Package 'lixoftConnectors'

October 11, 2024

Type Package

Title R connectors for Lixoft Suite (@Lixoft)

Version 2023.1

Date 2023-01-01

Author LIXOFT

Maintainer LIXOFT <support@lixoft.com>

Depends R (>= 3.0.0), RJSONIO

Imports ggplot2, stats, utils, gridExtra, gtable, grDevices

Encoding UTF-8

Collate apiTools.R displayTools.R lixoftEnvironment.R apiManager.R
commons-tools.R commons-maths.R commons-data.R
commons-projectManagement.R commons-settings.R
commons-scenario.R commons-results.R commons-stratify.R
commons-reporting.R mlx-settings.R mlx-scenario.R
mlx-populationParameters.R mlx-individualModel.R
mlx-covariateModel.R mlx-observationModel.R mlx-results.R
mlx-core.R mlx-modelBuilding.R mlx-assessment.R pkx-data.R
pkx-settings.R pkx-scenario.R pkx-results.R
smlx-projectManagement.R smlx-settings.R smlx-definition.R
smlx-scenario.R smlx-results.R plot-bins.R plot-checks.R
plot-common-charts-data.R plot-common-data-plots.R
plot-common-tools.R plot-data-tools.R plot-lixoft-checks.R
plot-mlx-charts-data-vpc.R plot-mlx-charts-data.R
plot-mlx-indivParameters-plot.R plot-mlx-prediction-plot.R
plot-mlx-predictive-checks-plot.R
plot-mlx-charts-data-diagnosis.R plot-mlx-diagnosis.R
plot-mlx-tools.R plot-pkx-ca-plot.R plot-pkx-charts-data.R
plot-pkx-nca-plot.R plot-pkx-be-plot.R plot-pkx-tools.R
plot-preferences.R plot-stratification.R plot-tools.R
plot-tte.R plot-utils.R

Description This package provides R connectors for Monolix - PKanalix - Simulx (@Lixoft) to create, edit and run Mlxtran projects from R command prompt.

License [BSD_2_clause + file LICENSE] (license lixoft)

RoxygenNote 7.2.1

NeedsCompilation no

R topics documented:

`.computeBins` *Generate Bins*

Description

Generate Bins

Usage

```
.computeBins(times, split = NULL, binsSettings = NULL)
```

Arguments

`times`	(*vector(double)*) vector of times.
`split`	(*vector*) split category associated with data (no split by default).
`binsSettings`	(*binSettingsClass*) (*optional*) a list of settings for bins computation:

- `is.fixedBins` (*bool*) If TRUE, specify manually bin vector (default FALSE).
- `fixedBins` (*double*) Define manually a vector of bins. To use when 'is.fixedBins' is set to TRUE.
- `criteria` (*string*) Bining criteria, one of 'equalwidth', 'equalsize', or 'least-square' methods. (default leastsquare).
- `is.fixedNbBins` (*bool*) If TRUE define a fixed number of bins, else define a range for automatic selection (default FALSE).
- `nbBins` (*int*) Define a fixed number of bins (default 10).
- `binRange` (*vector(int, int)*) Define a range for the number of bins (default c(5, 100)).
- `nbBinData` (*vector(int, int)*) Define a range for the number of data points per bin (default c(10, 200)).

Value

bins dataframe

Examples

```
## Not run:
  .computeBins(seq_len(100), binsSettings = list(nbBins = 3))

## End(Not run)
```

.computeCdfOnGrid *Compute CDF on grid*

Description

Compute CDF on grid

Usage

```
.computeCdfOnGrid(data, grid, ...)
```

Arguments

data	(*vector<double>*) Input data.
grid	(*vector<double>*) Computation grid (must be strictly sorted).
...	[optional] • normalize (*bool*) [optional] (default = FALSE) • outliers (*bool*) [optional] (default = FALSE) Should outliers be taken into account or not.

Value

A vector of doubles.

.computeDistributionFunctions *Compute PDF and CDF*

Description

Compute PDF and CDF

Usage

```
.computeDistributionFunctions(data, bounds)
```

Arguments

data	(*vector<double>*) Input data.
bounds	(*pair<double> OR 'normalLaw'*) CDF bounds.

Value

A list(cdf = vector<double>, pdf = vector<double>)

.computePercentiles *Compute percentiles*

Description

Compute percentiles

Usage

```
.computePercentiles(data, quantiles)
```

Arguments

data	(*vector*<*double*>) Input data.
quantiles	(*vector*<*double*>) Quantiles.

Value

A vector of doubles.

.computeSpline *Compute spline*

Description

Compute spline

Usage

```
.computeSpline(x, y, ...)
```

Arguments

x	(*vector*<*double*>) Input data abscissa.
y	(*vector*<*double*>) Input data ordinates.
...	[optional] • gridSize (*int*) Uniform grid size (default = 100)

Value

A vector of doubles.

.computeStatistics *Compute statistics*

Description

Compute statistics

Usage

```
.computeStatistics(
  data,
  mode = "arithmetic",
  errorMethod = "standardDeviation"
)
```

Arguments

data	(*vector<double>*) Input data.
mode	(*string*) Statistics computation mode: "arithmetic" (default) | "geometric".
errorMethod	(*string*) Error computation method: "standardDeviation" (default) | "standardError".

Value

A list of doubles corresponding to the requested statistics.

.computeSurvivalCurves *Compute survival curve and average event number*

Description

Compute survival curve and average event number

Usage

```
.computeSurvivalCurves(data, grid, ...)
```

Arguments

data	(*vector< vector< pair<double> > >*) Input data.
grid	(*vector<double>*) Computation grid (must be strictly sorted).
...	[optional] • exact (*bool*) (default = TRUE)

Value

A list(survivalFunction = vector<double>, averageEventNumber = vector<double>, censoredData = vector< pair< vector<int>, pair<double> > >).

.computeVisualGuides *Compute visual guides*

Description

Compute linear regression, spline and correlation coefficients on the same abscissa.

Usage

```
.computeVisualGuides(x, y)
```

Arguments

x	(*vector<double>*) Input data abscissa.
y	(*vector<double>*) Input data ordinates.

Value

A list(abscissa = vector<double>, spline = vector<double>, regression = vector<double>, correlation = double).

addAdditionalCovariate *[Monolix - PKanalix] Add an additional covariate*

Description

Create an additional covariate for stratification purpose. Notice that these covariates are available only if they are not contant through the dataset.
Available column transformations are:

[continuous]	'firstDoseAmount'	(first dose amount)
[continuous]	'doseNumber'	(dose number)
[discrete]	'administrationType'	(admninistration type)
[discrete]	'administrationSequence'	(administration sequence)
[discrete]	'dosingDesign'	(dose multiplicity)
[continuous]	'observationNumber'	(observation number per individual, for a given observation type)

Usage

```
addAdditionalCovariate(transformation, base = "", name = "")
```

Arguments

transformation	(*string*) applied transformation.
base	(*string*) [optional] base data on which the transformation is applied.
name	(*string*) [optional] name of the covariate.

See Also

deleteAdditionalCovariate

Examples

```
## Not run:
addAdditionalCovariate("firstDoseAmount")
addAdditionalCovariate(transformation = "observationNumberPerIndividual", headerName = "CONC")

## End(Not run)
```

---

addCategoricalTransformedCovariate
[Monolix] Add categorical transformed covariate

Description

Create a new categorical covariate by transforming an existing one. Transformed covariates cannot be use to produce new covariates. Call getCovariateInformation to know which covariates can be transformed.

Usage

```
addCategoricalTransformedCovariate(...)
```

Arguments

...	A list of comma-separated pairs {transformedCovariateName = { from = (*array<(string)>*)["basicCovariateNames"], transformed = (*array<array<string»*)"transformation"} }

See Also

getCovariateInformation removeCovariate

Examples

```
## Not run:
addCategoricalTransformedCovariate( Country2 = list(reference = "A1",
          from = "Country", transformed = list( A1 = c("A","B"), A2 = c("C")))
          )

## End(Not run)
```

addContinuousTransformedCovariate *[Monolix] Add continuous transformed covariate*

Description

Create a new continuous covariate by transforming an existing one. Transformed covariates cannot be use to produce new covariates. Call `getCovariateInformation` to know which covariates can be transformed.

Usage

```
addContinuousTransformedCovariate(...)
```

Arguments

...	A list of comma-separated pairs {transformedCovariateName = (*string*)"transformation"}

See Also

`getCovariateInformation` `removeCovariate`

Examples

```
## Not run:
addContinuousTransformedCovariate( tWt2 = "3*exp(Wt)"  )

## End(Not run)
```

addGroup *[Simulx] Add simulation group*

Description

Add a new simulation group.

Usage

```
addGroup(group)
```

Arguments

group	(*string*) Name of the group to add.

Details

Simulation groups can be added to the simulation as in Simulx GUI. By default, the elements of the newly added group are the same as the first simulation group. To check which elements have been set for this group, please use getGroups. To change a group element, use setGroupElement.

Note: when a Simulx project is created, a first group is created by default with the name "simulationGroup1".

See Also

getGroups

Examples

```
# create two groups with different treatments
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "4.exploration", "PKPD_exploration.smlx")
loadProject(project_name)
addGroup("simulationGroup2")
setGroupElement("simulationGroup2", elements = "Dose_4000")
```

---

addMixture *[Monolix] Add mixture to the covariate model*

Description

Add a new latent covariate to the current model giving its name and its modality number.

Usage

```
addMixture(...)
```

Arguments

...	A list of comma-separated pairs {latentCovariateName = (*int*)modalityNumber}

See Also

getCovariateInformation removeCovariate

Examples

```
## Not run:
addMixture(lcat = 2)

## End(Not run)
```

---

applyFilter *[Monolix - PKanalix] Apply filter*

Description

Apply a filter on the current data.

Usage

```
applyFilter(filter, name = "")
```

Arguments

`filter`	(*list< list< action = "headerName-comparator-value" > >* or "complement") filter definition. Existing actions are "selectLines", "selectIds", "removeLines" and "removeIds". First vector level is for set unions, the second one for set intersection. It is possible to give only a list of actions if there is only no high-level union.
`name`	(*string*) [optional] created data set name. If not defined, the default name is "currentDataSet_filtered".

Details

The possible actions are line selection (selectLines), line removal (removeLines), Ids selection (selectIds) or removal (removeIds).
The selection is a string containing the header name, a comparison operator and a value
selection = <string> "headerName*-comparator**-value" (ex: "id=='100'", "WEIGHT<70", "SEX!='M'")
Notice that :
- The headerName corresponds to the data set header or one of the header aliases defined in MONOLIX software preferences
- The comparator possibilities are "==", "!=" for all types of value and "<=", "<", ">=", ">" only for numerical types

Syntax:
* apply a simple filter:
applyFilter(filter = list(act = sel)), e.g. applyFilter(filter = list(removeIds = "WEIGHT<50"))
=> apply a filter with the action act on the selection sel. In this example, we apply a filter that removes all subjects with a weight less than 50.
* apply a filter with several concurrent conditions, i.e AND condition:
applyFilter(list(act1 = sel1, act2 = sel2)), e.g. applyFilter(filter = list(removeIds = "WEIGHT<50", removeIds = " AGE<20"))
=> apply a filter with both the action act1 on sel1 AND the action act2 on sel2. In this example, we apply a filter that removes all subjects with a weight less than 50 and an age less than 20. It corresponds to the intersecton of the subjects with a weight less than 50 and the subjects with an age less than 20.
* apply a filter with several non-concurrent conditions, i.e OR condition:
applyFilter(filter = list(list(act1 = sel1), list(act2 = sel2))), e.g. applyFilter(filter = list(list(removeIds = "WEIGHT<50"),list(removeIds = " AGE<20")))
=> apply a filter with the action act1 on sel1 OR the action act2 on sel2. In this example, we apply a filter that removes all subjects with a weight less than 50 and an age less than 20.
It corresponds to the union of the subjects with a weight less than 50 and the subjects with an age less than 20.
* It is possible to have any combination:
applyFilter(filter = list(list(act1 = sel1), list(act2 = sel2, act3 = sel3))) <=> act1,sel1 OR (act2,sel2 AND act3,sel3)
* It is possible to apply the complement of an existing filter:
applyFilter(filter = "complement")

See Also

`getAvailableData` `createFilter` `removeFilter`

Examples

```
## Not run:
-------------------------------------------------------------------------------------------
LINE [ int ]
applyFilter( filter = list(removeLines = "line>10") ) # keep only the 10th first rows
-------------------------------------------------------------------------------------------
ID [ string | int ]
If there are only integer identifiers within the data set, ids will be considered as integers. On the contrary, t
applyFilter( filter = list(selectIds = "id==100") ) # select the subject called '100'
applyFilter( filter = list(list(removeIds = "id!='id_2'")) ) # select all the subjects excepted the one called '
-------------------------------------------------------------------------------------------
ID INDEX [int]
applyFilter( filter = list(list(removeIds = "idIndex!=2"), list(selectIds = "id<5")) ) # select the 4 first subj
-------------------------------------------------------------------------------------------
OCC [ int ]
applyFilter( filter = list(selectIds = "occ1==1", removeIds = "occ2!=3") ) # select the subjects whose first occ
-------------------------------------------------------------------------------------------
TIME [ double ]
applyFilter( filter = list(removeIds='TIME>120') ) # remove the subjects who have time over 120
applyFilter( filter = list(selectLines='TIME>120') ) # remove the all the lines where the time is over 120
-------------------------------------------------------------------------------------------
OBSERVATION [ double ]
applyFilter( filter = list(selectLines = "CONC>=5.5", removeLines = "CONC>10")) # select the lines where CONC va
applyFilter( filter = list(removeIds = "CONC<0") ) # remove subjects who have negative CONC values
applyFilter( filter = list(removeIds = "E==0") ) # remove subjects for who E equals 0
-------------------------------------------------------------------------------------------
OBSID [ string ]
applyFilter( filter = list(removeIds = "y1==1") ) # remove subject who have at least one observation for y1
applyFilter( filter = list(selectLines = "y1!=2") ) # select all lines corresponding to observations exepected t
-------------------------------------------------------------------------------------------
AMOUNT [ double ]
applyFilter( filter = list(selectIds = "AMOUT==10") ) # select subjects who have a dose equals to 10
-------------------------------------------------------------------------------------------
INFUSION RATE AND INFUSION DURATION [ double ]
applyFilter( filter = list(selectIds = "RATE<10") ) # select subjects who have dose with a rate less than 10
-------------------------------------------------------------------------------------------
COVARIATE [ string (categorical) | double (continuous) ]
applyFilter( filter = list(selectIds = "SEX==M", selectIds = "WEIGHT<80") ) # select subjects who are men and wh
-------------------------------------------------------------------------------------------
REGERSSOR [ double ]
applyFilter( filter = list(selectLines = "REG>10") ) # select the lines where the regressor value is over 10
-------------------------------------------------------------------------------------------
COMPLEMENT
applyFilter(origin = "data_filtered", filter = "complement" )

## End(Not run)
```

`computeBins` *Compute bins*

Description

Compute bins values, middles, and data repartition over bins. Available options are:

"criteria"	(*string*)	Bins criteria: "equalwidth", "equalsize" or "leastsquare" (default).
"useFixedNb"	(*bool*)	TRUE to fix the number of bins, FALSE (default) to estimate it.
"fixedNb"	(*int*)	Fixed number of bins (default: 10).
"estimatedNb"	(*pair<int,int>*)	Bounds for bins number estimation (default: [5,30]).
"nbBinData"	(*pair<int,int>*)	Minimum and maximum number of data per bin for bins number estimation (default:

Usage

```
computeBins(data, options = list())
```

Arguments

data	(*vector<double>*) Input data.
options	(*list*) [optional] Computation options.

Value

A list bins values ("values"), middles ("middles") and the actual number of data per bin ("repartition").

Examples

```
## Not run:
computeBins(data = c(1, 1.25, 2.5, 5, 5.5, 5.75, 7.5, 15, 16.5), options = list(nbBinData = c(1,10)))

## End(Not run)
```

---

computeChartsData *[Monolix - PKanalix - Simulx] Compute the charts data*

Description

Compute (if needed) and export the charts data of a given plot or, if not specified, all the available project plots.

Usage

```
computeChartsData(plot = NULL, output = NULL, exportVPCSimulations = NULL)
```

Arguments

plot	(*character*) [optional][Monolix] Plot type. If not specified, all the available project plots will be considered. Available plots: bivariatedataviewer, covariateviewer, outputplot, indfits, obspred, residualsscatter, residualsdistribution, vpc, npc, predictiondistribution, parameterdistribution, randomeffects, covariancemodeldiagnosis, covariatemodeldiagnosis, likelihoodcontribution, fisher, saemresults, condmeanresults, likelihoodresults.
output	(*character*) [optional][Monolix] Plotted output (depending on the software, it can represent an observation, a simulation output, ...). By default, all available outputs are considered.

exportVPCSimulations
: (*bool*) [optional][Monolix] Should VPC simulations be exported if available. Equals FALSE by default. NOTE: If 'plot" argument is not provided, 'output' and "task' arguments are ignored.

Details

computeChartsData can be used to compute and export the charts data for plots available in the graphical user interface as in Monolix, PKanalix or Simulx, when you export > export charts data.

The exported charts data is saved as txt files in the result folder, in the ChartsData subfolder.

Notice that it does not impact the current scenario.

To get a ggplot equivalent to the plot in the GUI, but customizable in R with the ggplot2 library, better use one of the plot... functions available in the connectors for Monolix and PKanalix (not available for Simulx). To get the charts data for one of these plot functions as a dataframe, you can use getChartsData.

See Also

getChartsData

Examples

```
## Not run:
computeChartsData() # Monolix - PKanalix - Simulx
computeChartsData(plot = "vpc", output = "y1") # Monolix

## End(Not run)
```

---

computePredictions *[Monolix] Compute predictions from the structural model*

Description

[**MlxCore**][*Prediction*]
Call the monolix prediction function to compute observation models values on observation times for each subject of a set of individuals.

Usage

```
computePredictions(individualParameters, individualIds = NULL)
```

Arguments

individualParameters
: Individual parameter values associated to each one of the individual parameters present in the project, for a set of subjects which must be coherent with the list of individuals ids passed in "individualIds" field (ie, this length of the subject set must be the sum of the subject number of all the individuals selected by the "individualIds" field). This input field accepts a dataframe indexed by individual parameter names (columns) and subject indexes (rows).

individualIds
: [optional] *vector<int>* Ids of the individuals for which observation models should be computed. By default, all the individuals present in the project are considered.

Value

For each prediction names, a vector giving the computed prediction at observation times for each subject.

See Also

getIndividualParameterModel getEstimatedIndividualParameters

Examples

```
## Not run:
ids = c(1,4)
individualValuesForAllIndiv = getEstimatedIndividualParameters()$saem

predictions = computePredictions( individualParameters = individualValuesForAllIndiv[ids,],
                                  individualIds = ids )

predictions
  -> $Cc
       [3.8,6.75,...,3.4,5.1,...]
       |    id=1    |    id=4   |

## End(Not run)
```

---

createFilter *[Monolix - PKanalix] Create filter*

Description

Create a new filtered data set by applying a filter on an existing one and/or complementing it.

Usage

```
createFilter(filter, name = "", origin = "")
```

Arguments

filter	(*list< list< action = "headerName-comparator-value" > >* or "complement") [optional] filter definition. Existing actions are "selectLines", "selectIds", "removeLines" and "removeIds". First vector level is for set unions, the second one for set intersection. It is possible to give only a list of actions if there is only no high-level union.
name	(*string*) [optional] created data set name. If not defined, the default name is "currentDataSet_filtered".
origin	(*string*) [optional] name of the data set to be filtered. The current one is used by default.

Details

The possible actions are line selection (selectLines), line removal (removeLines), Ids selection (selectIds) or removal (removeIds).
The selection is a string containing the header name, a comparison operator and a value
selection = <string> "headerName*-comparator**-value" (ex: "id==’100’", "WEIGHT<70", "SEX!=’M’")
Notice that :
- The headerName corresponds to the data set header or one of the header aliases defined in MONOLIX software preferences
- The comparator possibilities are "==", "!=" for all types of value and "<=", "<", ">=", ">" only for numerical types

Syntax:
* create a simple filter:
createFilter(filter = list(act = sel)), e.g. createFilter(filter = list(removeIds = "WEIGHT<50"))
=> create a filter with the action act on the selection sel. In this example, we create a filter that removes all subjects with a weight less than 50.
* create a filter with several concurrent conditions, i.e AND condition:
createFilter(list(act1 = sel1, act2 = sel2)), e.g. createFilter(filter = list(removeIds = "WEIGHT<50", removeIds = " AGE<20"))
=> create a filter with both the action act1 on sel1 AND the action act2 on sel2. In this example, we create a filter that removes all subjects with a weight less than 50 and an age less than 20. It corresponds to the intersecton of the subjects with a weight less than 50 and the subjects with an age less than 20.
* create a filter with several non-concurrent conditions, i.e OR condition:
createFilter(filter = list(list(act1 = sel1), list(act2 = sel2))), e.g. createFilter(filter = list(list(removeIds = "WEIGHT<50"),list(removeIds = " AGE<20")))
=> create a filter with the action act1 on sel1 OR the action act2 on sel2. In this example, we create a filter that removes all subjects with a weight less than 50 and an age less than 20.
It corresponds to the union of the subjects with a weight less than 50 and the subjects with an age less than 20.
* It is possible to have any combinaison:
createFilter(filter = list(list(act1 = sel1), list(act2 = sel2, act3 = sel3))) <=> act1,sel1 OR (act2,sel2 AND act3,sel3)
* It is possible to create the complement of an existing filter:
createFilter(filter = "complement")

See Also

applyFilter

Examples

```
## Not run:
-------------------------------------------------------------------------------------
LINE [ int ]
createFilter( filter = list(removeLines = "line>10") ) # keep only the 10th first rows
-------------------------------------------------------------------------------------
ID [ string | int ]
If there are only integer identifiers within the data set, ids will be considered as integers. On the contrary, t
createFilter( filter = list(selectIds = "id==100") ) # select the subject called '100'
createFilter( filter = list(list(removeIds = "id!='id_2'")) ) # select all the subjects excepted the one called
-------------------------------------------------------------------------------------
ID INDEX [int]
```

```
createFilter( filter = list(list(removeIds = "idIndex!=2"), list(selectIds = "id<5")) ) # select the 4 first sub
-------------------------------------------------------------------------------------------
OCC [ int ]
createFilter( filter = list(selectIds = "occ1==1", removeIds = "occ2!=3") ) # select the subjects whose first oc
-------------------------------------------------------------------------------------------
TIME [ double ]
createFilter( filter = list(removeIds='TIME>120') ) # remove the subjects who have time over 120
createFilter( filter = list(selectLines='TIME>120') ) # remove the all the lines where the time is over 120
-------------------------------------------------------------------------------------------
OBSERVATION [ double ]
createFilter( filter = list(selectLines = "CONC>=5.5", removeLines = "CONC>10")) # select the lines where CONC v
createFilter( filter = list(removeIds = "CONC<0") ) # remove subjects who have negative CONC values
createFilter( filter = list(removeIds = "E==0") ) # remove subjects for who E equals 0
-------------------------------------------------------------------------------------------
OBSID [ string ]
createFilter( filter = list(removeIds = "y1==1") ) # remove subject who have at least one observation for y1
createFilter( filter = list(selectLines = "y1!=2") ) # select all lines corresponding to observations exepected
-------------------------------------------------------------------------------------------
AMOUNT [ double ]
createFilter( filter = list(selectIds = "AMOUT==10") ) # select subjects who have a dose equals to 10
-------------------------------------------------------------------------------------------
INFUSION RATE AND INFUSION DURATION [ double ]
createFilter( filter = list(selectIds = "RATE<10") ) # select subjects who have dose with a rate less than 10
-------------------------------------------------------------------------------------------
COVARIATE [ string (categorical) | double (continuous) ]
createFilter( filter = list(selectIds = "SEX==M", selectIds = "WEIGHT<80") ) # select subjects who are men and w
-------------------------------------------------------------------------------------------
REGERSSOR [ double ]
createFilter( filter = list(selectLines = "REG>10") ) # select the lines where the regressor value is over 10
-------------------------------------------------------------------------------------------
COMPLEMENT
createFilter(origin = "data_filtered", filter = "complement" )

## End(Not run)
```

`defineCovariateElement` *[Simulx] Define covariate element*

Description

Define or edit a covariate element. Covariate elements are defined and used for simulation as in Simulx GUI. As for other elements, the covariate elements can be defined or imported, and they are saved with the Simulx project if calling saveProject. Once a covariate element is defined, it needs to be added to a simulation group with setGroupElement to be used in simulation.

