihood.

cov The contents of the 'observed' slot are treated as a covariance matrix. The 'vector' argument is not required, but may be included for estimations involving means. The 'numObs' slot is required. Data of this type may use fit functions such as the mxFitFunctionML, depending on the specified model.

cor The contents of the 'observed' slot are treated as a correlation matrix. The 'vector' argument is not required, but may be included for estimations involving means. The 'numObs' slot is required. Data of this type may use fit functions such as the mxFitFunctionML, depending on the specified model.

The 'numObs' slot describes the number of observations in the data. If 'type' equals 'raw', then 'numObs' is automatically populated as the number of rows in the matrix or data frame in the 'observed' slot. If 'type' equals 'cov' or 'cor', then this slot must be input using the 'numObs' argument in the mxData function when the MxData argument is created.

MxData objects may not be included in MxAlgebra objects or use the mxFitFunctionAlgebra function. If these capabilities are desired, data should be appropriately input or transformed using the mxMatrix and mxAlgebra functions.

While column names are stored in the 'observed' slot of MxData objects, these names are not recognized as variable names in MxPath objects. Variable names must be specified using the 'manifestVars' argument of the mxModel function prior to use in MxPath objects.

The mxData function does not currently place restrictions on the size, shape, or symmetry of matrices input into the 'observed' argument. While it is possible to specify MxData objects as covariance or correlation matrices that do not have the properties commonly associated with these matrices, failure to correctly specify these matrices will likely lead to problems in model estimation.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

mxData for creating MxData objects, matrix and data.frame for objects which may be entered as arguments in the 'matrix' slot. More information about the OpenMx package may be found here.

`mxDataDynamic` *Create dynamic data*

Description

Create dynamic data

Usage

```
mxDataDynamic(type, ..., expectation, verbose = 0L)
```

Arguments

`type`	type of data
`...`	Not used. Forces remaining arguments to be specified by name.
`expectation`	the name of the expectation to provide the data
`verbose`	Increase runtime debugging output

MxDataStatic-class *Create static data*

Description

Internal static data class.

Details

Not to be used.

mxDataWLS *Create legacy MxData Object for Least Squares (WLS, DWLS, ULS) Analyses*

Description

This function creates a new MxData object of type “ULS” (unweighted least squares), “WLS” (weighted least squares) or “DWLS” (diagonally-weighted least squares). The appropriate fit function to include with these models is `mxFitFunctionWLS`

note: This function continues to work, but is deprecated. Use mxData and mxFitFunctionWLS instead.

Usage

```
mxDataWLS(data, type = "WLS", useMinusTwo = TRUE, returnInverted = TRUE,
 fullWeight = TRUE, suppressWarnings = TRUE, allContinuousMethod =
c("cumulants", "marginals"), silent=!interactive())
```

Arguments

`data`	A matrix or data.frame which provides raw data to be used for WLS.
`type`	A character string 'WLS' (default), 'DWLS', or 'ULS' for weighted, diagonally weighted, or unweighted least squares, respectively
`useMinusTwo`	Logical indicating whether to use -2LL (default) or -LL.
`returnInverted`	Logical indicating whether to return the information matrix (default) or the covariance matrix.
`fullWeight`	Logical determining if the full weight matrix is returned (default). Needed for standard error and quasi-chi-squared calculation.
`suppressWarnings`	Logical that determines whether to suppress diagnostic warnings. These warnings are likely only helpful to developers.
`allContinuousMethod`	A character string 'cumulants' (default) or 'marginals'. See mxFitFunctionWLS.
`silent`	Whether to report progress

Details

The mxDataWLS function creates an MxData object, which can be used in MxModel objects. This function takes raw data and returns an `MxData` object to be used in a model to fit with weighted least squares.

note: This function continues to work, but is deprecated. Use mxData and mxFitFunctionWLS instead.

Value

Returns a new MxData object.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Browne, M. W. (1984). Asymptotically Distribution-Free Methods for the Analysis of Covariance Structures. *British Journal of Mathematical and Statistical Psychology*, **37**, 62-83.

See Also

mxFitFunctionWLS. MxData for the S4 class created by mxData. matrix and data.frame for objects which may be entered as arguments in the 'observed' slot. More information about the OpenMx package may be found here.

Examples

```
# Create and fit a model using mxMatrix, mxAlgebra, mxExpectationNormal, and mxFitFunctionWLS

library(OpenMx)

# Simulate some data

x=rnorm(1000, mean=0, sd=1)
y= 0.5*x + rnorm(1000, mean=0, sd=1)
tmpFrame <- data.frame(x, y)
tmpNames <- names(tmpFrame)
wdata <- mxDataWLS(tmpFrame)

# Define the matrices


S <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(1,0,0,1),
              free=c(TRUE,FALSE,FALSE,TRUE), labels=c("Vx", NA, NA, "Vy"), name = "S")
A <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(0,1,0,0),
              free=c(FALSE,TRUE,FALSE,FALSE), labels=c(NA, "b", NA, NA), name = "A")
I <- mxMatrix(type="Iden", nrow=2, ncol=2, name="I")

# Define the expectation

expCov <- mxAlgebra(solve(I-A) %*% S %*% t(solve(I-A)), name="expCov")
expFunction <- mxExpectationNormal(covariance="expCov", dimnames=tmpNames)
```

```
# Choose a fit function

fitFunction <- mxFitFunctionWLS()

# Define the model

tmpModel <- mxModel(model="exampleModel", S, A, I, expCov, expFunction, fitFunction,
                    wdata)

# Fit the model and print a summary

tmpModelOut <- mxRun(tmpModel)
summary(tmpModelOut)
```

`mxDescribeDataWLS` *Determine whether a dataset will have weights and summary statistics for the means if used with mxFitFunctionWLS*

Description

Given either a data.frame or an mxData of type raw, this function determines whether `mxFitFunctionWLS` will generate expectations for means.

Usage

```
mxDescribeDataWLS(
  data,
  allContinuousMethod = c("cumulants", "marginals"),
  verbose = FALSE
)
```

Arguments

`data`	the (currently raw) data being used in a `mxFitFunctionWLS` model.
`allContinuousMethod`	the method used to process data when all columns are continuous.
`verbose`	logical. Whether to report diagnostics.

Details

All-continuous data processed using the "cumulants" method lack means, while all continuous data processed with allContinuousMethod = "marginals" will have means.

When data are not all continuous, allContinuousMethod is ignored, and means are modelled.

Value

- list describing the data.

See Also

- mxFitFunctionWLS, omxAugmentDataWithWLSSummary

Examples

```
# ====================================
# = All continuous, data.frame input =
# ====================================

tmp = mxDescribeDataWLS(mtcars, allContinuousMethod= "cumulants", verbose = TRUE)
tmp$hasMeans # FALSE - no means with cumulants
tmp = mxDescribeDataWLS(mtcars, allContinuousMethod= "marginals")
tmp$hasMeans # TRUE we get means with marginals

# ==========================
# = mxData object as input =
# ==========================
tmp = mxData(mtcars, type="raw")
mxDescribeDataWLS(tmp, allContinuousMethod= "cumulants", verbose = TRUE)$hasMeans # FALSE
mxDescribeDataWLS(tmp, allContinuousMethod= "marginals")$hasMeans  # TRUE

# =======================================
# = One var is a factor: Means modelled =
# =======================================
tmp = mtcars
tmp$cyl = factor(tmp$cyl)
mxDescribeDataWLS(tmp, allContinuousMethod= "cumulants")$hasMeans # TRUE - always has means
mxDescribeDataWLS(tmp, allContinuousMethod= "marginals")$hasMeans # TRUE
```

`MxDirectedGraph-class` *MxDirectedGraph*

Description

This is an internal class and should not be used directly. It is a class for directed graphs.

`mxEval` *Evaluate Values in MxModel*

Description

This function can be used to evaluate an arbitrary R expression that includes named entities from a MxModel object, or labels from a MxMatrix object.

Usage

```
mxEval(expression, model, compute = FALSE, show = FALSE, defvar.row = 1,
    cache = new.env(parent = emptyenv()), cacheBack = FALSE, .extraBack=0L)

mxEvalByName(name, model, compute = FALSE, show = FALSE, defvar.row = 1,
    cache = new.env(parent = emptyenv()), cacheBack = FALSE, .extraBack=0L)
```

Arguments

`expression`	An arbitrary R expression.
`model`	The model in which to evaluate the expression.
`compute`	If TRUE then compute the value of algebra expressions and populate square bracket substitutions.
`show`	If TRUE then print the translated expression.
`defvar.row`	The row number for definition variables when compute=TRUE; defaults to 1. When compute=FALSE, values for definition variables are always taken from the first (i.e., first before any automated sorting is done) row of the raw data.
`cache`	An R environment of matrix values used to speedup computation.
`cacheBack`	If TRUE then return the list pair (value, cache).
`name`	The character name of an object to evaluate.
`.extraBack`	Depth of original caller in count of stack frames (environments).

Details

[Stable] The argument ‘expression’ is an arbitrary R expression. Any named entities that are used within the R expression are translated into their current value from the model. Any labels from the matrices within the model are translated into their current value from the model. Finally the expression is evaluated and the result is returned. To enable debugging, the ‘show’ argument has been provided. The most common mistake when using this function is to include named entities in the model that are identical to R function names. For example, if a model contains a named entity named ‘c’, then the following mxEval call will return an error: `mxEval(c(A,B,C),model)`.

The `mxEvalByName` function is a wrapper around `mxEval` that takes a character instead of an R expression.

If ‘compute’ is FALSE, then MxAlgebra expressions return their current values as they have been computed by the optimization call (using mxRun). If the ‘compute’ argument is TRUE, then MxAlgebra expressions will be calculated in R and square bracket substitutions will be performed. Any references to an objective function that has not yet been calculated will return a 1 x 1 matrix with a value of NA.

The ‘cache’ is used to speedup calculation by storing previously computing values. The cache is a list of matrices, such that names(cache) must all be of the form “modelname.entityname”. Setting ‘cacheBack’ to TRUE will return the pair list(value, cache) where value is the result of the mxEval() computation and cache is the updated cache.

References

The OpenMx User’s guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

mxAlgebra to create algebraic expressions inside your model and mxModel for the model object mxEval looks inside when evaluating.

Examples

```
library(OpenMx)

# Set up a 1x1 matrix
matrixA <- mxMatrix("Full", nrow = 1, ncol = 1, values = 1, name = "A")

# Set up an algebra
algebraB <- mxAlgebra(A + A, name = "B")

# Put them both in a model
testModel <- mxModel(model="testModel3", matrixA, algebraB)

# Even though the model has not been run, we can evaluate the algebra
#   given the starting values in matrixA.
mxEval(B, testModel, compute=TRUE)

# If we just print the algebra, we can see it has not been evaluated
testModel$B
```

`mxEvaluateOnGrid` *Evaluate an algebra on an abscissa grid and collect column results*

Description

This function evaluates an algebra on a grid of points provided in an auxiliary abscissa matrix.

Usage

```
mxEvaluateOnGrid(algebra, abscissa)
```

Arguments

`algebra`	the name of the single column matrix to be evaluated.
`abscissa`	the name of the abscissa matrix. See details.

Details

The abscissa matrix must be in a specific format. The variables are in the rows. Abscissa row names must match names of free variables. The grid points are in columns. For each point (column), the free variables are set to the given values and the algebra is re-evaluated. The resulting columns are collected as the result.

Value

Returns the collected columns.

Examples

```
library(OpenMx)

test2 <- mxModel("test2",
mxMatrix(values=1.1, nrow=1, ncol=1, free=TRUE, name="thang"),
mxMatrix(nrow=1, ncol=1, labels="abscissa1", free=TRUE, name="currentAbscissa"),
mxMatrix(values=-2:2, nrow=1, ncol=5, name="abscissa",
 dimnames=list(c('abscissa1'), NULL)),
mxAlgebra(rbind(currentAbscissa + thang, currentAbscissa * thang), name="stuff"),
mxAlgebra(mxEvaluateOnGrid(stuff, abscissa), name="grid"))

test2 <- mxRun(test2)
omxCheckCloseEnough(test2$grid$result, matrix(c(-1:3 + .1, -2:2 * 1.1), ncol=5, nrow=2,byrow=TRUE))
```

`MxExpectation-class` *MxExpectation*

Description

This is an internal class and should not be used directly.

`mxExpectationBA81` *Create a Bock & Aitkin (1981) expectation*

Description

Used in conjunction with mxFitFunctionML, this expectation models ordinal data with a modest number of latent dimensions. Currently, only a multivariate Normal latent distribution is supported. An equal-interval quadrature is used to integrate over the latent distribution. When all items use the graded response model and items are assumed conditionally independent then item factor analysis is equivalent to a factor model.

Usage

```
mxExpectationBA81(
  ItemSpec,
  item = "item",
  ...,
  qpoints = 49L,
  qwidth = 6,
  mean = "mean",
  cov = "cov",
  verbose = 0L,
  weightColumn = NA_integer_,
  EstepItem = NULL,
  debugInternal = FALSE
)
```

Arguments

`ItemSpec`	a single item model (to replicate) or a list of item models in the same order as the column of `ItemParam`
`item`	the name of the mxMatrix holding item parameters with one column for each item model with parameters starting at row 1 and extra rows filled with NA
`...`	Not used. Forces remaining arguments to be specified by name.
`qpoints`	number of points to use for equal interval quadrature integration (default 49L)
`qwidth`	the width of the quadrature as a positive Z score (default 6.0)
`mean`	the name of the mxMatrix holding the mean vector
`cov`	the name of the mxMatrix holding the covariance matrix
`verbose`	the level of runtime diagnostics (default 0L)
`weightColumn`	the name of the column in the data containing the row weights (DEPRECATED)
`EstepItem`	a simple matrix of item parameters for the E-step. This option is mainly of use for debugging derivatives.
`debugInternal`	when enabled, some of the internal tables are returned in $debug. This is mainly of use to developers.

Details

The conditional likelihood of response x_{ij} to item j from person i with item parameters ξ_j and latent ability θ_i is

$$L(x_i|\xi, \theta_i) = \prod_j \mathrm{Pr}(\mathrm{pick} = x_{ij}|\xi_j, \theta_i).$$

Items are assumed to be conditionally independent. That is, the outcome of one item is assumed to not influence another item after controlling for ξ and θ_i.

The unconditional likelihood is obtained by integrating over the latent distribution θ_i,

$$L(x_i|\xi) = \int L(x_i|\xi, \theta_i)L(\theta_i)\mathrm{d}\theta_i.$$

With an assumption that examinees are independently and identically distributed, we can sum the individual log likelihoods,

$$\mathcal{L} = \sum_i \log L(x_i|\xi).$$

Response models $\Pr(\text{pick} = x_{ij} | \xi_j, \theta_i)$ are not implemented in OpenMx, but are imported from the RPF package. You must pass a list of models obtained from the RPF package in the 'ItemSpec' argument. All item models must use the same number of latent factors although some of these factor loadings can be constrained to zero in the item parameter matrix. The 'item' matrix contains item parameters with one item per column in the same order at ItemSpec.

The 'qpoints' and 'qwidth' argument control the fineness and width, respectively, of the equal-interval quadrature grid. The integer 'qpoints' is the number of points per dimension. The quadrature extends from negative qwidth to positive qwidth for each dimension. Since the latent distribution defaults to standard Normal, qwidth can be regarded as a value in Z-score units.

The optional 'mean' and 'cov' arguments permit modeling of the latent distribution in multigroup models (in a single group, the latent distribution must be fixed). A separate latent covariance model is used in combination with mxExpectationBA81. The point mass distribution contained in the quadrature is converted into a multivariate Normal distribution by mxDataDynamic. Typically mxExpectationNormal is used to fit a multivariate Normal model to these data. Some intricate programming is required. Examples are given in the manual. mxExpectationBA81 uses a sample size of N for the covariance matrix. This differs from mxExpectationNormal which uses a sample size of $N - 1$.

The 'verbose' argument enables diagnostics that are mainly of interest to developers.

When a two-tier covariance matrix is recognized, this expectation automatically enables analytic dimension reduction (Cai, 2010).

The optional 'weightColumn' is superseded by the weight argument in mxData. For data with many repeated response patterns, model evaluation time can be reduced. An easy way to transform your data into this form is to use compressDataFrame. Non-integer weights are supported except for EAPscores.

mxExpectationBA81 requires mxComputeEM. During a typical optimization run, latent abilities are assumed for examinees during the E-step. These examinee scores are implied by the previous iteration's parameter vector. This can be overridden using the 'EstepItem' argument. This is mainly of use to developers for checking item parameter derivatives.

Common univariate priors are available from univariatePrior. The standard Normal distribution of the quadrature acts like a prior distribution for difficulty. It is not necessary to impose any additional Bayesian prior on difficulty estimates (Baker & Kim, 2004, p. 196).

Many estimators are available for standard errors. Oakes is recommended (see mxComputeEM). Also available are Supplement EM (mxComputeEM), Richardson extrapolation (mxComputeNumericDeriv), likelihood-based confidence intervals (mxCI), and the covariance of the rowwise gradients.

References

Bock, R. D., & Aitkin, M. (1981). Marginal maximum likelihood estimation of item parameters: Application of an EM algorithm. *Psychometrika, 46*, 443-459.

Cai, L. (2010). A two-tier full-information item factor analysis model with applications. *Psychometrika, 75*, 581-612.

Pritikin, J. N., Hunter, M. D., & Boker, S. M. (2015). Modular open-source software for Item Factor Analysis. *Educational and Psychological Measurement, 75*(3), 458-474

Pritikin, J. N. & Schmidt, K. M. (in press). Model builder for Item Factor Analysis with OpenMx. *R Journal*.

Seong, T. J. (1990). Sensitivity of marginal maximum likelihood estimation of item and ability parameters to the characteristics of the prior ability distributions. *Applied Psychological Measurement, 14*(3), 299-311.

See Also

RPF

Examples

```
library(OpenMx)
library(rpf)

numItems <- 14

# Create item specifications
spec <- list()
for (ix in 1:numItems) { spec[[ix]] <- rpf.grm(outcomes=sample(2:7, 1)) }
names(spec) <- paste("i", 1:numItems, sep="")

# Generate some random "true" parameter values
correct.mat <- mxSimplify2Array(lapply(spec, rpf.rparam))

# Generate some example data
data <- rpf.sample(500, spec, correct.mat)

# Create a matrix of item parameters with starting values
imat <- mxMatrix(name="item",
                 values=mxSimplify2Array(lapply(spec, rpf.rparam)))
rownames(imat)[1] <- 'f1'
imat$free[!is.na(correct.mat)] <- TRUE
imat$values[!imat$free] <- NA

# Create a compute plan
plan <- mxComputeSequence(list(
  mxComputeEM('expectation', 'scores',
              mxComputeNewtonRaphson(), information="oakes1999",
              infoArgs=list(fitfunction='fitfunction')),
  mxComputeHessianQuality(),
  mxComputeStandardError(),
  mxComputeReportDeriv()))

# Build the OpenMx model
grmModel <- mxModel(model="grm1", imat,
                    mxData(observed=data, type="raw"),
                    mxExpectationBA81(ItemSpec=spec),
                    mxFitFunctionML(),
                    plan)

grmModel <- mxRun(grmModel)
summary(grmModel)
```

mxExpectationGREML *Create MxExpectationGREML Object*

Description

This function creates a new `MxExpectationGREML` object.

Usage

```
mxExpectationGREML(V, yvars=character(0), Xvars=list(), addOnes=TRUE, blockByPheno=TRUE,
             staggerZeroes=TRUE, dataset.is.yX=FALSE, casesToDropFromV=integer(0))
```

Arguments

`V`	Character string; the name of the `MxAlgebra` or `MxMatrix` to serve as the 'V' matrix (the model-expected covariance matrix). Internally, the 'V' matrix is assumed to be symmetric, and its elements above the main diagonal are ignored.
`yvars, Xvars, addOnes, blockByPheno, staggerZeroes`	Passed to `mxGREMLDataHandler()`.
`dataset.is.yX`	Logical; defaults to `FALSE`. If `TRUE`, then the first column of the raw dataset is taken as-is to be the 'y' phenotype vector, and the remaining columns are taken as-is to be the 'X' matrix of covariates. In this case, `mxGREMLDataHandler()` is never internally called at runtime, and all other arguments besides `V` and `casesToDropFromV` are ignored.
`casesToDropFromV`	Integer vector. Its elements are the numbers of the rows and columns of covariance matrix 'V' to be dropped at runtime, usually because they correspond to rows of 'y' or 'X' that contained missing observations. By default, no cases are dropped from 'V.' Ignored unless `dataset.is.yX=TRUE`.

Details

"GREML" stands for "genomic-relatedness-matrix restricted maximum-likelihood." In the strictest sense of the term, it refers to genetic variance-component estimation from matrices of subjects' pairwise degree of genetic relatedness, as calculated from genome-wide marker data. It is from this original motivation that some of the terminology originates, such as calling 'y' the "phenotype" vector. However, OpenMx's implementation of GREML is applicable for analyses from any subject-matter domain, and in which the following assumptions are reasonable:

1. Conditional on 'X' (the covariates), the phenotype vector (response variable) 'y' is a single realization from a multivariate-normal distribution having (in general) a dense covariance matrix, 'V.'
2. The parameters of the covariance matrix, such as variance components, are of primary interest.
3. The random effects are normally distributed.
4. Weighted least-squares regression, using the inverse of 'V' as a weight matrix, is an adequate model for the phenotypic means. Note that the regression coefficients are not actually free parameters to be numerically optimized.

Computationally, the chief distinguishing feature of an OpenMx GREML analysis is that the phenotype vector, 'y,' is a single realization of a random vector that, in general, cannot be partitioned into independent subvectors. For this reason, definition variables are not compatible (and should be unnecessary with) GREML expectation. GREML expectation can still be used if the covariance matrix is sparse, but as of this writing, OpenMx does not take advantage of the sparseness to improve performance. Because of the limitations of restricted maximum likelihood, GREML expectation is presently incompatible with ordinal variables.

Value

Returns a new object of class `MxExpectationGREML`.

References

Kirkpatrick RM, Pritikin JN, Hunter MD, & Neale, MC. (2021). Combining structural-equation modeling with genomic-relatedness matrix restricted maximum likelihood in OpenMx. In press at Behavior Genetics. https://doi.org/10.1007/s10519-020-10037-5

One of the first uses of the acronym "GREML":
Benjamin DJ, Cesarini D, van der Loos MJHM, Dawes CT, Koellinger PD, et al. (2012) The genetic architecture of economic and political preferences. Proceedings of the National Academy of Sciences 109: 8026-8031. doi: 10.1073/pnas.1120666109

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

See `MxExpectationGREML` for the S4 class created by `mxExpectationGREML()`. More information about the OpenMx package may be found here.

Examples

```
dat <- cbind(rnorm(100),rep(1,100))
colnames(dat) <- c("y","x")

ge <- mxExpectationGREML(V="V",yvars="y",Xvars=list("X"),addOnes=FALSE)
gff <- mxFitFunctionGREML(dV=c(ve="I"))
plan <- mxComputeSequence(freeSet=c("Ve"),steps=list(
  mxComputeNewtonRaphson(fitfunction="fitfunction"),
  mxComputeOnce('fitfunction',
    c('fit','gradient','hessian','ihessian')),
  mxComputeStandardError(),
  mxComputeReportDeriv(),
  mxComputeReportExpectation()
))

testmod <- mxModel(
  "GREMLtest",
  mxData(observed = dat, type="raw"),
  mxMatrix(type = "Full", nrow = 1, ncol=1, free=TRUE,
    values = 1, labels = "ve", lbound = 0.0001, name = "Ve"),
  mxMatrix("Iden",nrow=100,name="I",condenseSlots=TRUE),
  mxAlgebra(I %x% Ve,name="V"),
  ge,
  gff,
  plan
)
str(testmod)
```

MxExpectationGREML-class

Class "MxExpectationGREML"

Description

`MxExpectationGREML` is a type of expectation class. It contains the necessary elements for specifying a GREML model. For more information, see `mxExpectationGREML()`.

Objects from the Class

Objects can be created by calls of the form `mxExpectationGREML(V,yvars,Xvars,addOnes,blockByPheno,staggerZeroe`

Slots

- `V`: Object of class `"MxCharOrNumber"`. Identifies the `MxAlgebra` or `MxMatrix` to serve as the 'V' matrix.
- `yvars`: Character vector. Each string names a column of the raw dataset, to be used as a phenotypes.
- `Xvars`: A list of data column names, specifying the covariates to be used with each phenotype.
- `addOnes`: Logical; pertains to data-handling at runtime.
- `blockByPheno`: Logical; pertains to data-handling at runtime.
- `staggerZeroes`: Logical; pertains to data-handling at runtime.
- `dataset.is.yX`: Logical; pertains to data-handling at runtime.
- `y`: Object of class `"MxData"`. Its `observed` slot will contain the phenotype vector, 'y.'
- `X`: A matrix, to contain the 'X' matrix of covariates.
- `yXcolnames`: Character vector; used to store the column names of 'y' and 'X.'
- `casesToDrop`: Integer vector, specifying the rows and columns of the 'V' matrix to be removed at runtime.
- `b`: A matrix, to contain the vector of regression coefficients calculated at runtime.
- `bcov`: A matrix, to contain the sampling covariance matrix of the regression coefficients calculated at runtime.
- `numFixEff`: Integer number of covariates in 'X.'
- `dims`: Object of class `"character"`.
- `numStats`: Numeric; number of observed statistics.
- `dataColumns`: Object of class `"numeric"`.
- `name`: Object of class `"character"`.
- `data`: Object of class `"MxCharOrNumber"`.
- `.runDims`: Object of class `"character"`.

Extends

Class "MxBaseExpectation", directly. Class "MxBaseNamed", by class "MxBaseExpectation", distance 2. Class "MxExpectation", by class "MxBaseExpectation", distance 2.

Methods

No methods defined with class "MxExpectationGREML" in the signature.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation.

See Also

See mxExpectationGREML() for creating MxExpectationGREML objects, and for more information generally concerning GREML analyses, including a complete example. More information about the OpenMx package may be found here.

Examples

```
showClass("MxExpectationGREML")
```

---

mxExpectationHiddenMarkov

Hidden Markov expectation

Description

Used in conjunction with mxFitFunctionML, this expectation can express a mixture model (with the transition matrix omitted) or a Hidden Markov model.

Usage

```
mxExpectationHiddenMarkov(components, initial="initial", transition=NULL,
      ..., verbose=0L, scale=c('softmax', 'sum', 'none'))
```

Arguments

components	A character vector of model names.
initial	The name of the matrix or algebra column that specifies the initial probabilities.
transition	The name of the matrix or algebra that specifies the left stochastic transition probabilities.
...	Not used. Forces remaining arguments to be specified by name.
verbose	the level of runtime diagnostics
scale	How the probabilities are rescaled. For 'softmax', the coefficient-wise exponential is taken and then each column is divided by its column sum. For 'sum', each column is divided by its column sum. For 'none', no scaling is done.

Details

The initial probabilities given in `initial` must sum to one. So too must the *columns* of the transition matrix given in `transition`. The transitions go from a column to a row, similar to how regression effects in the RAM structural equation models go from the column variable to the row variable. This means `transition` is a left stochastic matrix.

For ease of use the raw free parameters of these matrices are rescaled by OpenMx according to the `scale` argument. When `scale` is set to "softmax" the softmax function is applied to the initial probabilities and the columns of the transition matrix. The softmax function is also sometimes called multinomial logistic regression. Softmax exponentiates each element in a vector and then divides each element by the sum of the exponentiated elements. In equation form the softmax function is

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{K}} e^{x_k}$$

When using the softmax scaling no free parameter bounds or constraints are needed. However, for model identification, one element of the initial probabilities vector must be fixed. If the softmax scaling is used, then the usual choice for the fixed parameter value is zero. The regime (or latent class or mixture component) that has its initial probability set to zero becomes the comparison against which other probabilities are evaluated. Likewise for model identification, one element in each column of the transition matrix must be fixed. When the softmax scaling is used, the typical choice is to fix one element in each column to zero. Generally, one row of the transition matrix is fixed to zero, or the diagonal elements of the transition matrix are fixed to zero.

When `scale` is set to "sum" then each element of the initial probabilities and each column of the transition matrix is internally divided by its sum. When using the sum scaling, the same model identification requirements are present. In particular, one element of the initial probabilities must be fixed and one element in each column of the transition matrix must be fixed. The typical value to fix these values at for sum scaling is one. Additionally when using sum scaling, all free parameters in the initial and transition probabilities must have lower bounds of zero. In equation form the sum scaling does the following:

$$sumscale(x_i) = \frac{x_i}{\sum_{k=1}^{K}} x_k$$

When `scale` is set to "none" then no re-scaling is done. The parameters of `initial` and `transition` are left "as is". This can be dangerous and is not recommended for novice users. It might produce nonsensical results particularly for hidden Markov models. However, some advanced users may find no scaling to be advantageous for certain applications (e.g., they are providing their own scaling), and thus it is provided as an option.

Parameters are estimated in the given scale. To obtain the initial column vector and left stochastic transition matrix in probability units then examine the expectation's `output` slot with for example `yourModel$expectation$output`

Definition variables can be used to assign a separate set of mixture probabilities to each row of data. Definition variables can be used in the initial column vector or in the transition matrix, but not in both at the same time.

Note that, when the transition matrix is omitted, this expectation is the same as mxExpectationMixture. mxGenerateData is not implemented for this type of expectation.

Examples

```
library(OpenMx)

start_prob <- c(.2,.4,.4)
transition_prob <- matrix(c(.8, .1, .1,
.3, .6, .1,
.1, .3, .6), 3, 3)
noise <- .5

  # simulate a trajectory
  state <- sample.int(3, 1, prob=transition_prob %*% start_prob)
  trail <- c(state)

  for (rep in 1:500) {
    state <- sample.int(3, 1, prob=transition_prob[,state])
    trail <- c(trail, state)
  }

  # add noise
  trailN <- sapply(trail, function(v) rnorm(1, mean=v, sd=sqrt(noise)))

  classes <- list()

  for (cl in 1:3) {
    classes[[cl]] <- mxModel(paste0("cl", cl), type="RAM",
                             manifestVars=c("ob"),
                             mxPath("one", "ob", value=cl, free=FALSE),
                             mxPath("ob", arrows=2, value=noise, free=FALSE),
                             mxFitFunctionML(vector=TRUE))
  }

  m1 <-
    mxModel("hmm", classes,
            mxData(data.frame(ob=trailN), "raw"),
            mxMatrix(nrow=3, ncol=1,
                     labels=paste0('i',1:3), name="initial"),
            mxMatrix(nrow=length(classes), ncol=length(classes),
                     labels=paste0('t', 1:(length(classes) * length(classes))),
                     name="transition"),
            mxExpectationHiddenMarkov(
              components=sapply(classes, function(m) m$name),
              initial="initial",
              transition="transition", scale="softmax"),
            mxFitFunctionML())

  m1$transition$free[1:(length(classes)-1), 1:length(classes)] <- TRUE

m1 <- mxRun(m1)

summary(m1)

print(m1$expectation$output)
```

mxExpectationLISREL *Create MxExpectationLISREL Object*

Description

This function creates a new MxExpectationLISREL object.

Usage

```
mxExpectationLISREL(LX=NA, LY=NA, BE=NA, GA=NA, PH=NA, PS=NA, TD=NA, TE=NA, TH=NA,
                    TX = NA, TY = NA, KA = NA, AL = NA,
                    dimnames = NA, thresholds = NA,
 threshnames = deprecated(), verbose=0L, ...,
  expectedCovariance=NULL, expectedMean=NULL, discrete = as.character(NA))
```

Arguments

`LX`	An optional character string indicating the name of the 'LX' matrix.
`LY`	An optional character string indicating the name of the 'LY' matrix.
`BE`	An optional character string indicating the name of the 'BE' matrix.
`GA`	An optional character string indicating the name of the 'GA' matrix.
`PH`	An optional character string indicating the name of the 'PH' matrix.
`PS`	An optional character string indicating the name of the 'PS' matrix.
`TD`	An optional character string indicating the name of the 'TD' matrix.
`TE`	An optional character string indicating the name of the 'TE' matrix.
`TH`	An optional character string indicating the name of the 'TH' matrix.
`TX`	An optional character string indicating the name of the 'TX' matrix.
`TY`	An optional character string indicating the name of the 'TY' matrix.
`KA`	An optional character string indicating the name of the 'KA' matrix.
`AL`	An optional character string indicating the name of the 'AL' matrix.
`dimnames`	An optional character vector that is currently ignored
`thresholds`	An optional character string indicating the name of the thresholds matrix.
`threshnames`	**[Deprecated]**
`verbose`	integer. Level of runtime diagnostic output.
`...`	Not used. Forces remaining arguments to be specified by name.
`expectedCovariance`	An optional character string indicating the name of a matrix for the model implied covariance.
`expectedMean`	An optional character string indicating the name of a matrix for the model implied mean.
`discrete`	An optional character string indicating the name of the discrete matrix.

Details

Expectation functions define the way that model expectations are calculated. The mxExpectation-LISREL calculates the expected covariance and means of a given MxData object given a LISREL model. This model is defined by LInear Structural RELations (LISREL; Jöreskog & Sörbom, 1982, 1996). Arguments 'LX' through 'AL' must refer to MxMatrix objects with the associated properties of their respective matrices in the LISREL modeling approach.

The full LISREL specification has 13 matrices and is sometimes called the extended LISREL model. It is defined by the following equations.

$$\eta = \alpha + B\eta + \Gamma\xi + \zeta$$

$$y = \tau_y + \Lambda_y\eta + \epsilon$$

$$x = \tau_x + \Lambda_x\xi + \delta$$

The table below is provided as a quick reference to the numerous matrices in LISREL models. Note that NX is the number of manifest exogenous (independent) variables, the number of Xs. NY is the number of manifest endogenous (dependent) variables, the number of Ys. NK is the number of latent exogenous variables, the number of Ksis or Xis. NE is the number of latent endogenous variables, the number of etas.

Matrix	Word	Abbreviation	Dimensions	Expression	Description
Λ_x	Lambda x	LX	NX x NK		Exogenous Factor Loading Matrix
Λ_y	Lambda y	LY	NY x NE		Endogenous Factor Loading Matrix
B	Beta	BE	NE x NE		Regressions of Latent Endogenous Variables Pre
Γ	Gamma	GA	NE x NK		Regressions of Latent Exogenous Variables Predi
Φ	Phi	PH	NK x NK	cov(ξ)	Covariance Matrix of Latent Exogenous Variable
Ψ	Psi	PS	NE x NE	cov(ζ)	Residual Covariance Matrix of Latent Endogenou
Θ_δ	Theta delta	TD	NX x NX	cov(δ)	Residual Covariance Matrix of Manifest Exogen
Θ_ϵ	Theta epsilon	TE	NY x NY	cov(ϵ)	Residual Covariance Matrix of Manifest Endogen
$\Theta_{\delta\epsilon}$	Theta delta epsilson	TH	NX x NY	cov(δ, ϵ)	Residual Covariance Matrix of Manifest Exogen
τ_x	tau x	TX	NX x 1		Residual Means of Manifest Exogenous Variable
τ_y	tau y	TY	NY x 1		Residual Means of Manifest Endogenous Variabl
κ	kappa	KA	NK x 1	mean(ξ)	Means of Latent Exogenous Variables
α	alpha	AL	NE x 1		Residual Means of Latent Endogenous Variables

From the extended LISREL model, several submodels can be defined. Subtypes of the LISREL model are defined by setting some of the arguments of the LISREL expectation function to NA. Note that because the default values of each LISREL matrix is NA, setting a matrix to NA can be accomplished by simply not giving it any other value.

The first submodel is the LISREL model without means.

$$\eta = B\eta + \Gamma\xi + \zeta$$

$$y = \Lambda_y\eta + \epsilon$$

$$x = \Lambda_x\xi + \delta$$

The LISREL model without means requires 9 matrices: LX, LY, BE, GA, PH, PS, TD, TE, and TH. Hence this LISREL model has TX, TY, KA, and AL as NA. This can be accomplished be leaving these matrices at their default values.

The TX, TY, KA, and AL matrices must be specified if either the mxData type is "cov" or "cor" and a means vector is provided, or if the mxData type is "raw". Otherwise the TX, TY, KA, and AL matrices are ignored and the model without means is estimated.

A second submodel involves only endogenous variables.

$$\eta = B\eta + \zeta$$

$$y = \Lambda_y \eta + \epsilon$$

The endogenous-only LISREL model requires 4 matrices: LY, BE, PS, and TE. The LX, GA, PH, TD, and TH must be NA in this case. However, means can also be specified, allowing TY and AL if the data are raw or if observed means are provided.

Another submodel involves only exogenous variables.

$$x = \Lambda_x \xi + \delta$$

The exogenous-model model requires 3 matrices: LX, PH, and TD. The LY, BE, GA, PS, TE, and TH matrices must be NA. However, means can also be specified, allowing TX and KA if the data are raw or if observed means are provided.

The model that is run depends on the matrices that are not NA. If all 9 matrices are not NA, then the full model is run. If only the 4 endogenous matrices are not NA, then the endogenous-only model is run. If only the 3 exogenous matrices are not NA, then the exogenous-only model is run. If some endogenous and exogenous matrices are not NA, but not all of them, then appropriate errors are thrown. Means are included in the model whenever their matrices are provided.

The MxMatrix objects included as arguments may be of any type, but should have the properties described above. The mxExpectationLISREL will not return an error for incorrect specification, but incorrect specification will likely lead to estimation problems or errors in the mxRun function.

Like the mxExpectationRAM, the mxExpectationLISREL evaluates with respect to an MxData object. The MxData object need not be referenced in the mxExpectationLISREL function, but must be included in the MxModel object. mxExpectationLISREL requires that the 'type' argument in the associated MxData object be equal to 'cov', 'cor', or 'raw'.

To evaluate, place mxExpectationLISREL objects, the mxData object for which the expected covariance approximates, referenced MxAlgebra and MxMatrix objects, and optional MxBounds and MxConstraint objects in an MxModel object. This model may then be evaluated using the mxRun function. The results of the optimization can be found in the 'output' slot of the resulting model, and may be obtained using the mxEval function.

Value

Returns a new MxExpectationLISREL object. One and only one MxExpectationLISREL object can be included with models using one and only one fit function object (e.g., MxFitFunctionML) and with referenced MxAlgebra, MxData and MxMatrix objects.

References

Jöreskog, K. G. & Sörbom, D. (1996). LISREL 8: User's Reference Guide. Lincolnwood, IL: Scientific Software International.

Jöreskog, K. G. & Sörbom, D. (1982). Recent developments in structural equation modeling. *Journal of Marketing Research, 19,* 404-416.

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

`demo("LISRELJointFactorModel")`

Examples

```
# Create and fit a model using mxExpectationLISREL, and mxFitFunctionML

library(OpenMx)

vNames <- paste("v",as.character(1:6),sep="")
dimList <- list(vNames, vNames)
covData <- matrix(
  c(0.9223099, 0.1862938, 0.4374359, 0.8959973, 0.9928430, 0.5320662,
    0.1862938, 0.2889364, 0.3927790, 0.3321639, 0.3371594, 0.4476898,
    0.4374359, 0.3927790, 1.0069552, 0.6918755, 0.7482155, 0.9013952,
    0.8959973, 0.3321639, 0.6918755, 1.8059956, 1.6142005, 0.8040448,
    0.9928430, 0.3371594, 0.7482155, 1.6142005, 1.9223567, 0.8777786,
    0.5320662, 0.4476898, 0.9013952, 0.8040448, 0.8777786, 1.3997558
    ), nrow=6, ncol=6, byrow=TRUE, dimnames=dimList)

# Create LISREL matrices

mLX <- mxMatrix("Full", values=c(.5, .6, .8, rep(0, 6), .4, .7, .5),
          name="LX", nrow=6, ncol=2,
          free=c(TRUE,TRUE,TRUE,rep(FALSE, 6),TRUE,TRUE,TRUE),
          dimnames=list(vNames, c("x1","x2")))
mTD <- mxMatrix("Diag", values=c(rep(.2, 6)),
          name="TD", nrow=6, ncol=6, free=TRUE,
          dimnames=dimList)
mPH <- mxMatrix("Symm", values=c(1, .3, 1),
          name="PH", nrow=2, ncol=2, free=c(FALSE, TRUE, FALSE),
          dimnames=list(c("x1","x2"),c("x1","x2")))

# Create a LISREL expectation with LX, TD, and PH matrix names

expFunction <- mxExpectationLISREL(LX="LX", TD="TD", PH="PH")

# Create fit function and data

tmpData <- mxData(observed=covData, type="cov", numObs=100)
fitFunction <- mxFitFunctionML()

# Create the model, fit it, and print a summary.
```

```
tmpModel <- mxModel(model="exampleModel",
                    mLX, mTD, mPH, expFunction, fitFunction, tmpData)
tmpModelOut <- mxRun(tmpModel)
summary(tmpModelOut)


#---------------------------------------
# Fit factor model with means

require(OpenMx)

data(demoOneFactor)
nvar <- ncol(demoOneFactor)
varnames <- colnames(demoOneFactor)

factorMeans <- mxMatrix("Zero", 1, 1, name="Kappa",
                        dimnames=list("F1", NA))
xIntercepts <- mxMatrix("Full", nvar, 1, free=TRUE, name="TauX",
                        dimnames=list(varnames, NA))
factorLoadings <- mxMatrix("Full", nvar, 1, TRUE, .6, name="LambdaX",
                           labels=paste("lambda", 1:nvar, sep=""),
                           dimnames=list(varnames, "F1"))
factorCovariance <- mxMatrix("Diag", 1, 1, FALSE, 1, name="Phi")
xResidualVariance <- mxMatrix("Diag", nvar, nvar, TRUE, .2, name="ThetaDelta",
                              labels=paste("theta", 1:nvar, sep=""))

liModel <- mxModel(model="LISREL Factor Model",
factorMeans, xIntercepts, factorLoadings,
factorCovariance, xResidualVariance,
mxExpectationLISREL(LX="LambdaX", PH="Phi",
TD="ThetaDelta", TX="TauX", KA="Kappa"),
mxFitFunctionML(),
mxData(cov(demoOneFactor), "cov",
means=colMeans(demoOneFactor), numObs=nrow(demoOneFactor))
)

liRun <- mxRun(liModel)

summary(liRun)
```

`mxExpectationMixture` *Mixture expectation*

Description

Used in conjunction with mxFitFunctionML, this expectation can express a mixture model.

Usage

```
mxExpectationMixture(components, weights="weights",
      ..., verbose=0L, scale=c('softmax', 'sum', 'none'))
```

Arguments

`components`	A character vector of model names.
`weights`	The name of the matrix or algebra column that specifies the component weights.
`...`	Not used. Forces remaining arguments to be specified by name.
`verbose`	the level of runtime diagnostics
`scale`	How the probabilities are rescaled. For 'softmax', the coefficient-wise exponential is taken and then each column is divided by its column sum. For 'sum', each column is divided by its column sum. For 'none', no scaling is done.

Details

The mixture probabilities given in `weights` must sum to one. As such for K mixture components, only $K - 1$ of the elements of `weights` can be estimated. The mixture probabilities in `weights` should be a column vector (i.e., a K by 1 matrix, or algebra with a K by 1 result).

For ease of use the raw free parameters of weights can be rescaled by OpenMx according to the `scale` argument. When `scale` is set to "softmax" the softmax function is applied to the weights. The softmax function is also sometimes called multinomial logistic regression. Softmax exponentiates each element in a vector and then divides each element by the sum of the exponentiated elements. In equation form the softmax function is

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{K}} e^{x_k}$$

When using the softmax scaling no free parameter bounds or constraints are needed. However, for model identification, one element of the weights vector must be fixed. If the softmax scaling is used, then the usual choice for the fixed parameter value is zero. The latent class or mixture component that has its raw weight set to zero becomes the comparison against which other probabilities are evaluated.

When `scale` is set to "sum" then each element of the weights matrix is internally divided by its sum. When using the sum scaling, the same model identification requirements are present. In particular, one element of the weights must be fixed. The typical value to fix this value at for sum scaling is one. Additionally when using sum scaling, all free parameters in the weights must have lower bounds of zero. In equation form the sum scaling does the following:

$$sumscale(x_i) = \frac{x_i}{\sum_{k=1}^{K}} x_k$$

When `scale` is set to "none" then no re-scaling is done. The weights are left "as is". This can be dangerous and is not recommended for novice users. However, some advanced users may find no scaling to be advantageous for certain applications (e.g., they are providing their own scaling), and thus it is provided as an option.

Parameters are estimated in the given scale. To obtain the weights column vector, examine the expectation's `output` slot with for example `yourModel$expectation$output`

An extension of this expectation to a Hidden Markov model is available with mxExpectationHiddenMarkov. mxGenerateData is not implemented for this type of expectation.

Examples

```
library(OpenMx)

set.seed(1)

trail <- c(rep(1,480), rep(2,520))
trailN <- sapply(trail, function(v) rnorm(1, mean=v))

classes <- list()

for (cl in 1:2) {
  classes[[cl]] <- mxModel(paste0("class", cl), type="RAM",
                           manifestVars=c("ob"),
                           mxPath("one", "ob", value=cl, free=FALSE),
                           mxPath("ob", arrows=2, value=1, free=FALSE),
                           mxFitFunctionML(vector=TRUE))
}

mix1 <- mxModel(
  "mix1", classes,
  mxData(data.frame(ob=trailN), "raw"),
  mxMatrix(values=1, nrow=1, ncol=2, free=c(FALSE,TRUE), name="weights"),
  mxExpectationMixture(paste0("class",1:2), scale="softmax"),
  mxFitFunctionML())

mix1Fit <- mxRun(mix1)
```

`mxExpectationNormal` *Create MxExpectationNormal Object*

Description

This function creates an MxExpectationNormal object.