Usage

```
defineCovariateElement(name, element)
```

Arguments

`name`	(*string*) Element name.
`element`	(*string* or *dataFrame*) Element definition from external file path or data frame with covariates as columns.

Details

Covariate elements can be defined only if the model used in the Simulx project contains a block [COVARIATE].

A covariate element can be defined as an external file (csv or txt) or as a data frame.

In any case, it can contain columns occasions (optional), and it should contain one column per covariate (mandatory). Covariate names and categorical covariate values must correspond to covariates and categories defined in the model (block [COVARIATE]). The occasion headers must correspond to the occasion names defined in the occasion element.

A data frame can be used only to define covariate elements of type 'common', i.e the same for all individuals (potentially occasion-wise). If you want to define subject-specific covariates, use an external file with an "id" column. Covariate definition with distributions is only possible in the GUI (in R, please sample from the desired distribution to generate an external file).

An external file can be used in all cases (common or subject-specific). It can contain a column id (optional) in addition to occasions (optional), and should contain one column per covariate (mandatory). When id and occasion columns are present, then they must be the first columns. When the id column is not present, the covariate is considered common.

If the project has a subject-specific occasion structure (defined with an external file with an ID column (see defineOccasionElement)), occasion-wise common elements are not allowed. Covariate elements must be either common over all subjects and all occasions, or can be defined with subject-specific occasion-wise values as an external table, with the same occasion structure.

See Also

getCovariateElements

Examples

```
# Create a manual and a subject-specific element
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "1.overview", "importFromMonolix_clinicalTrial.smlx")
loadProject(project_name)
defineCovariateElement(name = "wt_typical", element = data.frame(wt = 70, sex = 1, age = 35)) # manual
samples <- data.frame(id = 1:10, sex = sample(0:1, 10, replace = TRUE), age = rnorm(10, 30, 10))
samples$wt <- rnorm(10, mean = ifelse(samples$sex == 0, 62, 75), sd = 10) # mean weight dependent on sex
file_name <- tempfile("cov", fileext = ".csv")
write.csv(samples, file_name, row.names = FALSE)
defineCovariateElement(name = "wt_distribution", element = file_name) # subject-specific

# Create an element with common occasions
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.7.occasions", "occasions_common.smlx")
loadProject(project_name)
defineCovariateElement(name = "Fasted_Fed", element = data.frame(occ = c(1, 2), FOOD = c("Fasted", "Fed")))

# Create an element with subject-specific occasions
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.7.occasions", "occasions_external.smlx")
loadProject(project_name)
occasions <- getOccasionElements()
covariates <- data.frame(id = occasions$id, occ = unlist(occasions$occasions), FOOD = rep(c("Fasted", "Fed",
file_name <- tempfile("cov", fileext = ".csv")
write.csv(covariates, file_name, row.names = FALSE)
defineCovariateElement(name = "cov_external", element = file_name)
```

`defineEndpoint` *[Simulx] Define endpoint element*

Description

Define or edit an endpoint. Endpoints summarize the outcome values over all individuals, for each simulation group and each replicate. Endpoints are defined as in Simulx GUI.

Usage

```
defineEndpoint(name, element)
```

Arguments

`name`	(*string*) (required) Endpoint name.
`element`	(*list*) (required) List with the endpoint settings:

- `outcome` (*string* or *list*) (required) - Outcome on which the endpoint is based on. If one outcome, use a string containing its name. Use list to combine outcomes with:
  - `outcomes` (*vector* of strings) - Vector of outcome names
  - `outcomesGroupName` (*string*) - Name you want to give to the outcome combination
  - `operator` (*string*) - Way in which output should be combined. One of "and"/"or" (in case of boolean outcomes) or "min"/"max" (in other cases)
- `metric` (*string*) (optional) - Calculation method for the endpoint. One of **"arithmeticMean"** (default), "geometricMean" or "median" if value-based outcome. In case of event-based outcomes, "kaplanMeier" (median survival) will be used and in case of boolean outcomes, "percentTrue" will be used by default.
- `groupComparison` (optional) (*list*) - Group comparison settings. List of:
  - `type` (*string*) (optional) - one of "directComparison", **"statisticalTest"** (default).
  - `h1` (*list*) (optional) - a list containing hypothesis information:
    - `operator` (*string*) - one of **"!="** (default), ">" or "<",
    - `threshold` (*double*) - a real number indicating the threshold for difference/oddsRatio (0 by default)
    - `pvalue` (*double*) - a real number indicating the p-value (if `type` is "statisticalTest", 0.05 by default)

Details

To compute the defined endpoints, use runEndpoints and get the results with getEndpointsResults.

To specify if endpoints should be compared across simulation groups, use setGroupComparisonSettings. If group comparison is relevant, the way the comparison will be done for each endpoint (eg if statistical test and which p-value) should be defined in the endpoint element.

See Also

`getEndpoints`

Examples

```
# Endpoint with group comparison
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomeEndpoint_PK
loadProject(project_name)
defineEndpoint(name = "comparison", element = list(outcome = "changeFromBaseline", metric = "geometricMean", g

# Combine multiple outcomes
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomeEndpoint_PD
loadProject(project_name)
defineEndpoint(name = "combined_endpoint", element = list(outcome = list(names = c("Survival", "TimeToNADIR_As
```

`defineIndividualElement`

[Simulx] Define individual parameters element

Description

Define or edit an element of individual parameters. Individual parameter elements are defined and used for simulation as in Simulx GUI. As for other elements, the individual parameters elements can be defined or imported, and they are saved with the Simulx project if calling saveProject. Once an individual parameters element is defined, it needs to be added to a simulation group with setGroupElement to be used in simulation.

Usage

```
defineIndividualElement(name, element)
```

Arguments

`name`	(*string*) Element name.
`element`	(*string* or *dataFrame*) Element definition from external file path or data frame with individual parameters as columns.

Details

Individual parameters to be defined depend on the model of the simulx project. If only the [LONGITUDINAL] block is present in the model: all parameters of the input list, except those defined as regressors. If both the [LONGITUDINAL] and [INDIVIDUAL] blocks are present: parameters defined in the DEFINITION section of the [INDIVIDUAL] block.

An individual parameters element can be defined as an external file (csv or txt) or as a data frame. In any case, it can contain columns occasions (optional), and it should contain one column per individual parameter (mandatory). The parameter headers must correspond to the parameter names used in the model. The occasion headers must correspond to the occasion names defined in the occasion element.

A data frame can be used only to define individual parameter elements of type 'common', i.e the same for all individuals (but potentially occasion-wise). If you want to define subject-specific individual parameters, use an external file with an "id" column.

An external file can be used in all cases (common or subject-specific). It can contain a column id (optional) in addition to occasions (optional), and should contain one column per parameter (mandatory). When id and occasion columns are present, then they must be the first columns. When the id column is not present, the parameter element is considered common.

If the project has a subject-specific occasion structure (defined with an external file with an ID column (see defineOccasionElement)), occasion-wise common elements are not allowed. In this case, individual parameters elements have to be be either common over all subjects and all occasions, or can be defined with subject-specific occasion-wise values as an external table, with the same occasion structure.

See Also

getIndividualElements

Examples

```
# Defining elements with one and multiple sets of indiv params
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "2.models", "longitudinal_individual.smlx")
loadProject(project_name)
defineIndividualElement(name = "custom_params", element = data.frame(Tlag = 0.5, ka = 0.25, V = 70, Cl = 12, F
params <- data.frame(id = c(1, 2, 3), Tlag = 0.5, ka = 0.25, V = 70, Cl = 12, F = c(0.6, 0.7, 0.8))
file_name <- tempfile("cov", fileext = ".csv")
write.csv(params, file_name, row.names = FALSE)
defineIndividualElement(name = "different_F", element = file_name) # multiple sets

# Common occasions
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.7.occasions", "occasions_common.smlx")
loadProject(project_name)
defineIndividualElement(name = "params_per_occ", element = data.frame(occ = c(1, 2), ka = c(0.2, 0.4), V = 10,

# Subject-specific occasions
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.7.occasions", "occasions_external.smlx")
loadProject(project_name)
occasions <- getOccasionElements()
params <- data.frame(ID = occasions$id, occ = unlist(occasions$occasions))
params$ka <- rlnorm(27, log(0.2), 0.1 + 0.1)
params$V <- rlnorm(27, log(5), 0.2)
params$Cl <- rlnorm(27, log(0.3), 0.15)
file_name <- tempfile("cov", fileext = ".csv")
write.csv(params, file_name, row.names = FALSE)
defineIndividualElement(name = "params_external", element = file_name)
```

---

defineOccasionElement *[Simulx] Define occasion element*

Description

Define the occasion structure of the project.

Usage

```
defineOccasionElement(element)
```

Arguments

element	(*string* or *dataFrame*) Element definition from external file path or data frame with time and occasion levels as columns.

Details

The occasion structure of a project is defined and used for simulation as in Simulx GUI. The occasion element impacts the definition of other elements and the simulation. As for other elements, the occasion element can be defined or imported, and it is saved with the Simulx project if calling saveProject.

If can be defined as an external file (txt or csv) or as a data.frame.

In any case, it should contain a column time and one column per occasion level. The header names for these occasion levels are free and used by Simulx.

A data frame can be used only to define a common structure of occasions applied to all simulated subjects.

An external file can be used in all cases (common or subject-specific structure). It can contain a column id (optional) in addition to other columns. When the id column is not present, the occasion structure is considered common.

After defining a subject-specific occasion structure, other elements (parameters, covariates, treatments, outputs and regressors) must be either common over all subjects and all occasions, or can be defined with subject-specific occasion-wise values as an external table, with the same occasion structure.

See Also

getOccasionElements

Examples

```
# Example: define subject-specific occasions through external file
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "replicates.smlx")
loadProject(project_name)
occasions <- data.frame(id = c(1, 1, 2, 2), time = c(0, 24, 0, 36), occ = c(1, 2, 1, 2))
file_name <- tempfile("cov", fileext = ".csv")
write.csv(occasions, file_name, row.names = FALSE)
defineOccasionElement(element = file_name)

# Example: define common occasions through data.frame
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "replicates.smlx")
loadProject(project_name)
defineOccasionElement(element = data.frame(time = c(0, 0.5, 2), occ1 = c(1, 1, 2), occ2 = c(1, 2, 3)))
```

`defineOutcome` *[Simulx] Define outcome element*

Description

Define or edit an outcome. Outcomes represent a post-processing of the simulation outputs done for each individual. Outcomes are defined as in Simulx GUI. Outcomes can only be defined by the user (no outcome is imported), and they are saved with the Simulx project if calling saveProject. Contrary to the GUI, defining an outcome with the connectors does not automatically generate an endpoint. To compute the defined outcomes, use them in endpoints with defineEndpoint, run runEndpoints and get the results with getEndpointsResults.

Usage

```
defineOutcome(name, element)
```

Arguments

`name`		(*string*) Outcome name.
`element`		(*list*) List with the outcome settings:
`output`	(*string*)	Name of the output element on which the outcome is based.
`perOccasion`	(*bool*)	(optional) If occasions are present in the simulation, it indicates if the outcome is calculated fo

Then, if the outcome is based on a **continuous** or **categorical** output:

`value` (optional) - vector of boundaries if *operator* is "durationBelow", "durationAbove" or "durationBetween", or time point if *operator* is "timePoint" (0 by default).

`applyThreshold` (*list*) (optional) List of elements:

- `operator` - one of "==", "!=", ">=", ">", "<=" or "<",
- `value` - a real number indicating the threshold value.

Or if the outcome is based on a **TTE** output:

`event`	(*list*)	(required) List of arguments that define event settings: `type` (required) - one of "timeOfEvents" (in case o

See Also

- `getOutcomes`

Examples

```
# Define an outcome to calculate Cmax
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomeEndpoint_PK
loadProject(project_name)
defineOutcome(name = "Cmax_outcome", element = list(output = "Plasma_concentration", processing = list(operato

# Define time above MIC as percentage
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomesEndpoints_
loadProject(project_name)
defineOutcome(name = "TimeAboveMIC", element = list(output = "mlx_Cc", processing = list(operator = "durationAb

# Relative outcome with threshold per individual and occasion
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomeEndpoint_PK
loadProject(project_name)
defineOutcome(name = "custom_outcome", element = list(output = "prediction_Cc_per_id", perOccasion = TRUE, rel

# Event based outcome
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomeEndpoint_PD
loadProject(project_name)
defineOutcome(name = "survival_outcome", element = list(output = "Death", event = list(type = "timeOfEvents")))
```

---

defineOutputElement *[Simulx] Define output element*

Description

Define or edit an output element.

Usage

```
defineOutputElement(name, element)
```

Arguments

name	(*string*) Element name.
element	List with: • data (*string* or *dataFrame*): data frame or path to external file (csv or txt). • output (*string*): name of any variable from the [LONGITUDINAL] block of the model (variable in EQUATION, smooth prediction listed under OUTPUT or noisy observation listed under DEFINITION).

Details

Output elements are defined and used for simulation as in Simulx GUI. As for other elements, the output elements can be defined or imported, and they are saved with the Simulx project if calling saveProject. Once an output element is defined, it needs to be added to a simulation group with setGroupElement to be used in simulation.

To define an output element, in addition to the element name, you need to provide the time grid for simulation and the output to simulate.

The field data can be specified with a data frame or an external file (csv or txt).

To define a regular output, common to all individuals, you can use a data frame, with column headers start, interval and final. All time points from "start" to "final" by steps of "interval" will be used for simulation. If the project has a common occasion structure, this data frame can contain a column occasion and several lines to define the regular treatment occasion-wise.

To define an output by giving a specific list of times, both data frames and external files (txt or csv) can be used, with a column time. They can contain columns occasions (optional). The occasion headers must correspond to the occasion names defined in the occasion element.

Data frames can be used only to define output elements of type 'common', i.e the same for all individuals (potentially occasion-wise). If you want to define subject-specific output elements, you have to use an external file with an "id" column.

An external file can be used in all cases (common or subject-specific). It can contain a column id (optional) in addition to occasions (optional), and should contain one column time (mandatory). When id and occasion columns are present, then they must be the first columns. When the id column is not present, the covariate is considered common.

If the project has a subject-specific occasion structure (defined with an external file with an ID column (see defineOccasionElement)), occasion-wise common elements are not allowed. Output elements must be either common over all subjects and all occasions, or can be defined with subject-specific occasion-wise values as an external table, with the same occasion structure.

See Also

getOutputElements

Examples

```
# Define subject-specific outputs using an external file (saved in tmp directory)
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "replicates.smlx")
loadProject(project_name)
samplings <- data.frame(id = c(1, 1, 2, 2, 3, 3), time = c(0, 24, 0, 48, 0, 72))
file_name <- tempfile("cov", fileext = ".csv")
write.csv(samplings, file_name, row.names = FALSE)
defineOutputElement(name = "external_output", element = list(data = file_name, output = "Cc"))

# Define manual and regular output
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "1.overview", "importFromMonolix_clinicalTrial.smlx")
loadProject(project_name)
defineOutputElement(name = "AUC_0_24", element = list(data = data.frame(time = 24), output = "AUC"))
defineOutputElement(name = "Cc_7days", element = list(data = data.frame(start = 0, interval = 1, final = 168),

# Define manual and regular occasion-wise output
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.7.occasions", "occasions_two_levels.smlx")
loadProject(project_name)
defineOutputElement(name = "manualOcc", element = list(data = data.frame(time = c(0, 2, 24, 4, 36), occ1 = c(1
defineOutputElement(name = "regularOcc", element = list(data = data.frame(start = c(0, 0, 24, 24), interval =
```

`definePopulationElement` *[Simulx] Define population element*

Description

Define or edit an element of population parameters. Population parameter elements are defined and used for simulation as in Simulx GUI. As for other elements, the population parameters elements can be defined or imported, and they are saved with the Simulx project if calling saveProject. Once a population parameters element is defined, it needs to be added to a simulation group with setGroupElement to be used in simulation.

Usage

```
definePopulationElement(name, element)
```

Arguments

`name`	(*string*) Element name.
`element`	(*string* or *dataFrame*) Element definition from external file path or data frame with population parameters as columns.

Details

Definition of population parameters as simulation elements allows to simulate individual parameters from probability distributions. It is possible only if the model includes a block [INDIVIDUAL] to consider the inter-individual variability.

A population parameters element can be defined as an external file (csv or txt) or as a data frame. In any case, it should contain one column per population parameter (mandatory).

To check exactly which column names to use, you can use getPopulationElements and view the population parameters element that was automatically created after defining the model (if the model has an [INDIVIDUAL] block).

To define a population parameters element with several lines, with several sets to be used in different replicate simulations, an external file is required. In this case, you should also set the number of replicates for your simulation with setNbReplicates, otherwise only the first set of population parameters will be taken into account. Each replicate uses one set of population parameters with the order of the appearance in the table.

Note: It is not possible to define population elements with distributions with the connectors as in the GUI. To do this, please sample values from distributions in R and create the element with different rows as in the last example below.

See Also

`getPopulationElements`, `setNbReplicates`

Examples

```
## Not run:
  definePopulationElement(name = "name", element = "file/path")
  definePopulationElement(name = "name", element = data.frame(Cl_pop = 1, V_pop = 2.5))

## End(Not run)
# Define a pop param element with a data frame

 loadProject(file.path(getDemoPath(),"3.definition","3.3.population_parameters","pop_parameters_manual.sml
  # get the existing pop param element as an example
  ExamplePopParamData = getPopulationElements()$manual_parameters$data
 ExamplePopParamData[] = rep(1,11) #set the desired values, for simplicity we use all param =1
  definePopulationElement(name = "PopParam", element = ExamplePopParamData)

# Check impact of varying ka with replicates (external file required)

 loadProject(file.path(getDemoPath(),"3.definition","3.3.population_parameters","pop_parameters_manual.sml
  # get the existing pop param element as an example:
  ExamplePopParamData = getPopulationElements()$manual_parameters$data
  # add lines to the data frame to have different values for ka:
  PopParamReplicates = ExamplePopParamData[rep(1,10),]
  PopParamReplicates$ka_pop = rnorm(10,mean = 0.8, sd = 0.1)
  # write the table to an external file (required because it has several lines):
  file_name = tempfile("PopParamReplicates.csv")
  write.csv(PopParamReplicates, file_name, row.names = FALSE)
  # define the new element and use it in simulation:
  definePopulationElement(name = "PopParamReplicates", element = file_name)
  setGroupElement(group = "simulationGroup1", elements = "PopParamReplicates")
 setNbReplicates(nb = 10) # to simulate the project 10x, each time with a different population parameter element
  runSimulation()
 ## Not run: getSimulationResults()
```

`defineRegressorElement`

[Simulx] Define regressor element

Description

Define or edit a regressor element. Regressor elements are defined and used for simulation as in Simulx GUI. Once a regressor element is defined, it needs to be added to a simulation group with setGroupElement to be used in simulation. Regressor elements can be defined only if regressors are defined in the model loaded in the Simulx project.

Usage

```
defineRegressorElement(name, element)
```

Arguments

`name`	(*string*) Element name.
`element`	(*string* or *dataFrame*) Element definition from external file path or data frame with time and regressors as columns.

Details

To define a regressor element, in addition to the element name, you need to provide in the field data the time points and values of the regressor at each time point. To simulate the model for points outside of this grid, last value carried forward interpolation is used.

The field data can be specified with a data frame or an external file (csv or txt). They should contain a column time and a column for each regressor variable. They can contain columns occasions (optional). The occasion headers must correspond to the occasion names defined in the occasion element.

Data frames can be used only to define regressor elements of type 'common', i.e the same for all individuals (potentially occasion-wise). If you want to define subject-specific regressor elements, you have to use an external file with an additional "id" column.

An external file can be used in all cases (common or subject-specific). It can contain a column id (optional) in addition to occasions (optional), time (mandatory) and one column per regressor defined in the model (mandatory). When id and occasion columns are present, then they must be the first columns. When the id column is not present, the covariate is considered common.

If the project has a subject-specific occasion structure (defined with an external file with an ID column (see defineOccasionElement)), occasion-wise common elements are not allowed. In this case, regressors must be either common over all subjects and all occasions, or defined with subject-specific occasion-wise values as an external table, with the same occasion structure.