Usage

```
mxExpectationNormal(covariance, means=NA, dimnames = NA, thresholds = NA,
                    threshnames = dimnames, ...,
  discrete = as.character(NA), discreteSpec=NULL)
```

Arguments

`covariance`	A character string indicating the name of the expected covariance algebra.
`means`	A character string indicating the name of the expected means algebra.
`dimnames`	An optional character vector to be assigned to the dimnames of the covariance and means algebras.
`thresholds`	An optional character string indicating the name of the thresholds matrix.
`threshnames`	An optional character vector to be assigned to the column names of the thresholds matrix.
`...`	Not used. Forces remaining arguments to be specified by name.
`discrete`	An optional character string indicating the name of the discrete matrix.
`discreteSpec`	An optional matrix containing maximum counts and model IDs.

Details

Expectation functions define the way that model expectations are calculated. The mxExpectationNormal function uses the algebra defined by the 'covariance' and 'means' arguments to define the expected covariance and means under the assumption of multivariate normality. The 'covariance' argument takes an MxAlgebra object, which defines the expected covariance of an associated MxData object. The 'means' argument takes an MxAlgebra object, which defines the expected means of an associated MxData object. The 'dimnames' arguments takes an optional character vector. If this argument is not a single NA, then this vector is used to assign the dimnames of the means vector as well as the row and columns dimnames of the covariance matrix.

thresholds: The name of the thresholds matrix. When needed (for modelling ordinal data), this matrix should be created using mxMatrix(). The thresholds matrix must have as many columns as there are ordinal variables in the model, and number of rows equal to one fewer than the maximum number of levels found in the ordinal variables. The starting values of this matrix must also be set to reasonable values. Fill each column with a set of ordered start thresholds, one for each level of this column's factor levels minus 1. These thresholds may be free if you wish them to be estimated, or fixed. The unused rows in each column, if any, can be set to any value including NA.

threshnames: A character vector consisting of the variables in the thresholds matrix, i.e., the names of ordinal variables in a model. This is necessary for OpenMx to map the thresholds matrix columns onto the variables in your data. If you set the dimnames of the columns in the thresholds matrix then threshnames is not needed.

Usage Notes: dimnames must be supplied where the matrices referenced by the covariance and means algebras are not themselves labeled. Failure to do so leads to an error noting that the covariance or means matrix associated with the FIML objective does not contain dimnames.

mxExpectationNormal evaluates with respect to an MxData object. The MxData object need not be referenced in the mxExpectationNormal function, but must be included in the MxModel object. When the 'type' argument in the associated MxData object is equal to 'raw', missing values are permitted in the associated MxData object.

To evaluate, place an mxExpectationNormal object, the mxData object for which the expected covariance approximates, referenced MxAlgebra and MxMatrix objects, optional MxBounds or MxConstraint objects, and an mxFitFunction such as mxFitFunctionML in an MxModel object. This model may then be evaluated using the mxRun function.

The results of the optimization can be reported using the summary function, or accessed directly in the 'output' slot of the resulting model (i.e., modelName$output). Components of the output may be referenced using the Extract functionality.

Value

Returns an MxExpectationNormal object.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
# Create and fit a model using mxMatrix, mxAlgebra,
#  mxExpectationNormal, and mxFitFunctionML

library(OpenMx)

# Simulate some data

x=rnorm(1000, mean=0, sd=1)
y= 0.5*x + rnorm(1000, mean=0, sd=1)
tmpFrame <- data.frame(x, y)
tmpNames <- names(tmpFrame)

# Define the matrices

M <- mxMatrix(type = "Full", nrow = 1, ncol = 2, values=c(0,0),
              free=c(TRUE,TRUE), labels=c("Mx", "My"), name = "M")
S <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(1,0,0,1),
              free=c(TRUE,FALSE,FALSE,TRUE), labels=c("Vx", NA, NA, "Vy"),
              name = "S")
A <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(0,1,0,0),
              free=c(FALSE,TRUE,FALSE,FALSE), labels=c(NA, "b", NA, NA),
              name = "A")
I <- mxMatrix(type="Iden", nrow=2, ncol=2, name="I")

# Define the expectation

expCov <- mxAlgebra(solve(I-A) %*% S %*% t(solve(I-A)), name="expCov")
expFunction <- mxExpectationNormal(covariance="expCov", means="M",
dimnames=tmpNames)

# Choose a fit function

fitFunction <- mxFitFunctionML()

# Define the model

tmpModel <- mxModel(model="exampleModel", M, S, A, I,
                    expCov, expFunction, fitFunction,
                        mxData(observed=tmpFrame, type="raw"))

# Fit the model and print a summary

tmpModelOut <- mxRun(tmpModel)
summary(tmpModelOut)
```

`mxExpectationRAM` *Create an MxExpectationRAM Object*

Description

This function creates an MxExpectationRAM object.

Usage

```
mxExpectationRAM(A="A", S="S", F="F", M = NA, dimnames = NA, thresholds = NA,
                 threshnames = dimnames, ..., between=NULL, verbose=0L,
  .useSparse=NA, expectedCovariance=NULL, expectedMean=NULL,
  discrete = as.character(NA), selectionVector = as.character(NA),
  expectedFullCovariance=NULL, expectedFullMean=NULL)
```

Arguments

`A`	A character string indicating the name of the 'A' matrix.
`S`	A character string indicating the name of the 'S' matrix.
`F`	A character string indicating the name of the 'F' matrix.
`M`	An optional character string indicating the name of the 'M' matrix.
`dimnames`	An optional character vector to be assigned to the column names of the 'F' and 'M' matrices.
`thresholds`	An optional character string indicating the name of the thresholds matrix.
`threshnames`	An optional character vector to be assigned to the column names of the thresholds matrix.
`...`	Not used. Forces remaining arguments to be specified by name.
`between`	A character vector of matrices that specify cross model relationships.
`verbose`	integer. Level of runtime diagnostic output.
`.useSparse`	logical. Whether to use sparse matrices to compute the expectation. The default `NA` allows the backend to decide.
`expectedCovariance`	An optional character string indicating the name of a matrix for the observed model implied covariance.
`expectedMean`	An optional character string indicating the name of a matrix for the observed model implied mean.

`discrete`	An optional character string indicating the name of the discrete matrix.
`selectionVector`	An optional character string indicating the name of the Pearson selection vector matrix.
`expectedFullCovariance`	An optional character string indicating the name of a matrix for the full model implied covariance. Both latent and observed variables are included.
`expectedFullMean`	An optional character string indicating the name of a matrix for the full model implied mean. Both latent and observed variables are included.

Details

Expectation functions define the way that model expectations are calculated. The mxExpectation-RAM calculates the expected covariance and means of a given MxData object given a RAM model. This model is defined by reticular action modeling (McArdle and McDonald, 1984). The 'A', 'S', and 'F' arguments refer to MxMatrix objects with the associated properties of the A, S, and F matrices in the RAM modeling approach. *Note for advanced users*: these matrices may be replaced by mxAlgebras. Such a model will lack properties (labels, free, bounds) that other functions may be expecting.

The MxMatrix objects included as arguments may be of any type, but should have the properties described above. The mxExpectationRAM will not return an error for incorrect specification, but incorrect specification will likely lead to estimation problems or errors in the mxRun function.

The 'A' argument refers to the A or asymmetric matrix in the RAM approach. This matrix consists of all of the asymmetric paths (one-headed arrows) in the model. A free parameter in any row and column describes a regression of the variable represented by that row regressed on the variable represented in that column.

The 'S' argument refers to the S or symmetric matrix in the RAM approach, and as such must be square. This matrix consists of all of the symmetric paths (two-headed arrows) in the model. A free parameter in any row and column describes a covariance between the variable represented by that row and the variable represented by that column. Variances are covariances between any variable at itself, which occur on the diagonal of the specified matrix.

The 'F' argument refers to the F or filter matrix in the RAM approach. If no latent variables are included in the model (i.e., the A and S matrices are of both of the same dimension as the data matrix), then the 'F' should refer to an identity matrix. If latent variables are included (i.e., the A and S matrices are not of the same dimension as the data matrix), then the 'F' argument should consist of a horizontal adhesion of an identity matrix and a matrix of zeros.

The 'M' argument refers to the M or means matrix in the RAM approach. It is a 1 x n matrix, where n is the number of manifest variables + the number of latent variables. The M matrix must be specified if either the mxData type is “cov” or “cor” and a means vector is provided, or if the mxData type is “raw”. Otherwise the M matrix is ignored.

The 'dimnames' arguments takes an optional character vector. If this argument is not a single NA, then this vector be assigned to be the column names of the 'F' matrix and optionally to the 'M' matrix, if the 'M' matrix exists.

mxExpectationRAM evaluates with respect to an MxData object. The MxData object need not be referenced in the mxExpectationRAM function, but must be included in the MxModel object.

To evaluate an mxExpectationRAM object, place it, the mxData object which the expected covariance approximates, any referenced MxAlgebra and MxMatrix objects, and optional MxBounds and MxConstraint objects in an MxModel object and evaluate it using mxRun. The results of the optimization can be found in the 'output' slot of the resulting model, and may be obtained using the mxEval function.

Value

Returns a new MxExpectationRAM object. mxExpectationRAM objects should be included in a model, along with referenced MxAlgebra, MxData and MxMatrix objects.

References

McArdle, J. J. and MacDonald, R. P. (1984). Some algebraic properties of the Reticular Action Model for moment structures. *British Journal of Mathematical and Statistical Psychology, 37,* 234-251.

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
# Create and fit a model using mxMatrix, mxAlgebra,
#  mxExpectationNormal, and mxFitFunctionML

library(OpenMx)

# Simulate some data

x=rnorm(1000, mean=0, sd=1)
y= 0.5*x + rnorm(1000, mean=0, sd=1)
tmpFrame <- data.frame(x, y)
tmpNames <- names(tmpFrame)

# Define the matrices

matrixS <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(1,0,0,1),
              free=c(TRUE,FALSE,FALSE,TRUE), labels=c("Vx", NA, NA, "Vy"),
              name = "S")
matrixA <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(0,1,0,0),
              free=c(FALSE,TRUE,FALSE,FALSE), labels=c(NA, "b", NA, NA),
              name = "A")
matrixF <- mxMatrix(type="Iden", nrow=2, ncol=2, name="F")
matrixM <- mxMatrix(type = "Full", nrow = 1, ncol = 2, values=c(0,0),
              free=c(TRUE,TRUE), labels=c("Mx", "My"), name = "M")

# Define the expectation

expFunction <- mxExpectationRAM(M="M", dimnames = tmpNames)

# Choose a fit function

fitFunction <- mxFitFunctionML()
```

```
# Define the model

tmpModel <- mxModel(model="exampleRAMModel",
                    matrixA, matrixS, matrixF, matrixM,
                    expFunction, fitFunction,
                    mxData(observed=tmpFrame, type="raw"))

# Fit the model and print a summary

tmpModelOut <- mxRun(tmpModel)
summary(tmpModelOut)
```

---

mxExpectationStateSpace

Create an MxExpectationStateSpace Object

Description

This function creates a new MxExpectationStateSpace object.

Usage

```
mxExpectationStateSpace(A, B, C, D, Q, R, x0, P0, u,
                          dimnames = NA, thresholds = deprecated(),
                          threshnames = deprecated(),
                          ..., t = NA, scores=FALSE)
```

Arguments

`A`	A character string indicating the name of the 'A' matrix.
`B`	A character string indicating the name of the 'B' matrix.
`C`	A character string indicating the name of the 'C' matrix.
`D`	A character string indicating the name of the 'D' matrix.
`Q`	A character string indicating the name of the 'Q' matrix.
`R`	A character string indicating the name of the 'R' matrix.
`x0`	A character string indicating the name of the 'x0' matrix.
`P0`	A character string indicating the name of the 'P0' matrix.
`u`	A character string indicating the name of the 'u' matrix.
`dimnames`	An optional character vector to be assigned to the row names of the 'C' matrix.
`thresholds`	**[Deprecated]**
`threshnames`	**[Deprecated]**
`...`	Unused. Requires further arguments to be named.
`t`	Not to be used
`scores`	Not to be used

Details

This page presents details for both the `mxExpectationStateSpace` function and for state space modeling generally; however, for much more information on state space modeling see the paper by Hunter (2018) listed under references. Authors using state space modeling in OpenMx should cite Hunter (2018).

Expectation functions define the way that model expectations are calculated. When used in conjunction with the mxFitFunctionML, the mxExpectationStateSpace uses maximum likelihood prediction error decomposition (PED) to obtain estimates of free parameters in a model of the raw MxData object. State space expectations treat the raw data as a multivariate time series of equally spaced times with each row corresponding to a single occasion. This is not a model of the block Toeplitz lagged autocovariance matrix. State space expectations implement a classical Kalman filter to produce expectations.

The hybrid Kalman filter (combination of classical Kalman and Kalman-Bucy filters) for continuous latent time with discrete observations is implemented and is available as mxExpectationStateSpaceContinuousTime. The following alternative filters are not yet implemented: square root Kalman filter (in Cholesky or singular value decomposition form), extended Kalman filter for linear approximations to nonlinear state space models, unscented Kalman filter for highly nonlinear state space models, and Rauch-Tung-Striebel smoother for updating forecast state estimates after a complete forward pass through the data has been made.

Missing data handling is implemented in the same fashion as full information maximum likelihood for partially missing rows of data. Additionally, completely missing rows of data are handled by only using the prediction step from the Kalman filter and omitting the update step.

This model uses notation for the model matrices commonly found in engineering and control theory.

The 'A', 'B', 'C', 'D', 'Q', 'R', 'x0', and 'P0' arguments must be the names of MxMatrix or MxAlgebraobjects with the associated properties of the A, B, C, D, Q, R, x0, and P0 matrices in the state space modeling approach.

The state space expectation is defined by the following model equations.

$$x_t = Ax_{t-1} + Bu_t + q_t$$

$$y_t = Cx_t + Du_t + r_t$$

with q_t and r_t both independently and identically distributed random Gaussian (normal) variables with mean zero and covariance matrices Q and R, respectively.

The first equation is called the state equation. It describes how the latent states change over time. Also, the state equation in state space modeling is directly analogous to the structural model in LISREL structural equation modeling.

The second equation is called the output equation. It describes how the latent states relate to the observed states at a single point in time. The output equation shows how the observed output is produced by the latent states. Also, the output equation in state space modeling is directly analogous to the measurement model in LISREL structural equation modeling.

Note that the covariates, u, have "instantaneous" effects on both the state and output equations. If lagged effects are desired, then the user must create a lagged covariate by shifting their observed variable to the desired lag.

The state and output equations, together with some minimal assumptions and the Kalman filter, imply a new expected covariance matrix and means vector for every row of data. The expected covariance matrix of row t is

$$S_t = C(AP_{t-1}A^{\mathsf{T}} + Q)C^{\mathsf{T}} + R$$

The expected means vector of row t is

$$\hat{y}_t = Cx_t + Du_t$$

The 'dimnames' arguments takes an optional character vector.

The 'A' argument refers to the A matrix in the State Space approach. This matrix consists of time regressive coefficients from the latent variable in column j at time $t-1$ to the latent variable in row i at time t. Entries in the diagonal are autoregressive coefficients. Entries in the off-diagonal are cross-lagged regressive coefficients. If the A and B matrices are zero matrices, then the state space model reduces to a factor analysis. The A matrix is sometimes called the state-transition model.

The 'B' argument refers to the B matrix in the State Space approach. This matrix consists of regressive coefficients from the input (manifest covariate) variable j at time t to the latent variable in row i at time t. Note that the covariate effect is contemporaneous: the covariate at time t has influence on the latent state also at time t. A lagged effect can be created by lagged the observed variable. The B matrix is sometimes called the control-input model.

The 'C' argument refers to the C matrix in the State Space approach. This matrix consists of contemporaneous regression coefficients from the latent variable in column j to the observed variable in row i. This matrix is directly analogous to the factor loadings matrix in LISREL and Mplus models. The C matrix is sometimes called the observation model.

The 'D' argument refers to the D matrix in the State Space approach. This matrix consists of contemporaneous regressive coefficients from the input (manifest covariate) variable j to the observed variable in row i. The D matrix is sometimes called the feedthrough or feedforward matrix.

The 'Q' argument refers to the Q matrix in the State Space approach. This matrix consists of residual covariances among the latent variables. This matrix must be symmetric. As a special case, it is often diagonal. The Q matrix is the covariance of the process noise. Just as in factor analysis and general structural equation modeling, the scale of the latent variables is usually set by fixing some factor loadings in the C matrix, or fixing some factor variances in the Q matrix.

The 'R' argument refers to the R matrix in the State Space approach. This matrix consists of residual covariances among the observed (manifest) variables. This matrix must be symmetric As a special case, it is often diagonal. The R matrix is the covariance of the observation noise.

The 'x0' argument refers to the x_0 matrix in the State Space approach. This matrix consists of the column vector of the initial values for the latent variables. The state space expectation uses the x_0 matrix as the starting point to recursively estimate the latent variables' values at each time. These starting values can be difficult to pick, however, for sufficiently long time series they often do not greatly impact the estimation.

The 'P0' argument refers to the P_0 matrix in the State Space approach. This matrix consists of the initial values of the covariances of the error in the initial latent variable estimates given in x_0. That is, the P_0 matrix gives the covariance of $x_0 - xtrue_0$ where $xtrue_0$ is the vector of true initial values. P_0 is a measure of the accuracy of the initial latent state estimates. The Kalman filter uses this initial covariance to recursively generated a new covariance for each time point based on the previous time point. The Kalman filter updates this covariance so that it is as small as possible (minimum trace). Similar to the x_0 matrix, these starting values are often difficult to choose.

The 'u' argument refers to the u matrix in the State Space approach. This matrix consists of the inputs or manifest covariates of the state space expectation. The u matrix must be a column vector with the same number of rows as the B and D matrices have columns. If no inputs are desired, u can be a zero matrix. If time-varying inputs are desired, then they should be included as columns in the MxData object and referred to in the labels of the u matrix as definition variables. There is an example of this below.

The MxMatrix objects included as arguments may be of any type, but should have the properties described above. The mxExpectationStateSpace will not return an error for incorrect specification, but incorrect specification will likely lead to estimation problems or errors in the mxRun function.

mxExpectationStateSpace evaluates with respect to an MxData object. The MxData object need not be referenced in the mxExpectationStateSpace function, but must be included in the MxModel object. mxExpectationStateSpace requires that the 'type' argument in the associated MxData object be equal to 'raw'. Neighboring rows of the MxData object are treated as adjacent, equidistant time points increasing from the first to the last row.

To evaluate, place mxExpectationStateSpace objects, the mxData object for which the expected covariance approximates, referenced MxAlgebra and MxMatrix objects, and optional MxBounds and MxConstraint objects in an MxModel object. This model may then be evaluated using the mxRun function. The results of the optimization can be found in the 'output' slot of the resulting model, and may be obtained using the mxEval function.

Value

Returns a new MxExpectationStateSpace object. mxExpectationStateSpace objects should be included with models with referenced MxAlgebra, MxData and MxMatrix objects.

References

K.J. Åström and R.M. Murray (2010). *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press.

J. Durbin and S.J. Koopman. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press.

Hunter, M.D. (2018). State Space Modeling in an Open Source, Modular, Structural Equation Modeling Environment. *Structural Equation Modeling: A Multidisciplinary Journal, 25(2)*, 307-324. DOI: 10.1080/10705511.2017.1369354

R.E. Kalman (1960). A New Approach to Linear Filtering and Prediction Problems. *Basic Engineering, 82*, 35-45.

G. Petris (2010). An R Package for Dynamic Linear Models. *Journal of Statistical Software, 36*, 1-16.

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

mxExpectationStateSpaceContinuousTime

Examples

```
# Create and fit a model using mxMatrix, mxExpectationStateSpace, and mxFitFunctionML
require(OpenMx)
data(demoOneFactor)
nvar <- ncol(demoOneFactor)
varnames <- colnames(demoOneFactor)
ssModel <- mxModel(model="State Space Manual Example",
    mxMatrix("Full", 1, 1, TRUE, .3, name="A"),
    mxMatrix("Zero", 1, 1, name="B"),
    mxMatrix("Full", nvar, 1, TRUE, .6, name="C", dimnames=list(varnames, "F1")),
    mxMatrix("Zero", nvar, 1, name="D"),
    mxMatrix("Diag", 1, 1, FALSE, 1, name="Q"),
    mxMatrix("Diag", nvar, nvar, TRUE, .2, name="R"),
    mxMatrix("Zero", 1, 1, name="x0"),
    mxMatrix("Diag", 1, 1, FALSE, 1, name="P0"),
    mxMatrix("Zero", 1, 1, name="u"),
    mxData(observed=demoOneFactor[1:100,], type="raw"),#fewer rows = fast
    mxExpectationStateSpace("A", "B", "C", "D", "Q", "R", "x0", "P0", "u"),
    mxFitFunctionML()
)
ssRun <- mxRun(ssModel)
summary(ssRun)
# Note the freely estimated Autoregressive parameter (A matrix)
#  is near zero as it should be for the independent rows of data
#  from the factor model.

# Create and fit a model with INPUTS using mxMatrix, mxExpectationStateSpace, and mxFitFunctionML
require(OpenMx)
data(demoOneFactor)
nvar <- ncol(demoOneFactor)
varnames <- colnames(demoOneFactor)
#demoOneFactorInputs <- cbind(demoOneFactor, V1=rep(1, nrow(demoOneFactor)))
demoOneFactorInputs <- cbind(demoOneFactor, V1=rnorm(nrow(demoOneFactor)))
ssModel <- mxModel(model="State Space Inputs Manual Example",
    mxMatrix("Full", 1, 1, TRUE, .3, name="A"),
    mxMatrix("Full", 1, 1, TRUE, values=1, name="B"),
    mxMatrix("Full", nvar, 1, TRUE, .6, name="C", dimnames=list(varnames, "F1")),
    mxMatrix("Zero", nvar, 1, name="D"),
    mxMatrix("Diag", 1, 1, FALSE, 1, name="Q"),
    mxMatrix("Diag", nvar, nvar, TRUE, .2, name="R"),
    mxMatrix("Zero", 1, 1, name="x0"),
    mxMatrix("Diag", 1, 1, FALSE, 1, name="P0"),
    mxMatrix("Full", 1, 1, FALSE, labels="data.V1", name="u"),
    mxData(observed=demoOneFactorInputs[1:100,], type="raw"),#fewer rows = fast
    mxExpectationStateSpace("A", "B", "C", "D", "Q", "R", "x0", "P0", u="u"),
    mxFitFunctionML()
)
ssRun <- mxRun(ssModel)
summary(ssRun)
# Note the freely estimated Autoregressive parameter (A matrix)
#  and the freely estimated Control-Input parameter (B matrix)
#  are both near zero as they should be for the independent rows of data
# from the factor model that does not have inputs, covariates,
# or exogenous variables.
```

---

mxExpectationStateSpaceContinuousTime

Create an MxExpectationStateSpace Object

Description

This function creates a new MxExpectationStateSpace object.

Usage

```
mxExpectationStateSpaceContinuousTime(A, B, C, D, Q, R, x0, P0, u, t = NA,
                        dimnames = NA, thresholds = deprecated(),
      threshnames = deprecated(),  ..., scores=FALSE)
mxExpectationSSCT(A, B, C, D, Q, R, x0, P0, u, t = NA,
                        dimnames = NA, thresholds = deprecated(),
      threshnames = deprecated(),
                        ..., scores=FALSE)
```

Arguments

`A`	A character string indicating the name of the 'A' matrix.
`B`	A character string indicating the name of the 'B' matrix.
`C`	A character string indicating the name of the 'C' matrix.
`D`	A character string indicating the name of the 'D' matrix.
`Q`	A character string indicating the name of the 'Q' matrix.
`R`	A character string indicating the name of the 'R' matrix.
`x0`	A character string indicating the name of the 'x0' matrix.
`P0`	A character string indicating the name of the 'P0' matrix.
`u`	A character string indicating the name of the 'u' matrix.
`t`	A character string indicating the name of the 't' matrix.
`dimnames`	An optional character vector to be assigned to the row names of the 'C' matrix.
`thresholds`	**[Deprecated]**
`threshnames`	**[Deprecated]**
`...`	Unused. Requires further arguments to be named.
`scores`	Not to be used

Details

The `mxExpectationStateSpaceContinuousTime` and `mxExpectationSSCT` functions are identical. The latter is simply an abbreviated name. When using the former, tab completion is strongly encouraged to save tedious typing. Both of these functions are wrappers for the mxExpectationStateSpace function, which could be used for both discrete and continuous time modeling. However, there is a strong possibility of misunderstanding the model parameters when switching between discrete time and continuous time. The expectation matrices have the same names, but mean importantly different things so caution is warranted. The best practice is to use `mxExpectationStateSpace` for discrete time models, and `mxExpectationStateSpaceContinuousTime` for continuous time models.

Expectation functions define the way that model expectations are calculated. That is to say, expectation functions define how a set of model matrices get turned into expectations for the data. When used in conjunction with the mxFitFunctionML, the mxExpectationStateSpace uses maximum likelihood prediction error decomposition (PED) to obtain estimates of free parameters in a model of the raw MxData object. Continuous time state space expectations treat the raw data as a multivariate time series of possibly unevenly spaced times with each row corresponding to a single occasion. Continuous time state space expectations implement a hybrid Kalman filter to produce expectations. The hybrid Kalman filter uses a Kalman-Bucy filter for the prediction step and the classical Kalman filter for the update step. It is a hybrid between the classical Kalman filter used for the discrete (but possibly unequally spaced) measurement occasions and the continuous time Kalman-Bucy filter for latent variable predictions.

Missing data handling is implemented in the same fashion as full information maximum likelihood for partially missing rows of data. Additionally, completely missing rows of data are handled by only using the prediction step from the Kalman-Bucy filter and omitting the update step.

This model uses notation for the model matrices commonly found in engineering and control theory.

The 'A', 'B', 'C', 'D', 'Q', 'R', 'x0', and 'P0' arguments must be the names of MxMatrix or MxAlgebraobjects with the associated properties of the A, B, C, D, Q, R, x0, and P0 matrices in the state space modeling approach. The 't' matrix must be a 1x1 matrix using definition variables that gives the times at which measurements occurred.

The state space expectation is defined by the following model equations.

$$\frac{d}{dt}x(t) = Ax(t) + Bu_t + q(t)$$

$$y_t = Cx_t + Du_t + r_t$$

with $q(t)$ and r_t both independently and identically distributed random Gaussian (normal) variables with mean zero and covariance matrices Q and R, respectively. Subscripts or square brackets indicate discrete indices; parentheses indicate continuous indices. The derivative of $x(t)$ with respect to t is $\frac{d}{dt}x(t)$.

The first equation is called the state equation. It describes how the latent states change over time with a first-order linear differential equation. Unlike some other programs, we do not require that the continuous time A matrix has an inverse. This allows zero dynamics (i.e. no growth models) and many other important kinds of processes.

The second equation is called the output equation. It describes how the latent states relate to the observed states at a single point in time. The output equation shows how the observed output is produced by the latent states. Also, the output equation in state space modeling is directly analogous to the measurement model in LISREL structural equation modeling.

Note that the covariates, u, have "instantaneous" effects on both the state and output equations. If lagged effects are desired, then the user must create a lagged covariate by shifting their observed variable to the desired lag.

The state and output equations, together with some minimal assumptions and the Kalman filter, imply a new expected covariance matrix and means vector for every row of data. The expected covariance matrix of row t is

$$S_t = C(AP_{t-1}A^{\mathsf{T}} + Q)C^{\mathsf{T}} + R$$

The expected means vector of row t is

$$\hat{y}_t = Cx_t + Du_t$$

The 'dimnames' arguments takes an optional character vector.

The 'A' argument refers to the A matrix in the State Space approach. This matrix gives the dynamics. Entries in the diagonal give the strength of the influence of a variable's position on its slope. Entries in the off-diagonal give the coupling strength from one variable to another. The A matrix is sometimes called the state-transition model.

The 'B' argument refers to the B matrix in the State Space approach. This matrix consists of exogenous forces that influence the dynamics. Note that the covariate effect is contemporaneous: the covariate at time t has influence on the slope of the latent state also at time t. A lagged effect can be created by lagged the observed variable. The B matrix is sometimes called the control-input model.

The 'C' argument refers to the C matrix in the State Space approach. This matrix consists of contemporaneous regression coefficients from the latent variable in column j to the observed variable in row i. This matrix is directly analogous to the factor loadings matrix in LISREL and Mplus models. The C matrix is sometimes called the observation model.

The 'D' argument refers to the D matrix in the State Space approach. This matrix consists of contemporaneous regressive coefficients from the input (manifest covariate) variable j to the observed variable in row i. The D matrix is sometimes called the feedthrough or feedforward matrix.

The 'Q' argument refers to the Q matrix in the State Space approach. This matrix gives the covariance of the dynamic noise. The dynamic noise can be thought of as unmeasured covariate inputs active at all times. This matrix must be symmetric, diagonal, or zero. As a special case, it is often diagonal. The Q matrix is the covariance of the process noise. Just as in factor analysis and general structural equation modeling, the scale of the latent variables is usually set by fixing some factor loadings in the C matrix, or fixing some factor variances in the Q matrix.

The 'R' argument refers to the R matrix in the State Space approach. This matrix consists of residual covariances among the observed (manifest) variables. This matrix must be symmetric As a special case, it is often diagonal. The R matrix is the covariance of the observation noise.

The 'x0' argument refers to the x_0 matrix in the State Space approach. This matrix consists of the column vector of the initial values for the latent variables. The state space expectation uses the x_0 matrix as the starting point to recursively estimate the latent variables' values at each time. These starting values can be difficult to pick, however, for sufficiently long time series they often do not greatly impact the estimation.

The 'P0' argument refers to the P_0 matrix in the State Space approach. This matrix consists of the initial values of the covariances of the error in the initial latent variable estimates given in x_0. That is, the P_0 matrix gives the covariance of $x_0 - xtrue_0$ where $xtrue_0$ is the vector of true initial values. P_0 is a measure of the accuracy of the initial latent state estimates. The Kalman filter uses this initial covariance to recursively generated a new covariance for each time point based on the previous time point. The Kalman filter updates this covariance so that it is as small as possible (minimum trace). Similar to the x_0 matrix, these starting values are often difficult to choose.

The 'u' argument refers to the u matrix in the State Space approach. This matrix consists of the inputs or manifest covariates of the state space expectation. The u matrix must be a column vector with the same number of rows as the B and D matrices have columns. If no inputs are desired, u can be a zero matrix. If time-varying inputs are desired, then they should be included as columns in the MxData object and referred to in the labels of the u matrix as definition variables. There is an example of this below.

The 't' argument refers to the t matrix in the State Space approach. This matrix should be 1x1 (1 row and 1 column) and not free. The label for the element of this matrix should be 'data.YourTimeVariable'. The 'data' part does not change, but 'YourTimeVariable' should be a name in your data set that gives the times at which measurement happened. The units of time are up to you. Your choice of time units will influence of the values of the parameters you estimate. Also, recall that the model is given x_0 and P_0. These always happen at $t = 0$. So the first row of data happens some amount of time after zero.

The MxMatrix objects included as arguments may be of any type, but should have the properties described above. The mxExpectationStateSpace will not return an error for incorrect specification, but incorrect specification will likely lead to estimation problems or errors in the mxRun function.

mxExpectationStateSpaceContinuousTime evaluates with respect to an MxData object. The MxData object need not be referenced in the mxExpectationStateSpace function, but must be included in the MxModel object. mxExpectationStateSpace requires that the 'type' argument in the associated MxData object be equal to 'raw'. Neighboring rows of the MxData object are treated as adjacent, equidistant time points increasing from the first to the last row.

To evaluate, place an mxExpectationStateSpaceContinuousTime object, the mxData object for which the expected covariance approximates, referenced MxAlgebra and MxMatrix objects, and optional MxBounds and MxConstraint objects in an MxModel object. This model may then be evaluated using the mxRun function. The results of the optimization can be found in the 'output' slot of the resulting model, and may be obtained using the mxEval function.

Value

Returns a new MxExpectationStateSpace object. mxExpectationStateSpace objects should be included with models with referenced MxAlgebra, MxData and MxMatrix objects.

References

K.J. Åström and R.M. Murray (2010). *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press.

J. Durbin and S.J. Koopman. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press.

R.E. Kalman (1960). A New Approach to Linear Filtering and Prediction Problems. *Basic Engineering, 82*, 35-45.

R.E. Kalman and R.S. Bucy (1961). New Results in Linear Filtering and Prediction Theory. *Transactions of the ASME, Series D, Journal of Basic Engineering, 83*, 95-108.

G. Petris (2010). An R Package for Dynamic Linear Models. *Journal of Statistical Software, 36*, 1-16.

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

mxExpectationStateSpace

Examples

```
#--------------------------------------------------------------------------------
# Example 1
# Undamped linear oscillator, i.e. a noisy sine wave.
# Measurement error, but no dynamic error, single indicator.
# This example works great.

#----------------------------------------
# Data Generation

require(OpenMx)

set.seed(405)
tlen <- 200
t <- seq(1.2, 50, length.out=tlen)

freqParam <- .5
initialCond <- matrix(c(2.5, 0))
x <- initialCond[1,1]*cos(freqParam*t)
plot(t, x, type='l')

measVar <- 1.5
y <- cbind(obs=x+rnorm(tlen, sd=sqrt(measVar)), tim=t)

plot(t, y[,1], type='l')

#----------------------------------------
# Model Specification

#Note: the bounds are here only to keep SLSQP from
# stepping too far off a cliff.  With the bounds in
# place, SLSQP finds the right solution.  Without
# the bounds, SLSQP goes crazy.


cdim <- list('obs', c('ksi', 'ksiDot'))

amat <- mxMatrix('Full', 2, 2, c(FALSE, TRUE, FALSE, TRUE), c(0, -.1, 1, -.2),
name='A', lbound=-10)
bmat <- mxMatrix('Zero', 2, 1, name='B')
cmat <- mxMatrix('Full', 1, 2, FALSE, c(1, 0), name='C', dimnames=cdim)
dmat <- mxMatrix('Zero', 1, 1, name='D')
qmat <- mxMatrix('Zero', 2, 2, name='Q')
rmat <- mxMatrix('Diag', 1, 1, TRUE, .4, name='R', lbound=1e-6)
xmat <- mxMatrix('Full', 2, 1, TRUE, c(0, 0), name='x0', lbound=-10, ubound=10)
pmat <- mxMatrix('Diag', 2, 2, FALSE, 1, name='P0')
umat <- mxMatrix('Zero', 1, 1, name='u')
tmat <- mxMatrix('Full', 1, 1, name='time', labels='data.tim')

osc <- mxModel("LinearOscillator",
amat, bmat, cmat, dmat, qmat, rmat, xmat, pmat, umat, tmat,
mxExpectationSSCT('A', 'B', 'C', 'D', 'Q', 'R', 'x0', 'P0', 'u', 'time'),
mxFitFunctionML(),
mxData(y, 'raw'))


oscr <- mxRun(osc)


#---------------------------------------
# Results Examination

summary(oscr)


(ssFreqParam <- mxEval(sqrt(-A[2,1]), oscr))
freqParam


(ssMeasVar <- mxEval(R, oscr))
measVar


dampingParam <- 0
(ssDampingParam <- mxEval(-A[2,2], oscr))
dampingParam
```

`mxFactor` *Fail-safe Factors*

Description

This is a wrapper for the R function `factor`.

OpenMx requires ordinal data to be ordered. R's `factor` function doesn't enforce this, hence this wrapper exists to throw an error should you accidentally try and run with ordered = FALSE.

Also, the 'levels' parameter is optional in R's `factor` function. However, relying on the data to specify the data is foolhardy for the following reasons: The `factor` function will skip levels missing from the data: Specifying these in levels leaves the list of levels complete. Data will often not explore the min and max level that the user knows are possible. For these reasons this function forces you to write out all possible levels explicitly.

Usage

```
mxFactor(x = character(), levels, labels = levels,
    exclude = NA, ordered = TRUE, collapse = FALSE)
```

Arguments

`x`	either a vector of data or a data.frame object.
`levels`	a mandatory vector of the values that 'x' might have taken.
`labels`	_either_ an optional vector of labels for the levels, _or_ a character string of length 1.
`exclude`	a vector of values to be excluded from the set of levels.
`ordered`	logical flag to determine if the levels should be regarded as ordered (in the order given). Required to be TRUE.
`collapse`	logical flag to determine if duplicate labels should collapsed into a single level

Details

If ‘x’ is a data.frame, then all of the columns of ‘x’ are converted into ordered factors. If ‘x’ is a data.frame, then ‘levels’ and ‘labels’ may be either a list or a vector. When ‘levels’ is a list, then different levels are assigned to different columns of the constructed data.frame object. When ‘levels’ is a vector, then the same levels are assigned to all the columns of the data.frame object. The function will throw an error if ‘ordered’ is not TRUE or if ‘levels’ is missing. See `factor` for more information on creating ordered factors.

References

The OpenMx User’s guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
myVar <- c("s", "t", "a", "t", "i", "s", "t", "i", "c", "s")
ff    <- mxFactor(myVar, levels=letters)
# Note: letters is a built in list of all lowercase letters of the alphabet
ff
# [1] s t a t i s t i c s
# Levels: a < b < c < d < e < f < g < h < i < j < k < l < m < n < o < p < q <
#  r < s < t < u < v < w < x < y < z

as.integer(ff)  # the internal codes

factor(ff)      # NOTE: drops the levels that do not occur.
                # mxFactor prevents you doing this unintentionally.

# This example works on a dataframe
```

```
foo <- data.frame(x=c(1:3),y=c(4:6),z=c(7:9))

# Applys one set of levels to all three columns
mxFactor(foo, c(1:9))

# Apply unique sets of levels to each variable
mxFactor(foo, list(c(1:3), c(4:6), c(7:9)))


mxFactor(foo, c(1:9), labels=c(1,1,1,2,2,2,3,3,3), collapse=TRUE)
```

`mxFactorScores` *Estimate factor scores and standard errors*

Description

This function creates the factor scores and their standard errors under different methods for an MxModel object that has either a RAM or LISREL expectation.

Usage

```
mxFactorScores(model, type=c('ML', 'WeightedML', 'Regression'),
 minManifests=as.integer(NA))
```

Arguments

`model`	An MxModel object with either an MxExpectationLISREL or MxExpectation-RAM
`type`	The type of factor scores to compute
`minManifests`	Set scores to NA when there are less than minManifests non-NA manifest variables

Details

This is a helper function to compute or estimate factor scores along with their standard errors. The two maximum likelihood methods create a new model for each data row. They then estimate the factor scores as free parameters in a model with a single data row. For 'ML', the conditional likelihood of the data given the factor scores is optimized:

$$L(D|F)$$

. For 'WeightedML', the joint likelihood of the data and the factor scores is optimized:

$$L(D,F) = L(D|F)L(F)$$

. The WeightedML scores are akin to the empirical Bayes random effects estimates from mixed effects modeling. They display the same kind of shrinkage as random effects estimates, and for the same reason: they account for the latent variable distribution in their estimation.

In many cases, especially for ordinal data or missing data, the weighted ML scores are to be preferred over alternatives (Estabrook & Neale, 2013). For example, when using ordinal data, a person whose observations are all in the highest ordinal category theoretically has an 'ML' factor score of positive infinity. A similar situation arises for a person whose observations are all in the lowest ordinal category: their 'ML' factor score is theoretically negative infinity. Weighted ML factor scores in these cases remain reasonable.

For type='Regression', with LISREL expectation, factor scores are computed based on a simple formula. This formula is equivalent to the formula for the Kalman updated scores in a state space model with zero dynamics (Priestly & Subba Rao, 1975). Thus, to compute the regression factor scores, the appropriate state space model is set-up and the mxKalmanScores function is used to produce the factor scores and their standard errors. With RAM expectation, factor scores are predicted from the non-missing manifest variables for each row of the raw data, using a general linear prediction formula analytically equivalent to that used with LISREL expectation. The standard errors for regression-predicted RAM factor scores are the square roots of the indeterminate variances of the latent variables, given the data row's missing-data pattern and the values of any relevant definition variables. The RAM and LISREL methods for computing regression factor scores with their standard errors are analytically identical. They produce the same score and standard error estimates.

If you have missing data then you must specify `minManifests`. This option will set scores to NA when there are too few items to make an accurate score estimate. If you are using the scores as point estimates without considering the standard error then you should set minManifests as high as you can tolerate. This will increase the amount of missing data but scores will be more accurate. If you are carefully considering the standard errors of the scores then you can set minManifests to 0. When set to 0, all NA rows are scored to the prior distribution.

Note that for compatibility with `factanal`, type='regression' is also acceptable.

Value

An array with dimensions (Number of Rows of Data, Number of Latent Variables, 2). The third dimension has the scores in the first slot and the standard errors in the second slot. The rows are in the order of the *unsorted* data. Multigroup models are an exception, in that the returned value is instead a list of such arrays, containing one per group.

References

Estabrook, R. & Neale, M. C. (2013). A Comparison of Factor Score Estimation Methods in the Presence of Missing Data: Reliability and an Application to Nicotine Dependence. *Multivariate Behavioral Research, 48,* 1-27.

Priestley, M. & Subba Rao, T. (1975). The estimation of factor scores and Kalman filtering for discrete parameter stationary processes. *International Journal of Control, 21*, 971-975.

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

mxKalmanScores

Examples

```
# Create and estimate a factor model
require(OpenMx)
data(demoOneFactor)
manifests <- names(demoOneFactor)
latents <- c("G")
factorModel <- mxModel("OneFactor",
                       type="LISREL",
                       manifestVars=list(exo=manifests),
                       latentVars=list(exo=latents),
                       mxPath(from=latents, to=manifests),
                       mxPath(from=manifests, arrows=2),
                       mxPath(from=latents, arrows=2, free=FALSE, values=1.0),
                       mxPath(from='one', to=manifests),
                       mxData(observed=cov(demoOneFactor), type="cov", numObs=500,
                              means = colMeans(demoOneFactor)))
summary(factorRun <- mxRun(factorModel))

# Swap in raw data in place of summary data
factorRun <- mxModel(factorRun, mxData(observed=demoOneFactor[1:50,], type="raw"))

# Estimate factor scores for the model
r1 <- mxFactorScores(factorRun, 'Regression')
```

`mxFIMLObjective` *DEPRECATED: Create MxFIMLObjective Object*

Description

WARNING: Objective functions have been deprecated as of OpenMx 2.0.

Please use mxExpectationNormal() and mxFitFunctionML() instead. As a temporary workaround, mxFIMLObjective returns a list containing an MxExpectationNormal object and an MxFitFunctionML object.

All occurrences of

mxFIMLObjective(covariance, means, dimnames = NA, thresholds = NA, vector = FALSE, threshnames = dimnames)

Should be changed to

mxExpectationNormal(covariance, means, dimnames = NA, thresholds = NA, threshnames = dimnames) mxFitFunctionML(vector = FALSE)

Arguments

`covariance`	A character string indicating the name of the expected covariance algebra.
`means`	A character string indicating the name of the expected means algebra.

`dimnames`	An optional character vector to be assigned to the dimnames of the covariance and means algebras.
`thresholds`	An optional character string indicating the name of the thresholds matrix.
`vector`	A logical value indicating whether the objective function result is the likelihood vector.
`threshnames`	An optional character vector to be assigned to the column names of the thresholds matrix.

Details

NOTE: THIS DESCRIPTION IS DEPRECATED. Please change to using mxExpectationNormal and mxFitFunctionML as shown in the example below.

Objective functions were functions for which free parameter values are chosen such that the value of the objective function is minimized. The mxFIMLObjective function used full-information maximum likelihood to provide maximum likelihood estimates of free parameters in the algebra defined by the ’covariance’ and ’means’ arguments. The ’covariance’ argument takes an MxAlgebra object, which defines the expected covariance of an associated MxData object. The ’means’ argument takes an MxAlgebra object, which defines the expected means of an associated MxData object. The ’dimnames’ arguments takes an optional character vector. If this argument is not a single NA, then this vector is used to assign the dimnames of the means vector as well as the row and columns dimnames of the covariance matrix.

The ’vector’ argument is either TRUE or FALSE, and determines whether the objective function returns a column vector of the likelihoods, or a single -2*(log likelihood) value.

thresholds: The name of the thresholds matrix. When needed (for modelling ordinal data), this matrix should be created using mxMatrix(). The thresholds matrix must have as many columns as there are ordinal variables in the model, and number of rows equal to one fewer than the maximum number of levels found in the ordinal variables. The starting values of this matrix must also be set to reasonable values. Fill each column with a set of ordered start thresholds, one for each level of this column’s factor levels minus 1. These thresholds may be free if you wish them to be estimated, or fixed. The unused rows in each column, if any, can be set to any value including NA.

threshnames: A character vector consisting of the variables in the thresholds matrix, i.e., the names of ordinal variables in a model. This is necessary for OpenMx to map the thresholds matrix columns onto the variables in your data. If you set the dimnames of the columns in the thresholds matrix then threshnames is not needed.

Usage Notes: dimnames must be supplied where the matrices referenced by the covariance and means algebras are not themselves labeled. Failure to do so leads to an error noting that the covariance or means matrix associated with the FIML objective does not contain dimnames.

mxFIMLObjective evaluates with respect to an MxData object. The MxData object need not be referenced in the mxFIMLObjective function, but must be included in the MxModel object. mxFIMLObjective requires that the ’type’ argument in the associated MxData object be equal to ’raw’. Missing values are permitted in the associated MxData object.

To evaluate, place MxFIMLObjective objects, the mxData object for which the expected covariance approximates, referenced MxAlgebra and MxMatrix objects, and optional MxBounds and MxConstraint objects in an MxModel object. This model may then be evaluated using the mxRun function.

The results of the optimization can be reported using the summary function, or accessed directly in the 'output' slot of the resulting model (i.e., modelName$output). Components of the output may be referenced using the Extract functionality.

Value

Returns a list containing an MxExpectationNormal object and an MxFitFunctionML object.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
# Create and fit a model using mxMatrix, mxAlgebra, mxExpectationNormal, and mxFitFunctionML

library(OpenMx)

# Simulate some data

x=rnorm(1000, mean=0, sd=1)
y= 0.5*x + rnorm(1000, mean=0, sd=1)
tmpFrame <- data.frame(x, y)
tmpNames <- names(tmpFrame)

# Define the matrices

M <- mxMatrix(type = "Full", nrow = 1, ncol = 2, values=c(0,0),
              free=c(TRUE,TRUE), labels=c("Mx", "My"), name = "M")
S <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(1,0,0,1),
              free=c(TRUE,FALSE,FALSE,TRUE), labels=c("Vx", NA, NA, "Vy"), name = "S")
A <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(0,1,0,0),
              free=c(FALSE,TRUE,FALSE,FALSE), labels=c(NA, "b", NA, NA), name = "A")
I <- mxMatrix(type="Iden", nrow=2, ncol=2, name="I")

# Define the expectation

expCov <- mxAlgebra(solve(I-A) %*% S %*% t(solve(I-A)), name="expCov")
expFunction <- mxExpectationNormal(covariance="expCov", means="M", dimnames=tmpNames)

# Choose a fit function

fitFunction <- mxFitFunctionML()

# Define the model

tmpModel <- mxModel(model="exampleModel", M, S, A, I, expCov, expFunction, fitFunction,
                    mxData(observed=tmpFrame, type="raw"))

# Fit the model and print a summary

tmpModelOut <- mxRun(tmpModel)
```

```
summary(tmpModelOut)
```

`MxFitFunction-class` *MxFitFunction*

Description

This is an internal class and should not be used directly.

`mxFitFunctionAlgebra` *Create MxFitFunctionAlgebra Object*

Description

mxFitFunctionAlgebra returns an MxFitFunctionAlgebra object.

Usage

```
mxFitFunctionAlgebra(algebra, numObs = NA, numStats = NA, ..., gradient =
                 NA_character_, hessian = NA_character_, verbose = 0L,
   units="-2lnL", strict=TRUE)
```

Arguments

`algebra`	A character string indicating the name of an MxAlgebra or MxMatrix object to use for optimization.
`numObs`	(optional) An adjustment to the total number of observations in the model.
`numStats`	(optional) An adjustment to the total number of observed statistics in the model.
`...`	Not used. Forces remaining arguments to be specified by name.
`gradient`	(optional) A character string indicating the name of an MxAlgebra object.
`hessian`	(optional) A character string indicating the name of an MxAlgebra object.
`verbose`	(optional An integer to increase the level of runtime log output.
`units`	(optional) The units of the fit statistic.
`strict`	Whether to require that all derivative entries reference free parameters.

Details

If you want to fit a multigroup model, the preferred way is to use mxFitFunctionMultigroup.

Fit functions are functions for which free parameter values are chosen such that the value of the objective function is minimized. While the other fit functions in OpenMx require an expectation function for the model, the `mxAlgebraObjective` function uses the referenced MxAlgebra or MxMatrix object as the function to be minimized.

If a model's fit function is an `mxFitFunctionAlgebra` objective function, then the referenced algebra in the objective function must return a 1 x 1 matrix (when using OpenMx's default optimizer). There is no restriction on the dimensions of an fit function that is not the primary, or 'topmost', objective function.

To evaluate an algebra fit function, place the following objects in a MxModel object: a `mxFitFunctionAlgebra`, MxAlgebra and MxMatrix entities referenced by the `MxAlgebraObjective`, and optional MxBounds and MxConstraint objects. This model may then be evaluated using the mxRun function. The results of the optimization may be obtained using the mxEval function on the name of the MxAlgebra, after the model has been run.

First and second derivatives can be provided with the algebra fit function. The dimnames on the gradient and hessian MxAlgebras are matched against names of free variables. Names that do not match are ignored. The fit is assumed to be in deviance units (-2 log likelihood units). If you are working in log likelihood units, the -2 scaling factor is not applied automatically. You have to multiply by -2 yourself.

Value

Returns an MxFitFunctionAlgebra object. MxFitFunctionAlgebra objects should be included with models with referenced MxAlgebra and MxMatrix objects.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

Other fit functions: mxFitFunctionMultigroup, mxFitFunctionML, mxFitFunctionWLS, mxFitFunctionGREML, mxFitFunctionR, mxFitFunctionRow

To create an algebra suitable as a reference function to be minimized see: mxAlgebra

More information about the OpenMx package may be found here.

Examples

```
# Create and fit a very simple model that adds two numbers using mxFitFunctionAlgebra

library(OpenMx)

# Create a matrix 'A' with no free parameters
A <- mxMatrix('Full', nrow = 1, ncol = 1, values = 1, name = 'A')

# Create an algebra 'B', which defines the expression A + A
B <- mxAlgebra(A + A, name = 'B')
```

```
# Define the objective function for algebra 'B'
objective <- mxFitFunctionAlgebra('B')

# Place the algebra, its associated matrix and
# its objective function in a model
tmpModel <- mxModel(model="Addition", A, B, objective)

# Evalulate the algebra
tmpModelOut <- mxRun(tmpModel)

# View the results
tmpModelOut$output$minimum
```

`mxFitFunctionGREML` *Create MxFitFunctionGREML Object*

Description

This function creates a new `MxFitFunctionGREML` object.

Usage

```
mxFitFunctionGREML(dV=character(0), aug=character(0),
   augGrad=character(0), augHess=character(0),
   autoDerivType=c("semiAnalyt","numeric"),infoMatType=c("average","expected"))
```

Arguments

`dV`	Vector of character strings; defaults to a character vector of length zero. If a value of non-zero length is provided, it must be a *named* character vector. This vector's names must be the labels of each free parameter in the model. The vector's elements (i.e., the character strings themselves) must be the names of `MxAlgebra` or `MxMatrix` object(s), each of which equals the first partial derivative of the 'V' matrix with respect to the corresponding free parameter.
`aug`	Character string; defaults to a character vector of length zero. Any elements after the first are ignored. The string should name a 1x1 `MxMatrix` or an `MxAlgebra` that evaluates to a 1x1 matrix. The named object will be used as an "augmentation" to the GREML fitfunction–specifically, the [1,1] value of the object named by aug will be added to the GREML fitfunction value at each function evaluation during optimization. The augmentation can be used to regularize estimation with a prior likelihood, or to use penalty functions to approximate constraints.
`augGrad`	Character string; defaults to a character vector of length zero. Any elements after the first are ignored. The string should name a `MxMatrix` or an `MxAlgebra` that evaluates to the gradient of aug with respect to free parameters. The gradient can be either a column or row vector. The free parameters corresponding to the elements of the gradient vector are taken from the names of argument dV, e.g. if

	the third name of `dV` is `'va'`, then the third element of the gradient vector should be the first partial derivative of the augmentation function with respect to `'va'`. Ignored unless both `dV` and `aug` have nonzero length.
`augHess`	Character string; defaults to a character vector of length zero. Any elements after the first are ignored. The string should name a `MxMatrix` or an `MxAlgebra` that evaluates to the Hessian of `aug` with respect to free parameters. The free parameters corresponding to each row and column of this matrix are dictated by the names of argument `dV`, in the same manner as for the elements of `augGrad`. Ignored unless both `dV` and `aug` have nonzero length. Providing a nonzero-length value for `augHess` but not `augGrad` will result in an error at runtime.
`autoDerivType`	"Automatic derivative type." Character string, either "semiAnalyt" (default) or "numeric". See details below.
`infoMatType`	"Information matrix type." Character string, either "average" (default) or "expected". See details below.

Details

Making effective use of arguments `dV`, `augGrad`, and `augHess` will usually require a custom `mxComputeSequence()`. The derivatives of the REML loglikelihood function with respect to parameters can be internally computed from the derivatives of the 'V' matrix supplied via `dV`. The loglikelihood's first derivatives thus computed will always be exact, but its matrix of second partial derivatives (i.e., its Hessian matrix) will be approximated by either the average or expected information matrix, per the value of argument `infoMatType`. The average information matrix is faster to compute, but may not provide a good approximation to the Hessian if 'V' is not linear in the model's free parameters. The expected information matrix is slower to compute, but does not assume that 'V' is linear in the free parameters. Neither information matrix will be a good approximation to the Hessian unless the derivatives of 'V' evaluate to symmetric matrices the same size as 'V'. Note also that these loglikelihood derivatives do not reflect the influence of any parameter bounds or `MxConstraint`s. Internally, the derivatives of the 'V' matrix are assumed to be symmetric, and the elements above their main diagonals are ignored.

Formerly, if any derivatives were provided via `dV`, then derivatives had to be provided for *every* free parameter in the MxModel. Currently, users may provide derivatives of 'V' via `dV` with respect to some or all free parameters. Note that the gradient and Hessian of the augmentation must be complete, i.e. contain derivatives of the augmentation with respect to every parameter or pair of parameters respectively.

If there are any free parameters with respect to which the user did not provide an analytic derivative of 'V', OpenMx will automatically calculate the necessary loglikelihood derivatives according to `autoDerivType`. If `autoDerivType="semiAnalyt"`, the GREML fitfunction backend will calculate the missing derivatives in a "semi-analytic" fashion. Specifically, the backend will numerically differentiate 'V' with respect to the relevant parameter(s), and use those numeric matrix derivatives to analytically calculate the needed loglikelihood derivatives. If `autoDerivType="numeric"`, the needed loglikelihood derivatives will be calculated numerically, via finite-differences.

Argument `aug` is intended to allow users to provide penalty functions or prior likelihoods in order to approximate constraints or to regularize optimization. The user is warned that careless use of this augmentation feature may undermine the validity of his/her statistical inferences.

Value

Returns a new object of class MxFitFunctionGREML.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

See MxFitFunctionGREML for the S4 class created by mxFitFunctionGREML(). For more information generally concerning GREML analyses, including a complete example, see mxExpectationGREML().

Other fit functions: mxFitFunctionMultigroup, mxFitFunctionML, mxFitFunctionWLS, mxFitFunctionAlgebra, mxFitFunctionR, mxFitFunctionRow

More information about the OpenMx package may be found here.

Examples

```
gff <- mxFitFunctionGREML()
str(gff)
```

MxFitFunctionGREML-class

Class "MxFitFunctionGREML"

Description

MxFitFunctionGREML is the fitfunction class for GREML analyses.

Objects from the Class

Objects can be created by calls of the form mxFitFunctionGREML(dV).

Slots

dV: Object of class "MxCharOrNumber". Identifies the MxAlgebra or MxMatrix object(s) to serve as the derivatives of 'V' with respect to free parameters.

dVnames: Vector of character strings; names of the free parameters corresponding to slot dV.

MLfit: Object of class "numeric", equal to the maximum-likelihood fitfunction value (as opposed to the restricted maximum-likelihood value).

numObsAdjust: Object of class "integer". Number of observations adjustment.

aug: Object of class "MxCharOrNumber". Identifies the MxAlgebra or MxMatrix object used to "augment" the fitfunction value at each function evaluation during optimization.

augGrad: Object of class "MxCharOrNumber". Identifies the MxAlgebra or MxMatrix object(s) to serve as the first derivatives of aug with respect to free parameters.

`augHess`: Object of class `"MxCharOrNumber"`. Identifies the `MxAlgebra` or `MxMatrix` object(s) to serve as the second derivatives of aug with respect to free parameters.

`autoDerivType`: Object of class `"character"`. Dictates whether fitfunction derivatives automatically calculated by OpenMx should be numeric or "semi-analytic."

`infoMatType`: Object of class `"character"`. Dictates whether to calculate the average or expected information matrix.

`info`: Object of class `"list"`.

`dependencies`: Object of class `"integer"`.

`expectation`: Object of class `"integer"`.

`vector`: Object of class `"logical"`.

`rowDiagnostics`: Object of class `"logical"`.

`result`: Object of class `"matrix"`.

`name`: Object of class `"character"`.

Extends

Class `"MxBaseFitFunction"`, directly. Class `"MxBaseNamed"`, by class "MxBaseFitFunction", distance 2. Class `"MxFitFunction"`, by class "MxBaseFitFunction", distance 2.

Methods

No methods defined with class "MxFitFunctionGREML" in the signature.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation`.

See Also

See `mxFitFunctionGREML()` for creating MxFitFunctionGREML objects. See `mxExpectationGREML()` for creating MxExpectationGREML objects, and for more information generally concerning GREML analyses, including a complete example. More information about the OpenMx package may be found here.

Examples

```
showClass("MxFitFunctionGREML")
```

mxFitFunctionML *Create MxFitFunctionML Object*

Description

This function creates a new MxFitFunctionML object.

Usage

```
mxFitFunctionML(vector = FALSE, rowDiagnostics = FALSE, ..., fellner =
   as.logical(NA), verbose=0L, profileOut=c(),
  rowwiseParallel=as.logical(NA), jointConditionOn = c("auto", "ordinal", "continuous"))
```

Arguments

`vector`	A logical value indicating whether the objective function result is the likelihood vector.
`rowDiagnostics`	A logical value indicating whether the row-wise results of the objective function should be returned as an attribute of the fit function.
`...`	Not used. Forces remaining arguments to be specified by name.
`fellner`	Whether to fully expand the covariance matrix for maximum flexibility.
`verbose`	Level of diagnostic output
`profileOut`	Character vector naming constant coefficients to profile out of the likelihood (sometimes known as REML)
`rowwiseParallel`	For raw data only, whether to use OpenMP to parallelize the evaluation of rows
`jointConditionOn`	The evaluation strategy when both continuous and ordinal data are present.

Details

Fit functions are functions for which free parameter values are optimized such that the value of a cost function is minimized. The mxFitFunctionML function computes -2*(log likelihood) of the data given the current values of the free parameters and the expectation function (e.g., mxExpectationNormal or mxExpectationRAM) selected for the model.

The 'vector' argument is either TRUE or FALSE, and determines whether the objective function returns a column vector of the likelihoods, or a single -2*(log likelihood) value.

The 'rowDiagnostics' argument is either TRUE or FALSE, and determines whether the row likelihoods are returned as an attribute of the fit function. Additionally, the squared Mahalanobis distance and the number of observed (non-missing) variables for each row are returned under the names `rowDist` and `rowObs`, respectively. It is sometimes useful to inspect the likelihoods for outliers, diagnostics, or other anomalies. Each rowwise squared Mahalanobis distance should be chi-squared distributed with degrees of freedom equal to the number of observed variables. In the case of no missing data, all of the rowwise squared Mahalanobis distances should theoretically be chi-squared distributed with the same degrees of freedom. In the case of some missing data, the rowwise squared Mahalanobis distances should theoretically be a mixture of chi-squared distributions with mixing proportions equal to the proportions of each number of observed variables.

If there are ordinal data, then only the row likelihoods are returned among the row diagnostics.

When `vector=FALSE` and `rowDiagnostics=TRUE`, the fit function can be referenced in the model and included in algebras as a scalar. The row likelihoods, row distances, and row observations are then an attribute of the fit function but are not accessible in the model during optimization. The row likelihoods and other diagnostics are accessible to the user after the model has been run.

By default, `jointConditionOn='auto'` and a heuristic will be used to select the fastest algorithm. Conditioning the continuous data on ordinal will be superior when there are relatively few unique ordinal patterns. Otherwise, conditioning the ordinal data on continuous will perform better when there are relatively many ordinal patterns.

Usage Notes:

The results of the optimization can be reported using the summary function, or accessed directly in the 'output' slot of the resulting model (i.e., modelName$output). Components of the output may be referenced using the Extract functionality.

Value

Returns a new MxFitFunctionML object. One and only one MxFitFunctionML object should be included in each model along with an associated mxExpectationNormal or mxExpectationRAM object.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

Other fit functions: `mxFitFunctionMultigroup`, `mxFitFunctionWLS`, `mxFitFunctionAlgebra`, `mxFitFunctionGREML`, `mxFitFunctionR`, `mxFitFunctionRow`

More information about the OpenMx package may be found here.

Examples

```
# Create and fit a model using mxMatrix, mxAlgebra, mxExpectationNormal, and mxFitFunctionML

library(OpenMx)

# Simulate some data

x=rnorm(1000, mean=0, sd=1)
y= 0.5*x + rnorm(1000, mean=0, sd=1)
tmpFrame <- data.frame(x, y)
tmpNames <- names(tmpFrame)

# Define the matrices

M <- mxMatrix(type = "Full", nrow = 1, ncol = 2, values=c(0,0),
              free=c(TRUE,TRUE), labels=c("Mx", "My"), name = "M")
S <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(1,0,0,1),
              free=c(TRUE,FALSE,FALSE,TRUE), labels=c("Vx", NA, NA, "Vy"), name = "S")
A <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(0,1,0,0),
              free=c(FALSE,TRUE,FALSE,FALSE), labels=c(NA, "b", NA, NA), name = "A")
I <- mxMatrix(type="Iden", nrow=2, ncol=2, name="I")

# Define the expectation

expCov <- mxAlgebra(solve(I-A) %*% S %*% t(solve(I-A)), name="expCov")
expFunction <- mxExpectationNormal(covariance="expCov", means="M", dimnames=tmpNames)

# Choose a fit function

fitFunction <- mxFitFunctionML(rowDiagnostics=TRUE)
# also return row likelihoods, even though the fit function
#  value is still 1x1

# Define the model

tmpModel <- mxModel(model="exampleModel", M, S, A, I, expCov, expFunction, fitFunction,
                    mxData(observed=tmpFrame, type="raw"))

# Fit the model and print a summary

tmpModelOut <- mxRun(tmpModel)
summary(tmpModelOut)

fitResOnly <- mxEval(fitfunction, tmpModelOut)
attributes(fitResOnly) <- NULL
fitResOnly

# Look at the row likelihoods alone
fitLikeOnly <- attr(mxEval(fitfunction, tmpModelOut), 'likelihoods')
head(fitLikeOnly)
```

---

mxFitFunctionMultigroup

Create a fit function used to fit multiple-group models

Description

mxFitFunctionMultigroup creates a fit function consisting of the sum of the fit statistics from a list of submodels provided. Thus, it aggregates fit statistics from multiple submodels.

This total provides the optimization target for fitting a multi-group model.

In addition to being more compact and readable, using mxFitFunctionMultigroup has additional side effects which are valuable for multi-group modeling.

First, it aggregates analytic derivative calculations.

Second, it allows `mxRefModels` to compute saturated models for raw data, as this function can learn which are the constituent submodels.

Third, and finally, it allows `mxCheckIdentification` to evaluate the local identification of the multigroup model.

Usage

```
mxFitFunctionMultigroup(groups, ..., verbose = 0L)
```

Arguments

`groups`	vector of submodel names (strings)
`...`	Not used. Forces subsequent arguments to be specified by name.
`verbose`	the level of debugging output

Details

Conceptually, `mxFitFunctionMultigroup` is equivalent to summing the subModel objectives in an mxAlgebra, and using an mxFitFunctionAlgebra to optimize the model based on this summed likelihood.

e.g. this 1-line call to mxFitFunctionMultigroup:

`mxFitFunctionMultigroup(c("model1","model2"))`

is equivalent to the following pair of statements:

`mxAlgebra(name = "myAlgebra",model1.objective + model2.objective)`

`mxFitFunctionAlgebra("myAlgebra")`

Note: If needed, you can refer to the algebra generated by `mxFitFunctionMultigroup` as:

`modelName.fitfunction`

Where "modelName" is the name of the container or supermodel.

See Also

Other fit functions: `mxFitFunctionML`, `mxFitFunctionWLS`, `mxFitFunctionAlgebra`, `mxFitFunctionGREML`, `mxFitFunctionR`, `mxFitFunctionRow`

More information about the OpenMx package may be found here.

Examples

```
#------------------------------------------------
# Brief non-running example
require("OpenMx")
mxFitFunctionMultigroup(c("model1", "model2")) # names of sub-models to be jointly optimised



# ============================================
# = Longer, fully featured, running example =
# ============================================
# Create and fit a model using mxMatrix, mxExpectationRAM, mxFitFunctionML,
# and mxFitFunctionMultigroup.
# The model is multiple group regression.
# Only the residual variances are allowed to differ across groups.


library(OpenMx)

# Simulate some data

# Group 1
N1 = 100
x = rnorm(N1, mean= 0, sd= 1)
y = 0.5*x + rnorm(N1, mean= 0, sd= 1)
ds1 <- data.frame(x, y)
dsNames <- names(ds1)

# Group 2: y has greater variance; x & y slightly lower correlation...
N2= 150
x= rnorm(N2, mean= 0, sd= 1)
y= 0.5*x + rnorm(N2, mean= 0, sd= sqrt(1.5))
ds2 <- data.frame(x, y)


# Define the matrices (A matrix implementation of 2 RAM models)

I <- mxMatrix(name="I", type="Iden", nrow=2, ncol=2)
M <- mxMatrix(name = "M", type = "Full", nrow = 1, ncol = 2, values=0,
              free=TRUE, labels=c("Mean_x", "Mean_y"))
# A matrix containing a path "b" of x on y
A <- mxMatrix(name = "A", type = "Full", nrow = 2, ncol = 2, values=c(0,1,0,0),
              free=c(FALSE,TRUE,FALSE,FALSE), labels=c(NA, "b", NA, NA))

S1 <- mxMatrix(name = "S", type = "Diag", nrow = 2, ncol = 2, values=1,
              free=TRUE, labels=c("Var_x", "Resid_y_group1"))
S2 <- mxMatrix(name = "S", type = "Diag", nrow = 2, ncol = 2, values=1,
              free=TRUE, labels=c("Var_x", "Resid_y_group2"))

# Define the expectation
expect <- mxExpectationRAM('A', 'S', 'I', 'M', dimnames= dsNames)


# Choose a fit function
fitFunction <- mxFitFunctionML(rowDiagnostics=TRUE)
# Also return row likelihoods (the fit function value is still 1x1)

# Multiple-group fit function sums the model likelihoods
# from its component models
mgFitFun <- mxFitFunctionMultigroup(c('g1model', 'g2model'))


# Define model 1 and model 2
```

```
m1 = mxModel(model="g1model",
M, S1, A, I, expect, fitFunction,
    mxData(cov(ds1), type="cov", numObs=N1, means=colMeans(ds1))
)
m2 = mxModel(model="g2model",
M, S2, A, I, expect, fitFunction,
    mxData(cov(ds2), type="cov", numObs=N2, means=colMeans(ds2))
)

mg <- mxModel(model='multipleGroup', m1, m2, mgFitFun)
# note!: Paths with the same name in both submodels are
# constrained to the same value across models. i.e.,
# b has only 1 value, as does Var_x. But Resid_y can take distinct
# values in the two groups.

# Fit the model and print a summary
mg <- mxRun(mg)
summary(mg)

# Examine fit function results
# Fit in -2lnL units)
mxEval(fitfunction, mg)

# Fit function results for each submodel:
mxEval(g1model.fitfunction, mg)
mxEval(g2model.fitfunction, mg)

mg2 = omxSetParameters(mg,
  labels = c("Resid_y_group1", "Resid_y_group2"),
  newlabels = "Resid_y", name = "equated")
mg2 = omxAssignFirstParameters(mg2)
mg2 = mxRun(mg2)

mxCompare(mg, mg2)
# ouch... that was a significant loss in fit: the residuals definately are larger in group2!
```

`mxFitFunctionR` *Create MxFitFunctionR Object*

Description

mxFitFunctionR returns an MxFitFunctionR object.

Usage

```
mxFitFunctionR(fitfun, ..., units="-2lnL")
```

Arguments

`fitfun`	A function that accepts two arguments.
`...`	The initial state information to the objective function.
`units`	(optional) The units of the fit statistic.

Details

The mxFitFunctionR function evaluates a user-defined R function called the 'fitfun'. mxFitFunctionR is useful in defining new mxFitFunctions, since any calculation that can be performed in R can be treated as an mxFitFunction.

The 'fitfun' argument must be a function that accepts two arguments. The first argument is the mxModel that should be evaluated, and the second argument is some persistent state information that can be stored between one iteration of optimization to the next iteration. It is valid for the function to simply ignore the second argument.

The function must return either a single numeric value, or a list of exactly two elements. If the function returns a list, the first argument must be a single numeric value and the second element will be the new persistent state information to be passed into this function at the next iteration. The single numeric value will be used by the optimizer to perform optimization.

The initial default value for the persistent state information is NA.

Throwing an exception (via stop) from inside fitfun may result in unpredictable behavior. You may want to wrap your code in tryCatch while experimenting.

fitfun should not call R functions that use OpenMx's compiled backend, including (but not limited to) `omxMnor()`, because doing so can crash R.

Value

Returns an MxFitFunctionR object.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

Other fit functions: `mxFitFunctionMultigroup`, `mxFitFunctionML`, `mxFitFunctionWLS`, `mxFitFunctionAlgebra`, `mxFitFunctionGREML`, `mxFitFunctionRow`

More information about the OpenMx package may be found here.

Examples

```
# Create and fit a model using mxFitFunctionR

library(OpenMx)

A <- mxMatrix(nrow = 2, ncol = 2, values = c(1:4), free = TRUE, name = 'A')
squared <- function(x) { x ^ 2 }
```

```
# Define the objective function in R

objFunction <- function(model, state) {
    values <- model$A$values
    return(squared(values[1,1] - 4) + squared(values[1,2] - 3) +
        squared(values[2,1] - 2) + squared(values[2,2] - 1))
}

# Define the expectation function

fitFunction <- mxFitFunctionR(objFunction)

# Define the model

tmpModel <- mxModel(model="exampleModel", A, fitFunction)

# Fit the model and print a summary

tmpModelOut <- mxRun(tmpModel)
summary(tmpModelOut)
```

`mxFitFunctionRow` *Create an MxFitFunctionRow Object*

Description

mxFitFunctionRow returns an MxFitFunctionRow object.

Usage

```
mxFitFunctionRow(rowAlgebra, reduceAlgebra, dimnames,
   rowResults = "rowResults", filteredDataRow = "filteredDataRow",
   existenceVector = "existenceVector", units="-2lnL")
```

Arguments

`rowAlgebra`	A character string indicating the name of the algebra to be evaluated row-wise.
`reduceAlgebra`	A character string indicating the name of the algebra that collapses the row results into a single number which is then optimized.
`dimnames`	A character vector of names corresponding to columns be extracted from the data set.
`rowResults`	The name of the auto-generated "rowResults" matrix. See details.
`filteredDataRow`	The name of the auto-generated "filteredDataRow" matrix. See details.
`existenceVector`	The name of the auto-generated "existenceVector" matrix. See details.
`units`	(optional) The units of the fit statistic.

Details

Fit functions are functions for which free parameter values are optimized such that the value of a cost function is minimized. The mxFitFunctionRow function evaluates a user-defined MxAlgebra object called the ‘rowAlgebra’ in a row-wise fashion. It then stores results of the row-wise evaluation in another MxAlgebra object called the ‘rowResults’. Finally, the mxFitFunctionRow function collapses the row results into a single number which is then used for optimization. The MxAlgebra object named by the ‘reduceAlgebra’ collapses the row results into a single number.

The ‘filteredDataRow’ is populated in a row-by-row fashion with all the non-missing data from the current row. You cannot assume that the length of the filteredDataRow matrix remains constant (unless you have no missing data). The ‘existenceVector’ is populated in a row-by-row fashion with a value of 1.0 in column j if a non-missing value is present in the data set in column j, and a value of 0.0 otherwise. Use the functions omxSelectRows, omxSelectCols, and omxSelectRowsAndCols to shrink other matrices so that their dimensions will be conformable to the size of ‘filteredDataRow’.

Value

Returns a new MxFitFunctionRow object. Only one MxFitFunction object should be included in each model. There is no need for an MxExpectation object when using mxFitFunctionRow.

References

The OpenMx User’s guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

Other fit functions: `mxFitFunctionMultigroup`, `mxFitFunctionML`, `mxFitFunctionWLS`, `mxFitFunctionAlgebra`, `mxFitFunctionGREML`, `mxFitFunctionR`

More information about the OpenMx package may be found here.

Examples

```
# Model that adds two data columns row-wise, then sums that column
# Notice no optimization is performed here.

library(OpenMx)

xdat <- data.frame(a=rnorm(10), b=1:10) # Make data set
amod <- mxModel(model="example1",
            mxData(observed=xdat, type='raw'),
            mxAlgebra(sum(filteredDataRow), name = 'rowAlgebra'),
            mxAlgebra(sum(rowResults), name = 'reduceAlgebra'),
            mxFitFunctionRow(
                rowAlgebra='rowAlgebra',
                reduceAlgebra='reduceAlgebra',
                dimnames=c('a','b'))
)
amodOut <- mxRun(amod)
mxEval(rowResults, model=amodOut)
mxEval(reduceAlgebra, model=amodOut)
```

```
# Model that find the parameter that minimizes the sum of the
#  squared difference between the parameter and a data row.

bmod <- mxModel(model="example2",
            mxData(observed=xdat, type='raw'),
            mxMatrix(values=.75, ncol=1, nrow=1, free=TRUE, name='B'),
            mxAlgebra((filteredDataRow - B) ^ 2, name='rowAlgebra'),
            mxAlgebra(sum(rowResults), name='reduceAlgebra'),
            mxFitFunctionRow(
                rowAlgebra='rowAlgebra',
                reduceAlgebra='reduceAlgebra',
                dimnames=c('a'))
)
bmodOut <- mxRun(bmod)
mxEval(B, model=bmodOut)
mxEval(reduceAlgebra, model=bmodOut)
mxEval(rowResults, model=bmodOut)
```

---

mxFitFunctionWLS *Create MxFitFunctionWLS Object*

Description

This function creates a new MxFitFunctionWLS object.

Usage

```
mxFitFunctionWLS(type=c('WLS','DWLS','ULS'),
     allContinuousMethod=c("cumulants", "marginals"),
     fullWeight=TRUE)
```

Arguments

`type`	A character string 'WLS' (default), 'DWLS', or 'ULS' for weighted, diagonally weighted, or unweighted least squares, respectively
`allContinuousMethod`	A character string 'cumulants' (default) or 'marginals'. See Details.
`fullWeight`	Logical determining if the full weight matrix is returned (default). Needed for standard error and quasi-chi-squared calculation.

Details

As with other fit functions, `mxFitFunctionWLS` optimizes free parameter values such that the value of a cost function is minimized. For `mxFitFunctionWLS`, this cost function is the weighted least squares difference between the data and the model-implied expectations for the data based on the free parameters and the expectation function (e.g., `mxExpectationNormal` or `mxExpectationRAM`) selected for the model.

Bias and sensitivity to model misspecification Both ordinal and continuous data are supported, as well as combinations of these data types. All three methods ('WLS', 'ULS' and 'DWLS') are unbiased when the model is correct. Full 'WLS' is highly sensitive to model misspecification – it can heavily weight the fourth-order moments of the distribution, so small deviations between the observed fourth-order moments and those implied by the model can lead to poor estimates.

Behavior with all-continuous data When only continuous variables are present, the argument `allContinuousMethod` dictates how to process the data.

The default, *cumulants* is a good choice for non-normal data. This uses the asymptotically distribution free (ADF) method of Browne (1984) and computes the fourth order *cumulants* for the weight matrix: thus, the name. It is generally fast and ADF up to elliptical distributions. Data computed using cumulants should also be more accurate than via marginals (because the whole covariance is a single analytic expression, with no estimation involved).

note: The *cumulants* method does not handle missing data. It also does not return weights or summary statistics for the means.

The alternative option, 'marginals', uses methods similar to those used in processing ordinal and joint ordinal-continuous data. By contrast with cumulants, marginals returns weights and summary statistics for the means.

When data are not all continuous, `allContinuousMethod` is ignored, and means are modelled.

Usage Notes:

Model results can be reported using the summary function, or accessed directly in the 'output' slot of the model (i.e., `model$output`). Components of the output may also be accessed and used in the same way, i.e., via the `$` and `[]` `Extract` functions.

Summary statistics are returned in the MxData object in an `observedStats` list. If `observedStats` are already present and in the appropriate shape then they are reused. It is also possible to provide your own arbitrary user supplied observed statistics using this same approach.

Value

Returns a new MxFitFunctionWLS object. One and only one fit function object should be included in each model, along with an associated mxExpectationNormal or `mxExpectationRAM` object.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Browne, M. W. (1984). Asymptotically distribution-free methods for the analysis of covariance structures. *British Journal of Mathematical and Statistical Psychology*, **37**, 62-83.

See Also

Other fit functions: `mxFitFunctionMultigroup`, `mxFitFunctionML`, `mxFitFunctionAlgebra`, `mxFitFunctionGREML`, `mxFitFunctionR`, `mxFitFunctionRow`

More information about the OpenMx package may be found here.

Examples

```
# Create and fit a WLS model using RAM, and then using matrices.

library(OpenMx)

# Simulate some data where y = .5x + error

x = rnorm(1000, mean = 0, sd = 1)
y = 0.5*x + rnorm(1000, mean = 0, sd = 1)
tmpFrame = data.frame(x, y)
varNames = names(tmpFrame)

# ======================
# = A RAM model example =
# ======================

m1 = mxModel("my_first_WLS", type = "RAM",
manifestVars = c("x", "y"),
mxPath(c("x", "y"), arrows = 2, values = 1, labels = c("xVar", "yVar")),
mxPath("x", to = "y", labels = "x_to_y"),
mxFitFunctionWLS(),
mxData(tmpFrame, 'raw')
)

m1 = mxRun(m1)
summary(m1)$parameters

# Here are the cov, acov and Weight matrices:
print(m1$data$observedStats)

# Use a different weight matrix
m2 = m1
os <- m1$data$observedStats
os$asymCov <- solve(rWishart(n=1, df= nrow(tmpFrame), Sigma= diag(3))[,,1])
os$useWeight <- solve(os$asymCov * nrow(tmpFrame))
m2$data$observedStats <- os

# Set verbose to check if our new weights are used
m2$data$verbose <- 1L

# Run model
m2 <- mxRun(m2)

# SE indeed changed due to new weights
print(m2$output$standardErrors - m1$output$standardErrors)

# =========================
# = A matrix-based example =
# =========================

# Define matrices for Symmetric (S) and Asymmetric (A) paths and an Identity matrix.
```

```
S <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(1,0,0,1),
              free=c(TRUE,FALSE,FALSE,TRUE), labels=c("Vx", NA, NA, "Vy"), name = "S")
A <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(0,1,0,0),
              free=c(FALSE,TRUE,FALSE,FALSE), labels=c(NA, "b", NA, NA), name = "A")
I <- mxMatrix(type="Iden", nrow=2, ncol=2, name="I")

# Build the model

tmpModel <- mxModel(model="exampleModel",
# Add the S, A, and I matrices constructed above
S, A, I,

# Define the expectation
mxAlgebra(name="expCov", solve(I-A) %*% S %*% t(solve(I-A))),

# Choose a normal expectation and WLS as the fit function
mxExpectationNormal(covariance= "expCov", dimnames= varNames),
mxFitFunctionWLS(),

   # Add the data
mxData(tmpFrame, 'raw')
)

# Fit the model and print a summary
tmpModel <- mxRun(tmpModel)
summary(tmpModel)
```

`MxFlatModel-class` *MxFlatModel*

Description

This is an internal class and should not be used.

`mxGenerateData` *Generate data based on an mxModel (or a data.frame)*

Description

This function returns a new (simulated) data set based on either the model-implied distribution if a model is provided, OR saturated model if a data.frame is given in the model parameter.

See below for important details

Usage

```
mxGenerateData(model, nrows, returnModel=FALSE, use.miss = TRUE,
 ..., .backend=TRUE, subname=NULL, empirical=FALSE, nrowsProportion,
 silent=FALSE)
```

Arguments

`model`	A data.frame or MxModel object upon which the data are generated.
`nrows`	Numeric. The number of rows of data to generate (default = same as in the original data)
`returnModel`	Whether to return the model with new data, or just return the new data.frames (default)
`use.miss`	Whether to approximate the missingness pattern of the original data (TRUE by default).
`...`	Not used; forces remaining arguments to be specified by name.
`.backend`	Whether to use the backend to generate data (TRUE by default for speed)
`subname`	If given, limits data generation to this sub model.
`empirical`	Whether the generate data should match the distribution of the current data exactly. Uses mvrnorm instead of rmvnorm
`nrowsProportion`	Numeric. The number of rows of data to generate expressed as a proportion of the current number of rows.
`silent`	Logical. Whether to report progress during time consuming data generation.

Details

When given a data.frame as a model, the model is assumed to be saturated multivariate Gaussian and the expected distribution is obtained using mxDataWLS. In this case, the default number of rows is assumed to be the number of rows in the original data.frame, but any other number of rows can also be requested.

When given an MxModel, the model-implied means and covariance are extracted. It then generates data with the same mean and covariance. Data can be generated based on Normal (mxExpectationNormal), RAM (mxExpectationRAM), LISREL (mxExpectationLISREL), and state space (mxExpectationStateSpace) models.

Please note that this function samples data from the model-implied distribution(s); it does not sample from the data object in the model. That is, this function generates new data rather than pulling data that already exist from the model.

Thresholds and ordinal data are implemented by generating continuous data and then using cut and mxFactor to break the continuous data at the thresholds into an ordered factor.

If the model has definition variables, then a data set must be included in the model object and the number of rows requested must match the number of rows in the model data. In this case the means, covariance, and thresholds are reevaluated for each row of data, potentially creating a a different mean, covariance, and threshold structure for every generated row of data.

For state space models (i.e. models with an mxExpectationStateSpace or mxExpectationStateSpaceContinuousTime expectation), the data are generated based on the autoregressive structure of the model. The rows of data in a state space model are not independent replicates of a stationary process. Rather, they are the result of a latent (possibly non-stationary) autoregressive process. For state space models different rows of data often correspond to different times. As alluded to above, data generation works for discrete time state space models and hybrid continuous-discrete time state space models. The latter have a continuous process that is measured as discrete times.

The subname parameter is used to limit data generation to the given submodel. The reason you wouldn't pass the submodel in the `model` argument is that some parts of the submodel might depend on objects in other submodels that are part of the model.

Value

A data.frame, list of data.frames, or model populated with the new data (depending on the `returnModel` parameter). Raw data is always returned even if the original model contained covariance or some other non-raw data.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
# ====================================
# = Demonstration for empirical=TRUE =
# ====================================
popCov <- cov(Bollen[, 1:8])*(nrow(Bollen)-1)/nrow(Bollen)
got <- mxGenerateData(Bollen[, 1:8], nrows=nrow(Bollen), empirical = TRUE)
cov(got) - popCov  # pretty close, given 8 variables to juggle!
round(cov2cor(cov(got)) - cov2cor(popCov), 4)

# ===========================================
# = Create data based on state space model. =
# ===========================================

require(OpenMx)
nvar <- 5
varnames <- paste("x", 1:nvar, sep="")
ssModel <- mxModel(model="State Space Manual Example",
    mxMatrix("Full", 1, 1, TRUE, .3, name="A"),
    mxMatrix("Zero", 1, 1, name="B"),
    mxMatrix("Full", nvar, 1, TRUE, .6, name="C", dimnames=list(varnames, "F1")),
    mxMatrix("Zero", nvar, 1, name="D"),
    mxMatrix("Diag", 1, 1, FALSE, 1, name="Q"),
    mxMatrix("Diag", nvar, nvar, TRUE, .2, name="R"),
    mxMatrix("Zero", 1, 1, name="x0"),
    mxMatrix("Diag", 1, 1, FALSE, 1, name="P0"),
    mxMatrix("Zero", 1, 1, name="u"),
    mxExpectationStateSpace("A", "B", "C", "D", "Q", "R", "x0", "P0", "u"),
    mxFitFunctionML()
)

ssData <- mxGenerateData(ssModel, 200) # 200 time points

# Add simulated data to model and run
ssModel <- mxModel(ssModel, mxData(ssData, 'raw'))
ssRun <- mxRun(ssModel)

# Compare parameters from random data to the generating model
```

```
cbind(Rand = omxGetParameters(ssRun), Gen = omxGetParameters(ssModel))

# Note the parameters should be "close" (up to sampling error)
# to the generating values


# ========================================
# = Demo generating new data from a model =
# ========================================
require(OpenMx)
manifests <- paste0("x", 1:5)
originalModel <- mxModel("One Factor", type="RAM",
      manifestVars = manifests,
      latentVars = "G",
      mxPath(from="G", to=manifests, values=.8),
      mxPath(from=manifests, arrows=2, values=.2),
      mxPath(from="G"  , arrows=2, free=FALSE, values=1.0),
      mxPath(from = 'one', to = manifests)
)

factorData <- mxGenerateData(originalModel, 1000)
newData = mxData(cov(factorData), type="cov",
numObs=nrow(factorData), means = colMeans(factorData)
)
newModel <- mxModel(originalModel, newData)
newModel <- mxRun(newModel)
cbind(
Original = omxGetParameters(originalModel),
Generated = round(omxGetParameters(newModel), 4),
Delta = round(
omxGetParameters(originalModel) -
omxGetParameters(newModel), 3)
)

# And again with empirical = TRUE

factorData <- mxGenerateData(originalModel, 1000, empirical = TRUE)
newData = mxData(cov(factorData),
type = "cov",
numObs = nrow(factorData),
means = colMeans(factorData)
)

newModel <- mxModel(originalModel, newData)
newModel <- mxRun(newModel)

cbind(
Original  = omxGetParameters(originalModel),
Generated = round(omxGetParameters(newModel), 4),
Delta     = omxGetParameters(originalModel) -
      omxGetParameters(newModel)
)
```

mxGetExpected *Extract the component from a model's expectation*

Description

This function extracts the expected means, covariance, or thresholds from a model.

Usage

```
mxGetExpected(model, component, defvar.row=1, subname=model$name)
imxGetExpectationComponent(model, component, defvar.row=1, subname=model$name)
```

Arguments

`model`	MxModel object from which to extract the expectation component.
`component`	Character vector. The name(s) of the component(s) to extract. Recognized names are "covariance", "means", and "thresholds".
`defvar.row`	A row index. Which row to load for definition variables.
`subname`	Name of the submodel to evaluate.

Details

The expected means, covariance, or thresholds can be extracted from Normal (mxExpectationNormal), RAM (mxExpectationRAM), and LISREL (mxExpectationLISREL) models. When more than one component is requested, the components will be returned as a list.

If component 'vector' is requested then the non-redundant coefficients of the expected manifest distribution will be returned as a vector.

If component 'standVector' is requested then the same parameter structure as 'vector' is returned, but it is standardized. For Normal expectations the covariances are returned as correlations, the means are returned as zeros, and the thresholds are returned as z-scores. For the thresholds the z-scores are computed by using the model-implied means and variances.

Note that capitalization is ignored for the 'standVector' option, so 'standvector' is also acceptable.

Value

See details.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
# ===============================================
# = Build a 1-factor CFA, with bad start values =
# ===============================================
require(OpenMx)
manifests = paste("x", 1:5, sep="")
latents = c("G")
factorModel = mxModel("One Factor", type="RAM",
      manifestVars = manifests,
      latentVars = latents,
      mxPath(from = latents, to = manifests),
      mxPath(from = manifests, arrows = 2),
      mxPath(from = latents, arrows = 2, free = FALSE, values = 1.0),
      mxPath(from = 'one', to = manifests),
  mxData(demoOneFactor, type = "raw")
)

# ============================================================================
# = What do our starting values indicate about the expected data covariance? =
# ============================================================================
mxGetExpected(factorModel, "covariance")

# Oops. Starting values indicate an expected zero-covariance matrix.
# The model likely won't run from these start values.
# Let's adjust them:

factorModel = mxModel("One Factor", type = "RAM",
      manifestVars = manifests, latentVars = latents,
      # Reasonable start VALUES
  mxPath(from = latents, to = manifests, values = .2),
      mxPath(from = manifests, arrows = 2),
      mxPath(from = latents, arrows = 2, free = FALSE, values = 1.0),
      mxPath(from = 'one', to = manifests),
  mxData(demoOneFactor, type = "raw")
)

mxGetExpected(factorModel, "covariance")
#      x1   x2   x3   x4   x5
# x1 0.04 0.04 0.04 0.04 0.04
# x2 0.04 0.04 0.04 0.04 0.04
# x3 0.04 0.04 0.04 0.04 0.04
# x4 0.04 0.04 0.04 0.04 0.04
# x5 0.04 0.04 0.04 0.04 0.04

# And this version will run:
factorModel = mxRun(factorModel)
```

`mxGREMLDataHandler` *Helper Function for Structuring GREML Data*

Description

This function takes a dataframe or matrix and uses it to setup the 'y' and 'X' matrices for a GREML analysis; this includes trimming out `NA`s from 'X' and 'y.' The result is a matrix the first column of which is the 'y' vector, and the remaining columns of which constitute 'X.'

Usage

```
mxGREMLDataHandler(data, yvars=character(0), Xvars=list(), addOnes=TRUE,
                   blockByPheno=TRUE, staggerZeroes=TRUE)
```

Arguments

`data`	Either a dataframe or matrix, with column names, containing the variables to be used as phenotypes and covariates in 'y' and 'X,' respectively.
`yvars`	Character vector. Each string names a column of the raw dataset, to be used as a phenotype.
`Xvars`	A list of data column names, specifying the covariates to be used with each phenotype. The list should have the same length as argument `yvars`.
`addOnes`	Logical; should lead columns of ones (for the regression intercepts) be adhered to the covariates when assembling the 'X' matrix? Defaults to `TRUE`.
`blockByPheno`	Logical; relevant to polyphenotype analyses. If `TRUE` (default), then the resulting 'y' will contain phenotype #1 for individuals 1 thru *n*, phenotype #2 for individuals 1 thru *n*, ... If `FALSE`, then observations are "blocked by individual", and the resulting 'y' will contain individual #1's scores on phenotypes 1 thru *p*, individual #2's scores on phenotypes 1 thru *p*, ... Note that in either case, 'X' will be structured appropriately for 'y.'
`staggerZeroes`	Logical; relevant to polyphenotype analyses. If `TRUE` (default), then each phenotype's covariates in 'X' are "staggered," and 'X' is padded out with zeroes. If `FALSE`, then 'X' is formed simply by stacking the phenotypes' covariates; this requires each phenotype to have the same number of covariates (i.e., each character vector in `Xvars` must be of the same length). The default (`TRUE`) is intended for instances where the multiple phenotypes truly are different variables, whereas `staggerZeroes=FALSE` is intended for instances where the multiple "phenotypes" actually represent multiple observations on the same variable. One example of the latter case is longitudinal data where the multiple "phenotypes" are repeated measures on a single phenotype.

Details

For a monophenotype analysis (only), argument `Xdata` can be a character vector. In a polyphenotype analysis, if the same covariates are to be used with all phenotypes, then `Xdata` can be a list of length 1.