See Also

getRegressorElements

Examples

```
 # Define subject-specific regressors using an external file (saved in tmp directory)
 initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.6.regressors", "regressor_manual_paramCovRelati
 loadProject(project_name)
samplings <- data.frame(id = c(1, 1, 2, 2, 3, 3), time = c(0, 24, 0, 48, 0, 72), PCA = c(9, 15, 5, 20, 3, 14))
 file_name <- tempfile("cov", fileext = ".csv")
 write.csv(samplings, file_name, row.names = FALSE)
 defineRegressorElement(name = "reg_external", element = file_name)

 # Define manual regressor element
 initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.6.regressors", "regressor_manual_paramCovRelati
 loadProject(project_name)
defineRegressorElement(name = "reg_manual", element = data.frame(time = c(0, 0.5, 2), PCA = c(1, 2, 5.25)))

 # Define manual occasion-wise regressors
 initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.6.regressors", "regressor_manual_paramCovRelati
 loadProject(project_name)
defineOccasionElement(element = data.frame(time = c(0, 0, 0, 0), occ1 = c(1, 2, 1, 2), occ2 = c(1, 1, 2, 2)))
defineRegressorElement(name = "name", element = data.frame(time = c(0, 0.5, 2, 5, 6), PCA = c(1, 2, 5.25, 6, 7)
```

`defineTreatmentElement` *[Simulx] Define treatment element*

Description

Define or edit a treatment element.

Usage

```
defineTreatmentElement(name, element)
```

Arguments

`name`	(*string*) Element name.
`element`	(*list*) List with the treatment settings:

`data` (mandatory)	(data frame or external file, csv or txt) Column headers: [only data frame] for a regular treatment common to all ids:occ (optional, if common occasio
`admID` (optional)	(integer) same as integer in the model as administration type
`probaMissDose` (optional)	(double) probability to miss a dose (number in [0 1])
`repeats` (optional)	(vector) to repeat the specified treatment after a specific duration. Elements: duration after which the treatment will be repeated (can be longer than the treatment duration)
`scale` (optional)	(list) to scale the dose amount by a covariate. The scaled amount will be administered instea (string) covariate to use for scaling (same name as in the [COVARIATE] block of the model) (

Details

`cycleDuration` `NumberOfRepetitions` `covariate` `interceptdalities` `scaleDuration` Treatment elements are defined and used for simulation as in Simulx GUI. As for other elements, the treatment elements can be defined or created at import, and they are saved with the Simulx project if calling saveProject. Once an output element is defined, it needs to be added to a simulation group with setGroupElement to be used in simulation. Several treatment elements can be added to the same simulation group and they will be both administered to every individual in the group.

To define a treatment element, in addition to the element name, you need to provide a list with at list one field "data" containing the dose information. The field data can be specified with a data frame or an external file (csv or txt).

To define a regular treatment, common to all individuals, you can use a data frame, with column headers start, interval, nbDoses and amount. You can include an optional column tInf or rate to define an infusion. If the project has a common occasion structure (i.e. same occasions for all individuals), this data frame can contain a column occasion to define the regular treatment occasion-wise. The occasion headers must correspond to the occasion names defined in the occasion element.

To define a treatment by giving a specific list of times and amounts, both data frames and external files (txt or csv) can be used, with a column time. They can contain columns id and occasions (optional). The occasion headers must correspond to the occasion names defined in the occasion element.

Data frames can be used only to define output elements of type 'common', i.e the same for all individuals (potentially occasion-wise). If you want to define subject-specific treatment elements, you have to use an external file with an "id" column.

An external file can be used in all cases (common or subject-specific). It can contain a column id (optional) in addition to occasions (optional), and should contain one column time (mandatory) and one column amount (mandatory). When id and occasion columns are present, then they must be the first columns. When the id column is not present, the covariate is considered common.

Note

To define a regular schedule, it is advised to use a regular treatment without repeats, rather than a manual treatment with repeats. Repeats are useful to create more complex schedules in addition to a manual or regular definition, such as dosing regimen 3 weeks ON, 1 week OFF.

To see the impact of a treatment until the end of a dosing regimen, you should set an output element that spans the duration of the treatment to the same simulation group.

See Also

getTreatmentElements

Examples

```
## Not run:
  defineTreatmentElement(name = "name", element = list(data = "file/path"))
 defineTreatmentElement(name = "name", element = list(probaMissDose=0, admID=1, repeats=c(cycleDuration = 1, N
 defineTreatmentElement(name = "name", element = list(probaMissDose=0, admID=1, repeats=c(cycleDuration = 1, N
 defineTreatmentElement(name = "name", element = list(probaMissDose=0, admID=1, repeats=c(cycleDuration = 1, N
 defineTreatmentElement(name = "name", element = list(admID=1, scale=list(covariate="age", intercept=12), dat
 defineTreatmentElement(name = "name", element = list(admID=1, scale=list(covariate="sex", modalities=list(li

## End(Not run)

##### Working example with treatment combinations #####

# In this demo, the first group receives only the chemotherapy, while the second group receives both the chemothe
# Note that the chemotherapy treatment uses adm=1 to be applied to compartment 1 via the macro iv(adm=1, cmt=1) i
# The anti-angiogenic treatment is defined with adm=2 which is applied via the macro iv(adm=2, cmt=2) to compart

initializeLixoftConnectors("simulx")
loadProject(paste0(getDemoPath(), "/3.definition/3.1.treatments/treatment_combinations.smlx"))
# to see how the structural model is defined:
file.show(getStructuralModel())

defineTreatmentElement(name = "Chemotherapy", element = list(data=data.frame(start=10, interval=14, nbDoses=1
defineTreatmentElement(name = "AntiAngionenic_treatment", element = list(admID = 2, data=data.frame(start=10,

setGroupElement("simulationGroup1","Chemotherapy")
setGroupElement("simulationGroup2",c("Chemotherapy","AntiAngionenic_treatment"))
runSimulation()
# use ggplot or export to Monolix/PKanalix to plot trajectories
exportProject(settings = list(targetSoftware = "monolix"),force = TRUE)
## Not run:
plotObservedData( settings = list(dots = FALSE,  ylab = "Target Occupancy", legend = TRUE), stratify = list(colo

## End(Not run)
```

```
##### Working example with a treatment scaled by weight and based on genotype #####

#  In this demo, a weight-based dose is defined by indicating the dose per unit weight in the amount box (14 nmol/
# The "intercept" could be used to define a offset common to all weights (e.g 14nmol/kg + 10nmol).
# When an infusion duration or rate has been defined, the user can choose if the infusion duration or the infusio
# For categorical covariates, such as the genotype, a scaling factor and an intercept can be defined for each cat
# In this demo, the scaling for Homozygous is 1 meaning that they receive the dose defined in the amount box.
# For heterozygous, the scaling is 0.8, meaning that they receive 0.8 times the amount in the amount box.

initializeLixoftConnectors("simulx")
loadProject(paste0(getDemoPath(), "/3.definition/3.1.treatments/treatment_weight_and_genotype_based.smlx"))

defineTreatmentElement(name = "14nmolPerKg", element = list(data=data.frame(start=0, interval=21, nbDoses=5,
defineTreatmentElement(name = "1000nmol", element = list(data=data.frame(start=0, interval=21, nbDoses=5, amo
defineTreatmentElement(name = "1000nmolHomo_800nmolHetero", element = list(data=data.frame(start=0, interval=

setGroupElement("Weight_based","14nmolPerKg")
setGroupSize("Weight_based",20)
setGroupElement("Flat_dose","1000nmol")
setGroupSize("Flat_dose",20)
setGroupElement("Genotype_based","1000nmolHomo_800nmolHetero")
setGroupSize("Genotype_based",20)
runSimulation()
# use ggplot or export to Monolix/PKanalix to plot trajectories
exportProject(settings = list(targetSoftware = "monolix"),force = TRUE)
plotObservedData(obsName = "yTO", settings = list(dots = FALSE,  ylab = "Target Occupancy"), stratify = list(spl


##### Working example with a probability to miss a dose #####

# In this demo, the second treatment is defined with a probability to miss a dose of 0.1, meaning that on average

initializeLixoftConnectors("simulx")
loadProject(paste0(getDemoPath(), "/3.definition/3.1.treatments/treatment_non_adherence.smlx"))

defineTreatmentElement(name = "OncePerDay_full_compliance", element = list(data=data.frame(start=0, interval=
defineTreatmentElement(name = "OncePerDay_partial_compliance", element = list(data=data.frame(start=0, interv

setGroupElement("simulationGroup1","OncePerDay_full_compliance")
renameGroup("simulationGroup1","FullCompliance")
setGroupElement("simulationGroup2","OncePerDay_partial_compliance")
renameGroup("simulationGroup2","NonAdherence")
setGroupSize("FullCompliance",20)
setGroupSize("NonAdherence",20)
runSimulation()
# use ggplot or export to Monolix/PKanalix to plot trajectories
exportProject(settings = list(targetSoftware = "monolix"),force = TRUE)
plotObservedData( settings = list(dots = FALSE,  ylab = "Target Occupancy"), stratify = list(splitGroup = list(

##### Working example with an external file #####

# Demo: use an external file to define a dose by age group: 1-2 years 12.5 mg, 3-6 years 18.75 mg and 7-15 years 25
# The age also appears as covariate in the model and the covariate element is defined via an external file.
# To make sure the covariates are sampled from the covariate external file and the doses sampled from the treatme
```

```
initializeLixoftConnectors("simulx")
loadProject(paste0(getDemoPath(), "/3.definition/3.1.treatments/treatment_external_byAgeGroup.smlx"))
tableAge = getCovariateElements()$External_AGE_values$data
AmtByAgeGroups = (tableAge$AGE < 3)*12.5 + ((tableAge$AGE >=3) & (tableAge$AGE < 7))*18.75 + (tableAge$AGE >= 7)
Nid = length(AmtByAgeGroups)
dataAmtByAgeGroups = data.frame(id = tableAge$ID, time = rep(0,Nid), amount = AmtByAgeGroups)
file_name <- tempfile("trt", fileext = ".csv")
write.csv(dataAmtByAgeGroups, file_name, row.names = FALSE)

defineTreatmentElement(name = "doseByAgeGroup", element = list(data = file_name))

setGroupElement("simulationGroup1",c("doseByAgeGroup","External_AGE_values","regularCc"))
setSharedIds(c("covariate", "treatment"))
runSimulation()
# use ggplot or export to Monolix/PKanalix to plot trajectories
exportProject(settings = list(targetSoftware = "monolix"),force = TRUE)
plotObservedData(settings = list(dots = FALSE, ylab = "Cc",legend = TRUE, ylim = c(0,13)), stratify = list(split


##### Working example with washout #####

# In this demo, two different formulations are given.
# The reference formulation is given at time zero.
# The test formulation is given after a long washout period.
# In order not to simulate this washout period, the test dose is defined at time 48 and a washout is applied just

initializeLixoftConnectors("simulx")
loadProject(paste0(getDemoPath(), "/3.definition/3.1.treatments/treatment_washout.smlx"))
defineTreatmentElement(name = "ReferenceFormulation_atTime0", element = list(data=data.frame(time=0, amount=0
defineTreatmentElement(name = "TestFormulation_atTime48", element = list(admID = 2, data=data.frame(time=0, am
setGroupElement("simulationGroup1",c("ReferenceFormulation_atTime0","ReferenceFormulation_atTime0"))


##### Working example with a regular treatment and repeats #####

# The "repeat" option allows to repeat a base pattern several times with a defined periodicity.
# In this demo, the first treatment is defined as one dose per day during 112 days.
# The second treatment is defined as one dose per day during 14 days and this is repeated every 28 days leading to

initializeLixoftConnectors("simulx")
loadProject(paste0(getDemoPath(), "/3.definition/3.1.treatments/treatment_regular_cycles.smlx"))

defineTreatmentElement(name = "OncePerDay_4weeksOn", element = list(data=data.frame(start=0, interval=1, nbDo
defineTreatmentElement(name = "OncePerDay_2weeksOn2weeksOff", element = list(repeats=c(cycleDuration = 28, Nu

setGroupElement("simulationGroup1","OncePerDay_4weeksOn")
renameGroup("simulationGroup1","4weeksOn")
setGroupElement("simulationGroup2","OncePerDay_2weeksOn2weeksOff")
renameGroup("simulationGroup2","2weeksOn2weeksOff")
runSimulation()
# use ggplot or export to Monolix/PKanalix to plot trajectories
exportProject(settings = list(targetSoftware = "monolix"),force = TRUE)
plotObservedData(settings = list(dots = FALSE, ylab = "Cc",legend = TRUE), stratify = list(colorGroup = list(na
```

deleteAdditionalCovariate *[Monolix - PKanalix] Delete additional covariate*

Description

Delete a created additinal covariate.

Usage

```
deleteAdditionalCovariate(name)
```

Arguments

name	(*string*) name of the covariate.

See Also

addAdditionalCovariate

Examples

```
## Not run:
deleteAdditionalCovariate("firstDoseAmount")\cr
deleteAdditionalCovariate("observationNumberPerIndividual_y1")

## End(Not run)
```

deleteElement *[Simulx] Delete element*

Description

Delete an element of any type.

Usage

```
deleteElement(name)
```

Arguments

name	(*string*) Element name.

Details

Elements defined are created in the background and saved with the Simulx project if calling saveProject.

To check which elements of a certain type have been defined so far, please use one of the "get..Elements" connectors: getCovariateElements, getPopulationElements, getIndividualElements, getTreatmentElements, getOccasionElements, getRegressorElements.

Elements cannot be deleted if they are used for the simulation. To remove an element from the simulation, use removeGroupElement.

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "1.overview", "importFromMonolix_clinicalTrial.smlx")
loadProject(project_name)
deleteElement(name = "mlx_CovDist")
```

`deleteEndpoint` *[Simulx] Delete an endpoint*

Description

Delete an endpoint.

Usage

```
deleteEndpoint(name)
```

Arguments

`name`	(*string*) Endpoint name

Details

Endopints defined are created in the background and saved with the Simulx project if calling saveProject.

To check which endpoints have been defined, please use getEndpoints.

See Also

`deleteOutcome`

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomeEndpoint_PD
loadProject(project_name)
deleteEndpoint("mean_NADIR")
```

deleteFilter *[Monolix - PKanalix] Delete filter*

Description

Delete a data set. Only filtered data set which are not active and whose children are not active either can be deleted.

Usage

```
deleteFilter(name)
```

Arguments

name	(*string*) data set name.

See Also

createFilter

Examples

```
## Not run:
deleteFilter(name = "filter2")

## End(Not run)
```

deleteOccasionElement *[Simulx] Delete occasion element*

Description

Delete the occasion element.

Usage

```
deleteOccasionElement()
```

Details

The occasion element impacts the definition of other elements and the simulation. As for other elements, the occasion element can be defined or imported, and it is saved with the Simulx project if calling saveProject. To check if an occasion element has been defined, please use getOccasionElements. The occasion element may impact the definition of other elements. When deleting the occasion element, all other elements that depend on occasions are also deleted.

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.7.occasions", "occasions_common.smlx")
loadProject(project_name)
deleteOccasionElement()
```

deleteOutcome *[Simulx] Delete an outcome*

Description

Delete an outcome.

Usage

```
deleteOutcome(name)
```

Arguments

name	(*string*) Outcome name

Details

Outcomes defined are created in the background and saved with the Simulx project if calling saveProject.

To check which outcomes have been defined, please use getOutcomes.

An outcome used in an endpoint cannot be deleted. The related endpoint must be deleted first with deleteEndpoint.

See Also

deleteEndpoint

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomeEndpoint_PD
loadProject(project_name)
deleteEndpoint("mean_NADIR")
deleteOutcome("NADIR")
```

editFilter *[Monolix - PKanalix] Edit filter*

Description

Edit the definition of an existing filtered data set. Refere to createFilter for more details about syntax, allowed parameters and examples.
Notice that all the filtered data set which depend on the edited one will be deleted.

Usage

```
editFilter(filter, name = "")
```

Arguments

filter	(*list< list< action = "headerName-comparator-value" > >*) filter definition.
name	(*string*) [optional] data set name to edit (current one by default)

See Also

createFilter

---

exportChartDataSet *[Monolix] Export chart dataset*

Description

Export the data of a chart into Lixoft suite compatible data set format.
It can be generated only if the concerned chart has been built.
The file is written in the results folder of the current project.

Usage

```
exportChartDataSet(type, filePath = "")
```

Arguments

type	(*string*) Chart type whose data must be exported. Available types are: "vpc", "indfits".
filePath	[optional](*string*) Custom path for the exported file. By default, it is written in the DataFile folder of the current project.

See Also

computeChartsData runScenario

Examples

```
## Not run:
 exportChartDataSet(type = "vpc")
 exportChartDataSet(type = "indfits", filePath = "/path/to/exported/file.txt")

## End(Not run)
```

`exportProject` *[Monolix - PKanalix - Simulx] Export current project to Monolix, PKanalix or Simulx*

Description

Export the current project to another application of the MonolixSuite, and load the exported project. NOTE: This action switches the current session to the target software. Current unsaved modifications will be lost. The extensions are .mlxtran for Monolix, .pkx for PKanalix, .smlx for Simulx and .dxp for Datxplore.
WARNING: R is sensitive between '\' and '/', only '/' can be used.

Usage

```
exportProject(settings, force = F)
```

Arguments

`settings`	(*character*) Export settings: • targetSoftware (*character*) Target software ("monolix" | "simulx" | "pkanalix") • filesNextToProject (*boolean*) [optional][Monolix - PKanalix] Save data and/or structural model file next to exported project ([TRUE] | FALSE). Forced to TRUE for Simulx. • dataFilePath (emphcharacter) [optional][Monolix - Simulx] Path (filesNextToProject == FALSE) or name (filesNextToProject == TRUE) of the exported data file. Available only for generated datasets in Monolix (vpc, individual fits) • dataFileType (emphcharacter) [optional][Monolix] Dataset used in the exported project (["original"] | "vpc" | "individualFits") • modelFileName (emphcharacter) [optional][Simulx] Name of the exported model file.
`force`	(*bool*) [optional] Should software switch security be overpassed or not. Equals FALSE by default.

Details

At export, a new project is created in a temporary folder. By default, the file is created with a project setting filesNextToProject = TRUE, which means that file dependencies such as data and model files are copied and kept next to the new project (or in the result folder for Simulx). This new project can be saved to the desired location withsaveProject.

Exporting a Monolix or a PKanalix project to Simulx automatically creates elements that can be used for simulation, exactly as in the GUI.

To see which elements of some type have been created in the new project, you can use the get..Element functions: getOccasionElements, getPopulationElements, getPopulationElements, getIndividualElements, getCovariateElements, getTreatmentElements, getOutputElements, getRegressorElements.

See Also

`newProject`, `loadProject`, `importProject`

Examples

```
## Not run:
[PKanalix only]
exportProject(settings = list(targetSoftware = "monolix", filesNextToProject = F))
[Monolix only]
exportProject(settings = list(targetSoftware = "simulx", filesNextToProject = T, dataFilePath = "data.txt", da
exportProject(settings = list(targetSoftware = "simulx", filesNextToProject = F, dataFilePath = "/path/to/data

[Simulx only]
exportProject(settings = list(targetSoftware = "pkanalix", dataFilePath = "data.txt", modelFileName = "model.t
exportProject(settings = list(targetSoftware = "pkanalix", dataFilePath = "/path/to/data/data.txt"))

## End(Not run)
# Working example to export a Monolix project to Simulx. The resulting .smlx file can be opened from Simulx GUI.
initializeLixoftConnectors(software = "monolix", force = TRUE)
loadProject(file.path(getDemoPath(),"1.creating_and_using_models","1.1.libraries_of_models","warfarinPK_pro
runScenario()
exportProject(settings = list(targetSoftware = "simulx"), force = TRUE)
saveProject("importFromMonolix.smlx")
```

---

exportSimulatedData *[Simulx] Export simulated data*

Description

Export the simulated dataset into a MonolixSuite compatible format.
It contains treatment information and simulation results and can be generated only when simulation results are available.

Usage

```
exportSimulatedData(filePath = "")
```

Arguments

filePath	[optional](*string*) Custom path for the exported file. By default, it is written in the results folder of the current project, next to simulation results files. The default file name is simulatedData.csv.

Details

The generated dataset can then be loaded in Datxplore, PKanalix or Monolix.

Note: to export the simulated data and load it into another application in a single line, you can also use exportProject.