Note the synergy between the output of `mxGREMLDataHandler()` and arguments `dataset.is.yX` and `casesToDropFromV` to `mxExpectationGREML()`.

If the dataframe or matrix supplied for argument `data` has *n* rows, and argument `yvars` is of length *p*, then the resulting 'y' and 'X' matrices will have *np* rows. Then, if either matrix contains any `NA`'s, the rows containing the NA's are trimmed from both 'X' and 'y' before being returned in the output (in which case they will obviously have fewer than *np* rows). Function mxGREMLDataHandler() reports which rows of the full-size 'X' and 'y' were trimmed out due to missing observations. These row indices can be provided as argument casesToDropFromV to mxExpectationGREML().

Value

A list with these two components:

yX	Numeric matrix. The first column is the phenotype vector, 'y,' while the remaining columns constitute the 'X' matrix of covariates. If this matrix is used as the raw dataset for a model, then the model's GREML expectation can be constructed with dataset.is.yX=TRUE in mxExpectationGREML().
casesToDrop	Numeric vector. Contains the indices of the rows of the 'y' and 'X' that were dropped due to containing NA's. Can be provided as as argument casesToDropFromV to mxExpectationGREML().

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

For more information generally concerning GREML analyses, including a complete example, see mxExpectationGREML(). More information about the OpenMx package may be found here.

Examples

```
dat <- cbind(rnorm(100),rep(1,100))
colnames(dat) <- c("y","x")
dat[42,1] <- NA
dat[57,2] <- NA
dat2 <- mxGREMLDataHandler(data=dat, yvars="y", Xvars=list("x"),
  addOnes = FALSE)
str(dat2)
```

MxInterval-class *MxInterval*

Description

This is an internal class and should not be used directly.

See Also

mxCI

`mxJiggle` *Jiggle parameter values.*

Description

Jiggle free parameter values, subject to box constraints. `imxJiggle()` is called internally by `mxTryHard()` (q.v.). `mxJiggle()` provides a more user-friendly wrapper to `imxJiggle()`, and can alternately emulate the 'JIGGLE' behavior of classic Mx.

Usage

```
mxJiggle(model, classic=FALSE, dsn=c("runif","rnorm","rcauchy"), loc=1, scale=0.25)
imxJiggle(params, lbounds, ubounds, dsn, loc, scale)
```

Arguments

`model`	An object of class MxModel.
`classic`	Logical; should `mxJiggle()` emulate the classic-Mx behavior elicited by keyword JIGGLE? Defaults to `FALSE`. See below, under "Details," for additional information.
`dsn`	Character string naming which random-number distribution–either uniform (rectangular), normal (Gaussian), or Cauchy–to be used to perturb free-parameter values. Defaults to the uniform distribution (for `mxJiggle()`).
`loc, scale`	Numeric. The location and scale parameters of the distribution from which random values are drawn to perturb free-parameter values, defaulting respectively to 1 and 0.25 (for `mxJiggle()`).
`params`	Numeric vector of current free parameter values.
`lbounds`	Numeric vector of lower bounds on parameters.
`ubounds`	Numeric vector of upper bounds on parameters.

Details

If `mxJiggle()` argument `classic=FALSE` (the default), `mxJiggle()` calls `imxJiggle()`. In that case, `mxJiggle()` passes `imxJiggle()` its own values for arguments `dsn`, `loc`, and `scale`, and extracts values for arguments `params`, `lbounds`, and `ubounds` from `model`. Then, `model`'s free-parameter values are randomly perturbed before being re-assigned to it. The distributional family from which the perturbations are randomly generated is dictated by argument `dsn`. The distribution is parameterized by arguments `loc` and `scale`, respectively the location and scale parameters. The location parameter is the distribution's median. For the uniform distribution, `scale` is the absolute difference between its median and extrema (i.e., half the width of the rectangle); for the normal distribution, `scale` is its standard deviation; and for the Cauchy, `scale` is one-half its interquartile range. Free-parameter values are first multiplied by random draws from a distribution with the provided `loc` and `scale`, then added to random draws from a distribution with the same `scale` but with a median of zero.

If `mxJiggle()` argument `classic=TRUE`, then each free-parameter value x_i is replaced with $x_i + 0.1(x_i + 0.5)$; this is the same behavior elicited in classic Mx by keyword JIGGLE.

Value

imxJiggle() returns a numeric vector of randomly perturbed free-parameter values. mxJiggle() returns model, with its free parameter values altered according to the other function arguments.

See Also

mxTryHard()

Examples

```
data(demoOneFactor)
manifests <- names(demoOneFactor)
latents <- c("G")
factorModel <- mxModel(
"One Factor",
type="RAM",
manifestVars = manifests,
latentVars = latents,
mxPath(from=latents, to=manifests,values=0.8),
mxPath(from=manifests, arrows=2,values=1),
mxPath(from=latents, arrows=2,
 free=FALSE, values=1.0),
mxData(cov(demoOneFactor), type="cov",
 numObs=500)
)

iniPars <- coef(factorModel)
print(iniPars)

pars2 <- imxJiggle(params=iniPars,lbounds=NA,ubounds=NA,dsn="runif",loc=1,scale=0.05)
print(pars2)

mod2 <- mxJiggle(model=factorModel,scale=0.05)
coef(mod2)

mod3 <- mxJiggle(model=factorModel,classic=TRUE)
coef(mod3)
```

---

mxKalmanScores *Estimate Kalman scores and error covariance matrices*

Description

This function creates the Kalman predicted, Kalman updated, and Rauch-Tung-Striebel smoothed latent state and error covariance estimates for an MxModel object that has an MxExpectationStateSpace object.

Usage

```
mxKalmanScores(model, data=NA, frontend=TRUE)
```

Arguments

`model`	An MxModel object with an MxExpectationStateSpace.
`data`	An optional data.frame or matrix.
`frontend`	When TRUE, compute score in the frontend, otherwise use the backend.

Details

This is a helper function that computes the results of the classical Kalman filter. In particular, for every row of data there is a predicted latent score, an error covariance matrix for the predicted latent scores that provides an estimate of the predictions precision, an updated latent score, and an updated error covariance matrix for the updated latent scores. Additionally, the Rauch-Tung-Striebel (RTS) smoothed latent scores and error covariance matrices are returned.

Value

A list with components xPredicted, PPredicted, xUpdated, PUpdated, xSmoothed, PSmoothed, m2ll, and L. When using backend scores, this list also has components for yPredicted and SPredicted which have the same number of time points as the other components but relate to the observed variables instead of the latent variables. The rows of xPredicted, xUpdated, and xSmoothed correspond to different time points. The columns are the different latent variables. The third index of PPredicted, PUpdated, and PSmoothed corresponds to different times. This works nicely with the R default print method for arrays. At each time there is a covariance matrix of the latent variable scores. For all items listed below, the first element goes with the zeroth time point (See example).

xPredicted matrix of Kalman predicted scores

PPredicted array of Kalman predicted error covariances

xUpdated matrix of Kalman updated scores

PUpdated array of Kalman updated error covariances

xSmoothed matrix of RTS smoothed scores

PSmoothed array of RTS smoothed error covariances

m2ll minus 2 log likelihood

L likelihood, i.e., the multivariate normal probability density

yPredicted matrix of Kalman predicted scores for the observed variables, i.e., the predicted means. Only available for backend scores.

SPredicted array of Kalman predicted error covariances for the observed variables, i.e., the predicted covariances. Only available for backend scores.

References

J. Durbin and S.J. Koopman. (2001). *Time Series Analysis by State Space Methods*. Oxford University Press.

R.E. Kalman (1960). A New Approach to Linear Filtering and Prediction Problems. *Basic Engineering, 82*, 35-45.

H.E. Rauch, F. Tung, C.T. Striebel. (1965). Maximum Likelihood Estimates of Linear Dynamic Systems. *American Institute of Aeronautics and Astronautics Journal, 3*, 1445-1450.

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

mxExpectationStateSpace

Examples

```
# Create and fit a model using mxMatrix, mxExpectationStateSpace, and mxFitFunctionML
require(OpenMx)
data(demoOneFactor)
# Use only first 50 rows, for speed of example
data <- demoOneFactor[1:50,]
nvar <- ncol(demoOneFactor)
varnames <- colnames(demoOneFactor)
ssModel <- mxModel(model="State Space Manual Example",
    mxMatrix("Full", 1, 1, TRUE, .3, name="A"),
    mxMatrix("Zero", 1, 1, name="B"),
    mxMatrix("Full", nvar, 1, TRUE, .6, name="C", dimnames=list(varnames, "F1")),
    mxMatrix("Zero", nvar, 1, name="D"),
    mxMatrix("Diag", 1, 1, FALSE, 1, name="Q"),
    mxMatrix("Diag", nvar, nvar, TRUE, .2, name="R"),
    mxMatrix("Zero", 1, 1, name="x0"),
    mxMatrix("Diag", 1, 1, FALSE, 1, name="P0"),
    mxMatrix("Zero", 1, 1, name="u"),
    mxData(observed=data, type="raw"),
    mxExpectationStateSpace("A", "B", "C", "D", "Q", "R", "x0", "P0", "u"),
    mxFitFunctionML()
)
ssRun <- mxRun(ssModel)
summary(ssRun)
# Note the freely estimated Autoregressive parameter (A matrix)
#  is near zero as it should be for the independent rows of data
#  from the factor model.

ssScores <- mxKalmanScores(ssRun)

cor(cbind(ssScores$xPredicted[,1], ssScores$xUpdated[,1], ssScores$xSmoothed[,1]))
# Because the autoregressive dynamics are near zero, the predicted and updated scores
# correlate minimally, and the updated and smoothed latent state estimates
# are extremely close.

# The first few latent predicted scores
head(ssScores$xPredicted)

# The predicted latent score for time 10
ssScores$xPredicted[10+1,]

# The error covariance of the predicted score at time 10
ssScores$PPredicted[,,10+1]
```

MxLISRELModel-class *MxLISRELModel*

Description

This is an internal class and should not be used directly.

mxLISRELObjective *Create MxLISRELObjective Object*

Description

This function creates a new MxLISRELObjective object.

Usage

```
mxLISRELObjective(LX=NA, LY=NA, BE=NA, GA=NA, PH=NA, PS=NA, TD=NA, TE=NA, TH=NA,
   TX = NA, TY = NA, KA = NA, AL = NA,
   dimnames = NA, thresholds = NA, vector = FALSE, threshnames = dimnames)
```

Arguments

`LX`	An optional character string indicating the name of the 'LX' matrix.
`LY`	An optional character string indicating the name of the 'LY' matrix.
`BE`	An optional character string indicating the name of the 'BE' matrix.
`GA`	An optional character string indicating the name of the 'GA' matrix.
`PH`	An optional character string indicating the name of the 'PH' matrix.
`PS`	An optional character string indicating the name of the 'PS' matrix.
`TD`	An optional character string indicating the name of the 'TD' matrix.
`TE`	An optional character string indicating the name of the 'TE' matrix.
`TH`	An optional character string indicating the name of the 'TH' matrix.
`TX`	An optional character string indicating the name of the 'TX' matrix.
`TY`	An optional character string indicating the name of the 'TY' matrix.
`KA`	An optional character string indicating the name of the 'KA' matrix.
`AL`	An optional character string indicating the name of the 'AL' matrix.
`dimnames`	An optional character vector that is currently ignored
`thresholds`	An optional character string indicating the name of the thresholds matrix.
`vector`	A logical value indicating whether the objective function result is the likelihood vector.
`threshnames`	An optional character vector to be assigned to the column names of the thresholds matrix.

Details

Objective functions are functions for which free parameter values are chosen such that the value of the objective function is minimized. The mxLISRELObjective provides maximum likelihood estimates of free parameters in a model of the covariance of a given MxData object. This model is defined by LInear Structural RELations (LISREL; Jöreskog & Sörbom, 1982, 1996). Arguments 'LX' through 'AL' must refer to MxMatrix objects with the associated properties of their respective matrices in the LISREL modeling approach.

The full LISREL specification has 13 matrices and is sometimes called the extended LISREL model. It is defined by the following equations.

$$\eta = \alpha + B\eta + \Gamma\xi + \zeta$$

$$y = \tau_y + \Lambda_y\eta + \epsilon$$

$$x = \tau_x + \Lambda_x\xi + \delta$$

The table below is provided as a quick reference to the numerous matrices in LISREL models. Note that NX is the number of manifest exogenous (independent) variables, the number of Xs. NY is the number of manifest endogenous (dependent) variables, the number of Ys. NK is the number of latent exogenous variables, the number of Ksis or Xis. NE is the number of latent endogenous variables, the number of etas.

Matrix	Word	Abbreviation	Dimensions	Expression	Description
Λ_x	Lambda x	LX	NX x NK		Exogenous Factor Loading Matrix
Λ_y	Lambda y	LY	NY x NE		Endogenous Factor Loading Matrix
B	Beta	BE	NE x NE		Regressions of Latent Endogenous Variables Pre
Γ	Gamma	GA	NE x NK		Regressions of Latent Exogenous Variables Predi
Φ	Phi	PH	NK x NK	cov(ξ)	Covariance Matrix of Latent Exogenous Variable
Ψ	Psi	PS	NE x NE	cov(ζ)	Residual Covariance Matrix of Latent Endogeno
Θ_δ	Theta delta	TD	NX x NX	cov(δ)	Residual Covariance Matrix of Manifest Exogen
Θ_ϵ	Theta epsilon	TE	NY x NY	cov(ϵ)	Residual Covariance Matrix of Manifest Endoge
$\Theta_{\delta\epsilon}$	Theta delta epsilson	TH	NX x NY	cov(δ, ϵ)	Residual Covariance Matrix of Manifest Exogen
τ_x	tau x	TX	NX x 1		Residual Means of Manifest Exogenous Variable
τ_y	tau y	TY	NY x 1		Residual Means of Manifest Endogenous Variabl
κ	kappa	KA	NK x 1	mean(ξ)	Means of Latent Exogenous Variables
α	alpha	AL	NE x 1		Residual Means of Latent Endogenous Variables

From the extended LISREL model, several submodels can be defined. Subtypes of the LISREL model are defined by setting some of the arguments of the LISREL objective to NA. Note that because the default values of each LISREL matrix is NA, setting a matrix to NA can be accomplished by simply not giving it any other value.

The first submodel is the LISREL model without means.

$$\eta = B\eta + \Gamma\xi + \zeta$$

$$y = \Lambda_y\eta + \epsilon$$

$$x = \Lambda_x\xi + \delta$$

The LISREL model without means requires 9 matrices: LX, LY, BE, GA, PH, PS, TD, TE, and TH. Hence this LISREL model has TX, TY, KA, and AL as NA. This can be accomplished be leaving these matrices at their default values.

The TX, TY, KA, and AL matrices must be specified if either the mxData type is “cov” or “cor” and a means vector is provided, or if the mxData type is “raw”. Otherwise the TX, TY, KA, and AL matrices are ignored and the model without means is estimated.

A second submodel involves only endogenous variables.

$$\eta = B\eta + \zeta$$

$$y = \Lambda_y \eta + \epsilon$$

The endogenous-only LISREL model requires 4 matrices: LY, BE, PS, and TE. The LX, GA, PH, TD, and TH must be NA in this case. However, means can also be specified, allowing TY and AL if the data are raw or if observed means are provided.

Another submodel involves only exogenous variables.

$$x = \Lambda_x \xi + \delta$$

The exogenous-model model requires 3 matrices: LX, PH, and TD. The LY, BE, GA, PS, TE, and TH matrices must be NA. However, means can also be specified, allowing TX and KA if the data are raw or if observed means are provided.

The model that is run depends on the matrices that are not NA. If all 9 matrices are not NA, then the full model is run. If only the 4 endogenous matrices are not NA, then the endogenous-only model is run. If only the 3 exogenous matrices are not NA, then the exogenous-only model is run. If some endogenous and exogenous matrices are not NA, but not all of them, then appropriate errors are thrown. Means are included in the model whenever their matrices are provided.

The MxMatrix objects included as arguments may be of any type, but should have the properties described above. The mxLISRELObjective will not return an error for incorrect specification, but incorrect specification will likely lead to estimation problems or errors in the mxRun function.

Like the mxRAMObjective, the mxLISRELObjective evaluates with respect to an MxData object. The MxData object need not be referenced in the mxLISRELObjective function, but must be included in the MxModel object. mxLISRELObjective requires that the ’type’ argument in the associated MxData object be equal to ’cov’, ’cor’, or ’raw’.

To evaluate, place MxLISRELObjective objects, the mxData object for which the expected covariance approximates, referenced MxAlgebra and MxMatrix objects, and optional MxBounds and MxConstraint objects in an MxModel object. This model may then be evaluated using the mxRun function. The results of the optimization can be found in the ’output’ slot of the resulting model, and may be obtained using the mxEval function.

Value

Returns a new MxLISRELObjective object. MxLISRELObjective objects should be included with models with referenced MxAlgebra, MxData and MxMatrix objects.

References

Jöreskog, K. G. & Sörbom, D. (1996). LISREL 8: User's Reference Guide. Lincolnwood, IL: Scientific Software International.

Jöreskog, K. G. & Sörbom, D. (1982). Recent developments in structural equation modeling. *Journal of Marketing Research, 19,* 404-416.

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
#####------------------------------#####
##### Factor Model
mLX <- mxMatrix("Full", values=c(.5, .6, .8, rep(0, 6), .4, .7, .5),
   name="LX", nrow=6, ncol=2,
   free=c(TRUE,TRUE,TRUE,rep(FALSE, 6),TRUE,TRUE,TRUE))
mTD <- mxMatrix("Diag", values=c(rep(.2, 6)), name="TD", nrow=6, ncol=6,
   free=TRUE)
mPH <- mxMatrix("Symm", values=c(1, .3, 1), name="PH", nrow=2, ncol=2,
   free=c(FALSE, TRUE, FALSE))

# Create a LISREL objective with LX, TD, and PH matrix names
objective <- mxLISRELObjective(LX="LX", TD="TD", PH="PH")

testModel <- mxModel(model="testModel4", mLX, mTD, mPH, objective)
```

`MxListOrNull-class` *An optional list*

Description

An optional list

`mxMakeNames` *mxMakeNames*

Description

Adjust a character vector so that it is valid when used as MxMatrix column or row names.

Usage

```
mxMakeNames(names, unique = FALSE)
```

Arguments

`names`	a character vector
`unique`	whether to pass the result through make.unique

Details

note: OpenMx is (much) more restrictive than base R's make.names.

See Also

make.names

Examples

```
demo <- c("", "103", "data", "foo.bar[3,2]", "+!", "!+")
mxMakeNames(demo, unique=TRUE)
```

---

mxMarginalNegativeBinomial

Indicator with marginal Negative Binomial distribution

Description

Indicator with marginal Negative Binomial distribution

Usage

```
mxMarginalNegativeBinomial(
  vars,
  maxCount = NA,
  size,
  prob = c(),
  mu = c(),
  zeroInf = 0.01,
  free = TRUE,
  labels = NA,
  lbound = NA,
  ubound = NA
)
```

Arguments

`vars`	character vector of manifest indicators
`maxCount`	maximum observed count
`size`	positive target number of successful trials
`prob`	probability of success in each trial
`mu`	alternative parametrization via mean
`zeroInf`	zero inflation parameter in probability units
`free`	logical vector indicating whether paremeters are free
`labels`	character vector of parameter labels
`lbound`	numeric vector of lower bounds
`ubound`	numeric vector of upper bounds

Value

a list of MxMarginPoisson obects

`mxMarginalPoisson` *Indicator with marginal Poisson distribution*

Description

Indicator with marginal Poisson distribution

Usage

```
mxMarginalPoisson(
  vars,
  maxCount = NA,
  lambda,
  zeroInf = 0.01,
  free = TRUE,
  labels = NA,
  lbound = 0,
  ubound = c(1, NA)
)
```

Arguments

`vars`	character vector of manifest indicators
`maxCount`	maximum observed count
`lambda`	non-negative means
`zeroInf`	zero inflation parameter in probability units
`free`	logical vector indicating whether paremeters are free
`labels`	character vector of parameter labels
`lbound`	numeric vector of lower bounds
`ubound`	numeric vector of upper bounds

Value

a list of MxMarginPoisson obects

mxMatrix *Create MxMatrix Object*

Description

This function creates a new MxMatrix object.

Usage

```
mxMatrix(type = c('Full', 'Diag', 'Iden', 'Lower',
'Sdiag', 'Stand', 'Symm', 'Unit', 'Zero'), nrow = NA, ncol = NA,
    free = FALSE, values = NA, labels = NA, lbound = NA,
    ubound = NA, byrow = getOption('mxByrow'), dimnames = NA, name = NA,
    condenseSlots=getOption('mxCondenseMatrixSlots'),
    ..., joinKey=as.character(NA), joinModel=as.character(NA))
```

Arguments

type	A character string indicating the matrix type, where type indicates the range of values and equalities in the matrix. Must be one of: 'Diag', 'Full', 'Iden', 'Lower', 'Sdiag', 'Stand', 'Symm', 'Unit', or 'Zero'.
nrow	Integer; the desired number of rows. One or both of 'nrow' and 'ncol' is required when 'values', 'free', 'labels', 'lbound', and 'ubound' arguments are not matrices, depending on the desired MxMatrix type.
ncol	Integer; the desired number of columns. One or both of 'nrow' and 'ncol' is required when 'values', 'free', 'labels', 'lbound', and 'ubound' arguments are not matrices, depending on the desired MxMatrix type.
free	A vector or matrix of logicals for free parameter specification. A single 'TRUE' or 'FALSE' will set all allowable variables to free or fixed, respectively.
values	A vector or matrix of numeric starting values. By default, all values are set to zero.
labels	A vector or matrix of characters for variable label specification.
lbound	A vector or matrix of numeric lower bounds. Default bounds are specified with an NA.
ubound	A vector or matrix of numeric upper bounds. Default bounds are specified with an NA.
byrow	Logical; defaults to value of global option 'mxByRow'. If FALSE (default), the 'values', 'free', 'labels', 'lbound', and 'ubound' matrices are populated by column rather than by row.
dimnames	List. The dimnames attribute for the matrix: a list of length 2 giving the row and column names respectively. An empty list is treated as NULL, and a list of length one as row names. The list can be named, and the list names will be used as names for the dimensions.
name	An optional character string indicating the name of the MxMatrix object.

`condenseSlots`	Logical; defaults to value of global option 'mxCondenseMatrixSlots'. If `TRUE`, then the resulting MxMatrix will "condense" its 'labels', 'free', 'lbound', and 'ubound' down to 1x1 matrices if they contain only `FALSE` ('free') or `NA` (the other three). If `FALSE`, those four matrices and the 'values' matrix will all be of equal dimensions.
`...`	Not used. Forces remaining arguments to be specified by name.
`joinKey`	The name of the column in current model's raw data that is used as a foreign key to match against the primary key in the joinModel's raw data.
`joinModel`	The name of the model that this matrix joins against.

Details

The mxMatrix function creates MxMatrix objects, which consist of five matrices and a 'type' argument. The 'values' matrix is made up of numeric elements whose usage and capabilities in other functions are defined by the 'free' matrix. If an element is specified as a fixed parameter in the 'free' matrix, then the element in the 'values' matrix is treated as a constant value and cannot be altered or updated by an objective function when included in an mxRun function. If an element is specified as a free parameter in the 'free' matrix, the element in the 'value' matrix is considered a starting value and can be changed by an objective function when included in an mxRun function.

Element labels beginning with `'data.'` can be used if the MxMatrix is to be used in an MxModel object that has a raw dataset (i.e., an MxData object of `type="raw"`). Such a label instructs OpenMx to use a particular column of the raw dataset to fill in the value of that element. For historical reasons, the variable contained in that column is called a "definition variable." For example, if an MxMatrix element has the label `'data.x'`, then OpenMx will use the first value of the data column named "x" when evaluating the fitfunction for the first row, and will use the second value of column "x" when evaluating the fitfunction for the second row, and so on. After the call to `mxRun()`, the values for elements labeled with `'data.x'` are returned as the value from the first (i.e., first before any automated sorting is done) element of column "x" in the data.

Objects created by the `mxMatrix()` function are of a specific 'type', which specifies the number and location of parameters in the 'labels' matrix and the starting values in the 'values' matrix. Input 'values', 'free', and 'labels' matrices must be of appropriate shape and have appropriate values for the matrix type requested. Nine types of matrices are supported:

'Diag'	matrices must be square, and only elements on the principal diagonal may be specified as free parameters or take no
'Full'	matrices may be either rectangular or square, and all elements in the matrix may be freely estimated. This type is th
'Iden'	matrices must be square, and consist of no free parameters. Matrices of this type have a value of 1 for all entries on
'Lower'	matrices must be square, with a value of 0 for all entries in the upper triangle and no free parameters in the upper tr
'Sdiag'	matrices must be square, with a value of 0 for all entries in the upper triangle and along the diagonal. No free paran
'Symm'	matrices must be square, and elements in the principle diagonal and lower triangular portion of the matrix may be f
'Stand'	matrices are symmetric matrices (see 'Symm') with 1's along the main diagonal.
'Unit'	matrices may be either rectangular or square, and contain no free parameters. All elements in matrices of this type
'Zero'	matrices may be either rectangular or square, and contain no free parameters. All elements in matrices of this type

When 'type' is 'Lower' or 'Symm', then the arguments to 'free', 'values', 'labels', 'lbound', or 'ubound' may be vectors of length $N*(N+1)/2$, where N is the number of rows and columns of the matrix. When 'type' is 'Sdiag' or 'Stand', then the arguments to 'free', 'values', 'labels', 'lbound', or 'ubound' may be vectors of length $N * (N - 1)/2$.

Value

Returns a new MxMatrix object, which consists of a 'values' matrix of numeric starting values, a 'free' matrix describing free parameter specification, a 'labels' matrix of labels for the variable names, and 'lbound' and 'ubound' matrices of the lower and upper parameter bounds. This MxMatrix object can be used as an argument in the `mxAlgebra()`, `mxBounds()`, `mxConstraint()` and `mxModel()` functions.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

MxMatrix for the S4 class created by mxMatrix. More information about the OpenMx package may be found here.

Examples

```
# Create a 3 x 3 identity matrix

idenMatrix <- mxMatrix(type = "Iden", nrow = 3,
    ncol = 3, name = "I")

# Create a full 4 x 2 matrix from existing
# value matrix with all free parameters

vals <- matrix(1:8, nrow = 4)
fullMatrix <- mxMatrix(type = "Full", values = vals,
    free = TRUE, name = "foo")

# Create a 3 x 3 symmetric matrix with free off-
# diagonal parameters and starting values

symmMatrix <- mxMatrix(type = "Symm", nrow = 3, ncol = 3,
    free = c(FALSE, TRUE, TRUE, FALSE, TRUE, FALSE),
    values = c(1, .8, .8, 1, .8, 1),
    labels = c(NA, "free1", "free2", NA, "free3", NA),
    name = "bar")

# Create an mxMatrix from a character matrix.  All numbers are
# interpreted as fixed and non-numbers are interpreted as free
# parameters.

matrixFromChar <- function(inputm, name=NA) {
  inputmFixed <- suppressWarnings(matrix(
    as.numeric(inputm),nrow = nrow(inputm), ncol = ncol(inputm)))
  inputmCharacter <- inputm
  inputmCharacter[!is.na(inputmFixed)] <- NA
  mxMatrix(nrow=nrow(inputm), ncol=ncol(inputm),
              free=!is.na(inputmCharacter),
              values=inputmFixed,
              labels=inputmCharacter,
              dimnames=dimnames(inputm), name=name)
}


# Demonstrate some of the behavior of the condensed slots
# Create a 3x3 matrix with condensed slots

a <- mxMatrix('Full', 3, 3, values=1, condenseSlots=TRUE)
a@free # at operator returns the stored 1x1 matrix
a$free # dollar operator constructs full matrix for printing

# assignment with the dollar operator
#  de-condenses the slots to create the
#  full 3x3 matrix
a$free[1,1] <- TRUE
a@free
```

`MxMatrix-class` *MxMatrix Class*

Description

MxMatrix is a virtual S4 class that comprises the nine types of matrix objects used by OpenMx (see `mxMatrix()` for details). An MxMatrix object is a named entity. New instances of this class can be created using the function `mxMatrix()`. MxMatrix objects may be used as arguments in other functions from the OpenMx package, including `mxAlgebra()`, `mxConstraint()`, and `mxModel()`.

Objects from the Class

All nine types of object that the class comprises can be created via `mxMatrix()`.

Slots

`name`: Character string; the name of the MxMatrix object. Note that this is the object's "Mx name" (so to speak), which identifies it in OpenMx's internal namespace, rather than the symbol identifying it in R's workspace. Use of MxMatrix objects in an mxAlgebra or mxConstraint function requires reference by name.

`values`: Numeric matrix of values. If an element is specified as a fixed parameter in the 'free' matrix, then the element in the 'values' matrix is treated as a constant value and cannot be altered or updated by an objective function when included in an `mxRun()` function. If an element is specified as a free parameter in the 'free' matrix, the element in the 'value' matrix is considered a starting value and can be changed by an objective function when included in an `mxRun()` function.

`labels`**:** Matrix of character strings which provides the labels of free and fixed parameters. Fixed parameters with identical labels must have identical values. Free parameters with identical labels impose an equality constraint. The same label cannot be applied to a free parameter and a fixed parameter. A free parameter with the label 'NA' implies a unique free parameter, that cannot be constrained to equal any other free parameter.

`free`**:** Logical matrix specifying whether each element is free versus fixed. An element is a free parameter if-and-only-if the corresponding value in the 'free' matrix is 'TRUE'. Free parameters are elements of an MxMatrix object whose values may be changed by a fitfunction when that MxMatrix object is included in an MxModel object and evaluated using the `mxRun()` function.

`lbound`**:** Numeric matrix of lower bounds on free parameters.

`ubound`**:** Numeric matrix of upper bounds on free parameters.

`.squareBrackets`**:** Logical matrix; used internally by OpenMx. Identifies which elements have labels with square brackets in them.

`.persist`**:** Logical; used internally by OpenMx. Governs how `mxRun()` handles the MxMatrix object when it is inside the MxModel being run.

`.condenseSlots`**:** Logical; used internally by OpenMx. If `FALSE`, then the matrices in the 'values', 'labels', 'free', 'lbound', and 'ubound' slots are all of equal dimensions. If `TRUE`, then the last four of those slots will "condense" a matrix consisting entirely of `FALSE` or `NA` down to 1x1.

`display`**:** Character string; used internally by OpenMx when parsing MxAlgebras.

`dependencies`**:** Integer; used internally by OpenMx when parsing MxAlgebras.

Methods

$ `signature(x = "MxMatrix")`: ...

$<- `signature(x = "MxMatrix")`: ...

[`signature(x = "MxMatrix")`: ...

[<- `signature(x = "MxMatrix")`: ...

dim `signature(x = "MxMatrix")`: ...

dimnames `signature(x = "MxMatrix")`: ...

dimnames<- `signature(x = "MxMatrix")`: ...

length `signature(x = "MxMatrix")`: ...

names `signature(x = "MxMatrix")`: ...

ncol `signature(x = "MxMatrix")`: ...

nrow `signature(x = "MxMatrix")`: ...

print `signature(x = "MxMatrix")`: ...

show `signature(object = "MxMatrix")`: ...

Note that some methods are documented separately (see below, under "See Also").

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation`.

See Also

`mxMatrix()` for creating MxMatrix objects. Note that functions `imxCreateMatrix()`, `imxDeparse()`, `imxSquareMatrix()`, `imxSymmetricMatrix()`, and `imxVerifyMatrix()` are separately documented methods for this class. More information about the OpenMx package may be found here.

Examples

```
showClass("MxMatrix")
```

`mxMI` *Estimate Modification Indices for MxModel Objects*

Description

This function estimates the change in fit function value resulting from freeing currently fixed parameters.

Usage

```
mxMI(model, matrices=NA, full=TRUE)
```

Arguments

`model`	An MxModel for which modification indices are desired.
`matrices`	Character vector. The names of the matrices in which to search for modification
`full`	Logical. Whether or not to return the full modification index in addition to the restricted.

Details

Modification indices provide an estimate of how much the fit function value would change if a parameter that is currently fixed was instead freely estimated. There are two versions of this estimate: a restricted version and an full version. The restricted version is reported as the MI and is much faster to compute. The full version is reported as MI.Full. The full version accounts for the *total* change in fit function value resulting from the newly freed parameter. The restricted version only accounts for the change in the fit function due to the movement of the new free parameter. In particular, the restricted version does not account for the change in fit function value due to the other free parameters moving in response to the new parameter.

The algorithm respects fixed parameter labels. That is, when a fixed parameter has a label and occurs in more than one spot, then that fixed parameter is freed in all locations in which it occurs to evaluate the modification index for that fixed parameter.

When the fit function is in minus two log likelihood units (e.g. `mxFitFunctionML`), then the MI will be approximately chi squared distributed with 1 degree of freedom. Using a p-value of 0.01 has been suggested. Hence, a MI greater than `qchisq(p=1-0.01,df=1)`, or 6.63, is suggestive of a modification.

Users should be cautious in their use of modification indices. If a model was created with the aid of MIs, then it should *always* be reported. *Do not pretend that you have a theoretical reason for part of a model that was put there because it was suggested by a modification index. This is fraud.* When using modification indices there are two options for best practices. First, you can report the analyses as exploratory. Document all the explorations that you did, and know that your results may or may not generalize. Second, you can use cross-validation. Reserve part of your data for exploration, and use the remaining data to test if the exploratory model generalizes to new data.

Value

A named list with components

MI The restricted modification index.

MI.Full The full modification index.

plusOneParamModels A list of models with one additional free parameter

References

Sörbom, D. (1989). Model Modification. *Psychometrika, 54*, 371-384.

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
# Create a model
require(OpenMx)
data(demoOneFactor)
manifests <- names(demoOneFactor)
latents <- c("G")
factorModel <- mxModel("One Factor",
      type="RAM",
      manifestVars = manifests,
      latentVars = latents,
      mxPath(from=latents, to=manifests),
      mxPath(from=manifests, arrows=2),
      mxPath(from=latents, arrows=2,
            free=FALSE, values=1.0),
      mxPath(from = 'one', to = manifests),
      mxData(observed=cov(demoOneFactor), type="cov", numObs=500,
             means = colMeans(demoOneFactor)))
#No SEs for speed
factorModel <- mxOption(factorModel, 'Standard Errors', 'No')
factorRun <- mxRun(factorModel)

# See if it should be modified
# Notes
#  Using full=FALSE for faster performance
#  Using matrices= 'A' and 'S' to not get MIs for
#    the F matrix which is always fixed.
fim <- mxMI(factorRun, matrices=c('A', 'S'), full=FALSE)
round(fim$MI, 3)
plot(fim$MI, ylim=c(0, 10))
```

```
abline(h=qchisq(p=1-0.01, df=1)) # line of "significance"
```

`mxMLObjective` *DEPRECATED: Create MxMLObjective Object*

Description

WARNING: Objective functions have been deprecated as of OpenMx 2.0.

Please use mxExpectationNormal() and mxFitFunctionML() instead. As a temporary workaround, mxMLObjective returns a list containing an MxExpectationNormal object and an MxFitFunctionML object.

mxMLObjective(covariance, means = NA, dimnames = NA, thresholds = NA) All occurrences of

mxMLObjective(covariance, means = NA, dimnames = NA, thresholds = NA)

Should be changed to

mxExpectationNormal(covariance, means = NA, dimnames = NA, thresholds = NA, threshnames = dimnames) mxFitFunctionML(vector = FALSE)

Arguments

`covariance`	A character string indicating the name of the expected covariance algebra.
`means`	An optional character string indicating the name of the expected means algebra.
`dimnames`	An optional character vector to be assigned to the dimnames of the covariance and means algebras.
`thresholds`	An optional character string indicating the name of the thresholds matrix.

Details

NOTE: THIS DESCRIPTION IS DEPRECATED. Please change to using mxExpectationNormal and mxFitFunctionML as shown in the example below.

Objective functions are functions for which free parameter values are chosen such that the value of the objective function is minimized. The mxMLObjective function uses full-information maximum likelihood to provide maximum likelihood estimates of free parameters in the algebra defined by the 'covariance' argument given the covariance of an MxData object. The 'covariance' argument takes an MxAlgebra object, which defines the expected covariance of an associated MxData object. The 'dimnames' arguments takes an optional character vector. If this argument is not a single NA, then this vector be assigned to be the dimnames of the means vector, and the row and columns dimnames of the covariance matrix.

mxMLObjective evaluates with respect to an MxData object. The MxData object need not be referenced in the mxMLObjective function, but must be included in the MxModel object. mxMLObjective requires that the 'type' argument in the associated MxData object be equal to 'cov' or 'cov'. The 'covariance' argument of this function evaluates with respect to the 'matrix' argument of the associated MxData object, while the 'means' argument of this function evaluates with respect to the 'vector' argument of the associated MxData object. The 'means' and 'vector' arguments are optional in both functions. If the 'means' argument is not specified (NA), the optional 'vector' argument of the MxData object is ignored. If the 'means' argument is specified, the associated MxData object should specify a 'means' argument of equivalent dimension as the 'means' algebra.

dimnames must be supplied where the matrices referenced by the covariance and means algebras are not themselves labeled. Failure to do so leads to an error noting that the covariance or means matrix associated with the ML objective does not contain dimnames.

To evaluate, place MxMLObjective objects, the mxData object for which the expected covariance approximates, referenced MxAlgebra and MxMatrix objects, and optional MxBounds and MxConstraint objects in an MxModel object. This model may then be evaluated using the mxRun function. The results of the optimization can be found in the 'output' slot of the resulting model, or using the mxEval function.

Value

Returns a list containing an MxExpectationNormal object and an MxFitFunctionML object.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
# Create and fit a model using mxMatrix, mxAlgebra, mxExpectationNormal, and mxFitFunctionML

library(OpenMx)

# Simulate some data

x=rnorm(1000, mean=0, sd=1)
y= 0.5*x + rnorm(1000, mean=0, sd=1)
tmpFrame <- data.frame(x, y)
tmpNames <- names(tmpFrame)

# Define the matrices

S <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(1,0,0,1),
              free=c(TRUE,FALSE,FALSE,TRUE), labels=c("Vx", NA, NA, "Vy"), name = "S")
A <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(0,1,0,0),
              free=c(FALSE,TRUE,FALSE,FALSE), labels=c(NA, "b", NA, NA), name = "A")
I <- mxMatrix(type="Iden", nrow=2, ncol=2, name="I")

# Define the expectation

expCov <- mxAlgebra(solve(I-A) %*% S %*% t(solve(I-A)), name="expCov")
expFunction <- mxExpectationNormal(covariance="expCov", dimnames=tmpNames)

# Choose a fit function

fitFunction <- mxFitFunctionML()

# Define the model
```

```
tmpModel <- mxModel(model="exampleModel", S, A, I, expCov, expFunction, fitFunction,
                    mxData(observed=cov(tmpFrame), type="cov", numObs=dim(tmpFrame)[1]))

# Fit the model and print a summary

tmpModelOut <- mxRun(tmpModel)
summary(tmpModelOut)
```

mxModel *Create MxModel Object*

Description

Create or modify an MxModel.

Usage

```
mxModel(model = NA, ..., manifestVars = NA, latentVars = NA,
          remove = FALSE, independent = NA, type = NA, name = NA)
```

Arguments

`model`	This argument is either an MxModel object or a string. If 'model' is an MxModel object, then all elements of that model are placed in the resulting MxModel object. If 'model' is a string, then a new model is created with the string as its name. If 'model' is either unspecified or 'model' is a named entity, data source, or MxPath object, then a new model is created.
`...`	An arbitrary number of mxMatrix, mxPath, mxData, and other functions such as mxConstraints and mxCI. These will all be added or removed from the model as specified in the 'model' argument, based on the 'remove' argument.
`manifestVars`	For RAM-type models, A list of manifest variables to be included in the model.
`latentVars`	For RAM-type models, A list of latent variables to be included in the model.
`remove`	logical. If TRUE, elements listed in this statement are removed from the original model. If FALSE, elements listed in this statement are added to the original model.
`independent`	logical. If TRUE then the model is evaluated independently of other models.
`type`	character vector. The model type to assign to this model. Defaults to options("mxDefaultType"). See below for valid types
`name`	An optional character vector indicating the name of the object.

Details

The mxModel function is used to create MxModels. Models created by this function may be new, or may be modified versions of existing MxModel objects. By default a new MxModel object will be created: To create a modified version of an existing MxModel object, include this model in the 'model' argument.

Other named-entities may be added as arguments to the mxModel function, which are then added to or removed from the model specified in the 'model' argument. Functions you can use to add objects to the model to this way include mxPath, mxCI, mxAlgebra, mxBounds, mxConstraint, mxData, and mxMatrix objects, as well as fit functions and expectations (see below). You can also include sub-models as components of a model. These sub-models may be estimated separately or jointly depending on shared parameters and the 'independent' flag (see below). Only one MxData object and one fit function and expectation may be included per model, but there are no restrictions on the number of other named-entities included in an mxModel statement.

All other arguments must be named (i.e. 'latentVars = names'), or they will be interpreted as elements of the ellipsis list. The 'manifestVars' and 'latentVars' arguments specify the names of the manifest and latent variables, respectively, for use with the mxPath function.

The 'remove' argument may be used when mxModel is used to create a modified version of an existing MxMatrix object. When 'remove' is set to TRUE, the listed objects are removed from the model specified in the 'model' argument. When 'remove' is set to FALSE, the listed objects are added to the model specified in the 'model' argument.

Model independence may be specified with the 'independent' argument. If a model is independent ('independent = TRUE'), then the parameters of this model are not shared with any other model. An independent model may be estimated with no dependency on any other model. If a model is not independent ('independent = FALSE'), then this model shares parameters with one or more other models such that these models must be jointly estimated. These dependent models must be entered as arguments in another model, so that they are simultaneously optimized.

The model type is determined by a character vector supplied to the 'type' argument. The type of a model is a dynamic property, ie. it is allowed to change during the lifetime of the model. To see a list of available types, use the mxTypes command. When a new model is created and no type is specified, the type specified by `options("mxDefaultType")` is used.

Expectations and Fit functions

To be estimated, MxModel objects must include fit functions and expectations as arguments. Fit functions include mxFitFunctionML, mxFitFunctionMultigroup, mxFitFunctionWLS, mxFitFunctionAlgebra, mxFitFunctionGREML, mxFitFunctionR, and mxFitFunctionRow. Expectations include mxExpectationBA81, mxExpectationGREML, mxExpectationHiddenMarkov mxExpectationLISREL, mxExpectationMixture, mxExpectationNormal, mxExpectationRAM, mxExpectationStateSpace, mxExpectationStateSpaceContinuousTime. The 'type' of the model may imply a certain fit function or expectation (e.g. type = "RAM" implies mxExpectationRAM). The model data may also constrain which fit and expectation are appropriate.

The model, complete with fit function and expectation can then be executed using mxRun.

Accessing model components

You can view a model summary with summary. You can also access Named entities in MxModel directly via the $ symbol. For instance, for an MxModel named "yourModel" containing an MxMatrix named "yourMatrix", the contents of "yourMatrix" can be accessed as yourModel$yourMatrix. Slots (i.e., matrices, algebras, etc.) in an mxMatrix may also be referenced with the $ symbol (e.g., yourModel$matrices or yourModel$algebras). See the documentation for Classes and the examples in Classes for more information.

Value

Returns a new MxModel object. To be run, MxModel object must include a fit function and expectation.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

See mxCI for information about adding Confidence Interval calculations to a model. See mxPath for information about adding paths to RAM-type models. See mxMatrix for information about adding matrices to models. See mxData for specifying the data a model is to be evaluated against. Many advanced options can be set via mxOption. More information about the OpenMx package may be found here.

Examples

```
library(OpenMx)

# At the simplest, you can create an empty model,
#  placing it in an object, and add to it later
emptyModel <- mxModel(model="IAmEmpty")

# Create a model named 'firstdraft' with one matrix 'A'
firstModel <- mxModel(model='firstdraft',
                 mxMatrix(type='Full', nrow = 3, ncol = 3, name = "A"))

# Update 'firstdraft', and rename the model 'finaldraft'
finalModel <- mxModel(model=firstModel,
                 mxMatrix(type='Symm', nrow = 3, ncol = 3, name = "S"),
                 mxMatrix(type='Iden', nrow = 3, name = "F"),
                 name= "finaldraft")

# Add data to the model from an existing data frame in object 'data'
data(twinData)  # load some data
finalModel <- mxModel(model=finalModel, mxData(twinData, type='raw'))

# Two ways to view the matrix named "A" in MxModel object 'model'

finalModel$A

finalModel$matrices$A

# A working example using OpenMx Path Syntax
data(HS.ability.data)  # load the data

# The manifest variables loading on each proposed latent variable
```

```
Spatial   <- c("visual", "cubes", "paper")
Verbal    <- c("general", "paragrap", "sentence")
Math      <- c("numeric", "series", "arithmet")

latents   <- c("vis", "math", "text")
manifests <-  c(Spatial, Math, Verbal)

HSModel <- mxModel(model="Holzinger_and_Swineford_1939", type="RAM",
    manifestVars = manifests, # list the measured variables (boxes)
    latentVars   = latents,   # list the latent variables (circles)
    # factor loadings from latents to  manifests
    mxPath(from="vis",  to=Spatial),# factor loadings
    mxPath(from="math", to=Math),   # factor loadings
    mxPath(from="text", to=Verbal), # factor loadings

    # Allow latent variables to covary
    mxPath(from="vis" , to="math", arrows=2, free=TRUE),
    mxPath(from="vis" , to="text", arrows=2, free=TRUE),
    mxPath(from="math", to="text", arrows=2, free=TRUE),

    # Allow latent variables to have variance
    mxPath(from=latents, arrows=2, free=FALSE, values=1.0),
    # Manifest have residual variance
    mxPath(from=manifests, arrows=2),
    # the data to be analysed
    mxData(cov(HS.ability.data[,manifests]), type = "cov", numObs = 301))

fitModel <- mxRun(HSModel) # run the model
summary(fitModel) # examine the output: Fit statistics and path loadings
```

MxModel-class *MxModel Class*

Description

MxModel is an S4 class. An MxModel object is a named entity.

Details

The 'matrices' slot contains a list of the MxMatrix objects included in the model. These objects are listed by name. Two objects may not share the same name. If a new MxMatrix is added to an MxModel object with the same name as an MxMatrix object in that model, the added version replaces the previous version. There is no imposed limit on the number of MxMatrix objects that may be added here.

The 'algebras' slot contains a list of the MxAlgebra objects included in the model. These objects are listed by name. Two objects may not share the same name. If a new MxAlgebra is added to an MxModel object with the same name as an MxAlgebra object in that model, the added version replaces the previous version. All MxMatrix objects referenced in the included MxAlgebra objects must be included in the ‘matrices’ slot prior to estimation. There is no imposed limit on the number of MxAlgebra objects that may be added here.

The ‘constraints’ slot contains a list of the MxConstraint objects included in the model. These objects are listed by name. Two objects may not share the same name. If a new MxConstraint is added to an MxModel object with the same name as an MxConstraint object in that model, the added version replaces the previous version. All MxMatrix objects referenced in the included MxConstraint objects must be included in the ‘matrices’ slot prior to estimation. There is no imposed limit on the number of MxConstraint objects that may be added here.

The ‘intervals’ slot contains a list of the confidence intervals requested by included MxCI objects. These objects are listed by the free parameters, MxMatrices and MxAlgebras referenced in the MxCI objects, not the list of MxCI objects themselves. If a new MxCI object is added to an MxModel object referencing one or more free parameters MxMatrices or MxAlgebras previously listed in the ‘intervals’ slot, the new confidence interval(s) replace the existing ones. All listed confidence intervals must refer to free parameters MxMatrices or MxAlgebras in the model.

The ‘latentVars’ slot contains a list of latent variable names, which may be referenced by MxPath objects. This slot defaults to ’NA’, and is only used when the mxPath function is used. In the context of a RAM model, this slot accepts a character vector of variable names. However, the LISREL model is partitioned into exogenous and endogenous parts. Both exogenous and endogenous variables can be specified using a list like, `list(endo='a',exo='b')`. If a character vector is passed to a LISREL model then those variables will be assumed endogenous.

The ‘manifestVars’ slot contains a list of latent variable names, which may be referenced by MxPath objects. This slot defaults to ’NA’, and is only used when the mxPath function is used. In the context of a RAM model, this slot accepts a character vector of variable names. However, the LISREL model is partitioned into exogenous and endogenous parts. Both exogenous and endogenous variables can be specified using a list like, `list(endo='a',exo='b')`. If a character vector is passed to a LISREL model then those variables will be assumed endogenous.

The ‘data’ slot contains an MxData object. This slot must be filled prior to execution when a fitfunction referencing data is used. Only one MxData object may be included per model, but submodels may have their own data in their own ‘data’ slots. If an MxData object is added to an MxModel which already contains an MxData object, the new object replaces the existing one.

The ‘submodels’ slot contains references to all of the MxModel objects included as submodels of this MxModel object. Models held as arguments in other models are considered to be submodels. These objects are listed by name. Two objects may not share the same name. If a new submodel is added to an MxModel object with the same name as an existing submodel, the added version replaces the previous version. When a model containing other models is executed using mxRun, all included submodels are executed as well. If the submodels are dependent on one another, they are treated as one larger model for purposes of estimation.

The ‘independent’ slot contains a logical value indicating whether or not the model is independent. If a model is independent (independent=TRUE), then the parameters of this model are not shared with any other model. An independent model may be estimated with no dependency on any other model. If a model is not independent (independent=FALSE), then this model shares parameters with one or more other models such that these models must be jointly estimated. These dependent models must be entered as submodels of another MxModel objects, so that they are simultaneously optimized.

The ‘options’ slot contains a list of options for the model. The name of each entry in the list is the option name to be used at runtime. The values in this list are the values of the optimizer options.

The standard interface for updating options is through the mxOption function.

The 'output' slot contains a list of output added to the model by the mxRun function. Output includes parameter estimates, optimization information, model fit, and other information. If a model has not been optimized using the mxRun function, the 'output' slot will be 'NULL'.

Named entities in MxModel objects may be viewed and referenced by name using the $ symbol. For instance, for an MxModel named "yourModel" containing an MxMatrix named "yourMatrix", the contents of "yourMatrix" can be accessed as yourModel$yourMatrix. Slots (i.e., matrices, algebras, etc.) in an mxMatrix may also be referenced with the $ symbol (e.g., yourModel$matrices or yourModel$algebras). See the documentation for Classes and the examples in mxModel for more information.

Objects from the Class

Objects can be created by calls of the form `mxModel()`.

Slots

`name`**:** Character string. The name of the model object.

`matrices`**:** List of the model's MxMatrix objects.

`algebras`**:** List of the model's MxAlgebra objects.

`constraints`**:** List of the model's MxConstraint objects.

`intervals`**:** List of the model's MxInterval objects, requested via `mxCI()`.

`latentVars`**:** "Latent variables;" object of class `"MxCharOrList"`.

`manifestVars`**:** "Manifest variables;" object of class `"MxCharOrList"`.

`data`**:** Object of class MxData.

`submodels`**:** List of MxModel objects.

`expectation`**:** Object of class MxExpectation; dictates the model's specification.

`fitfunction`**:** Object of class MxFitFunction; dictates the cost function to be minimized when fitting the model.

`compute`**:** Object of class MxCompute–the model's compute plan, which contains instructions on what the model is to compute and how to do so.

`independent`**:** Logical; is the model to be run independently from other submodels?

`options`**:** List of model-specific options, set by `mxOption()`.

`output`**:** List of model output produced during a call to `mxRun()`.

`runstate`**:** List produced by `mxRun()`, which contains the pre-run state of the model object.

`.newobjects`**:** Logical; for internal use.

`.resetdata`**:** Logical; for internal use.

`.wasRun`**:** Logical; for internal use.

`.modifiedSinceRun`**:** Logical; for internal use.

`.version`**:** Object of class `"package_version"`; for internal use.

Methods

$ `signature(x = "MxModel")`: Accessor. Accesses slots by slot-name. Also accesses constituent named entities, by name.

$<- `signature(x = "MxModel")`: Assignment. Generally, this method will not allow the user to make unsafe changes to the MxModel object.

[[`signature(x = "MxModel")`: Accessor for constituent named entities.

[[<- `signature(x = "MxModel")`: Assignment for a named entity.

names `signature(x = "MxModel")`: Returns names of slots and named entities.

print `signature(x = "MxModel")`: "Print" method.

show `signature(object = "MxModel")`: "Show" method.

Note that `imxInitModel()`, `imxModelBuilder()`, `imxTypeName()`, and `imxVerifyModel()` are separately documented methods for class "MxModel".

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

mxExpectationRAM, mxExpectationLISREL, mxModel for creating MxModel objects. More information about the OpenMx package may be found here.

Examples

```
showClass("MxModel")
```

`mxModelAverage` *Information-Theoretic Model-Averaging and Multimodel Inference*

Description

`omxAkaikeWeights()` orders a list of MxModels (hereinafter, the "candidate set" of models) from best to worst AIC, reports their Akaike weights, and indicates which are in the confidence set for best-approximating model. `mxModelAverage()` calls `omxAkaikeWeights()` and includes its output, and also reports model-average point estimates and (if requested) their standard errors.

Usage

```
mxModelAverage(reference=character(0), models=list(),
include=c("onlyFree","all"), SE=NULL, refAsBlock=FALSE, covariances=list(),
type=c("AIC","AICc"), conf.level=0.95)

omxAkaikeWeights(models=list(), type=c("AIC","AICc"), conf.level=0.95)
```

Arguments

`reference`	Vector of character strings referring to parameters, MxMatrices, or MxAlgebras for which model-average estimates are to be computed. Defaults to `NULL`. If a zero-length value is provided, only the output of `omxAkaikeWeights()` is returned, with a warning.
`models`	The candidate set of models: a list of at least two MxModel objects, each of which must be uniquely identified by the value of its `name` slot. Defaults to an empty list.
`include`	Character string, either `"onlyFree"` (default) or `"all"`. When calculating model-average estimates for a given reference quantity, should all the MxModels in the candidate set be included in the calculations, or only those in which the quantity is freely estimated? See below, under "Details," for additional information.
`SE`	Logical; should standard errors be reported for the model-average point estimates? Defaults to `NULL`, in which case standard errors are reported if argument `include="onlyFree"`, and not reported otherwise.
`refAsBlock`	Logical. If `FALSE` (default), `mxModelAverage()` will include a matrix of model-conditional sampling variances for the reference quantities in its output, and model-average results may be based on different subsets of the candidate set if `include="onlyFree"`. If `TRUE`, `mxModelAverage()` will instead include a joint sampling covariance matrix for all reference quantities, and will throw an error if `include="onlyFree"` and if it is not the case that all reference quantities are freely estimated in all models in the candidate set.
`covariances`	Optional list of repeated-sampling covariance matrices of free parameter estimates (possibly from bootstrapping or the sandwich estimator); defaults to an empty list. A non-empty list must either be of the same length as `models`, or have named elements corresponding to names of MxModels in the candidate set. See below, under "Details," for additional information.
`type`	Character string specifying which information criterion to use: either `"AIC"` for the ordinary AIC (default), or `"AICc"` for Hurvich & Tsai's (1989) sample-size corrected AIC.
`conf.level`	Numeric proportion specifying the desired coverage probability of the confidence set for best-approximating model among the candidate set (Burnham & Anderson, 2002). Defaults to 0.95.

Details

If statistical inferences (hypothesis tests and confidence intervals) are the motivation for calculating model-average point estimates and their standard errors, then `include="onlyFree"` (the default) is recommended. Note that, if models in which a quantity is held fixed are included in calculating the quantity's model-average estimate, then that estimate cannot even asymptotically be normally distributed (Bartels, 1997).

If argument `covariances` is non-empty, then either it must be of the same length as argument `models`, or all of its elements must be named after an MxModel in `models` (an MxModel's name is the character string in its `name` slot). If `covariances` is of the same length as `models` but lacks element names, `mxModelAverage()` will assume that they are ordered so that the first element of covariances is to be used with the first MxModel, the second element is to be used with the second MxModel, and so on. Otherwise, mxModelAverage() assigns the elements of covariances to the MxModels by matching element names to MxModel names. If covariances doesn't provide a covariance matrix for a given MxModel–perhaps because it is empty, or only provides matrices for a nonempty proper subset of the candidate set–mxModelAverage() will fall back to its default behavior of calculating a covariance matrix from the Hessian matrix in the MxModel's output slot. If a covariance matrix cannot be thus calculated and SE=TRUE, SE is coerced to FALSE, with a warning.

The matrices in covariances must have complete row and column names, equal to the free parameter labels of the corresponding MxModel. These names indicate to which free parameter a given row or column corresponds.

Value

omxAkaikeWeights() returns a dataframe, with one row for each element of models. The rows are sorted by their MxModel's AIC (or AICc), from best to worst. The dataframe has five columns:

1. "model": Character string. The name of the MxModel.
2. "AIC" or "AICc": Numeric. The MxModel's AIC or AICc.
3. "delta": Numeric. The MxModel's AIC (or AICc) minus the best (smallest) AIC (or AICc) in the candidate set.
4. "AkaikeWeight": Numeric. The MxModel's Akaike weight. This column will sum to unity.
5. "inConfidenceSet": Character. Will contain an asterisk if the MxModel is in the confidence set for best-approximating model.

The dataframe also has an attribute, "unsortedModelNames", which contains the names of the MxModels in the same order as they appear in models (i.e., without sorting them by their AIC).

If a zero-length value is provided for argument reference, then mxModelAverage() returns only the output of omxAkaikeWeights(), with a warning. Otherwise, for the default values of its arguments, mxModelAverage() returns a list with four elements:

1. "Model-Average Estimates": A numeric matrix with one row for each distinct quantity specified by reference, and as many as two columns. Its rows are named for the corresponding reference quantities. Its first column, "Estimate", contains the model-average point estimates. If standard errors are being calculated, then its second column, "SE", contains the "model-unconditional" standard errors of the model-average point estimates. Otherwise, there is no second column.
2. "Model-wise Estimates": A numeric matrix with one row for each distinct quantity specified by reference (indicated by row name), and one column for each MxModel (indicated by column name). Each element is an estimate of the given reference quantity, from the given MxModel. Quantities that cannot be evaluated for a given MxModel are reported as NA.
3. "Model-wise Sampling Variances": A numeric matrix just like the one in list element 2, except that its elements are the estimated sampling variances of the corresponding model-conditional point estimates in list element 2. Variances for fixed quantities are reported as 0 if include="all", and as NA if include="onlyFree"; however, if no covariance matrix is available for a model, all of that model's sampling variances will be reported as NA.
4. "Akaike-Weights Table": The output from omxAkaikeWeights().

If `refAsBlock=TRUE`, list element 3 will instead contain be named `"Joint Covariance Matrix"`, and if `SE=TRUE`, it will contain the joint sampling covariance matrix for the model-average point estimates.

Note

The "best-approximating model" is defined as the model that truly ("in the population," so to speak) has the smallest Kullback-Leibler divergence from full reality, among the models in the candidate set (Burnham & Anderson, 2002).

A model's Akaike weight is interpretable as the relative weight-of-evidence for that model being the best-approximating model, given the observed data and the candidate set. It has a Bayesian interpretation as the posterior probability that the given model is the best-approximating model in the candidate set, assuming a "savvy" prior probability that depends upon sample size and the number of free parameters in the model (Burnham & Anderson, 2002).

The confidence set for best-approximating model serves to reflect sampling error in the AICs. When fitting the candidate set to data over repeated sampling, the confidence set is expected to contain the best-approximating model with probability equal to its confidence level.

The sampling variances and covariances of the model-average point estimates are calculated from Equations (4) and (5) in Burnham & Anderson (2004). The standard errors reported by `mxModelAverage()` are the square roots of those sampling variances.

For an example of model-averaging and multimodel inference applied to structural equation modeling using OpenMx v1.3 (i.e., well before the functions documented here were implemented), see Kirkpatrick, McGue, & Iacono (2015).

References

Bartels, L. M. (1997). Specification uncertainty and model averaging. *American Journal of Political Science, 41*(2), 641-674.

Burnham, K. P., & Anderson, D. R. (2002). *Model Selection and Multimodel Inference: A Practical Information-Theoretic Approach (2nd ed.).* New York: Springer.

Burnham, K. P., & Anderson, D. R. (2004). Multimodel inference: Understanding AIC and BIC in model selection. *Sociological Methods & Research, 33*(2), 261-304. doi:10.1177/0049124104268644

Hurvich, C. M., & Tsai, C-L. (1989). Regression and time series model selection in small samples. *Biometrika, 76*(2), 297-307.

Kirkpatrick, R. M., McGue, M., & Iacono, W. G. (2015). Replication of a gene-environment interaction via multimodel inference: Additive-genetic variance in adolescents' general cognitive ability increases with family-of-origin socioeconomic status. *Behavior Genetics, 45*, 200-214.

See Also

`mxCompare()`

Examples

```
require(OpenMx)
data(demoOneFactor)
factorModel1 <- mxModel(
"OneFactor1",
mxMatrix(
"Full", 5, 1, values=0.8,
labels=paste("a",1:5,sep=""),
free=TRUE, name="A"),
mxMatrix(
"Full", 5, 1, values=1,
labels=paste("u",1:5,sep=""),
free=TRUE, name="Udiag"),
mxMatrix(
"Symm", 1, 1, values=1,
free=FALSE, name="L"),
mxAlgebra(vec2diag(Udiag),name="U"),
mxAlgebra(A %*% L %*% t(A) + U, name="R"),
mxExpectationNormal(
covariance = "R",
dimnames = names(demoOneFactor)),
mxFitFunctionML(),
mxData(cov(demoOneFactor), type="cov", numObs=500))
factorFit1 <- mxRun(factorModel1)
#Constrain unique variances equal:
factorModel2 <- omxSetParameters(
model=factorModel1,labels=paste("u",1:5,sep=""),
newlabels="u",name="OneFactor2")
factorFit2 <- mxRun(factorModel2)
omxAkaikeWeights(models=list(factorFit1,factorFit2))

mxModelAverage(
reference=c("A","Udiag"), include="all",
models=list(factorFit1,factorFit2))
```

mxNormalQuantiles *mxNormalQuantiles*

Description

Get quantiles from a normal distribution

Usage

```
mxNormalQuantiles(nBreaks, mean = 0, sd = 1)
```

Arguments

nBreaks	the number of thresholds, or a vector of the number of thresholds
mean	the mean of the underlying normal distribution
sd	the standard deviation of the underlying normal distribution

Value

a vector of quantiles

Examples

```
mxNormalQuantiles(3)
mxNormalQuantiles(3, mean=7)
mxNormalQuantiles(2, mean=1, sd=3)
```

---

mxOption *Set or Clear an Optimizer Option*

Description

The function sets, shows, or clears an option that is specific to the optimizer in the back-end.

Usage

```
mxOption(model=NULL, key=NULL, value, reset = FALSE)
```

Arguments

`model`	An MxModel object or NULL
`key`	The name of the option.
`value`	The value of the option.
`reset`	If TRUE then reset all options to their defaults.

Details

mxOption is used to set, clear, or query an option (given in the 'key' argument) in the back-end optimizer. Valid option keys are listed below.

Use value = NULL to remove an existing option. Leaving value blank will return the current value of the option specified by 'key'.

To reset all options to their default values, use 'reset = TRUE'. When reset = TRUE, 'key' and 'value' are ignored.

If the 'model' argument is set to NULL, the default optimizer option (i.e those applying to all models by default) will be set.

To see the defaults, use `getOption('mxOptions')`.

Before the model is submitted to the back-end, all keys and values are converted into strings using the as.character function.

Optimizer specific options

The "Default optimizer" option can only be set globally (i.e., with `model=NULL`), and not locally (i.e., specifically to a given MxModel). Although the checkpointing options may be set globally, OpenMx's behavior is only affected by locally set checkpointing options (that is, global checkpointing options are ignored at runtime).

Gradient-based optimizers require the gradient of the fit function. When analytic derivatives are not available, the gradient is estimated numerically. There are a variety of options to control the numerical estimation of the gradient. One option for CSOLNP and SLSQP is the gradient algorithm. CSOLNP uses the `forward` method by default, while SLSQP uses the `central` method. The `forward` method requires 1 time "Gradient iterations" function evaluation per parameter per gradient, while `central` method requires 2 times "Gradient iterations" function evaluations per parameter per gradient. Users can change the default methods for either of these optimizers by setting the "Gradient algorithm" option. NPSOL usually uses the `forward` method, but adaptively switches to `central` under certain circumstances.

Options "Gradient step size", "Gradient iterations", and "Function precision" have on-load global defaults of `"Auto"`. If value `"Auto"` is in effect for any of these three options at runtime, then OpenMx selects a reasonable numerical value in its place. These automated numerical values are intended to (1) adjust for the limited precision of the algorithm for computing multivariate-normal probability integrals, and (2) calculate accurate numeric derivatives at the optimizer's solution. If the user replaces `"Auto"` with a valid numerical value, then OpenMx uses that value as-is.

By default, CSOLNP uses a step size of 10^-7 whereas SLSQP uses 10^-5. The purpose of this difference is to obtain roughly the same accuracy given other differences in numerical procedure. If you set a non-default "Gradient step size", it will be used as-is. NPSOL ignores "Gradient step size", and instead uses a function of mxOption "Function precision" to determine its gradient step size.

Option "Analytic Gradients" affects all three optimizers, but some options only affect certain optimizers. Option "Gradient algorithm" is used by CSOLNP and SLSQP, and ignored by NPSOL. Option "Gradient iterations" only affects SLSQP. Option "Gradient step size" is used slightly differently by SLSQP and CSOLNP, and is ignored by NPSOL (see `mxComputeGradientDescent()` for details).

If an mxModel contains mxConstraints, NPSOL is given .4 times the value of the option "Feasibility tolerance". If there are no constraints, NPSOL is given a hard-coded value of 1e-5 (its own native default).

Note: Where constraints are present, NPSOL is given 0.4 times the value of the mxOption "Feasibility Tolerance", and this is about a million times bigger than NPSOL's own native default. Values of "Feasibility Tolerance" around 1e-5 may be needed to get constraint performance similar to NPSOL's default. Note also that NPSOL's criterion for returning a status code of 0 versus 1 for a given solution depends partly on "Optimality tolerance".

For a block of `n` ordinal variables, the maximum number of integration points that OpenMx may use to calculate multivariate-normal probability integrals is given by `mvnMaxPointsA + mvnMaxPointsB*n + mvnMaxPointsC*n*n + exp(mvnMaxPointsD + mvnMaxPointsE * n * log(mvnRelEps))`. Integral approximation is stopped once either 'mvnAbsEps' or 'mvnRelEps' is satisfied. Use of 'mvnAbsEps' is deprecated.

The maximum number of major iterations (the option "Major iterations") for optimization for NPSOL can be specified either by using a numeric value (such as 50, 1000, etc) or by specifying a user-defined function. The user-defined function should accept two arguments as input, the number of parameters and the number of constraints, and return a numeric value as output.

OpenMx options

Calculate Hessian	[Yes | No]	calculate the Hessian explicitly after optimization.
Standard Errors	[Yes | No]	return standard error estimates from the explicitly calculate hessian.
Default optimizer	[NPSOL | SLSQP | CSOLNP]	the gradient-descent optimizer to use
Number of Threads	[0|1|2|...|10|...]	number of threads used for optimization. Default value is taken from
Feasibility tolerance	*r*	the maximum acceptable absolute violations in linear and nonlinear c
Optimality tolerance	*r*	the accuracy with which the final iterate approximates a solution to th
Gradient algorithm	see list	finite difference method, either 'forward' or 'central'.
Gradient iterations	1:4	the number of Richardson extrapolation iterations
Gradient step size	*r*	amount of change made to free parameters when numerically calculat
Analytic Gradients	[Yes | No]	should the optimizer use analytic gradients (if available)?
loglikelihoodScale	*i*	factor by which the loglikelihood is scaled.
Parallel diagnostics	[Yes | No]	whether to issue diagnostic messages about use of multiple threads
Nudge zero starts	[TRUE | FALSE]	Should OpenMx "nudge" starting values of zero to 0.1 at runtime?
Status OK	character vector	Status codes that are considered to indicate a successful optimization
Max minutes	numeric	Maximum backend elapsed time, in minutes

NPSOL-specific options

Nolist		this option suppresses printing of the options
Print level	*i*	the value of *i* controls the amount of printout produced by the major iterations
Minor print level	*i*	the value of *i* controls the amount of printout produced by the minor iterations
Print file	*i*	for $i > 0$ a full log is sent to the file with logical unit number *i*.
Summary file	*i*	for $i > 0$ a brief log will be output to file *i*.
Function precision	*r*	a measure of accuracy with which the fitfunction and constraint functions can be co
Infinite bound size	*r*	if $r > 0$ defines the "infinite" bound bigbnd.
Major iterations	*i* or a function	the maximum number of major iterations before termination.
Verify level	[-1:3 | Yes | No]	see NPSOL manual.
Line search tolerance	*r*	controls the accuracy with which a step is taken.
Derivative level	[0-3]	see NPSOL manual.
Hessian	[Yes | No]	return the Hessian (Yes) or the transformed Hessian (No).
Step Limit	*r*	maximum change in free parameters at first step of linesearch.

Checkpointing options

Always Checkpoint	[Yes | No]	whether to checkpoint all models during optimization.
Checkpoint Directory	path	the directory into which checkpoint files are written.
Checkpoint Prefix	string	the string prefix to add to all checkpoint filenames.
Checkpoint Fullpath	path	overrides the directory and prefix (useful to output to /dev/fd/2)
Checkpoint Units	see list	the type of units for checkpointing: 'minutes', 'iterations', or 'evaluations'.
Checkpoint Count	*i*	the number of units between checkpoint intervals.

Model transformation options

Error Checking	[Yes | No]	whether model consistency checks are performed in the OpenMx front-end
No Sort Data		character vector of model names for which FIML data sorting is not performed
RAM Inverse Optimization	[Yes | No]	whether to enable solve(I - A) optimization

RAM Max Depth	*i*	the maximum depth to be used when solve(I - A) optimization is enabled

Multivariate normal integration parameters

maxOrdinalPerBlock	*i*	maximum number of ordinal variables to evaluate together
mvnMaxPointsA	*i*	base number of integration points
mvnMaxPointsB	*i*	number of integration points per ordinal variable
mvnMaxPointsC	*i*	number of integration points per squared ordinal variables
mvnMaxPointsD	*i*	see details
mvnMaxPointsE	*i*	see details
mvnAbsEps	*i*	absolute error tolerance
mvnRelEps	*i*	relative error tolerance

Value

If a model is provided, it is returned with the optimizer option either set or cleared. If value is empty, the current value is returned.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

See `mxModel()`, as almost all uses of `mxOption()` are via an mxModel whose options are set or cleared. See `mxComputeGradientDescent()` for details on how different optimizers are affected by different options. See as.statusCode for information about the `Status OK` option.

Examples

```
# set the Numbder of Threads (cores to use)
mxOption(key="Number of Threads", value=imxGetNumThreads())

testModel <- mxModel(model = "testModel5") # make a model to use for example
testModel$options   # show the model options (none yet)
options()$mxOptions # list all mxOptions (global settings)

testModel <- mxOption(testModel, "Function precision", 1e-5) # set precision
testModel <- mxOption(testModel, "Function precision", NULL) # clear precision
# N.B. This is model-specific precision (defaults to global setting)

# may optimize for speed
# at cost of not getting standard errors
testModel <- mxOption(testModel, "Calculate Hessian", "No")
testModel <- mxOption(testModel, "Standard Errors"  , "No")

testModel$options # see the list of options you set
```

`MxOptionalChar-class` *An optional character*

Description

An optional character

`MxOptionalCharOrNumber-class` *A character, integer, or NULL*

Description

A character, integer, or NULL

`MxOptionalDataFrame-class` *An optional data.frame*

Description

An optional data.frame

`MxOptionalDataFrameOrMatrix-class` *An optional data.frame or matrix*

Description

An optional data.frame or matrix

`MxOptionalInteger-class` *An optional integer*

Description

An optional integer

MxOptionalLogical-class

An optional logical

Description

This is an internal class, the union of NULL and logical.

MxOptionalMatrix-class

An optional matrix

Description

An optional matrix

MxOptionalNumeric-class

An optional numeric

Description

An optional numeric

mxParametricBootstrap

Assess whether potential parameters should be freed using parametric bootstrap

Description

Data is simulated from 'nullModel'. The parameters named by 'labels' are freed to obtain the alternative model. The alternative model is fit against each simulated data set.

Usage

```
mxParametricBootstrap(nullModel, labels,
   alternative=c("two.sided", "greater", "less"),
   ..., alpha=0.05, correction=p.adjust.methods,
   previousRun=NULL, replications=400, checkHess=FALSE,
   signif.stars = getOption("show.signif.stars"))
```

Arguments

`nullModel`	The model specifying the null distribution
`labels`	A character vector of parameters to free to obtain the alternative model
`alternative`	a character string specifying the alternative hypothesis
`...`	Not used. Forces remaining arguments to be specified by name.
`alpha`	The false positive rate
`correction`	How to adjust the p values for multiple tests.
`replications`	The number of resampling replications. If available, replications from prior invocation will be reused.
`previousRun`	Results to re-use from a previous bootstrap.
`checkHess`	Whether to approximate the Hessian in each replication
`signif.stars`	logical. If TRUE, ‘significance stars’ are printed for each coefficient.

Details

When the model has a default compute plan and ‘checkHess’ is kept at FALSE then the Hessian will not be approximated or checked. On the other hand, ‘checkHess’ is TRUE then the Hessian will be approximated by finite differences. This procedure is of some value because it can be informative to check whether the Hessian is positive definite (see `mxComputeHessianQuality`). However, approximating the Hessian is often costly in terms of CPU time. For bootstrapping, the parameter estimates derived from the resampled data are typically of primary interest.

Value

A data frame is returned containing the test results. Details of the bootstrap replications are stored in the ‘bootData’ attribute on the data frame.

Examples

```
library(OpenMx)

data(demoOneFactor)
manifests <- names(demoOneFactor)
latents <- c("G")

base <- mxModel(
  "OneFactorCov", type="RAM",
  manifestVars = manifests,
  latentVars = latents,
 mxPath(from=latents, to=manifests, values=0, free=FALSE, labels=paste0('l',1:length(manifests))),
  mxPath(from=manifests, arrows=2, values=rlnorm(length(manifests)), lbound=.01),
  mxPath(from=latents, arrows=2, free=FALSE, values=1.0),
 mxPath(from = 'one', to = manifests, values=0, free=TRUE, labels=paste0('m',1:length(manifests))),
  mxData(demoOneFactor, type="raw"))

base <- mxRun(base)
```

```
# use more replications, 8 is for demonstration purpose only
mxParametricBootstrap(base, paste0('l', 1:length(manifests)),
                      "two.sided", replications=8)
```

`mxPath` *Create List of Paths*

Description

This function creates a list of paths.

Usage

```
mxPath(from, to = NA, connect = c("single", "all.pairs", "unique.pairs",
    "all.bivariate", "unique.bivariate"), arrows = 1,
    free = TRUE, values = NA, labels = NA,
    lbound = NA, ubound = NA, ..., joinKey = as.character(NA), step = c())
```

Arguments

`from`	character vector. These are the sources of the new paths.
`to`	character vector. These are the sinks of the new paths.
`connect`	String. Specifies the type of source to sink connection: "single", "all.pairs", "all.bivariate", "unique.pairs", "unique.bivariate". Default value is "single".
`arrows`	numeric value. Must be either 0 (for Pearson selection), 1 (for single-headed), or 2 (for double-headed arrows).
`free`	boolean vector. Indicates whether paths are free or fixed.
`values`	numeric vector. The starting values of the parameters.
`labels`	character vector. The names of the paths.
`lbound`	numeric vector. The lower bounds of free parameters.
`ubound`	numeric vector. The upper bounds of free parameters.
`...`	Not used. Allows OpenMx to catch the use of the deprecated 'all' argument.
`joinKey`	character vector. The name of the foreign key to join against some other model to create a cross model path (regression or factor loading.
`step`	numeric vector. The priority for processing arrows=0 paths. For example, step 1 is processed before step 2.

Details

The mxPath function creates MxPath objects. These consist of a list of paths describing the relationships between variables in a model using the RAM modeling approach (McArdle and MacDonald, 1984). Variables are referenced by name, and these names must appear in the 'manifestVars' and 'latentVars' arguments of the mxModel function.

Paths are specified as going "from" one variable (or set of variables) "to" another variable or set of variables using the ‘from’ and ‘to’ arguments, respectively. If ‘to’ is left empty, it will be set to the value of ‘from’.

‘connect’ has five possible connection types: "single", "all.pairs", "all.bivariate", "unique.pairs", "unique.bivariate". The default value is "single". Assuming the values c(‘a’,‘b’,‘c’) for the ‘to’ and ‘from’ fields the paths produced by each connection type are as follows:

"all.pairs": (a,a), (a,b), (a,c), (b,a), (b,b), (b,c), (c,a), (c,b), (c,c).

"unique.pairs": (a,a), (a,b), (a,c), (b,b), (b,c), (c,c).

"all.bivariate": (a,b), (a,c), (b,a), (b,c), (c,a), (c,b).

"unique.bivariate": (a,b), (a,c), (b,c).

"single": (a,a), (b,b), (c,c).

Multiple variables may be input as a vector of variable names. If the ‘connect’ argument is set to "single", then paths are created going from each entry in the ‘from’ vector to the corresponding entry in the ‘to’ vector. If the ‘to’ and ‘from’ vectors are of different lengths when the ‘connect’ argument is set to "single", the shorter vector is repeated to make the vectors of equal length.

The ‘free’ argument specifies whether the paths created by the mxPath function are free or fixed parameters. This argument may take either TRUE for free parameters, FALSE for fixed parameters, or a vector of TRUEs and FALSEs to be applied in order to the created paths.

The ‘arrows’ argument specifies the type of paths created. A value of 1 indicates a one-headed arrow representing regression. This path represents a regression of the ‘to’ variable on the ‘from’ variable, such that the arrow points to the ‘to’ variable in a path diagram. A value of 2 indicates a two-headed arrow, representing a covariance or variance. If multiple paths are created in the same mxPath function, then the ‘arrows’ argument may take a vector of 1s and 2s to be applied to the set of created paths.

The ‘values’ is a numeric vectors containing the starting values of the created paths. ‘values’ gives a starting value for estimation. The ‘labels’ argument specifies the names of the resulting MxPath object. The ‘lbound’ and ‘ubound’ arguments specify lower and upper bounds for the created paths.

Value

Returns a list of paths.

Note

The previous implementation of ‘all’ had unsafe features. Its use is now deprecated, and has been replaced by the new mechanism ‘connect’ which supports safe and controlled generation of desired combinations of paths.

References

McArdle, J. J. and MacDonald, R. P. (1984). Some algebraic properties of the Reticular Action Model for moment structures. *British Journal of Mathematical and Statistical Psychology, 37,* 234-251.

The OpenMx User’s guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

mxMatrix for a matrix-based approach to path specification; mxModel for the container in which mxPaths are embedded. More information about the OpenMx package may be found here.

Examples

```
# A simple Example: 1 factor Confirmatory Factor Analysis

library(OpenMx)

data(demoOneFactor)
manifests <- names(demoOneFactor)
latents   <- c("G")
factorModel <- mxModel(model="One Factor", type="RAM",
      manifestVars = manifests,
      latentVars   = latents,
      mxPath(from=latents, to=manifests),
      mxPath(from=manifests, arrows=2),
      mxPath(from=latents, arrows=2,free=FALSE, values=1.0),
      mxData(cov(demoOneFactor), type="cov",numObs=500)
)
factorFit <-mxRun(factorModel)
summary(factorFit)

# A more complex example using features of R to compress
#  what would otherwise be a long and error-prone script

# list of 100 variable names: "01"  "02"  "03"...
myManifest <- sprintf("%02d", c(1:100))

# the latent variables for the model
myLatent <- c("G1", "G2", "G3", "G4", "G5")


# Start building the model:
#  Define its type, and add the manifest and latent variable name lists
testModel <- mxModel(model="testModel6", type = "RAM",
                     manifestVars = myManifest, latentVars = myLatent)

# Create covariances between the latent variables and add to the model
# Here we use combn to create the covariances
# nb: To create the variances and covariances in one operation you could use
# expand.grid(myLatent,myLatent) to specify from and to

uniquePairs <- combn(myLatent,2)
covariances <- mxPath(from = uniquePairs[1,],
                      to=uniquePairs[2,], arrows = 2, free = TRUE, values = 1)
testModel <- mxModel(model=testModel, covariances)

# Create variances for the latent variables
variances <- mxPath(from = myLatent,
                    to=myLatent, arrows = 2, free = TRUE, values = 1)
```

```
testModel <- mxModel(model=testModel, variances) # add variances to the model

# Make a list of paths from each packet of 20 manifests
#  to one of the 5 latent variables
# nb: The first loading to each latent is fixed to 1 to scale its variance.
singles <- list()
for (i in 1:5) {
    j <- i*20
    singles <- append(singles, mxPath(
                        from = myLatent[i], to = myManifest[(j - 19):j],
                        arrows = 1,
                        free   = c(FALSE, rep(TRUE, 19)),
                        values = c(1, rep(0.75, 19))))
}

# add single-headed paths to the model
testModel <- mxModel(model=testModel, singles)
```

mxPearsonSelCov *Perform Pearson Aitken selection*

Description

[Maturing] These functions implement the Pearson Aitken selection formulae.

Usage

```
mxPearsonSelCov(origCov, newCov)
mxPearsonSelMean(origCov, newCov, origMean)
```

Arguments

origCov	covariance matrix. The covariance prior to selection.
newCov	covariance matrix. A subset of origCov to replace.
origMean	column vector. A mean vector to adjust.

Details

Which dimensions to condition on can be communicated in one of two ways: (1) newCov is a submatrix of origCov. The dimnames are matched to determine which partition of origCov to replace with newCov. Or (2) newCov is the same dimension as origCov. The matrix entries are inspected to determine which entries have changed. The changed entries determine which partition of origCov to replace with newCov.

Let the $n \times n$ covariance matrix R (origCov) be partitioned into non-empty, disjoint sets p and q. Let R_{ij} denote the covariance matrix between the p and q variables where the subscripts denote the variable subsets (e.g. R_{pq}). Let column vectors μ_p and μ_q contain the means of p and q variables, respectively. We wish to compute the conditional covariances of the variables in q for a subset of the population where R_{pp} and μ_p are known (or partially known)—that is, we wish to *condition* the covariances and means of q on those of p. Let V_{pp} (`newCov`) be an arbitrary covariance matrix of the same dimension as R_{pp}. If we replace R_{pp} by V_{pp} then the mean of q (`origMean`) is transformed as

$$\mu_q \to \mu_q + R_{qp} R_{pp}^{-1} \mu_p$$

and the covariance of p and q are transformed as

$$\left[\begin{array}{c|c} R_{pp} & R_{pq} \\ \hline R_{qp} & R_{qq} \end{array}\right] \to \left[\begin{array}{c|c} V_{pp} & V_{pp} R_{pp}^{-1} R_{pq} \\ \hline R_{qp} R_{pp}^{-1} V_{pp} & R_{qq} - R_{qp}(R_{pp}^{-1} - R_{pp}^{-1} V_{pp} R_{pp}^{-1}) R_{pq} \end{array}\right]$$

References

Aitken, A. (1935). Note on selection from a multivariate normal population. *Proceedings of the Edinburgh Mathematical Society (Series 2), 4*(2), 106-110. doi:10.1017/S0013091500008063

Examples

```
library(OpenMx)

m1 <- mxModel(
  'selectionTest',
  mxMatrix('Full', 10, 10, values=rWishart(1, 20, toeplitz((10:1)/10))[,,1],
           dimnames=list(paste0('c',1:10),paste0('c',1:10)), name="m1"),
  mxMatrix('Full', 2, 2, values=diag(2),
           dimnames=list(paste0('c',1:2),paste0('c',1:2)), name="m2"),
  mxMatrix('Full', 10, 1, values=runif(10),
           dimnames=list(paste0('c',1:10),c('v')), name="u1"),
  mxAlgebra(mxPearsonSelCov(m1, m2), name="c1"),
  mxAlgebra(mxPearsonSelMean(m1, m2, u1), name="u2")
)

m1 <- mxRun(m1)
```

`mxPowerSearch` *Power curve*

Description

`mxPower` is used to evaluate the power to distinguish between a hypothesized effect (`trueModel`) and a model where this effect is set to the value of the null hypothesis (`falseModel`).

`mxPowerSearch` evaluates power across a range of sample sizes or effect sizes, choosing intelligent values of, for example, sample size to explore.

Both functions are flexible, with multiple options to control *n*, `sig.level`, `method`, and other factors. Several parameters can take a vector as input. These options are described in detail below.

Evaluation can either use the non-centrality parameter (which can be very quick) or conduct an empirical evaluation.

note: During longer evaluations, the functions printout helpful progress consisting of a note about what task is being conducted, e.g. "Search n:power relationship for 'A11'" and, beneath that, an update on the run, the model being evaluated, and the current candidate value being considered, e.g. "R 15 1p N 79"

Usage

```
mxPowerSearch(trueModel, falseModel, n=NULL, sig.level=0.05, ...,
        probes=300L, previousRun=NULL,
        gdFun=mxGenerateData,
        method=c('empirical', 'ncp'),
        grid=NULL,
        statistic=c('LRT','AIC','BIC'),
OK=mxOption(trueModel, "Status OK"),
checkHess=FALSE,
silent=!interactive())

mxPower(trueModel, falseModel, n=NULL, sig.level=0.05, power=0.8, ...,
        probes=300L, gdFun=mxGenerateData,
        method=c('empirical', 'ncp'),
        statistic=c('LRT','AIC','BIC'),
        OK=mxOption(trueModel, "Status OK"), checkHess=FALSE,
        silent=!interactive())
```

Arguments

`trueModel`	The true generating model for the data.
`falseModel`	Model representing the null hypothesis that we wish to reject.
`n`	Total rows summing across all submodels proportioned as given in the trueModel. Default = NULL. If provided, result (power or alpha) solved-for at the given total sample size.
`sig.level`	A single value for the p-value (aka false positive or type-1 error rate). Default = .05.
`power`	One or values for power (a.k.a. 1 - type 2 error) to evaluate. Default = .80
`...`	Not used.
`probes`	The number of probes to use when method = ‘empirical’.
`previousRun`	Output from a prior run of ‘mxPowerSearch’ to build on.
`gdFun`	The function invoked to generate new data for each Monte Carlo probe. Default = `mxGenerateData`
`method`	To estimate power: ‘empirical’ (Monte Carlo method) or ‘ncp’ (average noncentrality method).
`grid`	A vector of locations at which to evaluate the power. If not provided, a reasonable default will be chosen.
`statistic`	Which test to use to compare models (Default = ’LRT’).
`OK`	The set of status codes that are considered successful.
`checkHess`	Whether to approximate the Hessian in each replication.
`silent`	Whether to show a progress indicator.

Details

Power is the chance of obtaining a significant difference when there is a true difference, i.e., (1 - false negative rate). The likelihood ratio test is used by default. There are two methods available to produce a power curve. The default, `method = empirical`, works for any model where the likelihood ratio test works. For example, definition variables and missing data are fine, but parameters estimated at upper or lower bounds will cause problems. The `method = empirical` can require a lot of time because the models need to be fit 100s of times. An alternate approach, `method = ncp`, is much more efficient and takes advantage of the fact that the non-null distribution of likelihood ratio test statistic is often $\chi^2(df_1 - df_0, N\lambda)$. That is, the non-centrality parameter, $\lambda(lambda)$, can be assumed, on average, to contribute equally per row. This permits essentially instant power curves without the burden of tedious simulation. However, definition variables, missing data, or unconventional models (e.g., mixture distributions) can violate this assumption. Therefore, we recommend verifying that the output from `method = empirical` roughly matches `method = ncp` on the model of interest before relying on `method = ncp`.

note: Unlike `method = empirical`, `method = ncp` does not use the `gdFun` argument and can be used with models that have summary statistics rather than raw data.

When `method = ncp`, parameters of both 'trueModel' and 'falseModel' are assumed to be converged to their desired values. In contrast, when `method = empirical`, 'trueModel' need not be run or even contain data. On each replication, data are generated from 'trueModel' at the given parameter vector. Then both 'trueModel' and 'falseModel' are fit against these data.

When `statistic = 'LRT'` then the models must be nested and `sig.level` is used to determine whether the test is rejected. For 'AIC' and 'BIC', the test is regarded as rejected when the 'trueModel' obtains a lower score than the 'falseModel'. In contrast to `statistic='LRT'`, there is no nesting requirement. For example, 'AIC' can be used to compare a 'trueModel' against its corresponding saturated model.

`mxPower` operates in many modes. When power is passed as `NULL` then power is calculated and returned. When power (as a scalar or vector) is given then sample or effect size is (are) returned. If you pass a list of models for 'falseModel', each model will be checked in turn and the results returned as a vector. If you pass a vector of sample sizes, each sample size will be checked in turn and the results returns as a vector.

mxPowerSearch

In contrast to `mxPower`, `mxPowerSearch` attempts to model the whole relationship between sample size or effect size and power. A naive Monte Carlo estimate of power examines a single candidate sample size at a time. To obtain the whole curve, and simultaneously, to reduce the number of simulation probes, `mxPowerSearch` employs a binomial family generalized linear model with a logit link to predict the power curve across the sample sizes of interest (similar to Schoemann et al, 2014).

The `mxPowerSearch` algorithm works in 3 stages. Without loss of generality, our description will use the sample size to power relationship, but a similar process is used when evaluating the relationship of parameter value to power. In the first stage, a crude binary search is used to find the range reasonable values for N. This stage is complete once we have at least two rejections and at least two non-rejections. At this point, the binomial intercept and slope model is fit to these data. If the *p*-value for the slope is less than 0.25 then we jump to stage 3. Otherwise, we fit an intercept only binomial model. Our goal is to nail down the intercept (where power=0.5) because this is the easiest point to find and is a necessary prerequisite to estimate the variance (a.k.a. slope). Therefore, we probe at the median of previous probes stepping by 10% in the direction of the model's predicted intercept. Eventually, after enough probes, we reach stage 3 where the *p*-value for the slope is less than 0.25. At stage 3, our goal is to nail down the interesting part of the power curve. Therefore, we cycle serially through probes at 0, 1, and 2 logits from the intercept. This process is continued for the permitted number of `probes`. Occasionally, the *p*-value for the slope in the stage 3 model grows larger than 0.25. In this case, we switch back to stage 2 (intercept only) until the stage 3 model start working again. There are no other convergence criteria. Accuracy continues to improves until the probe limit is reached.

note: After `mxPowerSearch` returns, you might find you wanted to run additional probes (i.e., bounds are wider than you'd like). You can run more probes without starting from scratch by passing the previous result back into the function using the `previousRun` argument.

When 'n' is fixed then `mxPowerSearch` helps answer the question, "how small of a true effect is likely to be detected at a particular sample size?" Only one parameter can be considered at a time. The way the simulation works is that a candidate value for the parameter of interest is loaded into the `trueModel`, data are generated, then both the true and false model are fit to the data to obtain the difference in fit. The candidate parameter is initially set to halfway between the `trueModel` and `falseModel`. The power curve will reflect the smallest distance, relative to the false model, required to have a good chance to reject the false model according to the chosen statistic.

Note that the default `grid` is chosen to show the interesting part of the power curve (from 0.25 to 0.97). Especially for `method=ncp`, this curve is practically identical for any pair of models (but located at a different range of sample sizes). However, if you wish to align power curves from more than one power analysis, you should select your own `grid` points (perhaps pass in the power array from the first to subsequent using `grid =`).

Note: CI is not given for method=ncp: The estimates are exact (to the precision of the true and null/false model solutions provided by the user).

Value

`mxPower` returns a vector of sample sizes, powers, or effect sizes.

`mxPowerSearch` returns a data.frame with one row for each 'grid' point. The first column is either the sample size 'N' (given as the proportion of rows provided in `trueModel`) or the parameter label. The second column is the power. If `method=empirical`, `lower` and `upper` provide a +/-2 SE confidence interval (CI95) for the power (as estimated by the binomial logit linear model). When `method=empirical` then the 'probes' attribute contains a data.frame record of the power estimation process.

References

Schoemann, A. M., Miller, P., Pornprasertmanit, S. & Wu, W. (2014). Using Monte Carlo simulations to determine power and sample size for planned missing designs. *International Journal of Behavioral Development*, **38**, 471-479.

See Also

`mxCompare`, `mxRefModels`

Examples