See Also

exportProject

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "1.overview", "newProject_TMDDmodel.smlx")
loadProject(project_name)
runSimulation()
exportSimulatedData()
```

`formatData` *[Monolix - PKanalix] Adapt and export a data file as a MonolixSuite formatted data set*

Description

Data formatting can be performed as in the Data Formatting Tab of Monolix and PKanalix interface. Look at the examples to see how each data formatting demo project could be created with the connectors.

Usage

```
formatData(
  dataFile,
  formattedFile,
  headerLines = 1,
  headers,
  linesToExclude = NULL,
  observationSettings = NULL,
  observations = NULL,
  treatmentSettings = NULL,
  treatments = NULL,
  additionalColumns = NULL
)
```

Arguments

`dataFile`	(*string*) Path to the original data file.
`formattedFile`	(*string*) Path to the data file that will be exported (must end with the .csv extension).
`headerLines`	(optional) (*int* or *vector<int>*) Line numbers containing headers (if multiple numbers are given, formatted headers will contain values from all header lines concatenated with the "_" character) - default: 1.
`headers`	(*list*) List of headers containing information about ID, time, volume (in case of urine data) and sort columns. • `id` (*string*) - Name of the column distinguishing data from different individuals. • `time` (*string*) - Name of the column containing observation times (in case of plasma data). • `sort` (*string* or *vector<string>*) - Name of the column(s) distinguishing different profiles. • `start` (*string*) - Name of the column containing urine collection start times (in case of urine data).

- `end` (*string*) - Name of the column containing urine collection end times (in case of urine data).
- `volume` (*string*) - Name of the column containing collected volume of urine samples (in case of urine data).

`linesToExclude` (optional) (*int* or *vector<int>*) Numbers of lines that should be removed from the data set.

`observationSettings`
(optional) (*list*) List containing settings applied when different observation columns are merged into a single column.

- `distinguishWithObsId` (*bool*) - If TRUE, different observations will be distinguished with the observation ID column (default), otherwise they will be distinguished with occasions.
- `duplicateInformation` (*bool*) - If TRUE, information from undefined columns will be duplicated (default) in the newly created rows.

`observations` (optional) (*list*) List of lists containing information about different observation types:

- `header` (*string*) - Name of the column containing observations.
- `censoring` (*list*) - List of lists containing information about different types of censored data (not necessary if there is no censored data):
  - `type` (*string*) - Type of censoring, one of "LLOQ", "ULOQ", or "interval".
  - `tags` (*string* or *vector<string>*) - Strings in the observation column indicating that the data is censored (e.g., "BLQ", "LLOQ", ...).
  - `limits` - Define limits of censored data. If censoring type is "LLOQ" or "ULOQ", the lower and upper limit is defined with one of the following arguments. If censoring type is "interval", the lower and upper limits of the censoring interval are defined with a list of two of the following arguments:
    - as *string* - The column with the indicated header will be used to define limits.
    - as *double* - The value will be used as a lower/upper limit.
    - as *list* - Used to give different values for different categories. List needs to be have two arguments:
      - `category` (*string*) - Name of the column containing the category.
      - `values` (*list*) - List containing modalities as keys and limit values as values (e.g., list(method1 = 0.06, method2 = 0.1)).

`treatmentSettings`
(optional) (*list*) List containing settings applied to all treatments.

- `infusionType` ("rate"|"duration", default = "duration") - Type of values defining infusion.
- `doseIntervalsAsOccasions` (default = FALSE) (*bool*) - If TRUE, occasions will be created for each dose interval.

`treatments` (optional) (*list*) List that can contain lists with information about different treatments or strings with paths to files that contain treatment information. Lists with information about different treatments need to have the following elements:

- `times` (*double* or *vector<double>*) - Times at which the dose is administered (R function seq can be used to define regular treatments).

- `amount` (*string*, *double* or *list*) - Administered amount. Can be defined in the same way as censoring limits (through a column name, as a fixed value or as values depending on categories).
- `infusion` (*string*, *double* or *list*) - Infusion rate or duration (see the treatmentSettings argument for more information). Can be defined in the same way as censoring limits (through a column name, as a fixed value or as values depending on categories). Does not need to be provided if the drug is not administered through an infusion.
- `admId` (*string*, *double* or *list*) - Administration ID. Can be defined in the same way as censoring limits (through a column name, as a fixed value or as values depending on categories). If not provided, default of 1 will be used.
- `repeatCycle` (*list*) - List containing repetition information (does not need to be provided if the treatment is not repeated):
  - `duration` (*double*) - Duration of a cycle.
  - `number` (*int*) - Number of repetitions.

`additionalColumns`
: (optional) (*string* or *vector*<*string*>) Path(s) to the file(s) containing additional columns (needs to have the ID column).

Examples

```
# example: create a new project with a dataset to format:
initializeLixoftConnectors(software = "pkanalix")
FormattedDataPath = tempfile("formatted_data", fileext = ".csv")
formatData(paste0(getDemoPath(),"/0.data_formatting/data/units_BLQ_tags_data.csv"),
           formattedFile = FormattedDataPath,
           headerLines = c(1,2),
           headers = c(id="ID", time="TIME_h"),
           observations = list(header="CONC_mg_L",
                               censoring = list(type="interval", tags = c("BLQ"),
                                                limits=list(0,"LLOQ_mg_L"))),
           treatments = list(times=0, amount=100))
colnames(read.csv(FormattedDataPath)) # to check column names of the generated file and tag them as desired
newProject(data = list(dataFile = FormattedDataPath, headerTypes = c("id","time","observation","contcov","con
plotObservedData()


# demo merge_occ_ParentMetabolite.pkx
formatData(paste0(getDemoPath(),"/0.data_formatting/data/parent_metabolite_data.csv"),
           formattedFile = FormattedDataPath,
           headers = c(id="ID", time="TIME"),
           observations = list(list(header="PARENT",
                       censoring = list(type="interval", tags = c("BLQ"), limits=list(0,0.01))),
                              list(header="METABOLITE")),
           observationSettings = list(distinguishWithObsId = FALSE),
           treatments = list(times=0, amount="DOSE"))

# demo merge_obsID_ParentMetabolite.pkx
formatData(paste0(getDemoPath(),"/0.data_formatting/data/parent_metabolite_data.csv"),
           formattedFile = FormattedDataPath,
           headers = c(id="ID", time="TIME"),
           observations = list(list(header="PARENT",
                       censoring = list(type="interval", tags = c("BLQ"), limits=list(0,0.01))),
                              list(header="METABOLITE")),
```

```
            treatments = list(times=0, amount="DOSE"))

    # demo DoseAndLOQ_byCategory.pkx
    formatData(paste0(getDemoPath(),"/0.data_formatting/data/units_BLQ_tags_data.csv"),
            formattedFile = FormattedDataPath,
            headerLines = c(1,2),
            headers = c(id="ID", time="TIME_h"),
            observations = list(header="CONC_mg_L",
                                censoring = list(type="interval", tags = c("BLQ"),
                                                 limits=list(0,list(category="STUDY",
                                              values=list("SD_400mg"=0.01, "SD_500mg"=0.1, "SD_600mg"=0.1)))),
            treatments = list(times=0, amount=list(category="STUDY",
                                values=list("SD_400mg"=400, "SD_500mg"=500, "SD_600mg"=600))))

    # demo DoseAndLOQ_fromData.pkx
    formatData(paste0(getDemoPath(),"/0.data_formatting/data/units_BLQ_tags_data.csv"),
            formattedFile = FormattedDataPath,
            headerLines = c(1,2),
            headers = c(id="ID", time="TIME_h"),
            observations = list(header="CONC_mg_L",
                                censoring = list(type="interval", tags = c("BLQ"),
                                                 limits=list(0,"LLOQ_mg_L"))),
            treatments = list(times=0, amount="STUDY"))

    # demo DoseAndLOQ_manual.pkx
    formatData(paste0(getDemoPath(),"/0.data_formatting/data/units_multiple_BLQ_tags_data.csv"),
            formattedFile = FormattedDataPath,
            headerLines = c(1,2),
            headers = c(id="ID", time="TIME_h"),
            observations = list(header="CONC_mg_L",
                      censoring = list(list(type="interval", tags = c("BLQ1"), limits=list(0,0.06)),
                                       list(type="interval", tags = c("BLQ2"), limits=list(0,0.1)))),
            treatments = list(times=0, amount=600))

    # demo Urine_LOQinObs.pkx
    formatData(paste0(getDemoPath(),"/0.data_formatting/data/urine_LOQinObs_data.csv"),
            formattedFile = FormattedDataPath,
            headers = c(id="ID", start="START_TIME", end="END_TIME", volume="VOLUME"),
            observations = list(header="CONC",
                                censoring=list(type="LLOQ", tags="<LOQ=1>", limits="CONC")),
           treatments = list(paste0(getDemoPath(),"/0.data_formatting/data/urine_data_doses.csv")))

    # demo CreateOcc_AdmIdbyCategory.pkx
    formatData(paste0(getDemoPath(),"/0.data_formatting/data/two_formulations_data.csv"),
            formattedFile = FormattedDataPath,
            linesToExclude = 1, headerLines = c(2,3),
            headers = c(id="ID", time="TIME_h", sort="FORM"),
            observations = list(header="CONC_mg_L",
                                censoring=list(type="LLOQ", tags="BLQ", limits=0.06)),
           treatments = list(times=0, amount=600, admId=list(category="FORM", values=list("ref"=1,"test"=2))))

    # MONOLIX EXAMPLES

    initializeLixoftConnectors(software = "monolix")
    FormattedDataPath = tempfile("formatted_data")

    # demo doseIntervals_as_Occ.mlxtran
```

```
formatData(paste0(getDemoPath(),"/0.data_formatting/data/data_multidose.csv"),
           formattedFile = FormattedDataPath,
           headers = c(id="ID", time="TIME"),
           observations = list(header="CONC"),
           treatments = list(times=seq(0,by=12,length=7), amount=40),
           treatmentSettings = list(doseIntervalsAsOccasions = TRUE))

# demo warfarin_PKPDseq_project.mlxtran
formatData(paste0(getDemoPath(),"/0.data_formatting/data/warfarin_data.csv"),
           formattedFile = FormattedDataPath,
           headers = c(id="id", time="time"),
       additionalColumns = paste0(getDemoPath(),"/0.data_formatting/data/warfarinPK_regressors.txt"))
```

`generateReport` *[Monolix - PKanalix] Generate report*

Description

Generate a Word project report with default options or from a custom .docx template.

Usage

```
generateReport(
  templateFile = NULL,
  tablesStyle = NULL,
  watermark = NULL,
  reportFile = NULL
)
```

Arguments

`templateFile`	[optional] (*character*) Path to the .docx template file used as reporting base. If not provided, a default report file is generated (as default option in the GUI).
`tablesStyle`	[optional] (*character*)
`watermark`	[optional] (*list*) • text (*character*) • fontFamily (*character*) ["Arial"] • fontSize (*int*) [36] • color (*vector<int>*) Rgb color [c(255, 0, 0)] • layout (*character*) • semiTransparent (*bool*) [true]
`reportFile`	[optional] (*list*) If not provided, the report will be saved next to the project file with the name <projectname>_report.docx. • nextToProject (*bool*) Generate report file next to project • path (*character*) Path (nextToProject == FALSE) or name (nextToProject == TRUE) of the generated report file

Details

Reports can be generated as in the GUI, either by using the default reporting or by using a custom template. Placeholders for tables can be used in the template, and they are replaced by result tables. It is not possible to replace plots placeholders with the connector, because this requires an interface to be open. If plots placeholders are present in the template, they will be replaced by nothing in the generated report.

Examples

```
## Not run:
generateReport()
generateReport(templateFile = "/path/to/template.docx")
generateReport(templateFile = "/path/to/template.docx", tablesStyle = "Plain Table 1", watermark = list(text =

## End(Not run)
# Working example to generate a default report ###
initializeLixoftConnectors("monolix")
loadProject(file.path(getDemoPath(),"1.creating_and_using_models","1.1.libraries_of_models","warfarinPK_pro
runScenario()
reportPath = tempfile("report", fileext = ".docx")
generateReport(reportFile = list(nextToProject = FALSE, path = reportPath))
file.show(reportPath)

# Working example to generate a report with a custom template###
# Note that only tables get replaced. It is not possible to add plots to a report via connectors, but it can be do
initializeLixoftConnectors("pkanalix")
loadProject(file.path(getDemoPath(),"2.case_studies","project_aPCSK9_SAD.pkx"))
runScenario()
reportPath = tempfile("report", fileext = ".docx")
generateReport(templateFile = file.path(getDemoPath(),"2.case_studies","report_templates","PK_report_templa
file.show(reportPath)
```

---

getAddLines *[Simulx] Get lines added to the model*

Description

Get the lines that were added to a model with setAddLines (or in the GUI).

Usage

```
getAddLines()
```

Details

Additional equations can be added to the model file as in Simulx GUI. It is useful in case of import from Monolix or PKanalix, in order to add equations to the model, eg to compute an additional variable, without modifying the model file used for estimation and without impacting elements already defined.

See Also

setAddLines

Examples

```
initializeLixoftConnectors("simulx")
file_name <- file.path(getDemoPath(), "3.definition", "3.5.outputs", "output_addLines.smlx")
loadProject(file_name)
getAddLines()
```

`getAssessmentResults` *[Monolix] Get the results of the assessment*

Description

Get the results of the assessment.

Usage

```
getAssessmentResults()
```

Value

A vector of lists containing, for each assessment run:

- populationParameters: results of population parameter estimation using SAEM:
  - nbexploratoryiterations (*int*) number of iterations during exploratory phase
  - nbsmoothingiterations (*int*) number of iterations during smoothing phase
  - convergence (*data.frame*) convergence history of estimated population parameters and convergence indicator (-2*log-likelihood)
- standardErrors: [optional] results of standard errors estimation:
  - method (*string*) fisher method used (stochasticApproximation or linearization)
  - values (*vector*) standard error associated to each population parameter
- loglikelihood: [optional] results of log-likelihood estimation
  - method (*string*) fisher method used (importanceSampling or linearization)
  - AIC (*double*)
  - BIC (*double*)
  - BICc (*double*) modified BIC
  - LL (*double*)
  - chosenDegree (*int*) [importanceSampling]
  - standardError (*double*) [importanceSampling]
  - convergence (*data.frame*) [importanceSampling]

See Also

`runAssessment`

Examples

```
## Not run:
getAssessmentResults()

## End(Not run)
```

`getAssessmentSettings` *[Monolix] Get assessment settings*

Description

Get the settings that will be used during the run of assessment.

Usage

```
getAssessmentSettings()
```

Value

The list of settings

- nbRuns: (*int*) number of runs
- extendedEstimation: (*boolean*) if TRUE, standard errors and log-likelihood are estimated
- useLin: (*boolean*) if TRUE, use linearization to estimate standard errors and log-likelihood instead of stochastic approximation (sd) and importance sampling (ll)
- initialParameters: (*list*) a list giving, for each parameter, if its initial value is fixed (fixed = [FALSE]|TRUE) and, if is it not the case, the bounds within which the initial value is drawn (min = *double*, max = *double*)

See Also

`runAssessment`

Examples

```
## Not run:
set = getAssessmentSettings()
set$nbRuns = 5
set$extendedEstimation = TRUE
set$useLin = FALSE

set$initialParameters
  parameters  fixed  min   max
1         ka  FALSE  0.5  0.75
2          V   TRUE  NaN   NaN

runAssessment(settings = set)

## End(Not run)
```

getAvailableData *[Monolix - PKanalix] Get data sets descriptions*

Description

Get information about the data sets and filters defined in the project.

Usage

```
getAvailableData()
```

Value

A list containing a list containing elements that describe the data set:

- `name`: a string, the name of the data set
- `file`: a string, the path of the data set file
- `current`: a boolean indicating if the data set is applied (currently in use)
- `children`: a list containing lists with information about data sets created from this one using filters
- `filter` (only if the dataset was created using filters): a list containing name of the `parent` and details about filter `definition`

Examples

```
## Not run:
getAvailableData()

## End(Not run)
```

getBioequivalenceResults *[PKanalix] Get Bioequivalence results*

Description

Get results for different steps in bioequivalence analysis.

Usage

```
getBioequivalenceResults(...)
```

Arguments

`...`	(*string*) Name of the step whose values must be displayed : "anova", "coefficientsOfVariation", "confidenceIntervals"

Examples

```
## Not run:
bioeqResults = getBioequivalenceResults() # retrieve all the results values.
bioeqResults = getBioequivalenceResults("anova", "confidenceIntervals") # retrieve anova and confidence interv

## End(Not run)
```

`getBioequivalenceSettings` *[PKanalix] Get the settings associated to the bioequivalence estimation.*

Description

Get the settings associated to the bioequivalence estimation. Associated settings are:

"level"	(*int*)	Level of the confidence interval
"bioequivalenceLimits"	(*vector*)	Limit in which the confidence interval must be to conclude the bi
"computedBioequivalenceParameters"	(*data.frame*)	Parameters to consider for the bioequivalence analysis and if they
"linearModelFactors"	(*list*)	The values are headers of the data set, except for reference where
"degreesFreedom"	(*string*)	t-test using the residuals degrees of freedom assuming equal varia
"bedesign"	(*string*)	automatically recognize BE design "crossover" or "parallel" (can

Usage

```
getBioequivalenceSettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setBioequivalenceSettings

Examples

```
## Not run:
getBioequivalenceSettings() # retrieve a list of all the bioequivalence methodology settings
getBioequivalenceSettings("level","bioequivalenceLimits") # retrieve a list containing only the value of the s

## End(Not run)
```

getCACost *[Pkanalix] Get the value of minimized cost in CA*

Description

Get the value of the cost function minimized in CA, and additional criteria to compare models. The detailed formulae for cost, -2LL, AIC and BIC are given in https://pkanalix.lixoft.com/ca-settings/.

- Cost: weighted sum of squared residuals. Weights are specified in setCAsettings. Additional weights are applied to balance the number of observations in each profile.
- -2LL: Twice the negative log-likelihood based on the obtained cost.
- AIC: Akaike Information Criteria calculated based on a penalization of -2LL by the number of optimized parameters.
- BIC: Bayesian Information Criteria calculated based on a penalization of -2LL by the number of optimized parameters and the total number of data points.

Usage

```
getCACost()
```

Value

A data frame with the values of total cost, -2LL, AIC and BIC computed during the last run.

Examples

```
## Not run:
getCACost()
 -> data.frame( Cost = -2.2, -2LL = 15.12, AIC = 17.54, BIC = 18.45 )

## End(Not run)
```

getCAData *[PKanalix] Get data used for CA computation*

Description

Get data used to compute CA estimation

Usage

```
getCAData()
```

Examples

```
## Not run:
data = getCAData()
       ID time concentration censored
  1     1  0.0          0.00   FALSE
  2     1  0.5          3.05   FALSE
  3     1  2.0          5.92   TRUE

## End(Not run)
```

`getCAIndividualParameters`
[PKanalix] Get CA individual parameters

Description

Get the estimated values for each subject of some of the individual CA parameters of the current project.

Usage

```
getCAIndividualParameters(...)
```

Arguments

`...`	(*string*) Name of the individual parameters whose values must be displayed.

Value

A data frame giving the estimated values of the individual parameters of interest for each subject and a list of information relative to these parameters (units)

Examples

```
## Not run:
indivParams = getCAIndividualParameters() # retrieve all the available individual parameters values.
indivParams = getCAIndividualParameters("ka", "V") # retrieve ka and V values for all individuals.

  $parameters
      id  ka    V
      1   0.8  1.2
      .   ...  ...
      N   0.4  2.2


## End(Not run)
```

`getCAInitialValues` *[PKanalix] Get the initial values of individual parameters for the compartmental analysis*

Description

Get the list of initial values for all parameters of the model used for compartmental analysis.
Each element of the parameter list is a list of

"value"	(*double*)	Initial value to use. Must be in the limits in case of bounded constraint.
"constraint"	(*string*)	Constraint on the parameter. Possible values are "none", "positive" or "bounded".
"limits"	(*vector of doubles*)	[optional] Limits in case of bounded constraint.

Usage

```
getCAInitialValues()
```

`getCAParametersByAutoInit` *[PKanalix] Automatically estimate initial parameters values.*

Description

Run automatic calculation of optimized parameters for CA initial parameters set in the project.

Usage

```
getCAParametersByAutoInit()
```

Examples

```
## Not run:
autoinitvalues <- getCAParametersByAutoInit()
setCAInitialValues(initialValues = autoinitvalues)

## End(Not run)
```

`getCAParameterStatistics` *[PKanalix] Get CA parameter statistics*

Description

Get statistics over the estimated values of some of the CA parameters of the current project. Statistics are computed on the different sets of individuals resulting from the stratification settings passed in argument or, if not given, the ones previously set.

Usage

```
getCAParameterStatistics(parameters = c(), stratification = NULL)
```

Arguments

`parameters`	[optional](*vector<string>*) Name of the parameters whose values must be displayed.
`stratification`	[optional] Stratification to apply on results. By default, project one is applied. Stratification is a list containing: • state Stratification state • groups Stratification groups list See `setCAResultsStratification` for more details about this argument structure.

Value

A data frame giving the statistics over the parameters of interest, and a list of information relative to these parameters (units)

See Also

`setCAResultsStratification` `getCAResultsStratification` `setResultsStratificationGroups`

Examples

```
## Not run:
indivParams = getCAParameterStatistics()
# retrieve all the available parameters values.

indivParams = getCAParameterStatistics("ka", "V")
# retrieve ka and V values for all individuals.

  $parameters
 parameter       min       Q1   median       Q3      max     mean       SD       SE     CV  geoMean   geoSD
      ka 0.05742669 0.08886395 0.1186787 0.1495961 0.1983748 0.1221367 0.03898449 0.007957675 31.91873 0.1159
       V  7.859237  13.51599 23.00674 30.73677 43.39608 22.95211  10.99187  2.243707 47.89047 20.23981 1.

#' statistics = getCAParameterStatistics(stratification = list(groups = list(name = "WEIGHT", definition = c(65
# retrieve the values of all the available parameters splitted by WEIGHT.

## End(Not run)
```

`getCAResultsStratification` *[PKanalix] Get CA results stratification*

Description

Get the stratification used to compute NCA parameters stratistics table. Stratification is defined by:

- stratification covariate groups which are shared by both NCA and CA results
- a stratification state which is specific to each task results

Usage

```
getCAResultsStratification()
```

Details

For each covariate, stratification groups can be defined as a list with:

name	*string*	covariate name
definition	*vector<double>*(continuous) || list<vector<string»(categorical)	group separations (continuous) || modality s

A stratification state is represented as a list with:

split	*vector<string>*	ordered list of splitted covariates
filter	*list< pair<string, vector<int» >*	list of paired containing a covariate name and the indexes of associated kept gr

Value

A list with stratification groups ('groups') and stratification state ('state').

See Also

`setCAResultsStratification`

Examples

```
## Not run:
getCAResultsStratification()

  $groups
    list(
      list( name = "WEIGHT",
            definition = c(70),
            type = "continuous",
            range = c(65,85) ),
      list( name = "TRT",
            definition = list(c("a","b"), "c")
            type = "categorical",
            categories = c("a","b","c") )
```

```
    )

  $state
    $split
      "WEIGHT"
    $filter
      list(list("WEIGHT", c(1,3), list("TRT", c(1)))

## End(Not run)
```

`getCASettings` *[PKanalix] Get the settings associated to the compartmental analysis*

Description

Get the settings associated to the compartmental analysis. Associated settings are:

"weightingCA"	(*string*)	Type of weighting objective function. One of "uniform", "Yobs", "Ypred", "Ypred2", "Yobs2
"pool"	(*logical*)	Fit with individual parameters or with the same parameters for all individuals.
"blqMethod"	(*string*)	Method by which the BLQ data should be replaced. One of "zero", "LOQ", "LOQ2" or "miss

Usage

```
getCASettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setCASettings

Examples

```
## Not run:
getCASettings() # retrieve a list of all the CA methodology settings
getCASettings("weightingca","blqmethod") # retrieve a list containing only the value of the settings whose name

## End(Not run)
```

`getChartsData` *[Monolix - PKanalix] Compute Charts data with custom stratification options and custom computation settings*

Description

[Monolix - PKanalix] Compute Charts data with custom stratification options and custom computation settings

Usage

```
getChartsData(
  plotName,
  computeSettings = NULL,
  ids = NULL,
  splitGroup = NULL,
  colorGroup = NULL,
  filter = NULL
)
```

Arguments

`plotName`	(*string*) Name of the plot function.
`computeSettings`	(*list*) list with computational settings (it can include arguments from the settings argument of the plot, as well as obsName)
`ids`	list of ids to display (by default all ids are displayed).
`splitGroup`	data group criteria. a list, or a list of list with fields: • `name` : The name of the covariate to use in grouping, • `breaks` : In case of a continuous covariate, a list of break values, • `groups` : [optional] In case of a categorical covariate, define groups of modalities. (by default no split is applied).
`colorGroup`	data group criteria. a list, or a list of list with fields: • `name` : The name of the covariate to use in grouping, or the name of the column id, • `breaks` : In case of a continuous covariate, a list of break values, • `groups` : [optional] In case of a categorical covariate, define groups of modalities. (by default no color group is defined).
`filter`	data filtering criteria. a list, or a list of list with fields: • `name` : the name of the covariate to filter, • `cat` : in case of a categorical covariate, the name of the category to filter, • `interval` : in case of a continuous covariate, a list of filtering intervals. (by default no filtering is applied).

Value

A dataframe object or a list of dataframe object to pass to "data" argument of plot functions

Examples

```
## Not run:
  initializeLixoftConnectors(software = "pkanalix")
  project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
  loadProject(project)
  data <- getChartsData(plotName = "plotObservedData", ids = c(1, 2, 3, 4))
  data <- getChartsData(plotName = "plotNCAParametersCorrelation")

  initializeLixoftConnectors(software = "monolix")
  project <- file.path(getDemoPath(), "1.creating_and_using_models",
                       "1.1.libraries_of_models", "theophylline_project.mlxtran")
  loadProject(project)
  xBinsSettings <- list(is.fixedNbBins = TRUE, nbBins = 10)
  data <- getChartsData(plotName = "plotVpc",
                        computeSettings = list(xBinsSettings = xBinsSettings))
  data <- getChartsData(plotName = "plotVpc", computeSettings = list(level = 75))

  splitGroup <- list(name = "WEIGHT", breaks = c(75))
  filter <- list(name = "WEIGHT", interval = c(75, 100))
  data <- getChartsData(plotName = "plotVpc", splitGroup = splitGroup)
  data <- getChartsData(plotName = "plotVpc", filter = filter)

## End(Not run)
```

---

getConditionalDistributionSamplingSettings

[Monolix] Get conditional distribution sampling settings

Description

Get the conditional distribution sampling settings. Associated settings are:

"ratio"	(*0< double <1*)	Width of the confidence interval.
"enableMaxIterations"	(*bool*)	Enable maximum of iterations.
"nbMinIterations"	(*int >=1*)	Minimum number of iterations.
"nbMaxIterations"	(*int >=1*)	Maximum number of iterations.
"nbSimulatedParameters"	(*int >=1*)	Number of replicates.

Usage

```
getConditionalDistributionSamplingSettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setConditionalDistributionSamplingSettings

Examples

```
## Not run:
getConditionalDistributionSamplingSettings()
# retrieve all the conditional distribution sampling settings
getConditionalDistributionSamplingSettings("ratio","nbMinIterations")
# retrieve only the ratio and nbMinIterations settings values

## End(Not run)
```

---

getConditionalModeEstimationSettings

[Monolix] Get conditional mode estimation settings

Description

Get the conditional mode estimation settings. Associated settings are:

"nbOptimizationIterationsMode"	(*int >=1*)	Maximum number of iterations.
"optimizationToleranceMode"	(*double >0*)	Optimization tolerance.