```
library(OpenMx)

# Make a 1-factor model of 5 correlated variables
data(demoOneFactor)
manifests <- names(demoOneFactor)
latents <- c("G")
factorModel <- mxModel("One Factor", type="RAM",
   manifestVars = manifests,
   latentVars = latents,
   mxPath(from= latents, to= manifests, values= 0.8),
   mxPath(from= manifests, arrows= 2,values= 1),
   mxPath(from= latents, arrows= 2, free= FALSE, values= 1),
   mxPath(from= "one", to= manifests),
   mxData(demoOneFactor, type= "raw")
)
factorModelFit <- mxRun(factorModel)

# The loading of x1 on G is estimated ~ 0.39
# Let's test fixing it to .3
indModel <- factorModelFit
indModel$A$values['x1','G'] <- 0.3
indModel$A$free['x1','G'] <- FALSE
indModel <- mxRun(indModel)

# What power do we have at different sample sizes
# to detect that the G to x1 factor loading is
# really 0.3 instead of 0.39?
mxPowerSearch(factorModelFit, indModel, method='ncp')

# If we want to conduct a study with 80% power to
# find that  the G to x1 factor loading is
# really 0.3 instead of 0.39, what sample size
# should we use?
mxPower(factorModelFit, indModel, method='ncp')
```

`MxRAMGraph-class` *MxRAMGraph*

Description

This is an internal class and should not be used directly. It is a class for RAM directed graphs.

`MxRAMModel-class` *MxRAMModel*

Description

This is an internal class and should not be used directly.

`mxRAMObjective` *DEPRECATED: Create MxRAMObjective Object*

Description

WARNING: Objective functions have been deprecated as of OpenMx 2.0.

Please use mxExpectationRAM() and mxFitFunctionML() instead. As a temporary workaround, mxRAMObjective returns a list containing an MxExpectationNormal object and an MxFitFunctionML object.

All occurrences of

mxRAMObjective(A, S, F, M = NA, dimnames = NA, thresholds = NA, vector = FALSE, threshnames = dimnames)

Should be changed to

mxExpectationRAM(A, S, F, M = NA, dimnames = NA, thresholds = NA, threshnames = dimnames) mxFitFunctionML(vector = FALSE)

Arguments

`A`	A character string indicating the name of the 'A' matrix.
`S`	A character string indicating the name of the 'S' matrix.
`F`	A character string indicating the name of the 'F' matrix.
`M`	An optional character string indicating the name of the 'M' matrix.
`dimnames`	An optional character vector to be assigned to the column names of the 'F' and 'M' matrices.
`thresholds`	An optional character string indicating the name of the thresholds matrix.
`vector`	A logical value indicating whether the objective function result is the likelihood vector.
`threshnames`	An optional character vector to be assigned to the column names of the thresholds matrix.

Details

NOTE: THIS DESCRIPTION IS DEPRECATED. Please change to using mxExpectationRAM and mxFitFunctionML as shown in the example below.

Objective functions were functions for which free parameter values are chosen such that the value of the objective function was minimized. The mxRAMObjective provided maximum likelihood estimates of free parameters in a model of the covariance of a given MxData object. This model is defined by reticular action modeling (McArdle and McDonald, 1984). The 'A', 'S', and 'F' arguments must refer to MxMatrix objects with the associated properties of the A, S, and F matrices in the RAM modeling approach.

The 'dimnames' arguments takes an optional character vector. If this argument is not a single NA, then this vector be assigned to be the column names of the 'F' matrix and optionally to the 'M' matrix, if the 'M' matrix exists.

The 'A' argument refers to the A or asymmetric matrix in the RAM approach. This matrix consists of all of the asymmetric paths (one-headed arrows) in the model. A free parameter in any row and column describes a regression of the variable represented by that row regressed on the variable represented in that column.

The 'S' argument refers to the S or symmetric matrix in the RAM approach, and as such must be square. This matrix consists of all of the symmetric paths (two-headed arrows) in the model. A free parameter in any row and column describes a covariance between the variable represented by that row and the variable represented by that column. Variances are covariances between any variable at itself, which occur on the diagonal of the specified matrix.

The 'F' argument refers to the F or filter matrix in the RAM approach. If no latent variables are included in the model (i.e., the A and S matrices are of both of the same dimension as the data matrix), then the 'F' should refer to an identity matrix. If latent variables are included (i.e., the A and S matrices are not of the same dimension as the data matrix), then the 'F' argument should consist of a horizontal adhesion of an identity matrix and a matrix of zeros.

The 'M' argument refers to the M or means matrix in the RAM approach. It is a 1 x n matrix, where n is the number of manifest variables + the number of latent variables. The M matrix must be specified if either the mxData type is "cov" or "cor" and a means vector is provided, or if the mxData type is "raw". Otherwise the M matrix is ignored.

The MxMatrix objects included as arguments may be of any type, but should have the properties described above. The mxRAMObjective will not return an error for incorrect specification, but incorrect specification will likely lead to estimation problems or errors in the mxRun function.

mxRAMObjective evaluates with respect to an MxData object. The MxData object need not be referenced in the mxRAMObjective function, but must be included in the MxModel object. mxRAMObjective requires that the 'type' argument in the associated MxData object be equal to 'cov' or 'cor'.

To evaluate, place MxRAMObjective objects, the mxData object for which the expected covariance approximates, referenced MxAlgebra and MxMatrix objects, and optional MxBounds and MxConstraint objects in an MxModel object. This model may then be evaluated using the mxRun function. The results of the optimization can be found in the 'output' slot of the resulting model, and may be obtained using the mxEval function..

Value

Returns a list containing an MxExpectationRAM object and an MxFitFunctionML object.

References

McArdle, J. J. and MacDonald, R. P. (1984). Some algebraic properties of the Reticular Action Model for moment structures. *British Journal of Mathematical and Statistical Psychology, 37,* 234-251.

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
# Create and fit a model using mxMatrix, mxAlgebra,
#  mxExpectationNormal, and mxFitFunctionML
```

```
library(OpenMx)

# Simulate some data

x=rnorm(1000, mean=0, sd=1)
y= 0.5*x + rnorm(1000, mean=0, sd=1)
tmpFrame <- data.frame(x, y)
tmpNames <- names(tmpFrame)

# Define the matrices

matrixS <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(1,0,0,1),
              free=c(TRUE,FALSE,FALSE,TRUE), labels=c("Vx", NA, NA, "Vy"),
              name = "S")
matrixA <- mxMatrix(type = "Full", nrow = 2, ncol = 2, values=c(0,1,0,0),
              free=c(FALSE,TRUE,FALSE,FALSE), labels=c(NA, "b", NA, NA),
              name = "A")
matrixF <- mxMatrix(type="Iden", nrow=2, ncol=2, name="F")
matrixM <- mxMatrix(type = "Full", nrow = 1, ncol = 2, values=c(0,0),
              free=c(TRUE,TRUE), labels=c("Mx", "My"), name = "M")

# Define the expectation

expFunction <- mxExpectationRAM(M="M", dimnames = tmpNames)

# Choose a fit function

fitFunction <- mxFitFunctionML()

# Define the model

tmpModel <- mxModel(model="exampleRAMModel",
                    matrixA, matrixS, matrixF, matrixM,
                    expFunction, fitFunction,
                    mxData(observed=cov(tmpFrame), type="cov", numObs=nrow(tmpFrame),
                              means = colMeans(tmpFrame)))

# Fit the model and print a summary

tmpModelOut <- mxRun(tmpModel)
summary(tmpModelOut)
```

mxRename *Rename a model or submodel*

Description

This function re-names a model. By default, the top model will be renamed. To rename a specific model, set oldname (see examples). Importantly, all internal references to the old model name (e.g. in algebras) will be updated to reference the new name.

Usage

```
mxRename(model, newname, oldname = NA)
```

Arguments

`model`	a MxModel object.
`newname`	the new name of the model.
`oldname`	the name of the target model to rename. If NA then rename top model.

Value

Return a mxModel object with the target model renamed.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
library(OpenMx)

# Create a parent model with two submodels:
modelC <- mxModel(model= 'modelC',
mxModel(model= 'modelA'),
mxModel(model= 'modelB')
)

# Rename modelC (the top model) to "model1"
model1 <- mxRename(modelC, 'model_1')

# Rename submodel "modelB" to "model_2"
model1 <- mxRename(model1, oldname = 'modelB', newname = 'model_2')

model1
```

`mxRestore` *Restore model state from a checkpoint file*

Description

Restore model state from a checkpoint file

Usage

```
mxRestore(
  model,
  chkpt.directory = mxOption(model, "Checkpoint directory"),
  chkpt.prefix = mxOption(model, "Checkpoint Prefix"),
  line = NULL,
  strict = FALSE
)

mxRestoreFromDataFrame(model, checkpoint, line = NULL)
```

Arguments

`model`	an MxModel object
`chkpt.directory`	character. Directory where the checkpoint file is located
`chkpt.prefix`	character. Prefix of the checkpoint file
`line`	integer. Which line from the checkpoint file to restore (defaults to the last line)
`strict`	logical. Require that the checkpoint name and model name match
`checkpoint`	a data.frame containing the model state

Details

In general, the arguments 'chkpt.directory' and 'chkpt.prefix' should be identical to the `mxOption`: 'Checkpoint Directory' and 'Checkpoint Prefix' that were specified on the model before execution.

Alternatively, the checkpoint file can be manually loaded as a data.frame in R and passed to `mxRestoreFromDataFrame`. Use `read.table` with the options `header=TRUE,sep="\t",stringsAsFactors=FALSE,check`

Value

Returns an MxModel object with free parameters updated to the last saved values. When 'line' is provided, the MxModel is updated to the values on that line within the checkpoint file.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation`

See Also

Other model state: `mxComputeCheckpoint()`, `mxSave()`

Examples

```
library(OpenMx)

# Simulate some data

x=rnorm(1000, mean=0, sd=1)
```

```
y= 0.5*x + rnorm(1000, mean=0, sd=1)
tmpFrame <- data.frame(x, y)
tmpNames <- names(tmpFrame)

dir <- tempdir()  # safe place to create files
mxOption(key="Checkpoint Directory", value=dir)

# Create a model that includes an expected covariance matrix,
# an expectation function, a fit function, and an observed covariance matrix

data <- mxData(cov(tmpFrame), type="cov", numObs = 1000)
expCov <- mxMatrix(type="Symm", nrow=2, ncol=2, values=c(.2,.1,.2), free=TRUE, name="expCov")
expFunction <- mxExpectationNormal(covariance="expCov", dimnames=tmpNames)
fitFunction <- mxFitFunctionML()
testModel <- mxModel(model="testModel", expCov, data, expFunction, fitFunction)

#Use mxRun to optimize the free parameters in the expected covariance matrix
modelOut <- mxRun(testModel, checkpoint = TRUE)
modelOut$expCov

#Use mxRestore to load the last checkpoint saved state of the model
modelRestore <- mxRestore(testModel)
modelRestore$expCov
```

---

mxRetro — *Return random classic Mx error message*

Description

This function allows you to obtain a classic Mx error message. The message returned is random.

Usage

```
mxRetro()
```

Details

If you're a nostalgic old sod and you miss the warm, fuzzy feelings you got from reading one of Mike Neale's patented error messages, then this function is here to save you from the depths of dire depression. All credit for this function is due to Sarah Medland, but of course the wisdom is from Mike Neale.

References

- https://en.wikipedia.org/wiki/OpenMx

See Also

- omxBrownie

Examples

```
require(OpenMx)
mxRetro()
```

`mxRObjective` *DEPRECATED: Create MxRObjective Object*

Description

WARNING: Objective functions have been deprecated as of OpenMx 2.0.

Please use mxFitFunctionR() instead. As a temporary workaround, mxRObjective returns a list containing a NULL MxExpectation object and an MxFitFunctionR object.

All occurrences of

mxRObjective(fitfun, ...)

Should be changed to

mxFitFunctionR(fitfun, ...)

Arguments

`objfun`	A function that accepts two arguments.
`...`	The initial state information to the objective function.

Details

NOTE: THIS DESCRIPTION IS DEPRECATED. Please change to using mxExpectationNormal and mxFitFunctionML as shown in the example below.

The fitfun argument must be a function that accepts two arguments. The first argument is the mxModel that should be evaluated, and the second argument is some persistent state information that can be stored between one iteration of optimization to the next iteration. It is valid for the function to simply ignore the second argument.

The function must return either a single numeric value, or a list of exactly two elements. If the function returns a list, the first argument must be a single numeric value and the second element will be the new persistent state information to be passed into this function at the next iteration. The single numeric value will be used by the optimizer to perform optimization.

The initial default value for the persistent state information is NA.

Throwing an exception (via stop) from inside fitfun may result in unpredictable behavior. You may want to wrap your code in tryCatch while experimenting.

Value

Returns a list containing a NULL mxExpectation object and an MxFitFunctionR object.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
# Create and fit a model using mxFitFunctionR

library(OpenMx)

A <- mxMatrix(nrow = 2, ncol = 2, values = c(1:4), free = TRUE, name = 'A')
squared <- function(x) { x ^ 2 }

# Define the objective function in R

objFunction <- function(model, state) {
    values <- model$A$values
    return(squared(values[1,1] - 4) + squared(values[1,2] - 3) +
        squared(values[2,1] - 2) + squared(values[2,2] - 1))
}

# Define the expectation function

fitFunction <- mxFitFunctionR(objFunction)

# Define the model

tmpModel <- mxModel(model="exampleModel", A, fitFunction)

# Fit the model and print a summary

tmpModelOut <- mxRun(tmpModel)
summary(tmpModelOut)
```

`mxRowObjective` *DEPRECATED: Create MxRowObjective Object*

Description

WARNING: Objective functions have been deprecated as of OpenMx 2.0.

Please use mxFitFunctionRow() instead. As a temporary workaround, mxRowObjective returns a list containing a NULL MxExpectation object and an MxFitFunctionRow object.

All occurrences of

mxRowObjective(rowAlgebra, reduceAlgebra, dimnames, rowResults = "rowResults", filteredDataRow = "filteredDataRow", existenceVector = "existenceVector")

Should be changed to

mxFitFunctionRow(rowAlgebra, reduceAlgebra, dimnames, rowResults = "rowResults", filteredDataRow = "filteredDataRow", existenceVector = "existenceVector")

Arguments

`rowAlgebra`	A character string indicating the name of the algebra to be evaluated row-wise.
`reduceAlgebra`	A character string indicating the name of the algebra that collapses the row results into a single number which is then optimized.
`dimnames`	A character vector of names corresponding to columns be extracted from the data set.
`rowResults`	The name of the auto-generated "rowResults" matrix. See details.
`filteredDataRow`	The name of the auto-generated "filteredDataRow" matrix. See details.
`existenceVector`	The name of the auto-generated "existenceVector" matrix. See details.

Details

Objective functions are functions for which free parameter values are chosen such that the value of the objective function is minimized. The mxRowObjective function evaluates a user-defined MxAlgebra object called the ‘rowAlgebra’ in a row-wise fashion. It then stores results of the row-wise evaluation in another MxAlgebra object called the ‘rowResults’. Finally, the mxRowObjective function collapses the row results into a single number which is then used for optimization. The MxAlgebra object named by the ‘reduceAlgebra’ collapses the row results into a single number.

The ‘filteredDataRow’ is populated in a row-by-row fashion with all the non-missing data from the current row. You cannot assume that the length of the filteredDataRow matrix remains constant (unless you have no missing data). The ‘existenceVector’ is populated in a row-by-row fashion with a value of 1.0 in column j if a non-missing value is present in the data set in column j, and a value of 0.0 otherwise. Use the functions omxSelectRows, omxSelectCols, and omxSelectRowsAndCols to shrink other matrices so that their dimensions will be conformable to the size of ‘filteredDataRow’.

Value

Please use mxFitFunctionRow() instead. As a temporary workaround, mxRowObjective returns a list containing a NULL MxExpectation object and an MxFitFunctionRow object.

References

The OpenMx User’s guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
# Model that adds two data columns row-wise, then sums that column
# Notice no optimization is performed here.

library(OpenMx)

xdat <- data.frame(a=rnorm(10), b=1:10) # Make data set
amod <- mxModel(model="example1",
            mxData(observed=xdat, type='raw'),
            mxAlgebra(sum(filteredDataRow), name = 'rowAlgebra'),
            mxAlgebra(sum(rowResults), name = 'reduceAlgebra'),
            mxFitFunctionRow(
                rowAlgebra='rowAlgebra',
                reduceAlgebra='reduceAlgebra',
                dimnames=c('a','b'))
)
amodOut <- mxRun(amod)
mxEval(rowResults, model=amodOut)
mxEval(reduceAlgebra, model=amodOut)

# Model that find the parameter that minimizes the sum of the
#  squared difference between the parameter and a data row.

bmod <- mxModel(model="example2",
            mxData(observed=xdat, type='raw'),
            mxMatrix(values=.75, ncol=1, nrow=1, free=TRUE, name='B'),
            mxAlgebra((filteredDataRow - B) ^ 2, name='rowAlgebra'),
            mxAlgebra(sum(rowResults), name='reduceAlgebra'),
            mxFitFunctionRow(
                rowAlgebra='rowAlgebra',
                reduceAlgebra='reduceAlgebra',
                dimnames=c('a'))
)
bmodOut <- mxRun(bmod)
mxEval(B, model=bmodOut)
mxEval(reduceAlgebra, model=bmodOut)
mxEval(rowResults, model=bmodOut)
```

---

mxRun *Run an OpenMx model*

Description

This function sends 'model' to the optimizer, and returns the optimized model if it ran without error.

If `intervals` = `TRUE`, then confidence intervals will be computed on any `mxCI`s added to the model.

During a run, 'mxRun' will print the context (e.g. optimizer name or step in the analysis e.g. MxComputeNumericDeriv), followed by the current evaluation count, fit value, and the change in fit compared to the last status report. e.g.:

`MxComputeGradientDescent(CSOLNP) evaluations 1258 fit 37702.6 change -0.05861`

This may be followed by progress on the numeric derivative

`MxComputeNumericDeriv 313/528`

note: For models that prove difficult to run, you might try using `mxTryHard` in place of mxRun.

Usage

```
mxRun(model, ..., intervals = NULL, silent = FALSE, suppressWarnings = FALSE,
    unsafe = FALSE, checkpoint = FALSE, useSocket = FALSE, onlyFrontend = FALSE,
    useOptimizer = TRUE, beginMessage=!silent)
```

Arguments

`model`	A MxModel object to be optimized.
`...`	Not used. Forces remaining arguments to be specified by name.
`intervals`	A boolean indicating whether to compute the specified confidence intervals.
`silent`	A boolean indicating whether to print status to terminal.
`suppressWarnings`	A boolean indicating whether to suppress warnings.
`unsafe`	A boolean indicating whether to ignore errors.
`checkpoint`	A boolean indicating whether to periodically write parameter values to a file.
`useSocket`	A boolean indicating whether to periodically write parameter values to a socket.
`onlyFrontend`	A boolean indicating whether to run only front-end model transformations.
`useOptimizer`	A boolean indicating whether to run only the log-likelihood of the current free parameter values but not move any of the free parameters.
`beginMessage`	A boolean indicating whether to print the number of parameters before invoking the backend.

Details

The mxRun function is used to optimize free parameters in MxModel objects based on an expectation function and fit function. MxModel objects included in the mxRun function must include an appropriate expectation and fit functions.

If the 'silent' flag is TRUE, then model execution will not print any status messages to the terminal.

If the 'suppressWarnings' flag is TRUE, then model execution will not issue a warning if NPSOL returns a non-zero status code.

If the 'unsafe' flag is TRUE, then many error conditions will not be detected. Any error condition detected will be downgraded to warnings. It is strongly recommended to use this feature only for debugging purposes.

Free parameters are estimated or updated based on the expectation and fit functions. These estimated values, along with estimation information and model fit, can be found in the 'output' slot of MxModel objects after mxRun has been used.

If a model is dependent on or shares parameters with another model, both models must be included as arguments in another MxModel object. This top-level MxModel object must include expectation and fit functions in both submodels, as well as an additional fit function describing how the results of the first two should be combined (e.g. `mxFitFunctionMultigroup`).

Value

Returns an MxModel object with free parameters updated to their final values. The return value contains an "output" slot with the results of optimization.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation`.

See Also

mxTryHard for running models which prove difficult to optimize; summary to print a summary of a run model; mxModel for more on the model itself; More information about the OpenMx package may be found here.

Examples

```
# Create and run the 1-factor CFA on the openmx.ssri.psu.edu front page

# 1. Load OpenMx and the demoOneFactor dataframe

library(OpenMx)
data(demoOneFactor)

# 2. Define the manifests (5 demo variables) and latents for use in the model

manifests <- names(demoOneFactor)
latents   <- c("G")

# 3. Build the model, adding paths and data
model <- mxModel(model="One Factor", type="RAM",
    manifestVars = manifests,
    latentVars   = latents,
    mxPath(from=latents, to=manifests, labels=paste("b", 1:5, sep="")),
    mxPath(from=manifests, arrows=2, labels=paste("u", 1:5, sep="")),
    mxPath(from=latents  , arrows=2, free=FALSE, values=1.0),
    mxData(cov(demoOneFactor), type="cov", numObs=500)
)

# 4. Run the model, returning the result into model
model <- mxRun(model)

# 5. Show a summary of the fitted model and parameter values
summary(model)

# 6. Print SE-based CIs for the fitted parameter values

confint(model)

# 7. Add likelihood-based CIs to the model and run these

model = mxModel(model, mxCI(paste0("b", 1:5)))
model <- mxRun(model, intervals = TRUE)
summary(model)$CI

#       lbound  estimate    ubound note
# b1 0.3675940 0.3967545 0.4285895
# b2 0.4690663 0.5031569 0.5405838
# b3 0.5384588 0.5766635 0.6186705
# b4 0.6572678 0.7020702 0.7514609
# b5 0.7457231 0.7954529 0.8503486
# 8. Demonstrate mxTryHard

model <- mxTryHard(model, intervals = TRUE)
```

mxSave *Save model state to a checkpoint file*

Description

The function saves the last state of a model to a checkpoint file.

Usage

```
mxSave(model, chkpt.directory = ".", chkpt.prefix = "")
```

Arguments

`model`	an MxModel object
`chkpt.directory`	character. Directory where the checkpoint file is located
`chkpt.prefix`	character. Prefix of the checkpoint file

Details

In general, the arguments 'chkpt.directory' and 'chkpt.prefix' should be identical to the `mxOption`: 'Checkpoint Directory' and 'Checkpoint Prefix' that were specified on the model before execution.

Alternatively, the checkpoint file can be manually loaded as a data.frame in R. Use `read.table` with the options `header=TRUE,sep="\t",stringsAsFactors=FALSE,check.names=FALSE`.

Value

Returns a logical indicating the success of writing the checkpoint file to the checkpoint directory.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation

See Also

Other model state: `mxComputeCheckpoint()`, `mxRestore()`

Examples

```
library(OpenMx)

# Simulate some data

x=rnorm(1000, mean=0, sd=1)
y= 0.5*x + rnorm(1000, mean=0, sd=1)
tmpFrame <- data.frame(x, y)
tmpNames <- names(tmpFrame)

dir <- tempdir()  # safe place to create files
mxOption(key="Checkpoint Directory", value=dir)

# Create a model that includes an expected covariance matrix,
# an expectation function, a fit function, and an observed covariance matrix

data <- mxData(cov(tmpFrame), type="cov", numObs = 1000)
expCov <- mxMatrix(type="Symm", nrow=2, ncol=2, values=c(.2,.1,.2), free=TRUE, name="expCov")
expFunction <- mxExpectationNormal(covariance="expCov", dimnames=tmpNames)
fitFunction <- mxFitFunctionML()
testModel <- mxModel(model="testModel", expCov, data, expFunction, fitFunction)

#Use mxRun to optimize the free parameters in the expected covariance matrix
modelOut <- mxRun(testModel)
modelOut$expCov

# Save the ending state of modelOut in a checkpoint file
mxSave(modelOut)

# Restore the saved model from the checkpoint file
modelSaved <- mxRestore(testModel)
modelSaved$expCov
```

---

mxSE *Compute standard errors in OpenMx*

Description

This function allows you to obtain standard errors for arbitrary expressions, named entities, and algebras.

Usage

```
mxSE(
  x,
  model,
  details = FALSE,
  cov,
  forceName = FALSE,
  silent = FALSE,
  ...,
  defvar.row = as.integer(NA),
  data = "data"
)
```

Arguments

`x`	the parameter to get SEs on (reference or expression)
`model`	the `mxModel` to use.
`details`	logical. Whether to provide further details, e.g. the full sampling covariance matrix of x.
`cov`	optional matrix of covariances among the free parameters. If missing, the inverse Hessian from the fitted model is used.
`forceName`	logical; defaults to `FALSE`. Set to `TRUE` if `x` is an R symbol that refers to a character string.
`silent`	logical; defaults to `FALSE`. If `TRUE`, message-printing is suppressed.
`...`	further named arguments passed to `mxEval`
`defvar.row`	which row to load for any definition variables
`data`	name of data from which to load definition variables

Details

`x` can be the name of an algebra, a bracket address, named entity or arbitrary expression. When the `details` argument is TRUE, the full sampling covariance matrix of `x` is also returned as part of a list. The square root of the diagonals of this sampling covariance matrix are the standard errors.

When supplying the `cov` argument, take care that the free parameter covariance matrix is given, not the information matrix. These two are inverses of one another.

This function uses the delta method to compute the standard error of arbitrary and possibly nonlinear functions of the free parameters. The delta method makes a first-order Taylor approximation of the nonlinear function. The nonlinear function is a map from all the free parameters to some transformed subset of parameters: the linearization of this map is given by the Jacobian J. In equation form, the delta method computes standard errors by the following:

$$J^T C J$$

where J is the Jacobian of the nonlinear parameter transformation and C is the covariance matrix of the free parameters (e.g., two times the inverse of the Hessian of the minus two log likelihood function).

Value

SE value(s) returned as a matrix when `details` is FALSE. When `details` is TRUE, a list of the SE value(s) and the full sampling covariance matrix.

References

- https://en.wikipedia.org/wiki/Standard_error

See Also

- mxCI

Examples

```
library(OpenMx)
data(demoOneFactor)
# ===============================
# = Make and run a 1-factor CFA =
# ===============================

latents  = c("G") # the latent factor
manifests = names(demoOneFactor) # manifest variables to be modeled
# ===========================
# = Make and run the model! =
# ===========================
m1 <- mxModel("One Factor", type = "RAM",
manifestVars = manifests, latentVars = latents,
mxPath(from = latents, to = manifests, labels=paste0('lambda', 1:5)),
mxPath(from = manifests, arrows = 2),
mxPath(from = latents, arrows = 2, free = FALSE, values = 1),
mxData(cov(demoOneFactor), type = "cov", numObs = 500)
)
m1 = mxRun(m1)
mxSE(lambda5, model = m1)
mxSE(lambda1^2, model = m1)
```

---

mxSetDefaultOptions   *Reset global options to the default*

Description

Reset global options to the default

Usage

```
mxSetDefaultOptions()
```

`mxSimplify2Array` *Like simplify2array but works with vectors of different lengths*

Description

Vectors are filled column-by-column into a matrix. Shorter vectors are padded with NAs to fill whole columns.

Usage

```
mxSimplify2Array(x, higher = FALSE)
```

Arguments

`x`	a list of vectors
`higher`	whether to produce a higher rank array (defaults to FALSE)

Examples

```
v1 <- 1:3
v2 <- 4:5
v3 <- 6:10
mxSimplify2Array(list(v1,v2,v3))

#      [,1] [,2] [,3]
# [1,]    1    4    6
# [2,]    2    5    7
# [3,]    3   NA    8
# [4,]   NA   NA    9
# [5,]   NA   NA   10
```

`mxStandardizeRAMpaths` *Standardize RAM models' path coefficients*

Description

Provides a dataframe containing the standardized values of all nonzero path coefficients appearing in the `A` and `S` matrices of models that use RAM expectation (either of `type="RAM"` or containing an explicit `mxExpectationRAM()` statement). These standardized values are what the path coefficients would be if all variables in the analysis–both manifest and latent–were standardized to zero mean and unit variance. If the means are being modeled in addition to the covariance structure, then the dataframe will also contain values of the nonzero elements of the `M` matrix after they have been re-scaled to standard deviation units. Can optionally include asymptotic standard errors for the standardized and re-scaled coefficients, computed via the delta method. Not intended for use with models that contain definition variables.

Usage

```
mxStandardizeRAMpaths(model,SE=FALSE,cov=NULL)
```

Arguments

`model`	An `mxModel` object, that either uses RAM expectation or contains at least one submodel that does.
`SE`	Logical. Should standard errors be included with the standardized point estimates? Defaults to `FALSE`. Certain conditions are required for use of `SE=TRUE`; see "Details" below.
`cov`	A repeated-sampling covariance matrix for the free-parameter estimates–say, from the robust "sandwich estimator," or from bootstrapping–used to calculate SEs for the standardized path coefficients. Defaults to `NULL`, in which case twice the inverse of the Hessian matrix at the ML solution is used. See below for details concerning concerning cases when `model` contains independent RAM submodels.

Details

Matrix `A` contains the *A*symmetric paths, i.e. the single-headed arrows. Matrix `S` contains the *S*ymmetric paths, i.e. the double-headed arrows. The function will work even if `mxMatrix` objects named "A" and "S" are absent from the model, since it identifies which matrices in the model have been assigned the roles of `A` and `S` in the `mxExpectationRAM` statement. Note that, in models of `type="RAM"`, the necessary matrices and expectation statement are automatically assembled from the `mxPath` objects. If present, the `M` matrix will contain the means of exogenous variables and the intercepts of endogenous variables.

If `model` contains any submodels with `independent=TRUE` that use RAM expectation, `mxStandardizeRAMpaths()` automatically applies itself recursively over those submodels. However, if a non-`NULL` matrix has been supplied for argument `cov`, that matrix is only used for the "container" model, and is not passed as argument to the recursive calls of the function. To provide a covariance matrix for calculating SEs in an independent submodel, use `mxStandardizeRAMpaths()` directly on that submodel.

Use of `SE=TRUE` requires that package `numDeriv` be installed. It also requires that `model` contain no `mxConstraint` statements. Finally, if `cov=NULL`, it requires `model` to have a nonempty `hessian` element in its output slot. There are three common reasons why the latter condition may not be met. First, the model may not have been run yet, i.e. it was not output by `mxRun()`. Second, `mxOption` `"Hessian"` might be set to `"No"`. Third, computing the Hessian matrix might possibly have been skipped per a user-defined `mxCompute*` statement (if any are present in the model). If `model` contains RAM-expectation submodels with `independent=TRUE`, these conditions are checked separately for each such submodel.

In any event, using these standard errors for hypothesis-testing or forming confidence intervals is *not* generally advised. Instead, it is considered best practice to conduct likelihood-ratio tests or compute likelihood-based confidence intervals (from `mxCI()`), as in examples below.

The user should note that `mxStandardizeRAMpaths()` only cares whether an element of `A` or `S` (or `M`) is nonzero, and not whether it is a fixed or free parameter. So, for instance, if the function is used on a model not yet run, any free parameters in `A` or `S` initialized at zero will not appear in the function's output.

The user is warned to interpret the output of `mxStandardizeRAMpaths()` cautiously if any elements of `A` or `S` depend upon "definition variables" (you have definition variables in your model if the `labels` of any `MxPath` or `MxMatrix` begin with `"data."`). Typically, either `mxStandardizeRAMpaths()`'s results will be valid only for the first row of the raw dataset (and any rows identical to it), or some of the standardized coefficients will be incorrectly reported as zero.

Value

If argument `model` is a single-group model that uses RAM expecation, then `mxStandardizeRAMpaths()` returns a dataframe, with one row for each nonzero path coefficient in `A` and `S` (and `M`, if present), and with the following columns:

`name`	Character strings that uniquely identify each nonzero path coefficient in terms of the model name, the matrix ("A", "S", or "M"), the row number, and the column number.
`label`	Character labels for those path coefficients that are labeled elements of an `mxMatrix` object, and `NA` for those that are not. Note that path coefficients having the same label (and therefore the same UNstandardized value) can have different standardized values, and therefore the same label may appear more than once in this dataframe.
`matrix`	Character strings of "A", "S", or "M", depending on which matrix contains the given path coefficient.
`row`	Character. The rownames of the matrix containing each path coefficient; row numbers are used instead if the matrix has no rownames.
`col`	Character. The colnames of the matrix containing each path coefficient; column numbers are used instead if the matrix has no colnames.
`Raw.Value`	Numeric values of the raw (i.e., UNstandardized) path coefficients.
`Raw.SE`	Numeric values of the asymptotic standard errors of the raw path coefficients if if `SE=TRUE`, or "not_requested" otherwise.
`Std.Value`	Numeric values of the standardized path coefficients.
`Std.SE`	Numeric values of the asymptotic standard errors of the standardized path coefficients if `SE=TRUE`, or "not_requested" otherwise.

If `model` is a multi-group model containing at least one submodel with RAM expectation, then `mxStandardizeRAMpaths()` returns a list. The list has a number of elements equal to the number of submodels that either have RAM expectation or contain a submodel that does. List elements corresponding to RAM-expectation submodels contain a dataframe, as described above. List elements corresponding to "container" submodels are themselves lists, of the kind described here.

See Also

`mxBootstrapStdizeRAMpaths()`

Examples

```
library(OpenMx)
data(demoOneFactor)
manifests <- names(demoOneFactor)
```

```
latents   <- c("G")
factorModel <- mxModel(model="One Factor", type="RAM",
      manifestVars = manifests,
      latentVars   = latents,
      mxPath(from=latents, to=manifests),
      mxPath(from=manifests, arrows=2, values=0.1),
      mxPath(from=latents, arrows=2,free=FALSE, values=1.0),
      mxData(cov(demoOneFactor), type="cov",numObs=500)
)
factorFit <-mxRun(factorModel)
summary(factorFit)$parameters
mxStandardizeRAMpaths(model=factorFit,SE=FALSE)

## Likelihood ratio test of variable x1's factor loading:
factorModelNull <- omxSetParameters(factorModel,labels="One Factor.A[1,6]",
                    values=0,free=FALSE)
factorFitNull <- mxRun(factorModelNull)
mxCompare(factorFit,factorFitNull)[2,"p"] #<--p-value

## Confidence intervals for all standardized paths:
factorModel2 <- mxModel(model=factorModel,
                        mxMatrix(type="Iden",nrow=nrow(factorModel$A),name="I"),
                 mxAlgebra( vec2diag(diag2vec( solve(I-A)%*%S%*%t(solve(I-A)) )%^%-0.5) ,
                                   name="InvSD"),
                        mxAlgebra( InvSD %*% A %*% solve(InvSD),
                                   name="Az",dimnames=dimnames(factorModel$A)),
                        mxAlgebra( InvSD %*% S %*% InvSD,
                                   name="Sz",dimnames=dimnames(factorModel$S)),
                        mxCI(c("Az","Sz"))
)

factorFit2 <- mxRun(factorModel2,intervals=TRUE)
## Contains point values and confidence limits for all paths:
summary(factorFit2)$CI
```

`mxThreshold` *Create List of Thresholds*

Description

This function creates a list of thresholds which mxModel can use to set up a thresholds matrix for a RAM model.

Usage

```
mxThreshold(vars, nThresh=NA,
free=FALSE, values=mxNormalQuantiles(nThresh), labels=NA,
lbound=NA, ubound=NA)
```

Arguments

`vars`	character vector. These are the variables for which thresholds are to be specified.
`nThresh`	numeric vector. These are the number of thresholds for each variables listed in 'vars'.
`free`	boolean vector. Indicates whether threshold parameters are free or fixed.
`values`	numeric vector. The starting values of the parameters.
`labels`	character vector. The names of the parameters.
`lbound`	numeric vector. The lower bounds of free parameters.
`ubound`	numeric vector. The upper bounds of free parameters.

Details

If you are new to ordinal data modeling and just want something quick to make your ordinal data work, we recommend you try the `umxThresholdMatrix` function in the `umx` package.

The mxPath function creates MxThreshold objects. These consist of a list of ordinal variables and the thresholds that define the relationship between the observed ordinal variable and the continuous latent variable assumed to underlie it. This function directly mirrors the usage of mxPath, but is used to specify thresholds rather than means, variances and bivariate relationships.

The 'vars' argument specifies which variables you wish to specify thresholds for. Variables are referenced by name, and these names must appear in the 'manifestVar' argument of the mxModel function if thresholds are to be correctly processed. Additionally, variables for which thresholds are specified must be specified as ordinal factors in whatever data is included in the model.

The 'nThresh' argument specifies how many thresholds are to be specified for the variable or variables included in the 'vars' argument. The number of thresholds for a particular variable should be one fewer than the number of categories specified for that variable.

The 'free' argument specifies whether the thresholds created by the mxThreshold function are free or fixed parameters. This argument may take either TRUE for free parameters, FALSE for fixed parameters, or a vector of TRUEs and FALSEs to be applied in order to the created thresholds.

'values' is a numeric vector containing the starting values of the created thresholds. This gives a starting point for estimation. The 'labels' argument specifies the names of the parameters in the resulting MxThreshold object. The 'lbound' and 'ubound' arguments specify lower and upper bounds for the created threshold parameters.

Thresholds for multiple variables may be specified simultaneously by including a vector of variable names to the 'vars' argument. When multiple variables are included in the 'vars' argument, the length of the 'vars' argument must be evenly divisable by the length of the 'nThresh' argument. All subsequent arguments ('free' through 'ubound') should have their lengths be a factor of the total number of thresholds specified for all variables.

If four variables are included in the 'vars' argument, then the 'nThresh' argument should contain either one, two or four elements. If the 'nThresh' argument specifies two thresholds for each variable, then 'free', 'values', and all subsequent arguments should specify eight values by including one, two, four or eight elements. Whenever fewer values are specified than are required (e.g., specify two values for eight thresholds), then the entire vector of values is repeated until the required number of values is reached, and will return an error if the correct number of values cannot be achieved by repeating the entire vector.

Value

Returns a list of thresholds.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

umxThresholdMatrix demo("mxThreshold") mxPath for comparable specification of paths. mxMatrix for a matrix-based approach to thresholds specification; mxModel for the container in which mxThresholds are embedded. More information about the OpenMx package may be found here.

Examples

```
library(OpenMx)
# threshold objects for three variables: 2 binary, and one ordinal.
mxThreshold(vars = c("z1", "z2", "z3"), nThresh = c(1,1,2),
  free = TRUE, values = c(-1, 0, -.5, 1.2) )
```

---

mxTryHard *Make multiple attempts to run a model*

Description

Makes multiple attempts to fit an MxModel object with `mxRun()` until the optimizer yields an acceptable solution or the maximum number of attempts (set by `extraTries`) is reached. Between attempts, start values are perturbed by random numbers (see details). Optimization-control parameters may also be altered. From among its attempts, the function returns the fitted model with the best fit (smallest fit-function value). If `bestInitsOutput` is TRUE, then in addition the start values used for the best-fitting model will be printed to the console.

note: If the model contains `mxConstraint`s, or if `mxFitFunctionWLS` is being used, the Hessian cannot be checked, and so `checkHess` will be coerced to FALSE.

Usage

```
mxTryHard(model, extraTries = 10, greenOK = FALSE, loc = 1, scale = 0.25,
initialGradientStepSize = imxAutoOptionValue("Gradient step size"),
initialGradientIterations = imxAutoOptionValue('Gradient iterations'),
initialTolerance=as.numeric(mxOption(NULL,'Optimality tolerance')),
checkHess = TRUE, fit2beat = Inf, paste = TRUE,iterationSummary=FALSE,
bestInitsOutput=TRUE, showInits=FALSE, verbose=0, intervals = FALSE,
finetuneGradient=TRUE, jitterDistrib=c("runif","rnorm","rcauchy"), exhaustive=FALSE,
maxMajorIter=3000, OKstatuscodes, wtgcsv=c("prev","best","initial"), silent=interactive())

mxTryHardOrig(model, finetuneGradient=FALSE, maxMajorIter=NA,
wtgcsv=c("prev","best"), silent=FALSE, ...)
```

```
mxTryHardctsem(model, initialGradientStepSize = .00001,
initialGradientIterations = 1,
initialTolerance=1e-12,jitterDistrib="rnorm", ...)

mxTryHardWideSearch(model, finetuneGradient=FALSE, jitterDistrib="rcauchy",
exhaustive=TRUE, wtgcsv="prev", ...)

mxTryHardOrdinal(model, greenOK = TRUE,checkHess = FALSE,
finetuneGradient=FALSE, exhaustive=TRUE,
OKstatuscodes=c(0,1,5,6), wtgcsv=c("prev","best"), ...)
```

Arguments

`model`	The MxModel to be run.
`extraTries`	The number of attempts to run the model *in addition to the first*. In effect, is the maximum number of attempts `mxTryHard()` will make, since the function will stop once an acceptable solution is reached. Defaults to 10 (for `mxTryHard()`), in which case a maximum of 11 total attempts will be made.
`greenOK`	Logical; is a solution with Mx status GREEN (optimizer status code 1) acceptable? Defaults to `FALSE` (for `mxTryHard()`). Ignored if a value is provided for `OKstatuscodes`.
`loc, scale`	Numeric. The location and scale parameters of the distribution from which random values are drawn to perturb start values between attempts, defaulting respectively to 1 and 0.25. See below, under "Details," for additional information.
`initialGradientStepSize, initialGradientIterations, initialTolerance`	Numeric. Initial values of optimization-control parameters passed to `mxComputeGradientDescent()` if `model` is using the default compute plan.
`checkHess`	Logical; is a positive-definite Hessian a requirement for an acceptable solution? Defaults to `TRUE` (for `mxTryHard()`). If `TRUE`, the Hessian and standard errors are calculated with each fit attempt, irrespective of the value of relevant options. The exception is if `model` or any of its submodels contains `MxConstraint`s, in which case `checkHess` is coerced to `FALSE`.
`fit2beat`	Numeric upper limit to the fitfunction value that an acceptable solution may have. Useful if a nested submodel of `model` has already been fitted, since `model`, with its additional free parameters, should not yield a fitfunction value any greater than that of the submodel.
`paste`	Logical. If `TRUE` (default), start values for the returned fitted model are printed to console as a comma-separated string. This is useful if the user wants to copy-paste these values into an R script, say, in an `omxSetParameters()` statement. If `FALSE`, the vector of start values is printed as-is. Note that this vector, from `omxGetParameters()`, has names corresponding to the free parameters; these names are not displayed when `paste=TRUE`.
`iterationSummary`	Logical. If `TRUE`, displays parameter estimates and fit values for every fit attempt, even if `silent=TRUE`. Defaults to `FALSE`.

`bestInitsOutput`	Logical. If `TRUE` and if `silent=FALSE`, `mxTryHard()` displays the starting values that resulted in the best fit, according to format specified by `paste` argument. Defaults to `TRUE`.
`showInits`	Logical. If `TRUE`, displays starting values for every fit attempt, even if `silent=TRUE`. Defaults to `FALSE`.
`verbose`	If `model` is using the default compute plan, is passed to `mxComputeGradientDescent()` to specify level of output printed to console during optimization.
`intervals`	Logical. If TRUE, OpenMx will estimate any specified confidence intervals.
`finetuneGradient`	Logical. If `TRUE` (default for `mxTryHard()`), then as repeated fit attempts appear to be improving, `mxTryHard()` will adjust optimization-control parameters `gradientStepSize`, `gradientIterations`, and `tolerance`, as well as argument `scale`, to "fine-tune" its convergence toward an optimal solution. `finetuneGradient=FALSE` is recommended for analyses involving thresholds.
`jitterDistrib`	Character string naming which random-number distribution–either uniform (rectangular), normal (Gaussian), or Cauchy–to be used to perturb start values. Defaults to the uniform distribution (for `mxTryHard()`). See below, under "Details," for additional information.
`exhaustive`	Logical. If `FALSE` (default for `mxTryHard()`), `mxTryHard()` stops making additional attempts once it reaches an acceptable solution. If `TRUE`, the function instead continues until it reaches its maximum number of attempts (as per `extraTries`), and returns the best solution it found.
`maxMajorIter`	Integer; passed to `mxComputeGradientDescent()`. Defaults to 3000, which was the internally hard-coded value `mxTryHard()` used in a prior version of OpenMx. Value of `NA` is permitted, in which case `mxTryHard()` will calculate a value via the on-load default formula for the "Major iterations" option.
`OKstatuscodes`	Optional integer vector containing optimizer status codes that an acceptable solution is permitted to have. `mxTryHard()` always considers a status code of 0 to be acceptable, this argument notwithstanding. By default, `mxTryHard()` will consider status code 0 acceptable, and, if `greenOK=TRUE`, status code 1 as well. If a value is supplied for `OKstatuscodes` that conflicts with `greenOK`, `OKstatuscodes` controls.
`wtgcsv`	Character vector. "Where to get current start values." See below, under "Details," for additional information.
`silent`	Logical; for `mxTryHard()`, defaults to `TRUE` if running interactively, and to `FALSE` otherwise. If `TRUE`, persistent message-printing during execution of `mxTryHard()` is suppressed, and non-persistent printing is used instead. The two exceptions are the persistent printing requested by `TRUE` values of `iterationSummary` and `showInits`.
`...`	Additional arguments to be passed to `mxTryHard()`.

Details

`mxTryHardOrig()`, `mxTryHardctsem()`, `mxTryHardWideSearch()`, and `mxTryHardOrdinal()` are wrapper functions to the main workhorse function `mxTryHard()`. Each wrapper function has default values for certain arguments that are tailored toward a specific purpose. `mxTryHardOrig()`

imitates the functionality of the earliest implementations of `mxTryHard()` in OpenMx's history; its chief purpose is to find good start values that lead to an acceptable solution. `mxTryHardctsem()` uses `mxTryHard()` to "zero in" on an acceptable solution with models that can be difficult to optimize, such as continuous-time state-space models. `mxTryHardWideSearch()` uses `mxTryHard()` to search a wide region of the parameter space, in hope of avoiding local fitfunction minima. `mxTryHardOrdinal()` attempts to use `mxTryHard()` as well as it can be used with models involving ordinal data.

Argument `wtgcsv` dictates where `mxTryHard()` is permitted to find free-parameter values, at the start of each fit attempt after the first, before randomly perturbing them to create the current fit attempt's start values. If `"prev"` is included, then `mxTryHard()` is permitted to use the parameter estimates of the most recent non-error fit attempt. If `"best"` is included, then `mxTryHard()` is permitted to use the parameter estimates at the best solution so far. If `"initial"` is included, then `mxTryHard()` is permitted to use the initial start values in `model`, as provided by the user. The default is to permit all three, in which case `mxTryHard()` is written to use the best solution's values if available, and otherwise to use the most recent solution's values, but to periodically revert to the initial values if recent fit attempts have not improved on the best solution.

Once the start values are located for the current fit attempt, they are randomly perturbed before being assigned to the MxModel. The distributional family from which the perturbations are randomly generated is dictated by argument `jitterDistrib`. The distribution is parameterized by arguments `loc` and `scale`, respectively the location and scale parameters. The location parameter is the distribution's median. For the uniform distribution, `scale` is the absolute difference between its median and extrema (i.e., half the width of the rectangle); for the normal distribution, `scale` is its standard deviation; and for the Cauchy, `scale` is one-half its interquartile range. Start values are first multiplied by random draws from a distribution with the provided `loc` and `scale`, then added to random draws from a distribution with the same `scale` but with a median of zero.

Value

Usually, `mxTryHard()` returns a post-`mxRun()` `MxModel` object. Specifically, this will be the fitted model having the smallest fit-function value found by `mxTryHard()` during its attempts. The start values used to obtain this fitted model are printed to console if `bestInitsOutput=TRUE`.

If every attempt at running `model` fails, `mxTryHard()` returns an object of class 'try-error'.

`mxTryHard()` throws a warning if the returned `MxModel` object has a nonzero status code (unless nonzero status codes are considered acceptable per argument `greenOK` or `OKstatuscodes`).

See Also

`mxRun()`, `mxComputeTryHard`

Examples

```
library(OpenMx)

data(demoOneFactor)  # load the demoOneFactor dataframe

manifests <- names(demoOneFactor) # set the manifest to the 5 demo variables
latents   <- c("G")  # define 1 latent variable
model <- mxModel(model="One Factor", type="RAM",
    manifestVars = manifests,
    latentVars   = latents,
    mxPath(from=latents, to=manifests, labels=paste("b", 1:5, sep="")),
    mxPath(from=manifests, arrows=2, labels=paste("u", 1:5, sep="")),
    mxPath(from=latents  , arrows=2, free=FALSE, values=1.0),
    mxData(cov(demoOneFactor), type="cov", numObs=500)
)
model <- mxTryHard(model) # Run the model, returning the result into model
summary(model) # Show summary of the fitted model
```

mxTypes *List Currently Available Model Types*

Description

This function returns a vector of the currently available type names.

Usage

```
mxTypes()
```

Value

Returns a character vector of type names.

Examples

```
    mxTypes()
```

mxVersion *Returns Current Version String*

Description

This function returns a string with the current version number of OpenMx. Optionally (with verbose = TRUE (the default)), it prints a message containing the version of R, the platform, and the optimizer.

Usage

```
mxVersion(model = NULL, verbose = TRUE)
```

Arguments

`model`	optional `MxModel` to request optimizer from (default = NULL)
`verbose`	Whether to print version information to the console (default = TRUE)

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
# Print useful version information.
mxVersion()
# If you just want the version, use this call.
x = mxVersion(verbose=FALSE)

library(OpenMx)
data(demoOneFactor)  # load the demoOneFactor dataframe
manifests <- names(demoOneFactor) # set the manifest to the 5 demo variables
latents   <- c("G")  # define 1 latent variable
model <- mxModel(model = "One Factor", type = "RAM",
    manifestVars = manifests,
    latentVars   = latents,
    mxPath(from = latents, to = manifests, labels = paste("b", 1:5, sep = "")),
    mxPath(from = manifests, arrows = 2  , labels = paste("u", 1:5, sep = "")),
    mxPath(from = latents   , arrows = 2  , free = FALSE, values = 1.0),
    mxData(cov(demoOneFactor), type = "cov", numObs = 500)
)
mxVersion(model, verbose = TRUE)
```

`MxVersionType-class` *A package_version or character*

Description

A package_version or character

`myAutoregressiveData` *Example data with autoregressively related columns*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("myAutoregressiveData")
```

Format

A data frame with 100 observations on the following variables.

`x1` x variable and time 1

`x2` x variable and time 2

`x3` x variable and time 3

`x4` x variable and time 4

`x5` x variable and time 5

Details

The rows are independently and identically distributed, but the columns are and auto-correlation structure.

Source

Simulated.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
data(myAutoregressiveData)
round(cor(myAutoregressiveData), 2)
# note the sub-diagonal correlations (lag 1)
#  x1-x2, x2-x3, x3-x4, x4-x5
# and the second sub-diagonal correlations (lag 2)
#  x1-x3, x2-x4, x3-x5
```

`myFADataRaw` *Example 500-row dataset with 12 generated variables*

Description

Twelve columns of generated numeric data: x1 x2 x3 x4 x5 x6 y1 y2 y3 z1 z2 z3.