Usage

```
getConditionalModeEstimationSettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setConditionalModeEstimationSettings

Examples

```
## Not run:
getConditionalModeEstimationSettings()
# retrieve a list of all the conditional mode estimation settings
getConditionalModeEstimationSettings("nbOptimizationIterationsMode")
# retrieve only the nbOptimizationIterationsMode setting value

## End(Not run)
```

`getConsoleMode` *[Monolix - Pkanalix - Simulx] Get console mode*

Description

Get console mode, ie current verbosity level (volume of output after running estimation tasks):

"none"	no output
"basic"	at each algorithm end, associated results are displayed
"complete"	each algorithm iteration and/or status is displayed

Usage

```
getConsoleMode()
```

Value

A string corresponding to current console mode

See Also

setConsoleMode

`getContinuousObservationModel`
[Monolix] Get continuous observation models information

Description

Get a summary of the information concerning the continuous observation models in the project. The following informations are provided.

- prediction: (*vector*<*string*>) name of the associated prediction
- formula: (*vector*<*string*>) formula applied on the observation
- distribution: (*vector*<*string*>) distribution of the observation in the Gaussian space. The distribution type can be "normal", "logNormal", or "logitNormal".
- limits: (*vector*< *pair*<*double,double*> >) lower and upper limits imposed to the observation. Used only if the distribution is logitNormal. If there is no logitNormal distribution, this field is empty.

- errormodel: (*vector<string>*) type of the associated error model
- autocorrelation: (*vector<bool>*) defines if there is auto correlation

Call `getObservationInformation` to get a list of the continuous observations present in the current project.

Usage

```
getContinuousObservationModel()
```

Value

A list associating each continuous observation to its model properties.

See Also

`getObservationInformation` `setObservationDistribution` `setObservationLimits` `setErrorModel` `setAutocorrelation`

Examples

```
## Not run:
obsModels = getContinuousObservationModel()
obsModels
 -> $prediction
      c(Conc = "Cc")
    $formula
      c(Conc = "Conc = Cc + (a+b*Cc)*e")
    $distribution
      c(Conc = "logitNormal")
    $limits
      list(Conc = c(0,11.5))
    $errormodel
      c(Conc = "combined1")
    $autocorrelation
      c(Conc = TRUE)

## End(Not run)
```

---

getCorrelationOfEstimates

[Monolix] Get the inverse of the Fisher Matrix

Description

Get the inverse of the last estimated Fisher matrix computed either by all the Fisher methods used during the last scenario run or by the specific one passed in argument.
WARNING: The Fisher matrix cannot be accessible until the Fisher algorithm has been launched once.
The user can choose to display only the Fisher matrix estimated with a specific method.
Existing Fisher methods :

Fisher by Linearization	"linearization"
Fisher by Stochastic Approximation	"stochasticApproximation"

WARNING: Only the methods which have been used during the last scenario run can provide results.

Usage

```
getCorrelationOfEstimates(method = "")
```

Arguments

method	[optional](*string*) Fisher method whose results should be displayed. If this field is not specified, the results provided by all the methods used during the last scenario run are displayed.

Value

A list whose each field contains the Fisher matrix computed by one of the available Fisher methods used during the ast scenario run. A matrix is defined as a structure containing the following fields :

rownames	list of row names
columnnames	list of column names
rownumber	number of rows
data	vector<...> containing matrix raw values (column major)

Examples

```
## Not run:
getCorrelationOfEstimates("linearization")
 -> list( linearization = list(data = c(1,0,0,0,1,-0.06,0,-0.06,1),
                               rownumber = 3,
                               rownames = c("Cl_pop","omega_Cl","a"),
                               columnnames = c("Cl_pop","omega_Cl","a")))


getCorrelationOfEstimates()
 -> list(linearization = list(...), stochasticApproximation = list(...) )

## End(Not run)
```

---

getCovariateElements *[Simulx] Get covariate elements*

Description

Get the list of all available covariate elements in the loaded project. To use one of these elements in simulation, please add it to a simulation group with setGroupElement.

Usage

```
getCovariateElements()
```

Details

Covariate elements can be defined with defineCovariateElement, or created by importing a Monolix project with importProject. Elements defined are created in the background and saved with the Simulx project if calling saveProject. They can be deleted with deleteElement.

Each element is a list of

"inputType"	(*string*)	Type of input definition: can be "manual", "distribution" or "external".
"file"	(*string*)	Path to the file if the inputType is external. NULL else wise.
"data"	(*data.frame*)	Values of the element.

Note that:
- if the project was created from a model file, a covariate element Covariates is created with all values equal 1.
- if the project was created by importing a Monolix project,

- a covariate element mlx_Cov is created with the values corresponding to the covariates values from the dataset of the Monolix project.
- a covariate element mlx_CovDist is created with the values corresponding to the estimation of the distribution of covariates in the dataset of the Monolix project.

The "distribution" type of covariate elements can only be created in the GUI. The "manual" type corresponds to elements created manually in the GUI, or with a data frame in connectors. The external type corresponds to elements created from an external file in the GUI or with the connectors.

See Also

defineCovariateElement

Examples

```
## Not run:
getCovariateElements()
$mlx_Cov
#'$mlx_Cov$inputType
[1] "external"

$mlx_Cov$file
[1] path/to/file

$mlx_Cov$data
   id WEIGHT
1   1   79.6
2   2   72.4
3   3   70.5
4   4   72.7
5   5   54.6
```

```
...

$covMan
$covMan$inputType
[1] "manual"

$covMan$file
NULL

$covMan$data
      id WEIGHT
1 common      75

## End(Not run)
# Get covariate elements in projects in which they were defined differently
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.2.covariates", "covariates_distribution.smlx")
loadProject(project_name)
getCovariateElements()

project_name <- file.path(getDemoPath(), "3.definition", "3.2.covariates", "covariates_external.smlx")
loadProject(project_name)
getCovariateElements()

project_name <- file.path(getDemoPath(), "3.definition", "3.2.covariates", "covariates_manual.smlx")
loadProject(project_name)
getCovariateElements()
```

`getCovariateInformation`

[Monolix - PKanalix] Get covariates information

Description

Get the name, the type and the values of the covariates present in the project.

Usage

```
getCovariateInformation()
```

Value

A list containing the following fields :

- name (*vector<string>*): covariate names
- type (*vector<string>*): covariate types. Existing types are "continuous", "continuoustransformed", "categorical", "categoricaltransformed"./ In Monolix mode, "latent" covariates are also allowed.
- [Monolix] modalityNumber (*vector<int>*): number of modalities (for latent covariates only)
- covariate: a data frame giving the values of continuous and categorical covariates for each subject. Latent covariate values exist only if they have been estimated, ie if the covariate is used and if the population parameters have been estimated. Call `getEstimatedIndividualParameters` to retrieve them.

Examples

```
## Not run:
info = getCovariateInformation() # Monolix mode with latent covariates
info
  -> $name
     c("sex","wt","lcat")
  -> $type
     c(sex = "categorical", wt = "continuous", lcat = "latent")
  -> $modalityNumber
     c(lcat = 2)
  -> $covariate
     id   sex    wt
      1    M   66.7
      .    .      .
      N    F   59.0

## End(Not run)
```

`getData` *[Monolix - PKanalix] Get project data*

Description

Get a description of the data used in the current project. Available informations are:

- dataFile (*string*): path to the data file
- header (*array<character>*): vector of header names
- headerTypes (*array<character>*): vector of header types
- observationNames (*vector<string>*): vector of observation names
- observationTypes (*vector<string>*): vector of observation types
- nbSSDoses (*int*): number of doses (if there is a SS column)

Usage

```
getData()
```

Value

A list describing project data.

See Also

`setData`

Examples

```
## Not run:
data = getData()
data
-> $dataFile
     "/path/to/data/file.txt"
   $header
```

```
    c("ID","TIME","CONC","SEX","OCC")
  $headerTypes
    c("ID","TIME","OBSERVATION","CATEGORICAL COVARIATE","IGNORE")
  $observationNames
    c("concentration")
  $observationTypes
    c(concentration = "continuous")

## End(Not run)
```

`getDataSettings` *[PKanalix] Get the data settings associated to the non compartmental analysis*

Description

Get the data settings associated to the non compartmental analysis. Associated settings are:

"urinevolume"	(*string*)	regressor name used as urine volume.
"datatype"	(*list*)	list("obsId" = *string*("plasma" or "urine")). The type of data associated with each *obsId*: observ
"units"	(*list*)	list with the units associated to "dose", "time", "volume" and "grading".
"scalings"	(*list*)	list with the scaling factor associated to "concentration", "dose", "time" and "urinevolume".
"enableunits"	(*bool*)	are units enabled or not.

Usage

```
getDataSettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setNCASettings

Examples

```
## Not run:
getDataSettings() # retrieve a list of all the NCA methodology settings
getDataSettings("urinevolume") # retrieve a list containing only the value of the settings whose name has been p

## End(Not run)
```

getDemoPath *Get Lixoft demos path*

Description

Get Lixoft demos path

Usage

```
getDemoPath()
```

Value

A string corresponding to Lixoft demos path corresponding to the currently active software.

Examples

```
## Not run:
  getDemoPath()

## End(Not run)
```

getEndpoints *[Simulx] Get endpoint elements*

Description

Get the list of all endpoints.

Usage

```
getEndpoints()
```

Details

Endpoints summarize the outcome values over all individuals, for each simulation group and each replicate. Endpoints are defined and calculated as in Simulx GUI.

To compute the defined endpoints, use runEndpoints and get the results with getEndpointsResults.

To check if endpoints will be compared across simulation groups, use getGroupComparisonSettings. If group comparison is relevant, the way the comparison will be done for each endpoint (eg if statistical test and which p-value) should be defined in the endpoint element.

Each element is a list of

- `outcome` (*string* or *list*) - Outcome on which the endpoint is based on. If one outcome, a string containing its name. If combined outcomes were used, a list with:
  - `names` (*vector* of strings) - Vector of outcome names
  - `groupName` (*string*) - Name of the outcome combination

  - `operator` (*string*) - Way in which output should be combined. One of "and"/"or" (in case of boolean outcomes) or "min"/"max" (in other cases)
- `metric` (*string*) - Calculation method for the endpoint. One of "arithmeticMean", "geometricMean" or "median" if value-based outcome. In case of event-based outcomes, "kaplan-Meier" (median survival) and in case of boolean outcomes, "percentTrue".
- `groupComparison` (*list*) - Group comparison settings containing:
  - `type` (*string*) - One of "directComparison", "statisticalTest"
  - `operator` (*string*) - One of "!=", ">" or "<",
  - `threshold` (*double*) - A real number indicating the threshold for difference/oddsRatio
  - `pvalue` (*double*) - A real number indicating the p-value (if `type` is "statisticalTest")

See Also

`defineEndpoint`

Examples

```
{
## Not run:
$median_average
$median_average$outcome
[1] "regularCc_average_id"

$median_average$metric
[1] "arithmeticMean"

$median_average$groupComparison
$median_average$groupComparison$type
[1] "statisticalTest"

$median_average$groupComparison$operator
[1] ">"

$median_average$groupComparison$threshold
[1] 0

$median_average$groupComparison$pvalue
[1] 0.05


$regular_between_003_005
$regular_between_003_005$metric
[1] "percentTrue"

$regular_between_003_005$outcome
$regular_between_003_005$outcome$groupName
[1] "regular_between_003_005_outcome"

$regular_between_003_005$outcome$names
[1] "regularCc_avg_idOcc_lower06"   "regularCc_avg_idOcc_greater03"

$regular_between_003_005$outcome$operator
[1] "and"
...
```

```
## End(Not run)
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomeEndpoint_PD
loadProject(project_name)
getEndpoints()
}
```

`getEndpointsResults` *[Simulx] Get endpoints results*

Description

Get the results of the outcomes & endpoints task. Outcomes, endpoints and group comparisons are calculated as in Simulx GUI with the task runEndpoints.The output is a list with outcomes, endpoints and comparison results if they have been computed.

Usage

```
getEndpointsResults()
```

See Also

runEndpoints

Examples

```
## Not run:
 getEndpointsResults()
 $outcomes
 $outcomes$outcome_per_id
       rep                   group  ID outcome_per_id
   1     1     washout_before_occ2   1        7.30854
   2     1     washout_before_occ2   2        7.36297
   3     1     washout_before_occ2   3        6.40377
   4     1     washout_before_occ2   4        7.49762
   5     1     washout_before_occ2   5        7.55129
    ...
 $endpoints
 $endpoints$endpoint_name
    rep                   group arithmeticMean standardDeviation
  1   1     washout_before_occ2        7.32404          0.775968
  2   2     washout_before_occ2        7.47649           1.12214
  3   3     washout_before_occ2        7.14622          0.827815
  ...
 $groupComparison
 $groupComparison$endpoint1
    rep                   group difference     p-value success
  1   1 no_whashout_before_occ2    1.09646 2.75024e-06    TRUE
  2   1               bothDoses    6.95433    <2.2e-16    TRUE
  3   2 no_whashout_before_occ2   0.990928 1.29062e-05    TRUE
  ...

## End(Not run)
initializeLixoftConnectors("simulx")
```

```
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomeEndpoint_PD
loadProject(project_name)
runScenario()
getEndpointsResults()
```

`getEstimatedIndividualParameters`
[Monolix] Get last estimated individual parameter values

Description

Get the last estimated values for each subject of some of the individual parameters present within the current project.
WARNING: Estimated individual parameters values cannot be accessible until the individual estimation algorithm has been launched once.
NOTE: The user can choose to display only the individual parameter values estimated with a specific method.
Existing individual estimation methods :

Conditional Mean SAEM	"saem"
Conditional Mean	"conditionalMean"
Conditional Mode	"conditionalMode"

WARNING: Only the methods which have been used during the last scenario run can provide estimation results.

Usage

```
getEstimatedIndividualParameters(..., method = "")
```

Arguments

`...`	(*string*) Name of the individual parameters whose values must be displayed. Call `getIndividualParameterModel` to get a list of the individual parameters present within the current project.
`method`	[optional](*string*) Individual parameter estimation method whose results should be displayed. If there are latent covariate used in the model, the estimated modality is displayed too If this field is not specified, the results provided by all the methods used during the last scenario run are displayed.

Value

A data frame giving, for each wanted method, the last estimated values of the individual parameters of interest for each subject with the corresponding standard deviation values.

See Also

`getEstimatedRandomEffects`

Examples

```
## Not run:
indivParams = getEstimatedIndividualParameters()
# retrieve the values of all the available individual parameters for all methods
  -> $saem
      id   Cl     V      ka
      1   0.28  7.71   0.29
      .   ...    ...    ...
      N   0.1047.62   1.51

indivParams = getEstimatedIndividualParameters("Cl", "V", method = "conditionalMean")
# retrieve the values of the individual parameters "Cl" and "V"
# estimated by the conditional mode method

## End(Not run)
```

`getEstimatedLogLikelihood`

[Monolix] Get Log-Likelihood values

Description

Get the values computed by using a log-likelihood algorithm during the last scenario run, with or without a method-based filter.
WARNING: The log-likelihood values cannot be accessible until the log-likelihood algorithm has been launched once.
The user can choose to display only the log-likelihood values computed with a specific method.
Existing log-likelihood methods :

Log-likelihood by Linearization	"linearization"
Log-likelihood by Important Sampling	"importanceSampling"

WARNING: Only the methods which have been used during the last scenario run can provide results.

Usage

```
getEstimatedLogLikelihood(method = "")
```

Arguments

`method`	[optional](*string*) Log-likelihood method whose results should be displayed. If this field is not specified, the results provided by all the methods used during the last scenario run are retrieved.

Value

A list associating the name of each method passed in argument to the corresponding log-likelihood values computed by during the last scenario run.

Examples

```
## Not run:
getEstimatedLogLikelihood()
 -> list(  linearization = [LL = -170.505, AIC = 350.280, BIC = 365.335] ,
           importanceSampling = [...] )

getEstimatedLogLikelihood("linearization")
 -> list(  linearization = [LL = -170.505, AIC = 350.280, BIC = 365.335] )

## End(Not run)
```

`getEstimatedPopulationParameters`

[Monolix] Get last estimated population parameter value

Description

Get the last estimated value of some of the population parameters present within the current project (fixed effects + individual variances + correlations + latent probabilities + error model parameters). WARNING: Estimated population parameters values cannot be accessible until the SAEM algorithm has been launched once.

Usage

```
getEstimatedPopulationParameters(..., coefficientsOfVariation = FALSE)
```

Arguments

`...`	[optional] (*array<string>*) Names of the population parameters whose value must be displayed. Call `getPopulationParameterInformation` to get a list of the population parameters present within the current project. If this field is not specified, the function will retrieve the values of all the available population parameters.
`coefficientsOfVariation`	[optional](*boolean*) if this option is TRUE, the standard deviations of random effects are also given as coefficients of variation (relative standard deviations as percentages) with _CV suffix.

Value

A named vector containing the last estimated value of each one of the population parameters passed in argument.

Examples

```
## Not run:
getEstimatedPopulationParameters("V_pop") -> [V_pop = 0.5]
getEstimatedPopulationParameters("V_pop","Cl_pop") -> [V_pop = 0.5, Cl_pop = 0.25]
getEstimatedPopulationParameters() -> [V_pop = 0.5, Cl_pop = 0.25, ka_pop = 0.05]

## End(Not run)
```

`getEstimatedRandomEffects`

[Monolix] Get estimated the random effects

Description

Get the random effects for each subject of some of the individual parameters present within the current project.
WARNING: Estimated random effects cannot be accessible until the individual estimation algorithm has been launched once.
The user can choose to display only the random effects estimated with a specific method.
NOTE: The random effects are defined in the gaussian referential, e.g. if ka is lognormally distributed around ka_pop, eta_i = log(ka_i)-log(ka_pop) Existing individual estimation methods :

Conditional Mean SAEM	"saem"
Conditional Mean	"conditionalMean"
Conditional Mode	"conditionalMode"

WARNING: Only the methods which have been used during the last scenario run can provide estimation results. Please call `getLaunchedTasks` to get a list of the methods whose results are available.

Usage

```
getEstimatedRandomEffects(..., method = "")
```

Arguments

`...`	(*string*) Name of the individual parameters whose random effects must be displayed. Call `getIndividualParameterModel` to get a list of the individual parameters present within the current project.
`method`	[optional](*string*) Individual parameter estimation method whose results should be displayed. If this field is not specified, the results provided by all the methods used during the last scenario run are displayed.

Value

A data frame giving, for each wanted method, the last estimated eta values of the individual parameters of interest for each subject with the corresponding standard deviation values.

See Also

`getEstimatedIndividualParameters`

Examples

```
## Not run:
etaParams = getEstimatedRandomEffects()
# retrieve the values of all the available random effects for all methods
# without the associated standard deviations
  -> $saem
```

```
      id    Cl     V      ka
       1   0.28  7.71   0.29
       .   ...    ...    ...
       N   0.1047.62   1.51

etaParams = getEstimatedRandomEffects("Cl", "V", method = "conditionalMode")
# retrieve the values of the individual parameters "Cl" and "V"
# estimated by the conditional mean from SAEM algorithm

## End(Not run)
```

`getEstimatedStandardErrors` *[Monolix] Get standard errors of population parameters*

Description

Get the last estimated standard errors of population parameters computed either by all the Fisher methods used during the last scenario run or by the specific one passed in argument.
WARNING: The standard errors cannot be accessible until the Fisher algorithm has been launched once.
Existing Fisher methods :

Fisher by Linearization	"linearization"
Fisher by Stochastic Approximation	"stochasticApproximation"

WARNING: Only the methods which have been used during the last scenario run can provide results.

Usage

```
getEstimatedStandardErrors(method = "")
```

Arguments

`method`	[optional](*string*) Fisher method whose results should be displayed. If this field is not specified, the results provided by all the methods used during the last scenario run are retrieved

Value

A list associating each retrieved Fisher algorithm method to a data frame containing the standard errors and relative standard errors (

Examples

```
## Not run:
getEstimatedStandardErrors() ->
  $linearization
     parameter          se       rse
        ka_pop 0.313449586 20.451556
         V_pop 0.020422507  4.483500
```

```
          omega_V 0.037975960 30.072470
         omega_Cl 0.062976601 23.270939

    $stochasticApproximation
       parameter          se       rse
          ka_pop 0.311284296 20.310278
           V_pop 0.020424882  4.484022
         omega_V 0.161053665 24.000077
        omega_Cl 0.035113095 27.805419

## End(Not run)
```

`getFixedEffectsByAutoInit` *[Monolix] Automatically estimate initial parameters value*

Description

Compute optimized values for initial population parameters. The values are returned in the same format as getPopulationParameterInformation.