Usage

```
data(myFADataRaw)
```

Details

The x variables intercorrelate around .6 with each other.

The y variables intercorrelate around .5 with each other, and correlate around .3 with the X vars.

There are three ordinal variables, z1, z2, and z3.

The data are used in some OpenMx examples, especially confirmatory factor analysis.

There are no missing data.

Examples

```
data(myFADataRaw)
str(myFADataRaw)
```

`myGrowthKnownClassData` *Data for a growth mixture model with the true class membership*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("myGrowthKnownClassData")
```

Format

A data frame with 500 observations on the following variables.

`x1` x variable and time 1

`x2` x variable and time 2

`x3` x variable and time 3

`x4` x variable and time 4

`x5` x variable and time 5

`c` Known class membership variable

Details

The same as myGrowthMixtureData, but with the class membership variable.

Source

Simulated.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(myGrowthKnownClassData)

#plot the observed trajectories
# blue lines are class 1, green lines are class 2
colSel <-c('blue', 'green')[myGrowthKnownClassData$c]
matplot(t(myGrowthKnownClassData[,-6]), type='l', lty=1, col=colSel)
```

`myGrowthMixtureData` *Data for a growth mixture model*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("myGrowthMixtureData")
```

Format

A data frame with 500 observations on the following variables.

`x1`	x variable and time 1
`x2`	x variable and time 2
`x3`	x variable and time 3
`x4`	x variable and time 4
`x5`	x variable and time 5

Details

The same as myGrowthKnownClassData, but without the class membership variable.

Source

Simulated.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(myGrowthMixtureData)

matplot(t(myGrowthMixtureData), type='l', lty=1)

data(myGrowthKnownClassData)
all(myGrowthKnownClassData[,-6]==myGrowthMixtureData)
```

---

myLongitudinalData *Data for a linear latent growth curve model*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("myLongitudinalData")
```

Format

A data frame with 500 observations on the following variables.

x1	x variable and time 1
x2	x variable and time 2
x3	x variable and time 3
x4	x variable and time 4
x5	x variable and time 5

Details

Linear growth model with mean intercept around 10, and slope of about 1.5.

Source

Simulated.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
data(myLongitudinalData)

matplot(t(myLongitudinalData), type='l', lty=1)
```

`myRegData` *Example regression data with correlated predictors*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("myRegData")
```

Format

A data frame with 100 observations on the following variables.

- `w` Predictor variable
- `x` Predictor variable
- `y` Predictor variable
- `z` Outcome variable

Details

w, x, and y are predictors of z. x and y are correlated.

Source

Simulated.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
data(myRegData)
summary(lm(z ~ ., data=myRegData))
```

`myRegDataRaw` *Example regression data with correlated predictors*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("myRegDataRaw")
```

Format

A data frame with 100 observations on the following variables.

`w`	Predictor variable
`x`	Predictor variable
`y`	Predictor variable
`z`	Outcome variable

Details

w, x, and y are predictors of z. x and y are correlated. Equal to myRegData.

Source

Simulated.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(myRegData)
data(myRegDataRaw)

all(myRegDataRaw == myRegData)
```

`myTwinData` *Duplicate of twinData*

Description

Legacy dataset from early teaching examples. See `twinData` for a more current file.

Usage

```
data("myTwinData")
```

Format

A data frame with 3808 observations on the following variables.

`fam`	Family ID variable
`age`	Age of the twin pair. Range: 17 to 88.
`zyg`	Integer codes for zygosity and gender combinations
`part`	Cohort
`wt1`	Weight in kilograms for twin 1
`wt2`	Weight in kilograms for twin 2
`ht1`	Height in meters for twin 1
`ht2`	Height in meters for twin 2
`htwt1`	Product of ht and wt for twin 1
`htwt2`	Product of ht and wt for twin 2
`bmi1`	Body Mass Index for twin 1
`bmi2`	Body Mass Index for twin 2

Details

Height and weight are highly correlated, and each individually highly heritable. These data present and opportunity for multivariate behavior genetics modeling.

Source

Timothy Bates

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
data(myTwinData)

plot( ht1 ~ wt1, myTwinData)
```

mzfData *Example twin extended kinship data: MZ female twins*

Description

Data for extended twin example ETC88.R

Usage

```
data("mzfData")
```

Format

A data frame with 3099 observations on the following 37 variables.

`famid` a numeric vector

`e1` a numeric vector

`e2` a numeric vector

`e3` a numeric vector

`e4` a numeric vector

`e5` a numeric vector

`e6` a numeric vector

`e7` a numeric vector

`e8` a numeric vector

`e9` a numeric vector

`e10` a numeric vector

`e11` a numeric vector

`e12` a numeric vector

`e13` a numeric vector

`e14` a numeric vector

`e15` a numeric vector

`e16` a numeric vector

`e17` a numeric vector

`e18` a numeric vector

`a1` a numeric vector

`a2` a numeric vector

`a3` a numeric vector

`a4` a numeric vector

`a5` a numeric vector

`a6` a numeric vector

`a7` a numeric vector

`a8` a numeric vector

`a9` a numeric vector

`a10` a numeric vector

`a11` a numeric vector

`a12` a numeric vector

`a13` a numeric vector

`a14` a numeric vector

`a15` a numeric vector

`a16` a numeric vector

`a17` a numeric vector

`a18` a numeric vector

Examples

```
data(mzfData)
str(mzfData)
```

`mzmData` *Example twin extended kinship data: MZ Male data*

Description

Data for extended twin example ETC88.R

Usage

```
data("mzmData")
```

Format

A data frame with 3019 observations on the following 37 variables.

`famid` a numeric vector

`e1` a numeric vector

`e2` a numeric vector

`e3` a numeric vector

`e4` a numeric vector

`e5` a numeric vector

`e6` a numeric vector

`e7` a numeric vector

`e8` a numeric vector

`e9` a numeric vector

`e10` a numeric vector

`e11` a numeric vector

`e12` a numeric vector

`e13` a numeric vector

`e14` a numeric vector

`e15` a numeric vector

`e16` a numeric vector

`e17` a numeric vector

`e18` a numeric vector

`a1` a numeric vector

`a2` a numeric vector

`a3` a numeric vector

`a4` a numeric vector

`a5` a numeric vector

`a6` a numeric vector

`a7` a numeric vector

`a8` a numeric vector

`a9` a numeric vector

`a10` a numeric vector

`a11` a numeric vector

`a12` a numeric vector

`a13` a numeric vector

`a14` a numeric vector

`a15` a numeric vector

`a16` a numeric vector

`a17` a numeric vector

`a18` a numeric vector

Examples

```
data(mzmData)
str(mzmData)
```

Named-entity *Named Entities*

Description

A named entity is an S4 object that can be referenced by name.

Details

Every named entity is guaranteed to have a slot called "name". Within a model, the named entities of that model can be accessed using the $ operator. Access is limited to one nesting depth, such that if 'B' is a submodel of 'A', and 'C' is a matrix of 'B', then 'C' must be accessed using A$B$C.

The following S4 classes are named entities in the OpenMx library: MxAlgebra, MxConstraint, MxMatrix, MxModel, MxData, and MxObjective.

Examples

```
library(OpenMx)

# Create a model, add a matrix to it, and then access the matrix by name.

testModel <- mxModel(model="anEmptyModel")

testMatrix <- mxMatrix(type="Full", nrow=2, ncol=2, values=c(1,2,3,4), name="yourMatrix")

yourModel <- mxModel(testModel, testMatrix, name="noLongerEmpty")

yourModel$yourMatrix
```

nuclear_twin_design_data *Twin data from a nuclear family design*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("nuclear_twin_design_data")
```

Format

A data frame with 1743 observations on the following variables.

`Twin1`

`Twin2`

`Father`

`Mother`

`zyg` Zygosity of the twin pair

Details

This is a wide format data set. A single variable has values for different member of the same nuclear family.

Source

Likely simulated.

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
data(nuclear_twin_design_data)

cor(nuclear_twin_design_data[,-5], use="pairwise.complete.obs")
```

`numHess1` *numeric Hessian data 1*

Description

data file used by the HessianTest.R script

Usage

```
data("numHess1")
```

Format

A 12 by 12 data frame containing Hessian (numeric variables a-l)

Examples

```
data(numHess1)
str(numHess1)
```

`numHess2` *numeric Hessian data 2*

Description

data file used by the HessianTest.R script

Usage

```
data("numHess2")
```

Format

A 12 by 12 data frame containing Hessian matrix (numeric variables a-l)

Examples

```
data(numHess2)
str(numHess2)
```

`omxAllInt` *All Interval Multivariate Normal Integration*

Description

`omxAllInt` computes the probabilities of a large number of cells of a multivariate normal distribution that has been sliced by a varying number of thresholds in each dimension. While the same functionality can be achieved by repeated calls to `omxMnor`, `omxAllInt` is more efficient for repeated operations on a single covariance matrix. `omxAllInt` returns an nx1 matrix of probabilities cycling from lowest to highest thresholds in each column with the rightmost variable in *covariance* changing most rapidly.

Usage

```
omxAllInt(covariance, means, ...)
```

Arguments

`covariance`	the covariance matrix describing the multivariate normal distribution.
`means`	a row vector containing means of the variables of the underlying distribution.
`...`	a matrix or set of matrices containing one column of thresholds for each column of `covariance`. Each column must contain a strictly increasing set of thresholds for the corresponding variable of the underlying distribution. `NA` values in these thresholds indicate that the list of thresholds in that column has ended.

Details

covariance and *means* contain the covariances and means of the multivariate distribution from which probabilities are to be calculated.

covariance must be a square covariance or correlation matrix with one row and column for each variable.

means must be a vector of length `nrows(covariance)` that contains the mean for each corresponding variable.

All further arguments are considered threshold matrices.

Threshold matrices contain locations of the hyperplanes delineating the intervals to be calculated. The first column of the first matrix corresponds to the thresholds for the first variable represented by the covariance matrix. Subsequent columns of the same matrix correspond to thresholds for subsequent variables in the covariance matrix. If more variables exist in the covariance matrix than in the first threshold matrix, the first column of the second threshold matrix will be used, and so on. That is, if *covariance* is a 4x4 matrix, and the three threshold matrices are specified, one with a single column and the others with two columns each, the first column of the first matrix will contain thresholds for the first variable in *covariance*, the two columns of the second matrix will correspond to the second and third variables of *covariance*, respectively, and the first column of the third threshold matrix will correspond to the fourth variable. Any extra columns will be ignored.

Each column in the threshold matrices must contain some number of strictly increasing thresholds, delineating the boundaries of a cell of integration. That is, if the integral from -1 to 0 and 0 to 1 are required for a given variable, the corresponding threshold column should contain the values -1, 0, and 1, in that order. Thresholds may be set to Inf or -Inf if a boundary at positive or negative infinity is desired.

Within a threshold column, a value of +Inf, if it exists, is assumed to be the largest threshold, and any rows after it are ignored in that column. A value of NA, if it exists, indicates that there are no further thresholds in that column, and is otherwise ignored. A threshold column consisting of only +Inf or NA values will cause an error.

For all i>1, the value in row i must be strictly larger than the value in row i-1 in the same column.

The return value of `omxAllInt` is a matrix consisting of a single column with one row for each combination of threshold levels.

See Also

`omxMnor`

Examples

```
data(myFADataRaw)

covariance <- cov(myFADataRaw[,1:5])
means <- colMeans(myFADataRaw[,1:5])

# Integrate from -Infinity to 0 and 0 to 1 on first variable
thresholdForColumn1 <- cbind(c(-Inf, 0,   1))
# Note: The first variable will never be calculated from 1 to +Infinity.

# These columns will be integrated from -Inf to -1, -1 to 0, etc.
```

```
thresholdsForColumn2 <- cbind(c(-Inf, -1, 0, 1, Inf))
thresholdsForColumns3and4 <- cbind(c(-Inf, 1.96, 2.326, Inf),
                                   c(-Inf, -1.96, 2.326, Inf))

# The integration
omxAllInt(covariance, means,
  thresholdForColumn1, thresholdsForColumn2,
  thresholdsForColumns3and4, thresholdsForColumn2)
# Notice that columns 2 and 5 are assigned identical thresholds.

#-------------------------------------------------------------
# An alternative specification of the same calculation follows
covariance <- cov(myFADataRaw[,1:5])
means <- colMeans(myFADataRaw[,1:5])

# Note NAs to indicate the end of the sequence of thresholds.
thresholds <- cbind(c(-Inf,     0,     1,  NA,  NA),
                    c(-Inf,    -1,     0,   1, Inf),
                    c(-Inf,  1.96,  2.32, Inf,  NA),
                    c(-Inf, -1.96,  2.32, Inf,  NA),
                    c(-Inf,    -1,     0,   1, Inf))
omxAllInt(covariance, means, thresholds)
```

omxApply *On-Demand Parallel Apply*

Description

If the snowfall library is loaded, then this function calls `sfApply`. Otherwise it invokes `apply`.

Usage

```
omxApply(x, margin, fun, ...)
```

Arguments

`x`	a vector (atomic or list) or an expressions vector. Other objects (including classed objects) will be coerced by `as.list`.
`margin`	a vector giving the subscripts which the function will be applied over.
`fun`	the function to be applied to each element of `x`.
`...`	optional arguments to `fun`.

See Also

`omxLapply`, `omxSapply`

Examples

```
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
omxApply(x, 2, mean, trim = .2)
```

`omxAssignFirstParameters` *Assign First Available Values to Model Parameters*

Description

Sometimes you may have a free parameter with two different starting values in your model. OpenMx will not run a model until all instances of a free parameter have the same starting value. It is often sufficient to arbitrarily select one of those starting values for optimization.

This function accomplishes that task of assigning valid starting values to the free parameters of a model. It selects an arbitrary current value (the "first" value it finds, where "first" is not defined) for each free parameter and uses that value for all instances of that parameter in the model.

Usage

```
omxAssignFirstParameters(model, indep = FALSE)
```

Arguments

`model`	a MxModel object.
`indep`	assign parameters to independent submodels.

See Also

`omxGetParameters`, `omxSetParameters`

Examples

```
A     <- mxMatrix('Full', 3, 3, values = c(1:9), labels = c('a','b', NA),
                  free = TRUE, name = 'A')
model <- mxModel(model=A, name = 'model')
model <- omxAssignFirstParameters(model)

# Note: All cells with the same label now have the same start value.
# Note also that NAs are untouched.

model$matrices$A

# $labels
#      [,1] [,2] [,3]
# [1,] "a"  "a"  "a"
# [2,] "b"  "b"  "b"
# [3,] NA   NA   NA
#
# $values
#      [,1] [,2] [,3]
# [1,]    1    1    1
# [2,]    2    2    2
# [3,]    3    6    9
```

---

omxAugmentDataWithWLSSummary

Estimate summary statistics used by the WLS fit function

Description

The summary statistics are returned in the observedStats slot of the MxData object.

Usage

```
omxAugmentDataWithWLSSummary(
  mxd,
  type = c("WLS", "DWLS", "ULS"),
  allContinuousMethod = c("cumulants", "marginals"),
  ...,
  exogenous = c(),
  fullWeight = TRUE,
  returnModel = FALSE,
  silent = TRUE
)
```

Arguments

`mxd`	an MxData object containing raw data
`type`	the type of WLS weight matrix
`allContinuousMethod`	which method to use when all indicators are continuous
`...`	Not used. Forces remaining arguments to be specified by name.
`exogenous`	names variables to be modelled as exogenous
`fullWeight`	whether to produce a fullWeight matrix
`returnModel`	whether to return the whole mxModel (TRUE) or just the mxData (FALSE)
`silent`	logical. Whether to print status to terminal.

See Also

mxFitFunctionWLS

Examples

```
omxAugmentDataWithWLSSummary(mxData(Bollen[,1:8], 'raw'))
```

---

omxBrownie *Make Brownies in OpenMx*

Description

This function returns a brownie recipe.

Usage

```
omxBrownie(quantity=1, walnuts=TRUE, wfpb=FALSE)
```

Arguments

`quantity`	Number of batches of brownies desired. Defaults to one.
`walnuts`	Logical. Indicates whether walnuts are to be included in the brownies. Defaults to TRUE.
`wfpb`	Logical. Indicates whether to display the whole food plant based version. Defaults to FALSE.

Details

Returns a brownie recipe. Alter the ‘quantity‘ variable to make more pans of brownies. Ingredients, equipment and procedure are listed, but neither ingredients nor equipment are provided.

Raw cocoa powder can be used instead of Dutch processed cocoa for approximately double the antioxidants and flavonols. However, raw cocoa powder is not as smooth and delicious in taste.

For the whole food plant based (wfpb) version of the recipe, we substitute coconut butter for dairy butter because dairy butter contains a large proportion of saturated fat that raises deadly LDL cholesterol (Trumbo & Shimakawa, 2011). In contrast, coconut butter has so much fiber that the considerable saturated fat that it contains is mostly not absorbed (Padmakumaran, Rajamohan & Kurup, 1999). You can substitute erythritol (den Hartog et al, 2010) for sucanat (Lustig, Schmidt, & Brindis, 2012) to improve the glycemic index and reduce calorie density. We substitute whole wheat flour for all-purpose wheat flour because whole grains are associated with improvement in blood pressure (Tighe et al, 2010).

Value

Returns a brownie recipe.

References

Padmakumaran Nair K.G, Rajamohan T, Kurup P.A. (1999). Coconut kernel protein modifies the effect of coconut oil on serum lipids. Plant Foods Hum Nutr. 53(2):133-44.

Tighe P, Duthie G, Vaughan N, Brittenden J, Simpson W.G, Duthie S, Mutch W, Wahle K, Horgan G, Thies F. (2010). Effect of increased consumption of whole-grain foods on blood pressure and other cardiovascular risk markers in healthy middle-aged persons: a randomized controlled trial. Am. J. Clin. Nutr. 92(4), 733-40.

R H Lustig, L A Schmidt, C D Brindis. (2012). Public health: The toxic truth about sugar. Nature. 482 27-29.

den Hartog G.J, Boots A.W, Adam-Perrot A, Brouns F, Verkooijen I.W, Weseler A.R, Haenen G.R, Bast A. (2010). Erythritol is a sweet antioxidant. Nutrition. 26(4), 449-58.

Trumbo P.R, Shimakawa T. (2011). Tolerable upper intake levels for trans fat, saturated fat, and cholesterol. Nutr. Rev. 69(5), 270-8. doi: 10.1111/j.1753-4887.2011.00389.x.

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

More information about the OpenMx package may be found here.

Examples

```
# Return a brownie recipe
omxBrownie()
```

`omxBuildAutoStartModel` *Build the model used for mxAutoStart*

Description

Build the model used for mxAutoStart

Usage

```
omxBuildAutoStartModel(model, type = c("ULS", "DWLS"))
```

Arguments

`model`	The MxModel for which starting values are desired
`type`	The type of starting values to obtain, currently unweighted or diagonally weighted least squares, ULS or DWLS

Value

an MxModel that can be run to obtain starting values

See Also

mxAutoStart

---

omxCheckCloseEnough *Approximate Equality Testing Function*

Description

This function tests whether two numeric vectors or matrixes are approximately equal to one another, within a specified threshold.

Usage

```
omxCheckCloseEnough(a, b, epsilon = 10^(-15))
```

Arguments

`a`	a numeric vector or matrix
`b`	a numeric vector or matrix
`epsilon`	a non-negative tolerance threshold

Details

Arguments 'a' and 'b' must be of the same type, ie. they must be either vectors of equal dimension or matrices of equal dimension. The two arguments are compared element-wise for approximate equality. If the absolute value of the difference of any two values is greater than the threshold, then an error will be thrown.

References

The OpenMx User's guide can be found at <https://openmx.ssri.psu.edu/documentation>.

See Also

`omxCheckWithinPercentError`, `omxCheckIdentical`, `omxCheckSetEquals`, `omxCheckTrue`, `omxCheckEquals`

Examples

```
omxCheckCloseEnough(c(1, 2, 3), c(1.1, 1.9 ,3.0), epsilon = 0.5)
omxCheckCloseEnough(matrix(3, 3, 3), matrix(4, 3, 3), epsilon = 2)
# Throws an error
try(omxCheckCloseEnough(c(1, 2, 3), c(1.1, 1.9 ,3.0), epsilon = 0.01))
```

omxCheckEquals *Equality Testing Function*

Description

This function tests whether two objects are equal using the '==' operator.

Usage

```
omxCheckEquals(...)
```

Arguments

`...`	arguments forwarded to expect_equivalent

Details

Performs the '==' comparison on the two arguments. If the two arguments are not equal, then an error will be thrown.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

`omxCheckCloseEnough`, `omxCheckWithinPercentError`, `omxCheckSetEquals`, `omxCheckTrue`, `omxCheckIdentical`

Examples

```
omxCheckEquals(c(1, 2, 3), c(1, 2, 3))

omxCheckEquals(FALSE, FALSE)

# Throws an error
try(omxCheckEquals(c(1, 2, 3), c(2, 1, 3)))
```

`omxCheckError` *Correct Error Message Function*

Description

This function tests whether the correct error message is thrown.

Usage

```
omxCheckError(expression, message)
```

Arguments

`expression`	an R expression that produces an error
`message`	a character string with the desired error message

Details

Arguments ‘expression’ and ‘message’ give the expression that generates the error and the message that is supposed to be generated, respectively.

References

The OpenMx User’s guide can be found at <https://openmx.ssri.psu.edu/documentation>.

See Also

`omxCheckWarning` `omxCheckWithinPercentError`, `omxCheckIdentical`, `omxCheckSetEquals`, `omxCheckTrue`, `omxCheckEquals`

`omxCheckIdentical` *Exact Equality Testing Function*

Description

This function tests whether two objects are equal.

Usage

```
omxCheckIdentical(...)
```

Arguments

`...`	arguments forwarded to expect_identical

Details

Performs the ‘identical’ comparison on the two arguments. If the two arguments are not equal, then an error will be thrown.

References

The OpenMx User’s guide can be found at https://openmx.ssri.psu.edu/documentation/.

See Also

`omxCheckCloseEnough`, `omxCheckWithinPercentError`, `omxCheckSetEquals`, `omxCheckTrue`, `omxCheckEquals`

Examples

```
omxCheckIdentical(c(1, 2, 3), c(1, 2, 3))

omxCheckIdentical(FALSE, FALSE)

# Throws an error
try(omxCheckIdentical(c(1, 2, 3), c(2, 1, 3)))
```

`omxCheckNamespace` *omxCheckNamespace*

Description

This is an internal function exported for those people who know what they are doing.

Usage

```
omxCheckNamespace(model, namespace)
```

Arguments

`model`	model
`namespace`	namespace

Details

This function checks that the named entities in the model are valid.

omxCheckSetEquals *Set Equality Testing Function*

Description

This function tests whether two vectors contain the same elements.

Usage

```
omxCheckSetEquals(...)
```

Arguments

`...`	arguments forwarded to expect_setequal

Details

Performs the 'setequal' function on the two arguments. If the two arguments do not contain the same elements, then an error will be thrown.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

`omxCheckCloseEnough`, `omxCheckWithinPercentError`, `omxCheckIdentical`, `omxCheckTrue`, `omxCheckEquals`

Examples

```
omxCheckSetEquals(c(1, 1, 2, 2, 3), c(3, 2, 1))

omxCheckSetEquals(matrix(1, 1, 1), matrix(1, 3, 3))

# Throws an error
try(omxCheckSetEquals(c(1, 2, 3, 4), c(2, 1, 3)))
```

`omxCheckTrue` *Boolean Equality Testing Function*

Description

This function tests whether an object is equal to TRUE.

Usage

```
omxCheckTrue(a)
```

Arguments

`a`	the value to test.

Details

Checks element-wise whether an object is equal to TRUE. If any of the elements are false, then an error will be thrown.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

`omxCheckCloseEnough`, `omxCheckWithinPercentError`, `omxCheckIdentical`, `omxCheckSetEquals`, `omxCheckEquals`

Examples

```
omxCheckTrue(1 + 1 == 2)

omxCheckTrue(matrix(TRUE, 3, 3))

# Throws an error
try(omxCheckTrue(FALSE))
```

`omxCheckWarning` *Correct Warning Message Function*

Description

This function tests whether the correct warning message is thrown. Arguments 'expression' and 'message' give the expression that generates the warning and the message that is supposed to be generated, respectively.

Usage

```
omxCheckWarning(expression, message)
```

Arguments

`expression`	an R expression that produces a warning
`message`	a character string with the desired warning message

Details

note: to test for no warning, set `message = NA`.

References

The OpenMx User's guide can be found at <https://openmx.ssri.psu.edu/documentation>.

See Also

`omxCheckError` `omxCheckWithinPercentError`, `omxCheckIdentical`, `omxCheckSetEquals`, `omxCheckTrue`, `omxCheckEquals`

Examples

```
foo <- omxCheckWarning(mxFIMLObjective('cov', 'mean'), "deprecated")

# Test for no warning
omxCheckWarning(2+2, message = NA)
```

`omxCheckWithinPercentError` *Approximate Percent Equality Testing Function*

Description

This function tests whether two numeric vectors or matrixes are approximately equal to one another, within a specified percentage.

Usage

```
omxCheckWithinPercentError(a, b, percent = 0.1)
```

Arguments

`a`	a numeric vector or matrix.
`b`	a numeric vector or matrix.
`percent`	a non-negative percentage.

Details

Arguments ‘a’ and ‘b’ must be of the same type, ie. they must be either vectors of equal dimension or matrices of equal dimension. The two arguments are compared element-wise for approximate equality. If the absolute value of the difference of any two values is greater than the percentage difference of ‘a’, then an error will be thrown.

References

The OpenMx User’s guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

See Also

`omxCheckCloseEnough`, `omxCheckIdentical`, `omxCheckSetEquals`, `omxCheckTrue`, `omxCheckEquals`

Examples

```
omxCheckWithinPercentError(c(1, 2, 3), c(1.1, 1.9 ,3.0), percent = 50)

omxCheckWithinPercentError(matrix(3, 3, 3), matrix(4, 3, 3), percent = 150)

# Throws an error
try(omxCheckWithinPercentError(c(1, 2, 3), c(1.1, 1.9 ,3.0), percent = 0.01))
```

omxConstrainMLThresholds *omxConstrainMLThresholds*

Description

Add constraint to ML model to keep thresholds in order

Usage

```
omxConstrainMLThresholds(model, dist = 0.1)
```

Arguments

`model`	the MxModel to which constraints should be added
`dist`	unused

Details

This function adds a nonlinear constraint to an ML model. The constraint keeps the thresholds in order. Constraints often slow model estimation, however, keeping the thresholds in increasing order helps ensure the likelihood function is well-defined. If you're having problems with ordinal data, this is one of the things to try.

Value

a new MxModel object with the constraints added

See Also

`demo("omxConstrainMLThresholds")`

omxDefaultComputePlan *Construct default compute plan*

Description

This function generates a default compute plan, where "default compute plan" refers to an object of class MxComputeSequence which is appropriate for use in a wide variety of cases. The exact specification of the plan will depend upon the arguments provided to `omxDefaultComputePlan()`.

Usage

```
omxDefaultComputePlan(modelName=NULL, intervals=FALSE,
  useOptimizer=TRUE, optionList=options()$mxOption)
```

Arguments

`modelName`	Optional (defaults to `NULL`) character string, providing the name of the MxModel the fitfunction of which is to be evaluated, and usually, optimized.
`intervals`	Logical; will confidence intervals be computed? Defaults to `FALSE`.
`useOptimizer`	Logical; will a fitfunction be minimized? Defaults to `TRUE`.
`optionList`	List of mxOptions. Defaults to the current list of global mxOptions.

Details

At minimum, argument `optionList` must include “Gradient algorithm”, “Gradient iterations”, “Gradient step size”, “Calculate Hessian”, and “Standard Errors”.

Value

Returns an object of class MxComputeSequence.

Examples

```
foo <- omxDefaultComputePlan(modelName="bar")
str(foo)
```

`omxDetectCores` *omxDetectCores*

Description

Detects the number of cores on the local machine

Usage

```
omxDetectCores(...)
```

Arguments

`...`	unused

Examples

```
omxDetectCores()
```

omxGetBootstrapReplications *omxGetBootstrapReplications*

Description

Checks a variety of conditions to ensure that bootstrap replications are available and valid. Throws exception if things go wrong. Otherwise, replications are returned to the caller.

Usage

```
omxGetBootstrapReplications(model)
omxBootstrapCov(model)
```

Arguments

`model`	an MxModel object

Value

a matrix or covariance matrix of bootstrap parameter estimates

omxGetNPSOL *omxGetNPSOL*

Description

Get the non-CRAN version of OpenMx from the OpenMx website.

Usage

```
omxGetNPSOL()
```

Details

This function

Value

Invisible NULL

Examples

```
omxGetNPSOL()
```

omxGetParameters *Fetch Model Parameters*

Description

Return a vector of the chosen parameters from the model.

Usage

```
omxGetParameters(model, indep = FALSE, free = c(TRUE, FALSE, NA),
    fetch = c('values', 'free', 'lbound', 'ubound', 'all'),
    labels = c())
```

Arguments

`model`	a MxModel object
`indep`	fetch parameters from independent submodels.
`free`	fetch either free parameters (TRUE), or fixed parameters or both types. Default value is TRUE.
`fetch`	which attribute of the parameters to fetch. Default choice is 'values'.
`labels`	additional labels to fetch

Details

The argument 'free' dictates whether to return only free parameters or only fixed parameters or both free and fixed parameters. The function can return unlabeled free parameters (parameters with a label of NA). These anonymous free parameters will be identified as 'modelname.matrixname[row,col]'. It will not return fixed parameters that have a label of NA.

If provided, the argument 'labels' takes precedent over the selection criteria specified by 'free'. Any labels mentioned in 'labels', including those of the form 'modelname.matrixname[row,col]', will be returned.

No distinction is made between ordinary labels, definition variables, and square bracket constraints. The function will return either a vector of parameter values, or free/fixed designations, or lower bounds, or upper bounds, depending on the 'fetch' argument. Using fetch with 'all' returns a data frame that is populated with all of the attributes.

See Also

`omxSetParameters`, `omxLocateParameters`, `omxAssignFirstParameters`

Examples

```
library(OpenMx)

A <- mxMatrix('Full', 2, 2, labels = c("A11", "A12", "A21", NA), values= 1:4,
    free = c(TRUE,TRUE,FALSE,TRUE), byrow=TRUE, name = 'A')
```

```
model <- mxModel(A, name = 'model')

# Request all free parameters in model
omxGetParameters(model)

# A11  A12 model.A[2,2]
#   1    2    4

# Request fixed parameters from model
omxGetParameters(model, free = FALSE)
# A21
#   3

A$labels
#      [,1]  [,2]
# [1,] "A11" "A12"
# [2,] "A21" NA

A$free
#       [,1] [,2]
# [1,]  TRUE TRUE
# [2,] FALSE TRUE

A$values
#      [,1] [,2]
# [1,]    1    2
# [2,]    3    4

# Example using un-labelled parameters

# Read in some demo data
data(demoOneFactor)
# Grab the names for manifestVars
manifestVars <- names(demoOneFactor)
nVar = length(manifestVars) # 5 variables
factorModel <- mxModel("One Factor",
    mxMatrix(name="A", type="Full", nrow=nVar, ncol=1, values=0.2, free=TRUE,
        lbound = 0.0, labels=letters[1:nVar]),
    mxMatrix(name="L", type="Symm", nrow=1, ncol=1, values=1, free=FALSE),
    # the "U" matrix has nVar (5) anonymous free parameters
    mxMatrix(name="U", type="Diag", nrow=nVar, ncol=nVar, values=1, free=TRUE),
    mxAlgebra(expression=A %&% L + U, name="R"),
    mxExpectationNormal(covariance="R", dimnames=manifestVars),
    mxFitFunctionML(),
    mxData(observed=cov(demoOneFactor), type="cov", numObs=500)
)

# Get all free parameters
params         <- omxGetParameters(factorModel)
lbound         <- omxGetParameters(factorModel, fetch="lbound")
# Set new values for these params, saving them in a new model
newFactorModel <- omxSetParameters(factorModel, names(params), values = 1:10)
# Read out the values from the new model
```

```
newParams      <- omxGetParameters(newFactorModel)
```

`omxGetRAMDepth` *omxGetRAMDepth*

Description

Get the potency of a matrix for inversion speed-up

Usage

```
omxGetRAMDepth(A, maxdepth = nrow(A) - 1)
```

Arguments

`A`	MxMatrix object
`maxdepth`	Numeric. maximum depth to check

Details

This function is used internally by the mxExpectationRAM function to determine how far to expand $(I - A)^{-1} = I + A + A^2 + A^3 + ...$. It is similarly used by mxExpectationLISREL in expanding $(I - B)^{-1} = I + B + B^2 + B^3 + ...$. In many situations A^2 is a zero matrix (nilpotent of order 2). So when A has large dimension it is much faster to compute $I + A$ than $(I - A)^{-1}$.

`omxGraphviz` *Show RAM Model in Graphviz Format*

Description

The function accepts a RAM style model and outputs a visual representation of the model in Graphviz format. The function will output either to a file or to the console. The recommended file extension for an output file is ".dot".

Usage

```
omxGraphviz(model, dotFilename = "")
```

Arguments

`model`	An RAM-type model.
`dotFilename`	The name of the output file. Use "" to write to console.

Value

Invisibly returns a string containing the model description in Graphviz format.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

`omxHasDefaultComputePlan` *omxHasDefaultComputePlan*

Description

Determine whether the model has a default complete plan (i.e., not custom).

Usage

```
omxHasDefaultComputePlan(model)
```

Arguments

`model`	model

`omxLapply` *On-Demand Parallel Lapply*

Description

If the snowfall library is loaded, then this function calls `sfLapply`. Otherwise it invokes `lapply`.

Usage

```
omxLapply(x, fun, ...)
```

Arguments

`x`	a vector (atomic or list) or an expressions vector. Other objects (including classed objects) will be coerced by `as.list`.
`fun`	the function to be applied to each element of `x`.
`...`	optional arguments to `fun`.

See Also

`omxApply`, `omxSapply`

Examples

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
# compute the list mean for each list element
omxLapply(x,mean)
```

---

omxLocateParameters *Get the location (model, matrix, row, column) and other info for a parameter*

Description

Returns a data.frame summarizing the *free* parameters in a model, possibly filtered using 'labels'.

For each located parameter, the label, model, matrix, row, col, value, and lbound & ubound are given as a row in the dataframe.

Duplicated labels return a row for each location in which they are found.

Usage

```
omxLocateParameters(model, labels = NULL, indep = FALSE, free = c(TRUE, FALSE, NA))
```

Arguments

`model`	a MxModel object
`labels`	optionally specify which free parameters to retrieve.
`indep`	fetch parameters from independent submodels.
`free`	fetch either free parameters (TRUE), or fixed parameters or both types. Default value is TRUE.

Details

Invoking the function with the default value for the 'labels' argument retrieves all the free parameters. The 'labels' argument can be used to select a subset of the free parameters. Note that 'NA' is a valid input to 'labels'.

See Also

`omxGetParameters`, `omxSetParameters`, `omxAssignFirstParameters`

Examples

```
A <- mxMatrix('Full', 2, 2, labels = c("A11", "A12", NA, "A11"), values= 1:4,
   free = TRUE, byrow = TRUE, name = 'A')

model <- mxModel(A, name = 'model')

# Request all free parameters in model
omxLocateParameters(model)

# Request free parameters "A11" and all NAs
omxLocateParameters(model, c("A11", NA))

# Works with submodel
B = mxMatrix(name = 'B', 'Full', 1, 2, labels = c("B11", "notme"),
  free = c(TRUE, FALSE), values= pi)
model <- mxModel(model, mxModel(B, name = 'subB'))

# nb: only returns free parameters ('notme' not shown)
omxLocateParameters(model)
```

`omxLogical` *Logical mxAlgebra() operators*

Description

`omxNot` computes the unary negation of the values of a matrix. `omxAnd` computes the binary and of two matrices. `omxOr` computes the binary or of two matrices. `omxGreaterThan` computes a binary greater than of two matrices. `omxLessThan` computes the binary less than of two matrices. `omxApproxEquals` computes a binary equals within a specified epsilon of two matrices.

Usage

```
omxNot(x)
omxAnd(x, y)
omxOr(x, y)
omxGreaterThan(x, y)
omxLessThan(x, y)
omxApproxEquals(x, y, epsilon)
```

Arguments

`x`	the first argument, the matrix which the logical operation will be applied to.
`y`	the second argument, applicable to binary functions.
`epsilon`	the third argument, specifies the error threshold for omxApproxEquals. Abs(x[i][j]-y[i][j]) must be less than epsilon[i][j].

Examples

```
A <- mxMatrix(values = runif(25), nrow = 5, ncol = 5, name = 'A')
B <- mxMatrix(values = runif(25), nrow = 5, ncol = 5, name = 'B')
EPSILON <- mxMatrix(values = 0.04*1:25, nrow = 5, ncol = 5, name = "EPSILON")

model <- mxModel(A, B, EPSILON, name = 'model')

mxEval(omxNot(A), model)
mxEval(omxGreaterThan(A,B), model)
mxEval(omxLessThan(B,A), model)
mxEval(omxOr(omxNot(A),B), model)
mxEval(omxAnd(omxNot(B), A), model)
mxEval(omxApproxEquals(A,B, EPSILON), model)
```

`omxManifestModelByParameterJacobian` *Estimate the Jacobian of manifest model with respect to parameters*

Description

The manifest model excludes any latent variables or processes. For RAM and LISREL models, the manifest model contains only the manifest variables with free means, covariance, and thresholds.

Usage

```
omxManifestModelByParameterJacobian(model, defvar.row = 1, standardize = FALSE)
```

Arguments

`model`	an mxModel
`defvar.row`	which row to use for definition variables
`standardize`	logical, whether or not to standardize the parameters

Details

The Jacobian is estimated by the central finite difference.

If the `standardize` argument is TRUE, then the Jacobian is for the standardized model. For Normal expectations the standardized manifest model has the covariances returned as correlations, the variances returned as ones, the means returned as zeros, and the thresholds are returned as z-scores. For the thresholds the z-scores are computed by using the model-implied means and variances.

Value

a matrix with manifests in the rows and original parameters in the columns

See Also

mxGetExpected

omxMatrixOperations *MxMatrix operations*

Description

`omxCbind` columnwise binding of two or more MxMatrices. `omxRbind` rowwise binding of two or more MxMatrices. `omxTranspose` transpose of MxMatrix.

Usage

```
omxCbind(..., allowUnlabeled =
    getOption("mxOptions")[["Allow Unlabeled"]],
    dimnames = NA, name = NA)
omxRbind(..., allowUnlabeled =
    getOption("mxOptions")[["Allow Unlabeled"]],
    dimnames = NA, name = NA)
omxTranspose(matrix, allowUnlabeled =
    getOption("mxOptions")[["Allow Unlabeled"]],
    dimnames = NA, name = NA)
```

Arguments

`...`	two or more MxMatrix objects
`matrix`	MxMatrix input
`allowUnlabeled`	whether or not to accept free parameters with NA labels
`dimnames`	list. The dimnames attribute for the matrix: a list of length 2 giving the row and column names respectively. An empty list is treated as NULL, and a list of length one as row names. The list can be named, and the list names will be used as names for the dimensions.
`name`	an optional character string indicating the name of the MxMatrix object

omxMnor *Multivariate Normal Integration*

Description

Given a covariance matrix, a means vector, and vectors of lower and upper bounds, returns the multivariate normal integral across the space between bounds.

Usage

```
omxMnor(covariance, means, lbound, ubound)
```

Arguments

covariance	the covariance matrix describing the multivariate normal distribution.
means	a row vector containing means of the variables of the underlying distribution.
lbound	a row vector containing the lower bounds of the integration in each variable.
ubound	a row vector containing the upper bounds of the integration in each variable.

Details

The order of columns in the 'means', 'lbound', and 'ubound' vectors are assumed to be the same as that of the covariance matrix. That is, means[i] is considered to be the mean of the variable whose variance is in covariance[i,i]. That variable will be integrated from lbound[i] to ubound[i] as part of the integration.

The value of ubound[i] or lbound[i] may be set to Inf or -Inf if a boundary at positive or negative infinity is desired.

For all i, ubound[i] must be strictly greater than lbound[i].