Usage

```
getFixedEffectsByAutoInit(parameters = NULL)
```

Examples

```
## Not run:
getFixedEffectsByAutoInit() -> optimizedParameters
setPopulationParameterInformation(optimizedParameters)

## End(Not run)
```

`getFormatting` *[Monolix - PKanalix] Get data formatting from a loaded project*

Description

[Monolix - PKanalix] Get data formatting from a loaded project

Usage

```
getFormatting()
```

getGeneralSettings *[Monolix] Get project general settings*

Description

Get a summary of the common settings for Monolix algorithms. Associated settings are:

"autoChains"	(*bool*)	Automatically adjusted the number of chains to have at least a minimum number of sul
"nbChains"	(*int >0*)	Number of chains. *Used only if "autoChains" is set to FALSE.*
"minIndivForChains"	(*int >0*)	Minimum number of individuals by chain.

Usage

```
getGeneralSettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setGeneralSettings

Examples

```
## Not run:
getGeneralSettings() # retrieve a list of all the general settings

getGeneralSettings("nbChains","autoChains")
# retrieve only the nbChains and autoChains settings values.

## End(Not run)
```

getGroupComparisonSettings *[Simulx] Get group comparison settings*

Description

Set settings related to the comparison of endpoints across groups.

Usage

```
getGroupComparisonSettings()
```

Details

Endpoints summarize the outcome values over all individuals, for each simulation group and each replicate. Endpoints are defined with defineEndpoint and can be compared across groups as in Simulx GUI.

getGroupComparisonSettings enables to check if endpoints will be compared across simulation groups and which group will be used as a reference. Group comparison is performed during the Endpoints task.

See Also

setGroupComparisonSettings

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.2.group_comparison", "groupComp_PDTTE_med
loadProject(project_name)
getGroupComparisonSettings()
```

---

getGroupRemaining *[Simulx] Get remaining parameters for a simulation group*

Description

Get the values of the remaining parameters (typically the error model parameters) for a group.

Usage

```
getGroupRemaining(group)
```

Arguments

group	(*character*) Group name

Details

Remaining parameters are all parameters that appear in the structural model (in the input line of [LONGITUDINAL]) and are neither individual parameters not regressors. They are typically error model parameters.

If an individual parameters element is selected for simulation, and the model includes remaining parameters, it is possible to set their values with setGroupRemaining.

It typically enables to make a simulation with measurement noise, with an individual element. These error model parameters will impact the simulation only if a noisy observation (from the DEFINITION section of the [LONGITUDINAL] block) is set as output element (instead of a smooth prediction in OUTPUT or variable in EQUATION).

If a population parameters element is selected, it is not possible to set remaining parameters because these parameters are already part of the population element.

See Also

setGroupRemaining

Examples

```
# get default values of error model parameters when importing a Monolix project
initializeLixoftConnectors("monolix")
monolix_project <- file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "warf
initializeLixoftConnectors("simulx")
importProject(monolix_project)
setGroupElement(group = "simulationGroup1", elements = "mlx_EBEs")
getGroupRemaining(group = "simulationGroup1")
```

---

getGroups *[Simulx] Get simulation groups*

Description

Get the list of simulation groups and elements in each group.

Usage

```
getGroups()
```

Details

Simulation groups are used for simulation as in Simulx GUI. At the creation of a Simulx project, a first group is created by default with the name "simulationGroup1". Use getGroups to check which groups have already been defined, and which elements are set in each group. To add a simulation group, use addGroup. To remove a simulation group, use removeGroup. To add or change a group element, use setGroupElement. To define new elements, use one of the define...Element functions.

See Also

addGroup

Examples

```
## Not run:
# Working example
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "simulationGroups_ethnicGroups.smlx")
loadProject(project_name)
getGroups()
[[1]]
[[1]]$name
[1] "g_caucasian"

[[1]]$covariate
[1] "caucasian"

[[1]]$parameter
[[1]]$parameter$type
[1] "population"

[[1]]$parameter$name
[1] "manual_popParam"
```

```
[[1]]$treatment
[1] "single_dose"

[[1]]$output
[1] "regularY1" "regularCc"

[[1]]$size
[1] 42


[[2]]
[[2]]$name
[1] "g_asian"

[[2]]$covariate
[1] "asian"

[[2]]$parameter
[[2]]$parameter$type
[1] "population"

[[2]]$parameter$name
[1] "manual_popParam"


[[2]]$treatment
[1] "single_dose"

[[2]]$output
[1] "regularY1" "regularCc"

[[2]]$size
[1] 42

## End(Not run)
```

`getIndividualElements` *[Simulx] Get individual parameters elements*

Description

Get the list of all available individual parameters elements for the simulation. To use one of these elements in simulation, please add it to a simulation group with setGroupElement.

Usage

```
getIndividualElements()
```

Details

Individual parameters elements can be defined with defineIndividualElement, or created by importing a Monolix or PKanalix project. Elements defined are created in the background and saved with the Simulx project if calling saveProject. They can be deleted with deleteElement.

Each element is a list of

"inputType"	(*string*)	Type of input definition: can be "manual" or "external".
"file"	(*string*)	Path to the file if the inputType is external. NULL else wise.
"data"	(*data.frame*)	Values of the element.

If the Simulx project was created from a model file, an individual element IndivParameters is created with all values equal 1.

If the Simulx project was created by importing or exporting a Monolix project, the following individual elements are created:

- mlx_IndivInit is created with a vector of initial population parameters if the population parameters were not estimated.
- mlx_PopIndiv is created with a vector of estimated population parameters (without the covariate(s) impact(s)) if the population parameters were estimated.
- mlx_PopIndivCov is created with a table of estimated population parameters with covariate impact if the population parameters were estimated.
- mlx_EBEs is created with a table of estimated EBEs if the EBEs were estimated.
- mlx_CondMean is created with a table of estimated conditional mean for each individual if the conditional distribution task was run.
- mlx_CondDistSample is created with a table including the first sample from the conditional distribution for each individual if the conditional distribution task was run.

If the Simulx project was created by importing or exporting a PKanalix project, the following individual elements are created:

- an individual element pkx_IndivInit is created with a vector of initial parameters if the CA task has not run.
- an individual element pkx_Indiv is created with a table of individual parameters estimated in PKanalix if the CA task has run with the method "individual fit".
- an individual element pkx_IndivPooled is created with a vector of parameters estimated in PKanalix if the CA task has run with the method "pooled fit".
- an individual element pkx_IndivGeoMean is created with a vector of individual parameters corresponding to the geometric mean of individual parameters estimated in PKanalix if the CA task has run with the method "individual fit".

See Also

defineIndividualElement

Examples

```
## Not run:
getIndividualElements()
$mlx_PopIndiv
$mlx_PopIndiv$inputType
[1] "manual"

$mlx_PopIndiv$file
NULL

$mlx_PopIndiv$data
      id       Cl       V1        Q2       V2      Q3       V3
1 common 2.462069 4.557752 0.1151846 4.355022 1.70242 8.229753

$mlx_EBEs
$mlx_EBEs$inputType
[1] "external"

$mlx_EBEs$file
[1] file/to/path

$mlx_EBEs$data
   id       Cl       V1         Q2        V2        Q3        V3
1   1 2.870556 5.801418 2.12758339  5.963084 0.6649303 13.069996
2   2 2.899189 4.443222 0.31646327  5.226719 2.3678264  9.182623
...

## End(Not run)
# Get individual elements in projects in which they were defined differently
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.4.individual_parameters", "indivparam_external.s
loadProject(project_name)
getIndividualElements()

project_name <- file.path(getDemoPath(), "3.definition", "3.4.individual_parameters", "indivparam_manual.sml
loadProject(project_name)
getIndividualElements()
```

---

getIndividualParameterModel

[Monolix] Get individual parameter model

Description

Get a summary of the information concerning the individual parameter model. The available informations are:

- name: (*string*) name of the individual parameter
- distribution: (*string*) distribution of the parameter values. The distribution type can be "normal", "logNormal", or "logitNormal".
- formula: (*string*) formula applied on individual parameters distribution
- variability: a list giving, for each variability level, if individual parameters have variability or not

- covariateModel: a list giving, for each individual parameter, if the related covariates are used or not. If no covariate is used, this field is empty.
- correlationBlocks : a list giving, for each variability level, the blocks of the correlation matrix of the random effects. A block is represented by a vector of individual parameter names. If there is no block, this field is empty.

Usage

```
getIndividualParameterModel()
```

Value

A list of individual parameter model properties.

See Also

setIndividualParameterDistribution setIndividualParameterVariability setCovariateModel

Examples

```
## Not run:
indivModel = getIndividualParameterModel()
indivModel
 -> $name
      c("ka","V","Cl")
    $distribution
      c(ka = "logNormal", V = "normal", Cl = "logNormal")
    $formula
      "\\tlog(ka) = log(ka_pop) + eta_ka\\n\\n
       \\tlog(V) = V_pop + eta_V\\n\\n
       \\tlog(Cl) = log(Cl_pop) + eta_Cl\\n\\n"
    $variability
      list( id = c(ka = TRUE, V = FALSE, Cl = TRUE) )
    $covariateModel
      list( ka = c(age = TRUE, sex = FALSE, wt = TRUE),
            V = c(age = FALSE, sex = FALSE, wt = FALSE),
            Cl = c(age = FALSE, sex = FALSE, wt = FALSE) )
    $correlationBlocks
      list( id = c("ka","V","Tlag") )

## End(Not run)
```

---

getInterpretedData *[Monolix - PKanalix] Get interpreted project data*

Description

Get data after interpretation done by the software, how it is displayed in the Data tab in the interface. Interpretation of data includes, but is not limited to, data formatting, addition of doses through the ADDL column and steady state settings, addition of additional covariates, interpolation of regressors.

Usage

```
getInterpretedData()
```

`getLastRunStatus` *[Monolix - PKanalix] Get last run status*

Description

Return an execution report about the last run with a summary of the error which could have occurred.

Usage

```
getLastRunStatus()
```

Value

A structure containing

1. a boolean which equals TRUE if the last run has successfully completed,
2. a summary of the errors which could have occurred.

Examples

```
## Not run:
lastRunInfo = getLastRunStatus()
lastRunInfo$status
 -> TRUE
lastRunInfo$report
 -> ""

## End(Not run)
```

`getLaunchedTasks` *[Monolix] Get tasks with results*

Description

Get a list of the tasks which have results to provide. A task is the association of:

- an algorithm (string)
- a vector of methods (string) relative to this algorithm for the standardErrorEstimation and the loglikelihoodEstimation, TRUE or FALSE for the other one.

Usage

```
getLaunchedTasks()
```

Value

The list of tasks with results, indexed by algorithm names.

Examples

```
## Not run:
tasks = getLaunchedTasks()
tasks
 -> $populationParameterEstimation = TRUE
    $conditionalModeEstimation = TRUE
   $standardErrorEstimation = "linearization"

## End(Not run)
```

`getLibraryModelContent` *[Monolix - PKanalix - Simulx] Get a library model's content*

Description

[Monolix - PKanalix - Simulx] Get a library model's content

Usage

```
getLibraryModelContent(filename, print = TRUE)
```

Arguments

`filename`	(*string*) The filename of the requested model. Can start with "lib:", end with ".txt", but neither are mandatory.
`print`	(*logical*) If TRUE (default), model's content is printed with human-readable line breaks (alongside regular output with "\n").

Value

The model's content as a raw string.

Examples

```
## Not run:
getLibraryModelContent("oral1_1cpt_kaVCl")
model <- getLibraryModelContent(filename = "lib:oral1_1cpt_kaVCl.txt", print = FALSE)

## End(Not run)
```

`getLibraryModelName` *[Monolix - PKanalix - Simulx] Get the name of a library model given a list of library filters.*

Description

Models can be loaded from a library based on a selection of filters as in PKanalix, Monolix and Simulx GUI. For a complete description of each model library, and guidelines on how to select models, please visit https://mlxtran.lixoft.com/model-libraries/.

Usage

```
getLibraryModelName(library, filters = list())
```

Arguments

`library`	(*string*) One of the MonolixSuite library of models. Possible values are "pk", "pd", "pkpd", "pkdoubleabs", "pm", "tmdd", "tte", "count" and "tgi".
`filters`	(*list(name = string)*) Named list of filters (optional), format: list(filterKey = "filterValue", ...). Default empty list. Since available filters are not in any particular order, filterKey should always be stated.

Details

getLibraryModelName enables to get the name of the model to be loaded. You can then use it in setStructuralModel or newProject to load the model in an existing or in a new project.

All possible keys and values for each of the libraries are listed below.

PK library:

key	values
administration	bolus, infusion, oral, oralBolus
delay	noDelay, lagTime, transitCompartments
absorption	zeroOrder, firstOrder
distribution	1compartment, 2compartments, 3compartments
elimination	linear, MichaelisMenten
parametrization	rate, clearance, hybridConstants
bioavailability	true, false

PD library:

key	values
response	immediate, turnover
drugAction	linear, logarithmic, quadratic, Emax, Imax, productionInhibition, degradationInhibition, degradationStimulation, productionStimulation
baseline	const, 1-exp, exp, linear, null
inhibition	partialInhibition, fullInhibition
sigmoidicity	true, false

PKPD library:

key	values
administration	bolus, infusion, oral, oralBolus
delay	noDelay, lagTime, transitCompartments
absorption	zeroOrder, firstOrder
distribution	1compartment, 2compartments, 3compartments
elimination	linear, MichaelisMenten
parametrization	rate, clearance
bioavailability	true, false
response	direct, effectCompartment, turnover
drugAction	Emax, Imax, productionInhibition, degradationInhibition, degradationStimulation, productionStimulation
baseline	const, null
inhibition	partialInhibition, fullInhibition
sigmoidicity	true, false

PK double absorption library:

key	values
firstAbsorption	zeroOrder, firstOrder
firstDelay	noDelay, lagTime, transitCompartments
secondAbsorption	zeroOrder, firstOrder
secondDelay	noDelay, lagTime, transitCompartments
absorptionOrder	simultaneous, sequential
forceLongerDelay	true, false
distribution	1compartment, 2compartments, 3compartments
elimination	linear, MichaelisMenten
parametrization	rate, clearance

Parent-metabolite library:

key	values
administration	bolus, infusion, oral, oralBolus
firstPassEffect	noFirstPassEffect, withDoseApportionment, withoutDoseApportionment
delay	noDelay, lagTime, transitCompartments
absorption	zeroOrder, firstOrder
transformation	unidirectional, bidirectional
parametrization	rate, clearance
parentDistribution	1compartment, 2compartments, 3compartments
parentElimination	linear, MichaelisMenten
metaboliteDistribution	1compartment, 2compartments, 3compartments
metaboliteElimination	linear, MichaelisMenten

TMDD library:

key	values
administration	bolus, infusion, oral, oralBolus
delay	noDelay, lagTime, transitCompartments

absorption	zeroOrder, firstOrder
distribution	1compartment, 2compartments, 3compartments
tmddApproximation	MichaelisMenten, QE, QSS, full, Wagner, constantRtot, constantRtotIB, irreversibleBinding
output	totalLigandLtot, freeLigandL
parametrization	rate, clearance

TTE library:

key	**values**
tteModel	exponential, Weibull, Gompertz, loglogistic, uniform, gamma, generalizedGamma
delay	true, false
numberOfEvents	singleEvent, repeatedEvents
typeOfEvent	intervalCensored, exact
dummyParameter	true, false

Count library:

key	**values**
countDistribution	Poisson, binomial, negativeBinomial, betaBinomial, generalizedPoisson, geometric, hypergeometric, logarithmic, Bernoulli
zeroInflation	true, false
timeEvolution	constant, linear, exponential, Emax, Hill
parametrization	probabilityOfSuccess, averageNumberOfCounts

TGI library:

key	**values**
shortcut	ClaretExponential, Simeoni, Stein, Wang, Bonate, Ribba, twoPopulation
initialTumorSize	asParameter, asRegressor
kinetics	true, false
model	linear, quadratic, exponential, generalizedExponential, exponentialLinear, Simeoni, Koch, logistic, generalizedLogistic, SimeoniLogisticHybrid, Gompertz, exponentialGompertz, vonBertalanffy, generalizedVonBertalanffy
additionalFeature	none, angiogenesis, immuneDynamics
treatment	none, pkModel, exposureAsRegressor, startAtZero, startTimeAsRegressor, armAsRegressor
killingHypothesis	logKill, NortonSimon
dynamics	firstOrder, MichaelisMenten, MichaelisMentenHill, exponentialKill, constant
resistance	ClaretExponential, resistantCells, none
delay	signalDistribution, cellDistribution, none
additionalTreatmentEffect	none, angiogenesisInhibition, immuneEffectorDecay

Value

Name of the filtered model, or vector of names of the available models if not all filters were selected. Names start with "lib:".

Examples

```
## Not run:
getLibraryModelName(library = "pk", filters = list(administration = "oral", delay = "lagTime", absorption = "fi
# returns "lib:oral1_1cpt_TlagkaVCl.txt"
getLibraryModelName("pd", list(response = "turnover", drugAction = "productionStimulation"))
# returns c("lib:turn_input_Emax.txt", "lib:turn_input_gammaEmax.txt")

## End(Not run)
```

---

getLixoftConnectorsState *Get lixoftConnectors API current state*

Description

Retrieve information about the Lixoft software the lixoftConnectors are currently interfacing with.

Usage

```
getLixoftConnectorsState(quietly = FALSE)
```

Arguments

quietly	*boolean* If TRUE, no warning is raised when lixoftConnectors package is not initialized. Equals FALSE by default.

Value

A structure containing:

- path: path to Lixoft installation directory
- software: software name
- version: Lixoft softwares suite version

---

getLixoftEnvInfo *Get information about LixoftEnvironment object*

Description

Get information about LixoftEnvironment object

Usage

```
getLixoftEnvInfo()
```

getLogLikelihoodEstimationSettings *[Monolix] Get LogLikelihood algorithm settings*

Description

Get the loglikelihood estimation settings. Associated settings are:

"nbFixedIterations"	(*int >0*)	Monte Carlo size for the loglikelihood evaluation.
"samplingMethod"	(*string*)	Should the loglikelihood estimation use a given number of freedom d
"nbFreedomDegrees"	(*int >0*)	Degree of freedom of the Student t-distribution. *Used only if "samplin*
"freedomDegreesSampling"	(*vector<int(>0)>*)	Sequence of freedom degrees to be tested. *Used only if "samplingMet*

Usage

```
getLogLikelihoodEstimationSettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setLogLikelihoodEstimationSettings

Examples

```
## Not run:
getLogLikelihoodEstimationSettings() # retrieve a list of all the loglikelihood estimation settings
getLogLikelihoodEstimationSettings("nbFixedIterations","samplingMethod")
# retrieve only nbFixedIterations and samplingMethod settings values

## End(Not run)
```

getMapping *[Monolix - PKanalix] Get mapping*

Description

Get mapping between data and model.

Usage

```
getMapping()
```

Value

A list of mapping information:

- mapping (*list<list>*) A list of lists representing a link between data and model. Each list contains:
  - data (*string*) Data name
  - prediction (*string*) Prediction name
  - model [Monolix] (*string*) Model observation name (for continuous observations only)
  - type (*string*) Type of linked data ("continuous" | "discrete" | "event")
- freeData (*list<list>*) A list of lists describing not mapped data:
  - data (*string*) Data name
  - type (*string*) Data type
- freePredictions (*list<list>*) A list of lists describing not mapped predictions:
  - prediction (*string*) Prediction name
  - type (*string*) Prediction type

See Also

setMapping

Examples

```
## Not run:
f = getMapping()
f$mapping
 -> list( list(data = "1", prediction = "Cc", model = "concentration", type = "continuous"),
          list(data = "2", prediction = "Level", type = "discrete") )
f$freeData
  -> list( list(data = "3", type = "event") )
f$freePredictions
  -> list( list(prediction = "Effect", type = "continuous") )

## End(Not run)
```

---

getMCMCSettings *[Monolix] Get MCMC algorithm settings*

Description

Get the MCMC algorithm settings of the current project. Associated settings are:

"strategy"	(*vector<int>[3]*)	Number of calls for each one of the three MCMC kernels.
"acceptanceRatio"	(*double*)	Target acceptance ratio.

Usage

```
getMCMCSettings(...)
```

Arguments

...	[optional] (string) Names of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setMCMCSettings

Examples

```
## Not run:
getMCMCSettings() # retrieve a list of all the MCMC settings
getMCMCSettings("strategy") # retrieve only the strategy setting

## End(Not run)
```

---

getModelBuildingResults

[Monolix] Get the results of the model building

Description

Get the results (detailed models) of the model building.

Usage

```
getModelBuildingResults()
```

Value

The results of model building All the detailed tried models are returned

- LL: result of -2*Log-Likelihood
- BICc: modified BIC.
- individualModels: (*data.frame*) individual model for each individual parameter. The columns are the covariates and the elements of the data.frame notes if a covariate is used or not for the current parameter.

COSSAC send 2 additional fields:

- tested: (*vector<string>*) first element is the individual parameter and the second one is the covariate. This combination notes if the covariate is tested or not with respect to the previous model.
- bestModel (*boolean*) best model amongst all the tried models according to the chosen criterion.

SAMBA send the error model and covariance model information if there are exist

- errorModels: chosen type for each error model
- covarianceModels: chosen correlations between individual parameters

See Also

runModelBuilding

Examples

```
## Not run:
getModelBuildingResults()

## End(Not run)
```

`getModelBuildingSettings`
[Monolix] Get model building settings

Description

Get the settings that will be used during the run of model building.

Usage

```
getModelBuildingSettings()
```

Value

The list of settings

- covariates: (*list<string>*) covariate names
- parameters: (*list<string>*) parameters names
- strategy: (*string*) strategy to search best model ([cossac], samba, covsamba, scm)
- criterion: (*string*) crtierion to search best model ([BIC], LRT)
- relationships: (*data.frame<parameters, covariates, locked>*) Use to lock relationships between parameters and covariates. By default, all the combinations are possible. This parameter forces the use or not of some combinations. See example where *ka* must have *SEX* and *V* must not have *WEIGHT*
- threshold$lrt: threshold used by criterion LRT to continue or not to improve the model (first element is for forward and the second one is for the backward method)
- threshold$correlation: threshold used by cossac to choose what combinations (parameter-covariate) must be tried as next candidate model (first element is for forward and the second one is for the backward method)
- useLin: (*boolean*) computes linearization ([TRUE]) or the Importance Sampling (FALSE)

See Also

runModelBuilding

Examples

```
## Not run:
set = getModelBuildingSettings()
set$relationships[1,] = c("ka", "SEX", TRUE)
set$relationships[2,] = c("V", "WEIGHT", FALSE)

-> set$relationships
  parameters covariates locked
1         ka        SEX   TRUE
2          V     WEIGHT  FALSE

runModelBuilding(settings = set)

## End(Not run)
```

---

getNbReplicates *[Simulx] Get number of replicates*

Description

Get the number of replicates of the simulation.

Usage

```
getNbReplicates()
```

Details

The number of replicates is the number of times that Simulx will simulate a given study. It should not be mixed up with the group size defined by setGroupSize.

To simulate one study, Simulx samples a number of individuals for each group, defined by setGroupSize (let's say NidsPerGroup). To simulate replicate studies, it will sample for each replicate NidsPerGroup other individuals for each group (it is like changing the seed).

If the parameter element is an individual element or a population parameter defined with a vector, replicates will always sample individuals using the same population parameters. In this case, they are useful to check the effect of changing the seed, to get for example the uncertainty of an endpoint due to limited sampling.

If the parameter element is a population element defined with a table containing several lines, or an imported element such as mlx_PopUncertainSA or mlx_TypicalUncertainSA, each replicate will use a different population parameter to simulate the study. In this case, it is possible to see the effect of changing the population parameters on the prediction (in addition to uncertainty due to limited sampling).

See Also

setNbReplicates

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "replicates.smlx")
loadProject(project_name)
getNbReplicates()
```

getNCAData *[PKanalix] Get data used for NCA computation*

Description

Get data used to compute lambda_Z in NCA estimation

Usage

```
getNCAData()
```

Examples

```
## Not run:
data = getNCAData()
       ID time concentration censored
  1     1  0.0          0.00   FALSE
  2     1  0.5          3.05   FALSE
  3     1  2.0          5.92   TRUE

## End(Not run)
```

getNCAIndividualParameters *[PKanalix] Get NCA individual parameters*

Description

Get the estimated values for each subject of some of the individual NCA parameters of the current project.