Examples

```
data(myFADataRaw)

covariance <- cov(myFADataRaw[,1:3])
means <- colMeans(myFADataRaw[,1:3])
lbound <- c(-Inf, 0,   1)    # Integrate from -Infinity to 0 on first variable
ubound <- c(0,    Inf, 2.5)  # From 0 to +Infinity on second, and from 1 to 2.5 on third
omxMnor(covariance, means, lbound, ubound)
# 0.0005995

# An alternative specification of the bounds follows
# Integrate from -Infinity to 0 on first variable
v1bound = c(-Inf, 0)
# From 0 to +Infinity on second
v2bound = c(0, Inf)
# and from 1 to 2.5 on third
v3bound = c(1, 2.5)
bounds <- cbind(v1bound, v2bound, v3bound)
lbound <- bounds[1,]
ubound <- bounds[2,]
omxMnor(covariance, means, lbound, ubound)
```

---

omxModelDeleteData *Remove all instances of data from a model*

Description

For very large data, it can be desirable to discard data after the model is run. That is what the purpose of this function.

Data is discarded from the model and all submodels recursively.

Usage

```
omxModelDeleteData(model)
```

Arguments

`model`	a MxModel object.

Examples

```
library(OpenMx)

data(demoOneFactor)
manifests <- names(demoOneFactor)
latents   <- c("G")
factorModel <- mxModel(model="One Factor", type="RAM",
      manifestVars = manifests,
      latentVars   = latents,
      mxPath(from=latents, to=manifests),
      mxPath(from=manifests, arrows=2),
      mxPath(from=latents, arrows=2,free=FALSE, values=1.0),
      mxData(cov(demoOneFactor), type="cov",numObs=500)
)
factorFit <-mxRun(factorModel)
object.size(factorFit)
factorFit <- omxModelDeleteData(factorFit)
object.size(factorFit)
factorFit$data
```

---

omxNameAnonymousParameters
omxNameAnonymousParameters

Description

Assign new names to the unnamed parameters

Usage

```
omxNameAnonymousParameters(model, indep = FALSE)
```

Arguments

`model`	the MxModel
`indep`	whether models are independent

Value

a list with components for the new MxModel with named parameters, and the new names.

omxParallelCI	*Calculate confidence intervals without re-doing the primary optimization.*

Description

OpenMx provides two functions to calculate confidence intervals for already-run MxModel objects that contain an MxInterval object (i.e., an `mxCI()` statement), without recalculating point estimates, fitfunction derivatives, or expectations.

The primary function is `omxRunCI()`. This is a wrapper for `omxParallelCI()` with arguments `run=TRUE` and `independentSubmodels=FALSE`, and is the recommended interface.

`omxParallelCI()` does the work of calculating confidence intervals. The "parallel" in the function's name refers to the not-yet-implemented feature of running independent submodels in parallel.

Usage

```
omxRunCI(model, verbose = 0, optimizer = "SLSQP")

omxParallelCI(model, run = TRUE, verbose = 0, independentSubmodels = TRUE,
optimizer = mxOption(NULL, "Default optimizer"))
```

Arguments

`model`	An MxModel object that contains an MxInterval object (i.e., an `mxCI()` statement).
`run`	Logical; For `omxParallelCI()`, determines if the model with its new compute plan is `mxRun()` before being returned. Hard-coded `TRUE` for omxRunCI.
`verbose`	Integer; defaults to zero; verbosity level passed to MxCompute* objects.
`independentSubmodels`	Logical; For `omxParallelCI()` defaults to `TRUE`. Hard coded FALSE for `omxRunCI()`. Also see "Details."
`optimizer`	Character string selecting the gradient-descent optimizer to be used to find confidence limits; one of "NPSOL", "CSOLNP", or "SLSQP". The default for `omxParallelCI()` is the current value of mxOption "Default optimizer", and for `omxRunCI()`, is "SLSQP".

Details

When `independentSubmodels=TRUE`, `omxParallelCI()` creates an independent MxModel object for each quantity specified in the 'reference' slot of `model`'s MxInterval object, and places these independent MxModels inside `model`. Each of these independent submodels calculates the confidence limits of its own quantity when the container model is run. When `independentSubmodels=FALSE`, no submodels are added to `model`. Instead, `model` is provided with a dedicated compute plan consisting only of an MxComputeConfidenceInterval step. Note that using `independentSubmodels=FALSE` will overwrite any compute plan already inside `model`.

Value

The functions return `model`, augmented with independent submodels (if `independentSubmodels=TRUE`) or with a non-default compute plan (if `independentSubmodels=FALSE`), and possibly having been passed through `mxRun()` (if `run=TRUE`). Naturally, if `run=FALSE`, the user can subsequently run the returned model to obtain confidence intervals. Users are cautioned that the returned model may not be very amenable to being further modified and re-fitted (e.g., having some free parameters fixed via `omxSetParameters()` and passed through `mxRun()` to get new point estimates) unless the added submodels or the non-default compute plan are eliminated. The exception is if `run=TRUE` and `independentSubmodels=TRUE` (which is always the case with `omxRunCI()`), since the non-default compute plan is set to be non-persistent, and will automatically be replaced with a default compute plan the next time the model is passed to `mxRun()`.

See Also

`mxCI()`, MxInterval, `mxComputeConfidenceInterval()`

Examples

```
require(OpenMx)
# 1. Build and run a model, don't compute intervals yet
data(demoOneFactor)
manifests <- names(demoOneFactor)
latents <- c("G")
factorModel <- mxModel("One Factor", type="RAM",
manifestVars=manifests,
latentVars=latents,
mxPath(from=latents, to=manifests),
  mxPath(from=manifests, arrows=2),
mxPath(from=latents, arrows=2, free=FALSE, values=1.0),
mxData(observed=cov(demoOneFactor), type="cov", numObs=500),
# Add confidence intervals for (free) params in A and S matrices.
mxCI(c('A', 'S'))
)
factorRun <- mxRun(factorModel)

# 2. Compute the CIs on factorRun, and view summary
factorCI1 <- omxRunCI(factorRun)
summary(factorCI1)$CI

# 3. Use low-level omxParallelCI interface
factorCI2 <- omxParallelCI(factorRun)

# 4. Build, but don't run the newly-created model
factorCI3 <- omxParallelCI(factorRun, run= FALSE)
```

---

omxQuotes *omxQuotes*

Description

Quote helper function, often for error messages.

Usage

```
omxQuotes(name)
```

Arguments

`name`	a character vector

Details

This is a helper function for creating a nicely put together formatted string.

Value

a character string

Examples

```
omxQuotes(c("Oh", "blah", "dee", "Oh", "blah", "da"))
omxQuotes(c("A", "S", "F"))
omxQuotes("Hello World")
```

`omxRAMtoML` *omxRAMtoML*

Description

Convert a RAM model to an ML model

Usage

```
omxRAMtoML(model)
```

Arguments

`model`	the MxModel

Details

This is a legacy function that was once used to convert RAM models to ML models in the old (1.0 release of OpenMx) objective function style.

Value

an ML model with an ML objective

`omxReadGRMBin`	*Read a GCTA-Format Binary GRM into R.*

Description

This simple function is adapted from syntax in the GCTA User Manual. It loads a binary genomic-relatedness matrix (GRM) from disk into R's workspace.

Usage

```
omxReadGRMBin(prefix, AllN=FALSE, size=4, returnList=FALSE)
```

Arguments

`prefix`	Character string: everything in the path, relative to R's working directory, and filenames of the GRM files preceding '.grm.*'. See below, under "Details"
`AllN`	Logical. If `FALSE` (default), then when `omxReadGRMBin()` calls `readBin()`, it passes a value of 1 for argument `n`. if `TRUE`, then it instead passes a value equal to the number of nonredundant elements in the GRM.
`size`	Passed to `readBin()`.
`returnList`	Logical. If `FALSE` (default), `omxReadGRMBin` returns the GRM. If `TRUE`, then `omxReadGRMBin` returns a list as described below, under "Value".

Details

A GRM calculated in GCTA that is saved to disk in binary format comprises three files, the filenames of which have the same stem but different extensions. The first, with extension "grm.bin", is the actual binary file containing the GRM elements. The second, with extension "grm.N.bin", contains information about how many genetic markers were used to calculate the GRM. The third, with extension "grm.id", is a text file containing two columns of data, respectively, the participant family and individual IDs. `omxReadGRMBin()` is meant to be used with all three files together in the same directory. Thus, argument `prefix` should be everything in the path (relative to R's working directory) and filenames of those GRM files, up to the first period in their extensions. In practice, it is simplest to set R's working directory to whichever directory contains the files, and simply provide the filename stem for argument `prefix`.

`omxReadGRMBin()` opens three file connections, one for each file.

Value

If `returnList=FALSE` (the default), then the GRM itself is returned as a numeric matrix, with each row and column named as the sum of the corresponding participant's family ID and individual ID. Otherwise, a list of the following four elements is returned:

1. `"diag"`: Numeric vector containing the GRM's diagonal elements.
2. `"off"`: Numeric vector containing the GRM's off-diagonal elements.

3. "id": Dataframe containing the family and individual IDs corresponding to the rows and columns of the GRM.
4. "N": Numeric; number of markers used to calculate the GRM.

References

Yang J, Lee SH, Goddard ME, Visscher, PM. GCTA: A tool for genome-wide complex trait analysis. American Journal of Human Genetics 2011;88:76-82. doi: 10.1016/j.ajhg.2010.11.011.

Yang J, Lee SH, Goddard ME, Visscher, PM. Genome-wide complex trait analysis (GCTA): Methods, data analyses, and interpretations. In Gondro, C et al. (Eds.), *Genome-Wide Association Studies and Genomic Prediction*. New York: Springer;2013. p. 215-236.

GCTA website: https://cnsgenomics.com/software/gcta/#Overview.

Code for omxReadGRMBin() was adapted from syntax by Jiang Yang in the GCTA User Manual, version 1.24, dated 28 July 2014, retrieved from http://cnsgenomics.com/software/gcta/GCTA_UserManual_v1.24.pdf
.

---

omxRMSEA *Get the RMSEA with confidence intervals from model*

Description

This function calculates the Root Mean Square Error of the Approximation (RMSEA) for a model and computes confidence intervals for that fit statistic.

Usage

```
omxRMSEA(model, lower=.025, upper=.975, null=.05, ...)
```

Arguments

model	An MxModel object for which the RMSEA is desired
lower	The lower confidence bound for the confidence interval
upper	The upper confidence bound for the confidence interval
null	Value of RMSEA used to test for close fit
...	Further named arguments passed to summary

Details

To help users obtain fit statistics related to the RMSEA, this function confidence intervals and a test for close fit. The user determines how close the fit is required to be by setting the null argument to the value desired for comparison.

Value

A named vector with elements lower, est.rmsea, upper, null, and ‘Prob(x <= null)‘.

References

Browne, M. W. & Cudeck, R. (1992). Alternative Ways of Assessing Model Fit. *Sociological Methods and Research*, **21**, 230-258.

Examples

```
require(OpenMx)
data(demoOneFactor)
manifests <- names(demoOneFactor)
latents <- c("G")
factorModel <- mxModel("One Factor",
                       type="RAM",
                       manifestVars=manifests,
                       latentVars=latents,
                       mxPath(from=latents, to=manifests),
                       mxPath(from=manifests, arrows=2),
                       mxPath(from=latents, arrows=2, free=FALSE, values=1.0),
                       mxData(observed=cov(demoOneFactor), type="cov", numObs=500))
factorRun <- mxRun(factorModel)
factorSat <- mxRefModels(factorRun, run=TRUE)
summary(factorRun, refModels=factorSat)
# Gives RMSEA with 95% confidence interval

omxRMSEA(factorRun, .05, .95, refModels=factorSat)
# Gives RMSEA with 90% confidence interval
#  and probability of 'close enough' fit
```

---

omxSapply — *On-Demand Parallel Sapply*

Description

If the snowfall library is loaded, then this function calls `sfSapply`. Otherwise it invokes `sapply`.

Usage

```
omxSapply(x, fun, ..., simplify = TRUE, USE.NAMES = TRUE)
```

Arguments

`x`	a vector (atomic or list) or an expressions vector. Other objects (including classed objects) will be coerced by `as.list`.
`fun`	the function to be applied to each element of `x`.
`...`	optional arguments to `fun`.
`simplify`	logical; should the result be simplified to a vector or matrix if possible?
`USE.NAMES`	logical; if `TRUE` and if `x` is a character, use `x` as `names` for the result unless it had names already.

See Also

`omxApply`, `omxLapply`

Examples

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
# compute the list mean for each list element
omxSapply(x, quantile)
```

`omxSaturatedModel` *Create Reference (Saturated and Independence) Models*

Description

This function creates and, optionally, runs saturated and independence (null) models of a base model or data set for use with mxSummary to enable fit indices that depend on these models. Note that there are cases where this function is not valid for use, or should be used with caution (see below, under "Warnings").

Usage

```
mxRefModels(x, run=FALSE, ..., distribution="default", equateThresholds = TRUE)
```

Arguments

`x`	A MxModel object, data frame, or matrix.
`run`	logical. If TRUE, runs the models before returning; otherwise returns built models without running.
`...`	Not used. Forces remaining arguments to be specified by name.
`distribution`	character. Which distribution to assume.
`equateThresholds`	logical. Whether ordinal thresholds should be constrained equal across groups.

Details

For typical structural equation models the saturated model is the free-est possible model. Not only all variances (and, when possible, all means) are estimated, but also all covariances. In the case of ordinal data, the ordinal means are fixed to zero and the thresholds are estimated. For binary variables, those variances are also constrained to one. This is the free-est possible model that is identified. The saturated model is used in calculating fit statistics such as the RMSEA, and Chi-squared fit indices.

The independence model, sometimes called the null model, is a model in which each variable is treated as being completely independent of every other variable. As such, all the variances and, when possible, all means are estimated. However, covariances are set to zero. Ordinal variables are handled the same for the independence and saturated models. The independence model is used, along with the saturated model, to create CFI and TLI fit indices.

The saturated and independence models could be used to create further fit indices. However, OpenMx does not recommend using GFI, AGFI, NFI (aka Bentler-Bonett), or SRMR. The page for `mxSummary` has information about why.

When the mxFitFunctionMultigroup fit function is used, `mxRefModels` creates the appropriate multi-group saturated and independence models. Saturated and independence models are created separately for each group. Each group has its own saturated and independence model. The multi-group saturated model is a multi-group model where each group has its own saturated model, and similarly for the independence model.

When an MxModel has been run, some effort is made to make the reference models for only the variables used in the model. For covariance data, all variables are modeled by default. For raw data when the model has been run, only the modeled variables are used in the reference models. This matches the behavior of mxModel.

In general, it is best practice to give `mxRefModels` a model that has already been run.

Multivariate normal models with all ordinal data and no missing values can use the saturated multinomial distribution. This is much faster than estimation of the saturated multivariate normal model. Use `distribution='multinomial'` to avail this option.

Warnings

One potentially important limitation of the `mxRefModels` function is for behavior-genetic models. If variables 'x', 'y', and 'z' are measured on twins 1 and 2 creating the modeled variables 'x1', 'y1', 'z1', 'x2', 'y2', 'z2', then this function may not create the intended saturated or independence models. In particular, the means of 'x1' and 'x2' are estimated separately. Similarly, the covariance of 'x1' with 'y1' and 'x2' with 'y2' are allowed be be distinct: $cov(x1, y1)! = covx2, y2$. Moreover, the cross-twin covariances are estimated: e.g. $cov(x1, y2)! = 0$.

Another potential misuse of this function is for models with definition variables. If definition variables are used, the saturated and independence model may not be correct because they do not account for the definition variables.

The are a few considerations specific to IFA models (mxExpectationBA81). The independence model preserves equality constraints among item parameters from the original model. The saturated model is a multinomial distribution with the proportions equal to the proportions in your data. For example, if you have 2 dichotomous items then there are 4 possible response patterns: 00, 01, 10, 11. A multinomial distribution for these 2 items is fully specified by 3 proportions or 3 parameters: a, b, c, $1.0 - (a + b + c)$. Hence, there is no need to optimize the saturated model. When there is no missing data, the deviance is immediately known as $-2 * sum(logproportions)$. Typical Bayesian priors involve latent factors (various densities on the pseudo-guessing lower bound, log norm on loading, and uniqueness prior). These priors cannot be included in the independence model because there are no latent factors. Therefore, exercise caution when comparing the independence model to a model that includes Bayesian priors.

`mxRefModels()` is not compatible with GREML expectation, as there is no sensible general definition for a saturated GREML-type model.

References

The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation/`.

Examples

```
require(OpenMx)
data(demoOneFactor)
manifests <- names(demoOneFactor)
latents <- c("G")
factorModel <- mxModel("OneFactor", type = "RAM",
    manifestVars = manifests, latentVars = latents,
    mxPath(from = latents, to=manifests, values = diag(var(demoOneFactor))*.2),
    mxPath(from = manifests, arrows = 2, values = diag(var(demoOneFactor))*.8),
    mxPath(from = latents, arrows = 2, free = FALSE, values = 1),
    mxData(cov(demoOneFactor), type = "cov", numObs = 500)
)
factorRun <- mxRun(factorModel)
factorSat <- mxRefModels(factorRun, run=TRUE)
summary(factorRun)
summary(factorRun, refModels=factorSat)

# A raw-data example where using mxRefModels adds fit indices

m1 <- mxModel("OneFactor", type = "RAM",
    manifestVars = manifests, latentVars = latents,
    mxPath(latents, to=manifests, values = diag(var(demoOneFactor))*.2),
    mxPath(manifests, arrows = 2, values = diag(var(demoOneFactor))*.8),
    mxPath(latents, arrows = 2, free = FALSE, values = 1),
    mxPath("one", to = latents, free = FALSE, values = 0),
    mxPath("one", to = manifests, values = 0),
    mxData(demoOneFactor, type = "raw")
)
m1 <- mxRun(m1)
summary(m1) # CFI, TLI, RMSEA missing
summary(m1, refModels=mxRefModels(m1, run = TRUE))
```

`omxSelectRowsAndCols` *Filter rows and columns from an mxMatrix*

Description

This function filters rows and columns from a matrix using a single row or column R matrix as a selector.

Usage

```
omxSelectRowsAndCols(x, selector)
omxSelectRows(x, selector)
omxSelectCols(x, selector)
```

Arguments

`x`	the matrix to be filtered
`selector`	A single row or single column R matrix indicating which values should be filtered from the mxMatrix.

Details

omxSelectRowsAndCols, omxSelectRows, and omxSelectCols returns the filtered entries in a target matrix specified by a single row or single column selector matrix. Each entry in the selector matrix is treated as a logical data indicating if the corresponding entry in the target matrix should be excluded (0 or FALSE) or included (not 0 or TRUE). Typically the function is used to filter data from a target matrix using an existence vector which specifies what data entries are missing. This can be seen in the demo: RowObjectiveFIMLBivariateSaturated.

Value

Returns a new matrix with the filtered data.

References

The function is most often used when filtering data for missingness. This can be seen in the demo: RowObjectiveFIMLBivariateSaturated. The OpenMx User's guide can be found at `https://openmx.ssri.psu.edu/documentation`. The omxSelect* functions share some similarity to the Extract function in the R programming language.

Examples

```
loadings <- matrix(1:9, 3, 3, byrow= TRUE)
existenceList <- c(1, 0, 1)
existenceList <- matrix(existenceList, 1, 3, byrow= TRUE)
rowsAndCols <- omxSelectRowsAndCols(loadings, existenceList)
rows <- omxSelectRows(loadings, existenceList)
cols <- omxSelectCols(loadings, existenceList)
```

`omxSetParameters` *Assign Model Parameters*

Description

Modify the attributes of parameters in a model. This function cannot modify parameters that have NA labels. Often you will want to call `omxAssignFirstParameters` after using this, to force the starting values of equated parameters to the same value (otherwise the model cannot begin to be evaluated)

Usage

```
omxSetParameters(model, labels=names(coef(model)), free = NULL, values = NULL,
    newlabels = NULL, lbound = NULL, ubound = NULL, indep = FALSE,
    strict = TRUE, name = NULL)
```

Arguments

`model`	an MxModel object.
`labels`	a character vector of target parameter names.
`free`	a boolean vector of parameter free/fixed designations.
`values`	a numeric vector of parameter values.
`newlabels`	a character vector of new parameter names.
`lbound`	a numeric vector of lower bound values.
`ubound`	a numeric vector of upper bound values.
`indep`	boolean. set parameters in independent submodels.
`strict`	boolean. If TRUE then throw an error when a label does not appear in the model.
`name`	character string. (optional) a new name for the model.

See Also

`omxGetParameters`, `omxAssignFirstParameters`

Examples

```
A <- mxMatrix('Full', 3, 3, labels = c('a','b', NA), free = TRUE, name = 'A')
model <- mxModel(model="testModel7", A, name = 'model')

# set value of cells labelled "a" and "b" to 1 and 2 respectively
model <- omxSetParameters(model, c('a', 'b'), values = c(1, 2))

# set label of cell labelled "a" to "b" and vice versa
model <- omxSetParameters(model, c('a', 'b'), newlabels = c('b', 'a'))

# set label of cells labelled "a" to "b"
model <- omxSetParameters(model, c('a'), newlabels = 'b')

# ensure initial values are the same for each instance of a labeled parameter
model <- omxAssignFirstParameters(model)
```

`omxSymbolTable` *Internal OpenMx algebra operations*

Description

This is an internal table used in the OpenMx backend.

OpenMx *OpenMx: An package for Structural Equation Modeling and Matrix Algebra Optimization*

Description

OpenMx is a package for structural equation modeling, matrix algebra optimization and other statistical estimation problems. Try the example below. We try and have useful help files: for instance help(`mxRun`) to learn more. Also the reference manual

Details

OpenMx solves algebra optimization and statistical estimation problems using matrix algebra. Most users use it for Structural equation modeling.

The core function is `mxModel`, which makes a model. Models are containers for `mxData`, `matrices`, `mxPaths` `algebras`, mxBounds, `confidence intervals`, and `mxConstraints`. Most models require an expectation (see the list below) to calculate the expectations for the model. Models also need a fit function, several of which are built-in (see below). OpenMx also allows user-defined fit functions for purposes not covered by the built-in functions. (e.g., `mxFitFunctionR` or `mxFitFunctionAlgebra`).

Note, for mxModels of `type="RAM"`, the expectation and fit-function are set for you automatically.

Running and summarizing a model

Once built, the resulting mxModel can be run (i.e., optimized) using `mxRun`. This returns the fitted model.

You can summarize the results of the model using `summary`(yourModel)

Additional overview of model making and getting started

The OpenMx manual is online (see references below). However, `mxRun`, `mxModel`, `mxMatrix` all have working examples that will help get you started as well.

The main OpenMx functions are: `mxAlgebra`, `mxBounds`, `mxCI`, `mxConstraint`, `mxData`, `mxMatrix`, `mxModel`, and `mxPath`.

Expectation functions include `mxExpectationNormal`, `mxExpectationRAM`, `mxExpectationLISREL`, and `mxExpectationStateSpace`;

Fit functions include mxFitFunctionML, mxFitFunctionAlgebra, mxFitFunctionRow and mxFitFunctionR.

Datasets built into OpenMx

`OpenMx` comes with over a dozen useful datasets built-in. Discover them using `data(package="OpenMx")`, and open them with, for example, `data("jointdata",package ="OpenMx",verbose= TRUE)`

Please cite the 'OpenMx' package in any publications that make use of it:

Michael C. Neale, Michael D. Hunter, Joshua N. Pritikin, Mahsa Zahery, Timothy R. Brick Robert M. Kirkpatrick, Ryne Estabrook, Timothy C. Bates, Hermine H. Maes, Steven M. Boker. (2016). OpenMx 2.0: Extended structural equation and statistical modeling. *Psychometrika*, **81**, 535–549. DOI: 10.1007/s11336-014-9435-8

Steven M. Boker, Michael C. Neale, Hermine H. Maes, Michael J. Wilde, Michael Spiegel, Timothy R. Brick, Jeffrey Spies, Ryne Estabrook, Sarah Kenny, Timothy C. Bates, Paras Mehta, and John Fox. (2011) OpenMx: An Open Source Extended Structural Equation Modeling Framework. *Psychometrika*, 306-317. DOI:10.1007/s11336-010-9200-6

Steven M. Boker, Michael C. Neale, Hermine H. Maes, Michael J. Wilde, Michael Spiegel, Timothy R. Brick, Ryne Estabrook, Timothy C. Bates, Paras Mehta, Timo von Oertzen, Ross J. Gore, Michael D. Hunter, Daniel C. Hackett, Julian Karch, Andreas M. Brandmaier, Joshua N. Pritikin, Mahsa Zahery, Robert M. Kirkpatrick, Yang Wang, and Charles Driver. (2016) OpenMx 2 User Guide. http://openmx.ssri.psu.edu/docs/OpenMx/latest/OpenMxUserGuide.pdf

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation

Examples

```
library(OpenMx)
data(demoOneFactor)
# ===============================
# = Make and run a 1-factor CFA =
# ===============================

latents  = c("G") # the latent factor
manifests = names(demoOneFactor) # manifest variables to be modeled
# ====================
# = Make the MxModel =
# ====================
m1 <- mxModel("One Factor", type = "RAM",
manifestVars = manifests, latentVars = latents,
mxPath(from = latents, to = manifests),
mxPath(from = manifests, arrows = 2),
mxPath(from = latents, arrows = 2, free = FALSE, values = 1.0),
mxData(cov(demoOneFactor), type = "cov", numObs = 500)
)

# ===============================
# = mxRun it and get a summary! =
# ===============================

m1 = mxRun(m1)
summary(m1)
```

---

ordinalTwinData *Data for ordinal twin model*

Description

Example data for ordinal twin-data modelling. Three variables measured in each twin.

Usage

```
data("ordinalTwinData")
```

Format

A data frame with 139 observations on the following 7 variables.

`zyg`	a numeric vector
`var1_twin1`	a numeric vector
`var2_twin1`	a numeric vector
`var3_twin1`	a numeric vector
`var1_twin2`	a numeric vector
`var2_twin2`	a numeric vector
`var3_twin2`	a numeric vector

Examples

```
data(ordinalTwinData)
str(ordinalTwinData)
```

`Oscillator` *Oscillator Data for Latent Differential Equations*

Description

Data set used in some of OpenMx's examples, for instance the LDE demo. The data were simulated by Steven M. Boker according to a noisy oscillator model.

Usage

```
data("Oscillator")
```

Format

A data frame with 100 observations on the following 1 numeric variable.

`x` Noisy oscillator value

Details

The data appear to be sinusoidal with exponential decay on the amplitude. The rows of data are different times. The column is the variable.

Source

Simulated. Pulled from https://openmx.ssri.psu.edu/thread/144

References

Boker, S., Neale, M., & Rausch, J. (2004). Latent differential equation modeling with multivariate multi-occasion indicators. In *Recent developments on structural equation models* van Montfort, K., Oud, H, and Satorra, A. (Eds.). 151-174. Springer, Dordrecht.

Examples

```
data(Oscillator)
plot(Oscillator$x, type='l')
```

`predict.MxModel` `predict` *method for* `MxModel` *objects*

Description

`predict` method for `MxModel` objects

Usage

```
## S3 method for class 'MxModel'
predict(
  object,
  newdata = NULL,
  interval = c("none", "confidence", "prediction"),
  method = c("ML", "WeightedML", "Regression", "Kalman"),
  level = 0.95,
  type = c("latent", "observed"),
  ...
)
```

Arguments

`object`	an MxModel object from which predictions are desired
`newdata`	an optional `data.frame` object. See details.
`interval`	character indicating what kind of intervals are desired. 'none' gives no intervals, 'confidence', gives confidence intervals, 'prediction' gives prediction intervals.
`method`	character the method used to create the predictions. See details.
`level`	the confidence or predictions level, ignored if not using intervals
`type`	character the type of thing you want predicted: latent variables or manifest variables.
`...`	further named arguments

Details

The `newdata` argument is either a `data.frame` or `MxData` object. In the latter case is replaces the data in the top level model. In the former case, it is passed as the `observed` argument of `mxData` with `type='raw'` and must accept the same further arguments as the data in the model passed in the `object` argument.

The available methods for prediction are 'ML', 'WeightedML', 'Regression', and 'Kalman'. See the help page for `mxFactorScores` for details on the first three of these. The 'Kalman' method uses the Kalman filter to create predictions for state space models.

`rvectorize` *Vectorize By Row*

Description

This function returns the vectorization of an input matrix in a row by row traversal of the matrix. The output is returned as a column vector.

Usage

```
rvectorize(x)
```

Arguments

`x`	an input matrix.

See Also

`cvectorize`, `vech`, `vechs`

Examples

```
rvectorize(matrix(1:9, 3, 3))
rvectorize(matrix(1:12, 3, 4))
```

`summary.MxModel` *Model Summary*

Description

This function returns summary statistics of a model. These include model statistics (parameters, degrees of freedom and likelihood), fit statistics such as AIC, parameter estimates and standard errors (when available), as well as version and timing information and possible warnings about estimates.

Usage

```
## S3 method for class 'MxModel'
summary(object, ..., verbose=FALSE)
```

Arguments

`object`	A MxModel object.
`...`	Any number of named arguments (see below).
`verbose`	Whether to include extra diagnostic information.

Details

mxSummary allows the user to set or override the following parameters of the model:

numObs Numeric. Specify the total number of observations for the model.

numStats Numeric. Specify the total number of observed statistics for the model.

refModels List of MxModel objects. Specify a saturated and independence likelihoods in single argument for testing.

SaturatedLikelihood Numeric or MxModel object. Specify a saturated likelihood for testing.

SaturatedDoF Numeric. Specify the degrees of freedom of the saturated likelihood for testing.

IndependenceLikelihood Numeric or MxModel object. Specify an independence likelihood for testing.

IndependenceDoF Numeric. Specify the degrees of freedom of the independence likelihood for testing.

indep [Deprecated]

verbose logical. Changes the printing style for summary (see Details)

boot.quantile numeric. A vector of quantiles to be used to summarize bootstrap replication.

boot.SummaryType character. One of 'quantile' or 'bcbci'.

Standard Output

The standard output consists of a table of free parameters, tables of model and fit statistics, information on the time taken to run the model, the optimizer used, and the version of OpenMx.

Table of free parameters

Free parameters in the model are reported in a table with columns for the name (label) of the parameter, the matrix, row and col containing the parameter, the parameter estimate itself, and any lower or upper bounds set for the parameter.

note: An exclamation mark ("!") printed after a bound in the lbound or ubound columns indicates that the solution was sufficiently close to the bound that the optimizer could not ignore the bound during its last few iterations.

Additional columns: standard errors and 'A' (asymmetry) warning column

When the information matrix is available, either approximated by the Hessian or from bootstrap resampling, standard errors are reported in the column "Std.Error".

If the information matrix was estimated using finite differences then an additional diagnostic column 'A' is displayed. An exclamation point in the 'A' column indicates that the gradient appears to be asymmetric and the standard error may not accurately reflect the variability of that parameter. As a precaution, it is recommended that you compare the SEs with likelihood-based or bootstrap confidence intervals.

Fit statistics

AIC and BIC `Information Criteria` are reported in a table showing different versions of the information criteria obtained using different penalties. AIC is reported with both a Parameters Penalty and a Degrees of Freedom Penalty version. AIC generally takes the form $Fit + 2 * k$. With the Parameters Penalty version, k is the number of free parameters: $AIC.param = Fit + 2 * param$. With the Degrees of Freedom Penalty, k is minus one times the model degrees of freedom. So the penalty is subtracted instead of added: $AIC.param = Fit - 2 * df$. The Degrees of Freedom penalty was used in Classic Mx. BIC is defined similarly: $Fit + k * log(N)$ where k is either the number of free parameters or minus one times the model degrees of freedom. The Sample-Size-Adjusted BIC is only defined for the parameters penalty: $Fit + k * log((N + 2)/24)$. Similarly, the Sample-Size-Adjusted AIC is $Fit + 2 * k + 2 * k * (k + 1)/(N - k - 1)$. For raw data models, Fit is the minus 2 log likelihood, $-2LL$. For covariance data, Fit is the Chi-squared statistic. The $-2LL$ and saturated likelihood values reported under covariance data are not necessarily meaningful on their own, but their difference yields the Chi-squared value.

Additional fit statistics

When the model has a saturated likelihood, several additional fit indices are printed, including Chi-Squared, CFI, TLI, RMSEA and p RMSEA <= 0.05. For covariance data, saturated and independence models are fitted automatically so all fit indices are reported.

For raw data (to save computational time), the reference models needed to compute these absolute statistics are **not estimated** by default. They are available once you fit reference models.

The `refModels`, `SaturatedLikelihood`, `SaturatedDoF`, `IndependenceLikelihood`, and `IndependenceDoF` arguments can be used to obtain these additional fit statistics. An easy way to make reference models for most cases is provided by the mxRefModels function (see the example given in mxRefModels).

When the `SaturatedLikelihood` or `IndependenceLikelihood` arguments are used, OpenMx attempts to calculate the appropriate degrees of freedom. However, depending on the model, it may sometimes be necessary for the user to also explicitly provide the corresponding `SaturatedDoF` and/or `IndependenceDoF`. Again, for the vast majority of cases, the mxRefModels function handles these situations effectively and conveniently.

Notes on fit statistics

With regard to RMSEA, it is important to note that OpenMx does not currently make a multigroup adjustment that some other structural equation modeling programs make. In particular, we do not multiply the single-group RMSEA by the square root of the number of groups as suggested by Steiger (1998). The RMSEA we use is based on the model likelihood (and degrees of freedom) as compared to the saturated model likelihood (and degrees of freedom), and we do not feel the adjustment is appropriate in this case.

OpenMx does not recommend (and does not compute) some fit indices including GFI, AGFI, NFI, and SRMR. The Goodness of Fit Index (GFI) and Adjusted Goodness of Fit Index (AGFI) are not recommended because they are strongly influenced by sample size and have rather high Type I error rates (Sharma, Mukherjee, Kumar, & Dillon, 2005). The Normed Fit Index (NFI) has no penalty for model complexity. That is, adding more parameters to a model always improves the NFI, regardless of how useful those parameters are. Because the Non-Normed Fit Index (NNFI), also known as the Tucker-Lewis Index (TLI), does adjust for model complexity it is used instead. Lastly, the Standardized Root Mean Square Residual (SRMR) is not reported because it (1) only applies to covariance models, having no direct extension to missing data, (2) has no penalty for model complexity, similar to the NFI, and (3) is positively biased (Hu & Bentler, 1999).

verbose

The `verbose` argument changes the printing style for the `summary` of a model. When `verbose=FALSE`, a relatively minimal amount of information is printed: the free parameters, the likelihood, and a few fit indices. When `verbose=TRUE`, the compute plan, data summary, and additional timing information are always printed. Moreover, available fit indices are printed regardless of whether or not they are defined. The undefined fit indices are printed as `NA`. In addition, the condition number of the information matrix, and the maximum absolute gradient may also be shown.

note: The `verbose` argument only changes the printing style, all of the same information is calculated and exists in the output of `summary`. More information is displayed when `verbose=TRUE`, and less when `verbose=FALSE`.

Summary for bootstrap replications

Summarization of bootstrap replications is controlled by two options: ‘boot.quantile’ and ‘boot.SummaryType’. To obtain a two-sided 95% width confidence interval, use `boot.quantile=c(.025,.975)`. Options for ‘boot.SummaryType’ are ‘quantile’ (using R’s standard `stats::quantile` function) and ‘bcbci’ for bias-corrected bootstrap confidence intervals. The latter, ‘bcbci’, is the default due to its superior theoretical properties.

References

The OpenMx User’s guide can be found at https://openmx.ssri.psu.edu/documentation/.

Hu, L., & Bentler, P. M. (1999). Cutoff criteria for fit indexes in covariance structure analysis: Conventional criteria versus new alternatives. *Structural Equation Modeling, 6,* 1-55.

Savalei, V. (2012). The relationship between root mean square error of approximation and model misspecification in confirmatory factor analysis models. *Educational and Psychological Measurement, 72*(6), 910-932.

Sharma, S., Mukherjee, S., Kumar, A., & Dillon, W.R. (2005). A simulation study to investigate the use of cutoff values for assessing model fit in covariance structure models. *Journal of Business Research, 58,* 935-43.

Steiger, J. H. (1998). A note on multiple sample extensions of the RMSEA fit index. *Structural Equation Modeling: A Multidisciplinary Journal, 5(4)*, 411-419. DOI: 10.1080/10705519809540115

See Also

mxBootstrap mxCI as.statusCode

Examples

```
library(OpenMx)
data(demoOneFactor)  # load the demoOneFactor dataframe
manifests <- names(demoOneFactor) # set the manifest to the 5 demo variables
latents <- c("G")  # define 1 latent variable
model <- mxModel(model="One Factor", type="RAM",
    manifestVars = manifests,
    latentVars = latents,
    mxPath(from = latents, to=manifests, labels = paste("b", 1:5, sep = "")),
    mxPath(from = manifests, arrows = 2, labels = paste("u", 1:5, sep = "")),
    mxPath(from = latents, arrows = 2, free = FALSE, values = 1.0),
    mxData(cov(demoOneFactor), type = "cov", numObs = 500)
)
model <- mxRun(model) # Run the model, returning the result into model

# Show summary of the fitted model
summary(model)

# Compute the summary and store in the variable "statistics"
statistics <- summary(model)

# Access components of the summary
statistics$parameters
statistics$SaturatedLikelihood

# Specify a saturated likelihood for testing
summary(model, SaturatedLikelihood = -3000)

# Add a CI and view it in the summary
model = mxRun(mxModel(model=model, mxCI("b5")), intervals = TRUE)
summary(model)
```

`tr` *trace*

Description

This function returns the trace of an n-by-n square matrix x, defined to be the sum of the elements on the main diagonal (the diagonal from the upper left to the lower right).

Usage

```
tr(x)
```

Arguments

`x`	an input matrix. Must be square

Details

The input matrix must be square.

See Also

`vech`, `rvectorize`, `cvectorize`

Examples

```
tr(matrix(1:9, 3, 3))
tr(matrix(1:12, 3, 4))
```

`twinData` *Australian twin sample biometric data.*

Description

Australian twin data with 3,808 observations on the 12 variables including body mass index (BMI) assessed in both MZ and DZ twins.

Questionnaires were mailed to 5,967 pairs age 18 years and over. These data consist of completed questionnaires returned by both members of 3,808 (64 percent) pairs. There are two cohort blocks in the data: a younger group (zyg 1:5), and an older group (zyg 6:10)

It is a wide dataset, with two individuals per line. Families are identified by the variable “fam”.

Data include zygosity (zyg), along with heights in meters, weights in kg, and the derived variables BMI in kg/m^2 (stored as “htwt1” and “htwt2”), as well as the 7 times the natural log of this variable, stored as bmi1 and bmi2. The logged values are more closely normally distributed while scaling by 7 places them into a similar range to the original variable.

For convenience, zyg is broken out into separate “zygosity” and “cohort” factors. “zygosity” is coded as a factor with 5-levels: MZFF, MZMM, DZFF, DZMM, DZOS. DZOS are in Female/Male wide order.

Usage

```
data(twinData)
```

Format

A data frame with 3808 observations on the following 12 variables.

`fam` The family ID

`age` Age in years (of both twins)

`zyg` Code for zygosity and cohort (see details)

`part` A numeric vector

`wt1` Weight of twin 1 (kg)

`wt2` Weight of twin 2 (kg)

`ht1` Height of twin 1 (m)

`ht2` Height of twin 2 (m)

`htwt1` Raw BMI of twin 1 (kg/m^2)

`htwt2` Raw BMI of twin 2 (kg/m^2)

`bmi1` 7*log(BMI) of twin 1

`bmi2` 7*log(BMI) of twin 2

`cohort` Either “younger” or “older”

`zygosity` Zygosity factor with levels: MZFF, MZMM, DZFF, DZMM, DZOS

`age1` Age of Twin 1

`age2` Age of Twin 2

Details

“zyg” codes twin-zygosity as follows: 1 == MZFF (i.e MZ females) 2 == MZMM (i.e MZ males) 3 == DZFF 4 == DZMM 5 == DZOS opposite sex pairs

Note: zyg 6:10 are for an older cohort in the sample. So: 6 == MZFF (i.e MZ females) 7 == MZMM (i.e MZ males) 8 == DZFF 9 == DZMM 10 == DZOS opposite sex pairs

The “zygosity” and “cohort” variables take care of this for you (conventions differ).

References

Martin, N. G. & Jardine, R. (1986). Eysenck’s contribution to behavior genetics. In S. Modgil & C. Modgil (Eds.), *Hans Eysenck: Consensus and Controversy.* Falmer Press: Lewes, Sussex.

Martin, N. G., Eaves, L. J., Heath, A. C., Jardine, R., Feingold, L. M., & Eysenck, H. J. (1986). Transmission of social attitudes. *Proceedings of the National Academy of Science*, **83**, 4364-4368.

Examples

```
data(twinData)
str(twinData)
plot(wt1 ~ wt2, data = twinData)
selVars = c("bmi1", "bmi2")
mzData <- subset(twinData, zyg == 1, selVars)
dzData <- subset(twinData, zyg == 3, selVars)
```

```
# equivalently
mzData <- subset(twinData, zygosity == "MZFF", selVars)

# Disregard sex, pick older cohort
mz <- subset(twinData, zygosity %in% c("MZFF","MZMM") & cohort == "older", selVars)
```

`twin_NA_dot` *Twin biometric data (Practice cleaning: "." for missing data, wrong data types etc.)*

Description

Data set used in some of OpenMx's examples.

Usage

```
data("twin_NA_dot")
```

Format

A data frame with 3808 observations on the following variables.

`fam` Family ID variable

`age` Age of the twin pair. Range: 17 to 88, coded as factor

`zyg` Integer codes for zygosity and gender combinations

`part` Cohort

`wt1` Weight in kilograms for twin 1 (this and following have "." embedded as NA...)

`wt2` Weight in kilograms for twin 2

`ht1` Height in meters for twin 1

`ht2` Height in meters for twin 2

`htwt1` Product of ht and wt for twin 1

`htwt2` Product of ht and wt for twin 2

`bmi1` Body Mass Index for twin 1

`bmi2` Body Mass Index for twin 2

Details

Same as myTwinData but has . as the missing data value instead of NA.

Source

Timothy Bates

References

The OpenMx User's guide can be found at https://openmx.ssri.psu.edu/documentation/.

Examples

```
data(twin_NA_dot)
summary(twin_NA_dot)
# Note that all variables are treated as factors because of the missing data coding.
```

---

vec2diag *Create Diagonal Matrix From Vector*

Description

Given an input row or column vector, `vec2diag` returns a diagonal matrix with the input argument along the diagonal.

Usage

```
vec2diag(x)
```

Arguments

`x`	a row or column vector.

Details

Similar to the function `diag`, except that the input argument is always treated as a vector of elements to place along the diagonal.

See Also

`diag2vec`

Examples

```
vec2diag(matrix(1:4, 1, 4))
vec2diag(matrix(1:4, 4, 1))
```

`vech` *Half-vectorization*

Description

This function returns the half-vectorization of an input matrix as a column vector.

Usage

```
vech(x)
```

Arguments

`x`	an input matrix.

Details

The half-vectorization of an input matrix consists of the elements in the lower triangle of the matrix, including the elements along the diagonal of the matrix, as a column vector. The column vector is created by traversing the matrix in column-major order.

See Also

`vech2full`, `vechs`, `rvectorize`, `cvectorize`

Examples

```
vech(matrix(1:9, 3, 3))
vech(matrix(1:12, 3, 4))
```

`vech2full` *Inverse Half-vectorization*

Description

This function returns the symmetric matrix constructed from a half-vectorization.

Usage

```
vech2full(x)
```

Arguments

`x`	an input single column or single row matrix.

Details

The half-vectorization of an input matrix consists of the elements in the lower triangle of the matrix, including the elements along the diagonal of the matrix, as a column vector. The column vector is created by traversing the matrix in column-major order. The inverse half-vectorization takes a vector and reconstructs a symmetric matrix such that `vech2full(vech(x))` is identical to `x` if `x` is symmetric.

Note that very few vectors have the correct number of elements to construct a symmetric matrix. For example, vectors with 1, 3, 6, 10, and 15 elements can be used to make a symmetric matrix, but none of the other numbers between 1 and 15 can. An error is thrown if the number of elements in `x` cannot be used to make a symmetric matrix.

See Also

`vechs2full`, `vech`, `vechs`, `rvectorize`, `cvectorize`

Examples

```
vech2full(1:10)

matrix(1:16, 4, 4)
vech(matrix(1:16, 4, 4))
vech2full(vech(matrix(1:16, 4, 4)))
```

`vechs` *Strict Half-vectorization*

Description

This function returns the strict half-vectorization of an input matrix as a column vector.

Usage

```
vechs(x)
```

Arguments

`x`	an input matrix.

Details

The half-vectorization of an input matrix consists of the elements in the lower triangle of the matrix, excluding the elements along the diagonal of the matrix, as a column vector. The column vector is created by traversing the matrix in column-major order.

See Also

`vech`, `rvectorize`, `cvectorize`

Examples

```
vechs(matrix(1:9, 3, 3))
vechs(matrix(1:12, 3, 4))
```

`vechs2full` *Inverse Strict Half-vectorization*

Description

This function returns the symmetric matrix constructed from a strict half-vectorization.

Usage

```
vechs2full(x)
```

Arguments

`x`	an input single column or single row matrix.

Details

The strict half-vectorization of an input matrix consists of the elements in the lower triangle of the matrix, excluding the elements along the diagonal of the matrix, as a column vector. The column vector is created by traversing the matrix in column-major order. The inverse strict half-vectorization takes a vector and reconstructs a symmetric matrix such that `vechs2full(vechs(x))` is equal to `x` with zero along the diagonal if `x` is symmetric.

Note that very few vectors have the correct number of elements to construct a symmetric matrix. For example, vectors with 1, 3, 6, 10, and 15 elements can be used to make a symmetric matrix, but none of the other numbers between 1 and 15 can. An error is thrown if the number of elements in `x` cannot be used to make a symmetric matrix.

See Also

`vech2full`, `vech`, `vechs`, `rvectorize`, `cvectorize`

Examples

```
vechs2full(1:10)

matrix(1:16, 4, 4)
vechs(matrix(1:16, 4, 4))
vechs2full(vechs(matrix(1:16, 4, 4)))
```

Index