Usage

```
getNCAIndividualParameters(...)
```

Arguments

...	(*string*) Name of the individual parameters whose values must be displayed. Possible parameters: • parameters related to the calculation of lambda_z: "Rsq", "Rsq_adjusted", "Corr_XY", "No_points_lambda_z", "Lambda_z", "Lambda_z_lower", "Lambda_z_upper", "HL_Lambda_z", "Lambda_z_intercept", "Span" • parameters calculated in case of plasma data: "Tlag", "T0", "Dose", "N_Samples", "C0", "Tmax", "Cmax", "Cmax_D", "Tlast", "Clast", "AUClast", "AUClast_D", "AUMClast", "MRTlast", "MRTlast", "AUCall", "AUCINF_obs", "AUCINF_D_obs", "AUC_PerCentExtrap_obs", "AUC_PerCentBack_Ext_obs", "AUMCINF_obs", "AUMC_PerCentExtrap_obs", "MRTINF_obs", "MRTINF_obs", "Vz_F_obs", "Cl_F_obs", "Vz_obs", "Cl_obs", "Vss_obs", "Clast_pred", "AUCINF_pred", "AUCINF_D_pred", "AUC_PerCentExtrap_pred", "AUC_PerCentBack_Ext_ "AUMCINF_pred", "AUMC_PerCentExtrap_pred", "MRTINF_pred", "MRTINF_pred", "Vz_F_pred", "Cl_F_pred", "Vz_pred", "Cl_pred", "Vss_pred"

- parameters calculated in case of plasma data if partial AUC calculation intervals are provided through the *partialAucTime*: "AUC_lower_upper", "AUC_lower_upper_D", "CAVG_lower_upper"
- parameters calculated only for multiple dose data: "Tau", "Ctau", "Ctrough", "AUC_TAU", "AUC_TAU_D", "AUC_TAU_PerCentExtrap", "AUMC_TAU", "Vz_F", "Vz", "CLss_F", "CLss", "Cavg", "FluctuationPerCent", "FluctuationPerCent_Tau", "Accumulation_Index", "Swing", "Swing_Tau", "Tmin", "Cmin", "Cmax", "MRTINF_obs"
- parameters calculated in case of urine data: "T0", "Dose", "N_Samples", "Tlag", "Tmax_Rate", "Max_Rate", "Mid_Pt_last", "Rate_last", "AURC_last", "AURC_last_D", "Vol_UR", "Amount_Recovered", "Percent_Recovered", "AURC_all", "AURC_INF_obs", "AURC_PerCentExtrap_obs", "AURC_INF_pred", "AURC_PerCentExtrap_pred", "AURC_lower_upper", "Rate_last_pred"
- parameters calculated in case of urine data if partial AUC calculation intervals are provided through the *partialAucTime*: "AURC_lower_upper", "AURC_lower_upper_D"

Value

A data frame giving the estimated values of the individual parameters of interest for each subject, and a list of information relative to these parameters (units & CDISC names)

Examples

```
## Not run:
indivParams = getNCAIndividualParameters()
# retrieve the values of all the available parameters.

indivParams = getNCAIndividualParameters("Tmax","Clast")
# retrieve only the values of Tmax and Clast for all individuals.

   $parameters
      id  Tmax Clast
      1   0.8  1.2
      .   ...  ...
      N   0.4  2.2

   $information
          CDISC
     Tmax  TMAX
     Clast CLST


## End(Not run)
```

---

getNCAParameterStatistics

[PKanalix] Get NCA parameter statistics

Description

Get statistics over the estimated values of some of the NCA parameters of the current project. Statistics are computed on the different sets of individuals resulting from the stratification settings passed in argument or, if not given, the ones previously set.

Usage

```
getNCAParameterStatistics(parameters = c(), stratification = NULL)
```

Arguments

`parameters` [optional](*vector<string>*) Name of the parameters whose values must be displayed and a list of information relative to these parameters (units & CDISC names). Possible parameters:

- parameters related to the calculation of lambda_z: "Rsq", "Rsq_adjusted", "Corr_XY", "No_points_lambda_z", "Lambda_z", "Lambda_z_lower", "Lambda_z_upper", "HL_Lambda_z", "Lambda_z_intercept", "Span"
- parameters calculated in case of plasma data: "Tlag", "T0", "Dose", "N_Samples", "C0", "Tmax", "Cmax", "Cmax_D", "Tlast", "Clast", "AUClast", "AUClast_D", "AUMClast", "MRTlast", "MRTlast", "AUCall", "AUCINF_obs", "AUCINF_D_obs", "AUC_PerCentExtrap_obs", "AUC_PerCentBack_Ext_obs", "AUMCINF_obs", "AUMC_PerCentExtrap_obs", "MRTINF_obs", "MRTINF_obs", "Vz_F_obs", "Cl_F_obs", "Vz_obs", "Cl_obs", "Vss_obs", "Clast_pred", "AUCINF_pred", "AUCINF_D_pred", "AUC_PerCentExtrap_pred", "AUC_PerCentBack_Ext_ "AUMCINF_pred", "AUMC_PerCentExtrap_pred", "MRTINF_pred", "MRTINF_pred", "Vz_F_pred", "Cl_F_pred", "Vz_pred", "Cl_pred", "Vss_pred"
- parameters calculated in case of plasma data if partial AUC calculation intervals are provided through the *partialAucTime*: "AUC_lower_upper", "AUC_lower_upper_D", "CAVG_lower_upper"
- parameters calculated only for multiple dose data: "Tau", "Ctau", "Ctrough", "AUC_TAU", "AUC_TAU_D", "AUC_TAU_PerCentExtrap", "AUMC_TAU", "Vz_F", "Vz", "CLss_F", "CLss", "Cavg", "FluctuationPerCent", "FluctuationPerCent_Tau", "Accumulation_Index", "Swing", "Swing_Tau", "Tmin", "Cmin", "Cmax", "MRTINF_obs"
- parameters calculated in case of urine data: "T0", "Dose", "N_Samples", "Tlag", "Tmax_Rate", "Max_Rate", "Mid_Pt_last", "Rate_last", "AURC_last", "AURC_last_D", "Vol_UR", "Amount_Recovered", "Percent_Recovered", "AURC_all", "AURC_INF_obs", "AURC_PerCentExtrap_obs", "AURC_INF_pred", "AURC_PerCentExtrap_pred", "AURC_lower_upper", "Rate_last_pred"
- parameters calculated in case of urine data if partial AUC calculation intervals are provided through the *partialAucTime*: "AURC_lower_upper", "AURC_lower_upper_D"

`stratification` [optional] Stratification to apply on results. By default, project one is applied. Stratification is a list containing:

- state Stratification state
- groups Stratification groups list

See `setNCAResultsStratification` for more details about this argument structure.

See Also

`setNCAResultsStratification` `getNCAResultsStratification` `setResultsStratificationGroups`

Examples

```
## Not run:
statistics = getNCAParameterStatistics()
```

```
# retrieve the values of all the available parameters.

statistics = getNCAParameterStatistics(parameters = c("Tmax","Clast"))
# retrieve only the values of Tmax and Clast for all individuals.

  $statistics
 parameter    min    Q1  median     Q3    max    mean      SD      SE     CV Ntot Nobs Nmiss  geoMean   geoSD
    Tmax    2.5   2.5   2.75      3     3    2.75  0.3535534     0.25 12.85649   2   2    0 2.738613  1.1376
   Clast 0.76903 0.76903 0.85836  0.94769 0.94769  0.85836  0.1263317   0.08933  14.7178   2   2    0 0.853699

  $information
          units CDISC
  Tmax          h  Tmax
  Clast mg.mL^-1 Clast

statistics = getNCAParameterStatistics(stratification = list(groups = list(name = "WEIGHT", definition = c(65.
# retrieve the values of all the available parameters splitted by WEIGHT.

## End(Not run)
```

`getNCAResultsStratification`

[PKanalix] Get NCA results stratification

Description

Get the stratification used to compute NCA parameters stratistics table. Stratification is defined by:

- stratification covariate groups which are shared by both NCA and CA results
- a stratification state which is specific to each task results

Usage

```
getNCAResultsStratification()
```

Details

For each covariate, stratification groups can be defined as a list with:

name	*string*	covariate name
definition	*vector<double>*(continuous) || list<vector<string»(categorical)	group separations (continuous) || modality s

A stratification state is represented as a list with:

split	*vector<string>*	ordered list of splitted covariates
filter	*list< pair<string, vector<int»* >	list of paired containing a covariate name and the indexes of associated kept gr

Value

A list with stratification groups ('groups') and stratification state ('state').

See Also

setNCAResultsStratification

Examples

```
## Not run:
getNCAResultsStratification()

  $groups
    list(
      list( name = "WEIGHT",
            definition = c(70),
            type = "continuous",
            range = c(65,85) ),
      list( name = "TRT",
            definition = list(c("a","b"), "c")
            type = "categorical",
            categories = c("a","b","c") )
    )

  $state
    $split
      "WEIGHT"
    $filter
      list("Span", list("TRT", c(1)))

## End(Not run)
```

`getNCASettings` *[PKanalix] Get the settings associated to the non compartmental analysis*

Description

Get the settings associated to the non compartmental analysis. Associated settings are:

"obsidtouse"	(*string*)	The observation id from data section to use for computations
"administrationType"	(*list*)	list(key = "admId", value = *string*("intravenous" or "extravascular")). *admId*
"integralMethod"	(*string*)	Method for AUC and AUMC calculation and interpolation.
"partialAucTime"	(*list*)	The first element of the list is a bolean describing if this setting is used. The
"blqMethodBeforeTmax"	(*string*)	Method by which the BLQ data before Tmax should be replaced.
"blqMethodAfterTmax"	(*string*)	Method by which the BLQ data after Tmax should be replaced.
"ajdr2AcceptanceCriteria"	(*list*)	The first element of the list is a boolean describing if this setting is used. Th
"extrapAucAcceptanceCriteria"	(*list*)	The first element of the list is a boolean describing if this setting is used. Th

"spanAcceptanceCriteria"	(*list*)	The first element of the list is a boolean describing if this setting is used. Th
"lambdaRule"	(*string*)	Main rule for the lambda_Z estimation.
"timeInterval"	(*vector*)	Time interval for the lambda_Z estimation when "*lambdaRule*" = "interval"
"timeValuesPerId"	(*list*)	list("idName" = idTimes,...): *idTimes* Observation times to use for the calcu
"nbPoints"	(*integer*)	Number of points for the lambda_Z estimation when "*lambdaRule*" = "poin
"maxNbOfPoints"	(*list*)	The first element of the list is a boolean describing if this setting is used. Th
"startTimeNotBefore"	(*list*)	The first element of the list is a boolean describing if this setting is used. Th
"weightingNCA"	(*string*)	Weighting method used for the regression that estimates lambda_Z.
"computedNCAParameters"	(*vector*)	All the parameters to compute during the analysis."

Usage

```
getNCASettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setNCASettings

Examples

```
## Not run:
getNCASettings() # retrieve a list of all the NCA methodology settings
getNCASettings("lambdaRule","integralMethod") # retrieve a list containing only the value of the settings whose

## End(Not run)
```

---

getObservationInformation

[Monolix - PKanalix] Get observations information

Description

Get the name, the type and the values of the observations present in the project.

Usage

```
getObservationInformation()
```

Value

A list containing the following fields :

- name (*vector<string>*): observation names.
- type (*vector<string>*): observation generic types. Existing types are "continuous", "discrete", "event".
- [Monolix] detailedType (*vector<string>*): observation specialized types set in the structural model. Existing types are "continuous", "bsmm", "wsmm", "categorical", "count", "exactEvent", "intervalCensoredEvent".
- [Monolix] mapping (*vector<string>*): mapping between the observation names (defined in the mlxtran project) and the name of the corresponding entry in the data set.
- ["obsName"] (*data.frame*): observation values for each observation id.

In PKanalix mode, the observation type is not provided as only continuous observations are allowed. Neither do the mapping as dataset names are always used.

Examples

```
## Not run:
info = getObservationInformation()
info
  -> $name
     c("concentration")
  -> $type # [Monolix]
     c(concentration = "continuous")
  -> $detailedType # [Monolix]
     c(concentration = "continuous")
  -> $mapping # [Monolix]
     c(concentration = "CONC")
  -> $concentration
       id   time concentration
        1    0.5     0.0
        .    .       .
        N    9.0    10.8

## End(Not run)
```

---

getOccasionElements *[Simulx] Get occasion elements*

Description

Get the content of the occasion element that will be used for the simulation. The element will be automatically used for simulation and does not need to be added to a simulation group.

Usage

```
getOccasionElements()
```

Details

The occasion element can be defined with defineOccasionElement, or created during the import of a Monolix or PKanalix project with importProject. It can be deleted with deleteOccasionElement.

The element is a list of

"id"	(*string*)	Ids of the occasions
"names"	(*string*)	Name of the occasions. If empty, no occasions will be used in the project.
"times"	(*data.frame*)	Times of the occasions.
"occasions"	(*data.frame*)	Values of the element. If empty, no occasions will be used in the project.

The structure of the occasion elements impacts the definition of all other elements in the Simulx project. If the occasion element is subject-specific, other elements (parameters, covariates, treatments, outputs and regressors) must be either common over all subjects and all occasions, or they must be defined with subject-specific occasion-wise values as an external table (with id and occ columns), with the same occasion structure.

See Also

defineOccasionElement

Examples

```
# Get occasion elements in projects in which they were defined differently
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.7.occasions", "occasions_common.smlx")
loadProject(project_name)
getOccasionElements()

project_name <- file.path(getDemoPath(), "3.definition", "3.7.occasions", "occasions_common_washout.smlx")
loadProject(project_name)
getOccasionElements()

project_name <- file.path(getDemoPath(), "3.definition", "3.7.occasions", "occasions_external.smlx")
loadProject(project_name)
getOccasionElements()

project_name <- file.path(getDemoPath(), "3.definition", "3.7.occasions", "occasions_two_levels.smlx")
loadProject(project_name)
getOccasionElements()
```

---

getOutcomes *[Simulx] Get outcome elements*

Description

Get the list of all available outcomes.
The function returns a list containing the following elements:

output	(*string*)	Name of the output element on which the outcome is based.

perOccasion (*bool*) If occasions are present in the simulation, it indicates if the outcome is calculated for each id or

Then, if the outcome is based on a continuous or categorical output:

value - vector of boundaries if *operator* is "durationBelow", "durationAbove" or "durationBetween", or time point if *operator* is "timePoint".

applyThreshold (*list*) List of elements:

- operator - one of "==", "!=", ">=", ">", "<=" or "<",
- value - a real number indicating the threshold value.

Or if the outcome is based on a TTE output:

event (*list*) List of arguments that define event settings: type - one of "timeOfEvents" (in case of repeated TTE), "hasA

Usage

- `getOutcomes()`

See Also

`defineOutcome`

Examples

```
{
## Not run:
$outcome_per_id
$outcome_per_id$perOccasion
[1] FALSE

$outcome_per_id$output
[1] "regularCc"

$outcome_per_id$processing
$outcome_per_id$processing$operator
[1] "avg"


$hemorrhaging_id_timeOf_2nd
$hemorrhaging_id_timeOf_2nd$perOccasion
[1] FALSE

$hemorrhaging_id_timeOf_2nd$output
[1] "Hemorrhaging_until_150"

$hemorrhaging_id_timeOf_2nd$event
$hemorrhaging_id_timeOf_2nd$event$type
[1] "timeOfEvents"

$hemorrhaging_id_timeOf_2nd$event$rank
```

```
[1] 2
...

## End(Not run)

initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.1.outcome_endpoints", "OutcomeEndpoint_PD
loadProject(project_name)
getOutcomes()
}
```

`getOutputElements` *[Simulx] Get output elements*

Description

Get the list of all available output elements for simulation.

Usage

```
getOutputElements()
```

Details

Output elements can be defined with defineOutputElement, or created by importing a Monolix or a PKanalix project. Elements defined are created in the background and saved with the Simulx project if calling saveProject. They can be deleted with deleteElement.

Each element is a list of

"output"	(*string*)	Output name.
"inputType"	(*string*)	Type of input definition: can be "manual" or "external".
"file"	(*string*)	Path to the file if the inputType is external. NULL else wise.
"data"	(*data.frame*)	Values of the element.

Importantly, all the variables in the model file can be used as an output. The user is not constrained to the ones defined in the OUTPUT section of the model.

If the project was created from a model file, an output element is created for each output of the section OUTPUT of the model, with regular grid from 0 to 100 by steps of 1.

If the project was created by importing a Monolix project, with for example Cc as a prediction and y as measurement,

- an output element mlx_y1 is created as an external file with the times of the output for each id in the original dataset of the Monolix project.
- an output element mlx_Cc is created with a regular grid starting from the first time to the final time measured in the original dataset.

If the project was created by importing a PKanalix project, with for example Cc as a prediction, #'

- an output element pkx_Cc_OriginalTimes is created as an external file with the times of the output for each id in the original dataset of the Monolix project.
- an output element pkx_Cc_FineGrid is created with a regular grid starting from the first time to the final time measured in the original dataset.

See Also

defineOutputElement

Examples

```
## Not run:
$output
[1] "y1"

$inputType
[1] "manual"

$file
NULL

$data
        id time
1   common    0
2   common    1
3   common    2
...

## End(Not run)
  # Get output elements in projects in which they were defined differently
  initializeLixoftConnectors("simulx")
 project_name <- file.path(getDemoPath(), "3.definition", "3.5.outputs", "output_addLines.smlx")
  loadProject(project_name)
  getOutputElements()

 project_name <- file.path(getDemoPath(), "3.definition", "3.5.outputs", "output_intermOutputs.smlx")
  loadProject(project_name)
  getOutputElements()

 project_name <- file.path(getDemoPath(), "3.definition", "3.5.outputs", "output_mainOutputs.smlx")
  loadProject(project_name)
  getOutputElements()
```

---

getPlotPreferences *Define Preferences to customize plots*

Description

Define Preferences to customize plots

Usage

```
getPlotPreferences(plotName = NULL, update = NULL, ...)
```

Arguments

plotName	(*string*) Name of the plot function. if plotName is NULL, all preferences are returned
update	list containing the plot elements to be updated.
...	additional arguments - dataType for some plots

Details

This function creates a theme that customizes how a plot looks, i.e. legend, colors fills, transparencies, linetypes an sizes, etc. For each curve, list of available customizations:

- color: color (when lines or points)
- fill: color (when surfaces)
- opacity: color transparency
- radius: size of points
- shape: shape of points
- lineType: linetype
- lineWidth: line size
- legend: name of the legend (if NULL, no legend is displayed for the element)

Value

A list with theme specifiers

See Also

setPlotPreferences resetPlotPreferences

Examples

```
## Not run:
  preferences <- getPlotPreferences(update = list(
    obs = list(color = "red", legend = "Observations"),
    obsCens = list(color = rgb(70, 130, 180, maxColorValue = 255))
  ))
  # preferences that are used by default in the plots
  preferences <- getPlotPreferences()

  # preferences that are used by default in plotObservedData
  preferences <- getPlotPreferences(plotName = "plotObservedData")

## End(Not run)
```

---

getPointsIncludedForLambdaZ

[PKanalix] Get points included in lambda_Z computation

Description

Get points used to compute lambda_Z in NCA estimation

Usage

```
getPointsIncludedForLambdaZ()
```

Examples

```
## Not run:
pointsIncluded = getPointsIncludedForLambdaZ()
       ID time concentration BLQ includedForLambdaZ
  1     0  0.0          0.00   0                  0
  2     0  0.5          3.05   0                  1
  3     0  2.0          5.92   1                  1

## End(Not run)
```

---

getPopulationElements *[Simulx] Get population parameters elements*

Description

Get the list of all available population parameters elements for the simulation. To use one of these elements in simulation, please add it to a simulation group with setGroupElement.

Usage

```
getPopulationElements()
```

Details

Population parameters elements can be defined with definePopulationElement, or created at the import of a Monolix project with importProject. Elements defined are created in the background and saved with the Simulx project if calling saveProject. They can be deleted with deleteElement.

Each element is a list of

"inputType"	(*string*)	Type of input definition: can be "manual", "distribution" or "external".
"file"	(*string*)	Path to the file if the inputType is external. NULL else wise.
"data"	(*data.frame*)	Values of the element.

Notice that:
- if the project was created from a model file, a population element PopParameters is created with all values equal 1.
- if the project was created using by importing a Monolix project (with importProject),

- a population element mlx_Pop is created with the population parameters estimated in the Monolix project.
- if the parameters were not estimated in the Monolix project, a population element mlx_PopInit is created instead of mlx_Pop, with the initial values of the population parameters.
- a population element mlx_PopUncertainSA (resp. mlx_PopUncertainLin) is created which enables to sample population parameters using the covariance matrix of the estimates computed by Monolix if the Standard Error task (Estimation of the Fisher Information matrix) was performed by stochastic approximation (resp. by linearization). To sample several population parameter sets, this element needs to be used with replicates (usesetNbReplicates.
- a population element mlx_Typical is created with the population parameters estimated in the Monolix projects and all omega parameters set to zero. It is useful to simulate a typical individual with different covariate values than the reference in the model.

- a population element mlx_TypicalUncertainSA (resp. mlx_TypicalUncertainLin) is created which is the same as mlx_PopUncertain but with all omegas set to zero to remove the inter-individual variability. It is useful to propagate the uncertainty of population parameters to the prediction of a typical individual.

See Also

definePopulationElement

Examples

```
## Not run:
getPopulationElements()
$mlx_Pop
$mlx_Pop$inputType
[1] "manual"

$mlx_Pop$file
NULL

$mlx_Pop$data
    id  Cl_pop  V1_pop   Q2_pop  V2_pop Q3_pop  V3_pop omega_Cl omega_Q2 omega_Q3 omega_V1 omega_V2 omega
1 common 2.462069 4.557752 0.1151846 4.355022 1.70242 8.229753 0.2272315  0.92641 0.5745623 0.4080015 0.8127517

## End(Not run)
  # Get population elements in projects in which they were defined differently
  initializeLixoftConnectors("simulx")
 project_name <- file.path(getDemoPath(), "3.definition", "3.3.population_parameters", "pop_parameters_distr
  loadProject(project_name)
  getPopulationElements()

 project_name <- file.path(getDemoPath(), "3.definition", "3.3.population_parameters", "pop_parameters_exter
  loadProject(project_name)
  getPopulationElements()

 project_name <- file.path(getDemoPath(), "3.definition", "3.3.population_parameters", "pop_parameters_manua
  loadProject(project_name)
  getPopulationElements()
```

---

getPopulationParameterEstimationSettings

[Monolix] Get population parameter estimation settings

Description

Get the population parameter estimation settings. Associated settings are:

"nbBurningIterations"	(*int >=0*)	Number of iterations in the burn-in phase.
"nbExploratoryIterations"	(*int >=0*)	If "exploratoryAutoStop" is set to FALSE, it is the number of iteration
"exploratoryAutoStop"	(*bool*)	Should the exploratory step automatically stop.
"exploratoryInterval"	(*int >0*)	Minimum number of interation in the exploratory phase. *Used only if*
"exploratoryAlpha"	(*0<= double <=1*)	Convergence memory in the exploratory phase. *Used only if "explorat*
"nbSmoothingIterations"	(*int >=0*)	If "smoothingAutoStop" is set to FALSE, it is the number of iterations

"smoothingAutoStop"	(*bool*)	Should the smoothing step automatically stop.
"smoothingInterval"	(*int >0*)	inimum number of interation in the smoothing phase. *Used only if "sm*
"smoothingAlpha"	(*0.5< double <=1*)	Convergence memory in the smoothing phase. *Used only if "smoothin*
"smoothingRatio"	(*0< double <1*)	Width of the confidence interval. *Used only if "smoothingAutoStop" is*
"simulatedAnnealing"	(*bool*)	Should annealing be simulated.
"tauOmega"	(*double >0*)	Proportional rate on variance. *Used only if "simulatedAnnealing" is T*
"tauErrorModel"	(*double >0*)	Proportional rate on error model. *Used only if "simulatedAnnealing" i*
"variability"	(*string*)	Estimation method for parameters without variability: "firstStage" | "d
"nbOptimizationIterations"	(*int >=1*)	Number of optimization iterations.
"optimizationTolerance"	(*double >0*)	Tolerance for optimization.

Usage

```
getPopulationParameterEstimationSettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setPopulationParameterEstimationSettings

Examples

```
## Not run:
getPopulationParameterEstimationSettings()
# retrieve a list of all the population parameter estimation settings

getPopulationParameterEstimationSettings("nbBurningIterations","smoothingInterval")
# retrieve only the nbBurningIterations and smoothingInterval settings values

## End(Not run)
```

---

getPopulationParameterInformation

[Monolix] Get population parameters information

Description

Get the name, the initial value, the estimation method and, if relevant, MAP parameters value of the population parameters present in the project. It is available for fixed effects, random effects, error model parameters, and latent covariates probabilities.

Usage

```
getPopulationParameterInformation()
```

Value

A data frame giving, for each population parameter, the corresponding :

- initialValue : (*double*) initial value
- method : (*string*) estimation method
- priorValue : (*double*) [MAP] typical value
- priorSD : (*double*) [MAP] standard deviation

See Also

setPopulationParameterInformation

Examples

```
## Not run:
info = getPopulationParameterInformation()
info
    name      initialValue   method   typicalValue  stdDeviation
  ka_pop           1.0      MLE             NA            NA
  V_pop           10.0      MAP           10.0           0.5
  omega_ka         1.0     FIXED            NA            NA

## End(Not run)
```

---

getPreferences *[Monolix - PKanalix - Simulx] Get project preferences*

Description

Get a summary of the project preferences. Preferences are:

"relativepath"	(*bool*)	Use relative path for save/load operations.
"threads"	(*int >0*)	Number of threads.
"temporarydirectory"	(*string*)	Path to the directory used to save temporary files.
"timestamping"	(*bool*)	Create an archive containing result files after each run.
"delimiter"	(*string*)	Character use as delimiter in exported result files.
"exportchartsdata"	(*bool*)	Should charts data be exported.
"exportchartsdatasets"	(*bool*)	[Monolix] Should charts datasets be exported if possible.
"exportvpcsimulations"	(*bool*)	[Monoliw] Should vpc simulations be exported if possible.
"exportsimulationfiles"	(*bool*)	[Simulx] Should simulation results files be exported.
"headeraliases"	(*list("header" = vector<string>)*)	For each header, the list of the recognized aliases.
"ncaparameters"	(*vector<string>*)	[PKanalix] Defaulty computed NCA parameters.
"units"	(*list("type" = string)*	[PKanalix] Time, amount and/or volume units.

Usage

```
getPreferences(...)
```

Arguments

...	[optional] (string) Name of the preference whose value should be displayed. If no argument is provided, all the preferences are returned.

Value

An array which associates each preference name to its current value.

Examples

```
## Not run:
getPreferences() # retrieve a list of all the general settings

getPreferences("imageFormat","exportCharts")
# retrieve only the imageFormat and exportCharts settings values

## End(Not run)
```

---

getProjectSettings *[Monolix - PKanalix - Simulx] Get project settings*

Description

Get a summary of the project settings. Associated settings for Monolix projects are:

"directory"	(*string*)	Path to the folder where simulation results will be saved. It shou
"exportResults"	(*bool*)	Should results be exported.
"seed"	(*0< int <2147483647*)	Seed used by random generators.
"grid"	(*int*)	Number of points for the continuous simulation grid.
"nbSimulations"	(*int*)	Number of simulations.
"dataandmodelnexttoproject"	(*bool*)	Should data and model files be saved next to project.
"project"	(*string*)	Path to the Monolix project.

Associated settings for PKanalix projects are:

"directory"	(*string*)	Path to the folder where simulation results will be saved. It should be a wr
"seed"	(*0< int <2147483647*)	Seed used by random generators.
"datanexttoproject"	(*bool*)	Should data and model (in case of CA) files be saved next to project.

Associated settings for Simulx projects are:

"directory"	(*string*)	Path to the folder where simulation results will be saved. It should be
"seed"	(*0< int <2147483647*)	Seed used by random generators.
"userfilesnexttoproject"	(*bool*)	Should user files be saved next to project.

Usage

```
getProjectSettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

`getProjectSettings`

`getRegressorElements` *[Simulx] Get regressor elements*

Description

Get the list of all available regressor elements for simulation. To use one of these elements in simulation, please add it to a simulation group with setGroupElement. To simulate the model for times outside of the specified grid, last value carried forward interpolation is used.

Usage

```
getRegressorElements()
```

Details

Regressor elements can be defined with defineRegressorElement, or created by importing a Monolix or a PKanalix project with importProject. Elements defined are created in the background and saved with the Simulx project if calling saveProject. They can be deleted with deleteElement.

Each element is a list of

"inputType"	(*string*)	Type of input definition: can be "manual" or "external".
"file"	(*string*)	Path to the file if the inputType is external. NULL else wise.
"data"	(*data.frame*)	Values of the element.

Note that:

- if the project was created from a model file with regressors, a regressor element Regressors is created with one time point at 0 and all regressors equal 1.
- if the project was created by importing a Monolix or a PKanalix project with regressors, a regressor element mlx_Reg is created based on an external file with ids, times and regressor values and names read from the dataset of the Monolix project.

See Also

`defineRegressorElement`

Examples

```
## Not run:
$inputType
[1] "manual"

$file
NULL

$data
        id time
1   common    0
2   common    1
3   common    2
...

## End(Not run)
  # Get regressor elements in projects in which they were defined differently
  initializeLixoftConnectors("simulx")
 project_name <- file.path(getDemoPath(), "3.definition", "3.6.regressors", "regressor_external_PKPD.smlx")
  loadProject(project_name)
  getRegressorElements()

 project_name <- file.path(getDemoPath(), "3.definition", "3.6.regressors", "regressor_manual_paramCovRelati
  loadProject(project_name)
  getRegressorElements()
```

---

getResultsStratificationGroups

[Monolix - PKanalix - Simulx] Get results stratification groups

---

Description

Get the stratification covariate groups used to compute statistics over individual parameters. These groups are shared by all the task results.

Usage

```
getResultsStratificationGroups()
```

Details

For each covariate, stratification groups can be defined as a list with:

name	*string*	covariate name
definition	*vector*<*double*>(continuous) || list<vector<string»(categorical)	group separations (continuous) || modality s

Value

Stratification groups list.

See Also

setResultsStratificationGroups

Examples

```
## Not run:
getResultsStratificationGroups()

 list(
  list( name = "WEIGHT",
        definition = c(70),
        type = "continuous",
        range = c(65,85) ),
  list( name = "TRT",
        definition = list(c("a","b"), "c")
        type = "categorical",
       categories = c("a","b","c") )
  )

## End(Not run)
```

`getSAEMiterations` *[Monolix] Get SAEM algorithm iterations*

Description

Retrieve the successive values of some of the population parameters present within the current project (fixed effects + individual variances + correlations + latent probabilities + error model parameters) during the previous run of the SAEM algorithm.
WARNING: Convergence history of population parameters values cannot be accessible until the SAEM algorithm has been launched once.

Usage

```
getSAEMiterations(...)
```

Arguments

`...`	[optional] (*array<string>*) Names of the population parameters whose convergence history must be displayed. Call `getPopulationParameterInformation` to get a list of the population parameters present within the current project. If this field is not specified, the function will retrieve the values of all the available population parameters.

Value

A list containing a pair composed by the number of exploratory and smoothing iterations and a data frame which associates each wanted population parameter to its successive values over SAEM algorithm iterations.

Examples

```
## Not run:
report = getSAEMiterations()
report
  -> $iterationNumbers
       c(50,25)
```

```
$estimates
     V    Cl
  0.25     0
   0.3   0.5
     .     .
  0.35  0.25

## End(Not run)
```

`getSameIndividualsAmongGroups` *[Simulx] Get same individuals among groups*

Description

Get the information if the same individuals are simulated among all groups.

Usage

```
getSameIndividualsAmongGroups()
```

Details

setSameIndividualsAmongGroups(value = TRUE) allows to have the same individual parameters in all groups. It is available if the following elements (required for the sampling) are the same for all groups: size of groups, parameters (population or individual) and covariates.

The main goal is to make the comparison between groups easier.

In particular, it is used to compare different treatments on the same individuals - subjects with the same individual parameters.

Selecting same individuals among groups ensures that the differences between groups are only due to the treatment itself. To obtain the same conclusion without this option enabled, simulation should be performed on a very large number of individuals to averaged out the individual differences.

All options of the Simulx scenario are the same as in Simulx GUI. Check the online doc of Simulx to get more guidance on how to use them.

See Also

`setSameIndividualsAmongGroups`

Examples

```
 initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "simulationGroups_sameIdsInGroups.smlx")
 loadProject(project_name)
 getSameIndividualsAmongGroups()
```

getSamplingMethod *[Simulx] Get sampling method*

Description

Get which sampling method is used for the simulation. The possibilities are:

- keepOrder (default): individual values are taken in the same order as they appear in a table.
- withReplacement: individual values are sampled from a table with replacement.
- withoutReplacement: individual values are sampled from a table without replacement. This option is available only if tables contain at least the same number of individual values as a group size.

All of the above sampling methods are general and apply to all tables in a simulation scenario.

Usage

```
getSamplingMethod()
```

Details

All options of the Simulx scenario are the same as in Simulx GUI. Check the online doc of Simulx to get more guidance on how to use them.

See Also

setSamplingMethod

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "samplingOptions.smlx")
loadProject(project_name)
getSamplingMethod()
```

getScenario *[Monolix - PKanalix - Simulx] Get current scenario*

Description

Get the list of tasks that will be run at the next call to runScenario. For Monolix, get in addition the associated method (linearization true or false), and the associated list of plots.

Usage

```
getScenario()
```

Details

For Monolix, getScenario returns a given list of tasks, the linearization option and the list of plots. Every task in the list is associated to a boolean.
NOTE: Within a MONOLIX scenario, the order according to which the different algorithms are run is fixed:

Algorithm	Algorithm Keyword
Population Parameter Estimation	"populationParameterEstimation"
Conditional Mode Estimation (EBEs)	"conditionalModeEstimation"
Sampling from the Conditional Distribution	"conditionalDistributionSampling"
Standard Error and Fisher Information Matrix Estimation	"standardErrorEstimation"
LogLikelihood Estimation	"logLikelihoodEstimation"
Plots	"plots"

For PKanalix, getScenario returns a given list of tasks. Every task in the list is associated to a boolean.
NOTE: Within a PKanalix scenario, the order according to which the different algorithms are run is fixed:

Algorithm	Algorithm keyword
Non Compartmental Analysis	"nca"
Bioequivalence estimation	"be"

For Simulx, setScenario returns a given list of tasks. Every task in the list is associated to a boolean.
NOTE: Within a Simulx scenario, the order according to which the different algorithms are run is fixed:

Algorithm	Algorithm keyword
Simulation	"simulation"
Outcomes and endpoints	"endpoints"

Note: every task can also be run separately with a specific function, such as runSimulation in Simulx, runEstimation in Monolix. The CA task in PKanalix cannot be part of a scenario, it must be run with runCAEstimation.

Value

The list of tasks that corresponds to the current scenario, indexed by task names.

See Also

setScenario

Examples

```
## Not run:
[MONOLIX]
scenario = getScenario()
scenario
```

```
 -> $tasks
   populationParameterEstimation conditionalDistributionSampling conditionalModeEstimation standardErrorEsti
   TRUE                 TRUE  TRUE              FALSE             FALSE             FALSE
    $linearization = T
    $plotList = "outputplot", "vpc"

[PKANALIX]
scenario = getScenario()
scenario
    nca     be
   TRUE  FALSE

[SIMULX]
scenario = getScenario()
scenario
  simulation  endpoints
        TRUE      FALSE

## End(Not run)
```

`getSharedIds` *[Simulx] Get element types sharing individuals*

Description

Get the element types that will share the same individuals in the simulation.

Usage

```
getSharedIds()
```

Details

If several elements are defined with tables of individual values and set to some simulation groups, the option "shared ids" allows to create an intersection of ids present in these tables. After that, ids from this intersection are sampled to create the data for simulation.

All options of the Simulx scenario are the same as in Simulx GUI. Check the online doc of Simulx to get more guidance on how to use them.

See Also

`setSharedIds`

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "sharedIds.smlx")
loadProject(project_name)
getSharedIds()
```

`getSimulatedIndividualParameters` *[Monolix] Get simulated individual parameters*

Description

Get the simulated values for each replicate of each subject of some of the individual parameters present within the current project.
WARNING: Simulated individual parameters values cannot be accessible until the individual estimation with conditional mean algorithm has been launched once.

Usage

```
getSimulatedIndividualParameters(...)
```

Arguments

`...`	(*string*) Name of the individual parameters whose values must be displayed. Call `getIndividualParameterModel` to get a list of the individual parameters present within the current project.

Value

A list giving the last simulated values of the individual parameters of interest for each replicate of each subject.

See Also

`getSimulatedRandomEffects`

Examples

```
## Not run:
simParams = getSimulatedIndividualParameters()
# retrieve the values of all the available individual parameters

simParams
     rep   id    Cl     V     ka
      1    1   0.022  0.37  1.79
      1    2   0.033  0.42  -0.92
      .    .    ...    ...  ...
      2    1   0.021  0.33  1.47
      .    .    ...    ...  ...

## End(Not run)
```

`getSimulatedRandomEffects` *[Monolix] Get simulated random effects*

Description

Get the simulated values for each replicate of each subject of some of the individual random effects present within the current project.
WARNING: Simulated individual random effects values cannot be accessible until the individual estimation algorithm with conditional mean has been launched once.

Usage

```
getSimulatedRandomEffects(...)
```

Arguments

`...`	(*string*) Name of the individual parameters whose values must be displayed. Call `getIndividualParameterModel` to get a list of the individual parameters present within the current project.

Value

A list giving the last simulated values of the individual random effects of interest for each replicate of each subject.

See Also

`getIndividualParameterModel`

Examples

```
## Not run:
simEtas = getSimulatedRandomEffects()
# retrieve the values of all the available individual random effects

simEtas
     rep   id    Cl     V     ka
      1    1   0.022  0.37  1.79
      1    2   0.033  0.42  -0.92
      .    .    ...    ...  ...
      2    1   0.021  0.33  1.47
      .    .    ...    ...  ...

## End(Not run)
```

`getSimulationResults` *[Simulx] Get simulation results*

Description

Get the results of the simulation.
The output is a list of four elements: res, IndividualParameters, populationParameters and doses.

- res corresponds to the list of the output(s) of the simulation. It includes a data frame for each output with the columns id, time, outputName, and group (corresponding to the group name if there are several groups).
- IndividualParameters corresponds to a list of data frames, one for each simulation group, with the individual parameters sampled in the group.
- PopulationParameters corresponds to a list of data frames, one for each simulation group, with the population parameters sampled in the group (can be several sets only in case of replicates).
- doses corresponds to a list of data frames, one for each simulation group, with the dose amounts administered to each individual in the group.

Usage

```
getSimulationResults(id = NULL, rep = NULL)
```

Arguments

`id`	[optional](*string*) If provided, results are retrieved only for this id.
`rep`	[optional](*int*) If provided, results are retrieved only for this replicate.

See Also

`runSimulation`

Examples

```
 # retrieve complete results
 initializeLixoftConnectors("simulx")
project_file <- file.path(getDemoPath(), "1.overview", "importFromMonolix_resimulateProject.smlx")
 loadProject(project_file)
 runSimulation()
 ## Not run:
   getSimulationResults()
   getSimulationResults()$IndividualParameters
   getSimulationResults()$doses

## End(Not run)

 # retrieve results of a specific individual and replicate
 initializeLixoftConnectors("simulx")
project_file <- file.path(getDemoPath(), "1.overview", "importFromMonolix_clinicalTrial.smlx")
 loadProject(project_file)
 runSimulation()
 getSimulationResults(id = "1", rep = 10)
```

getStandardErrorEstimationSettings *[Monolix] Get standard error estimation settings*

Description

Get the standard error estimation settings. Associated settings are:

"minIterations"	(*int >=1*)	Minimum number of iterations.
"maxIterations"	(*int >=1*)	Maximum number of iterations.

Usage

```
getStandardErrorEstimationSettings(...)
```

Arguments

...	[optional] (string) Name of the settings whose value should be displayed. If no argument is provided, all the settings are returned.

Value

An array which associates each setting name to its current value.

See Also

setStandardErrorEstimationSettings

Examples

```
## Not run:
getStandardErrorEstimationSettings()
# retrieve a list of all the standard error estimation settings

getStandardErrorEstimationSettings("minIterations","maxIterations")
# retrieve only minIterations and maxIterations settings values

## End(Not run)
```

getStructuralModel *[Monolix - PKanalix - Simulx] Get structural model file*

Description

Get the model file for the structural model used in the current project.

Usage

```
getStructuralModel()
```

Details

For Simulx, this function will return the path to the structural model only if the project was imported from Monolix, and the path to the full custom model otherwise. Note that a custom model in Simulx may include also a statistical part. For Simulx, there is no associated function getStructuralModel() because setting a new model is equivalent to creating a new project. Use newProject instead.

If a model was loaded from the libraries, the returned character is not a path, but the name of the library model, such as "lib:model_name.txt". To see the content of a library model, use getLibraryModelContent.

Value

A string corresponding to the path to the structural model file.

See Also

For Monolix and PKanalix only: setStructuralModel

Examples

```
## Not run:
getStructuralModel() => "/path/to/model/inclusion/modelFile.txt"

## End(Not run)
# Get the name and see the content of the model used in warfarin demo project
initializeLixoftConnectors("monolix", force = TRUE)
loadProject(file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "warfarinPK_p
libModelName <- getStructuralModel()
getLibraryModelContent(libModelName)

# Get the name of the model file used in Simulx
initializeLixoftConnectors("simulx", force = TRUE)
project_name <- file.path(getDemoPath(), "1.overview", "newProject_TMDDmodel.smlx")
loadProject(project_name)
getStructuralModel()

# Get the name of the model file imported to Simulx
initializeLixoftConnectors("monolix", force = TRUE)
project_name <- file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "warfarin
loadProject(project_name)
getStructuralModel()
initializeLixoftConnectors("simulx", force = TRUE)
importProject(project_name)
getStructuralModel()
```

---

getTests *[Monolix] Get statistical tests results*

Description

Get the results of performed statistical tests.
Existing tests: Wald, Individual parameters normality, individual parameters marginal distribution, random effects normality,

random effects correlation, individual parameters vs covariates correlation, random effects vs covariates correlation,
residual normality and residual symmetry.
WARNING: Only the tests performed during the last scenario run can provide results.

Usage

```
getTests()
```

Value

A list associating the name of the test to the corresponding results values computed during the last scenario run.

Examples

```
## Not run:
getTests()
 -> list(  wald = [...] ,
           individualParametersNormality = [...] ,
           ... )

## End(Not run)
```

---

getTreatmentElements *[Simulx] Get treatment elements*

Description

Get the list of all available treatments elements for the simulation. To use one or several of these elements in simulation, please add it to a simulation group with setGroupElement.

Usage

```
getTreatmentElements()
```

Details

Treatment elements can be defined with defineTreatmentElement, or created by importing a Monolix or a PKanalix project with importProject. Elements defined are created in the background and saved with the Simulx project if calling saveProject. They can be deleted with deleteElement.

Each element is a list of

"inputType"	(*string*)	Type of input definition: can be "manual" or "external".
"file"	(*string*)	Path to the file if the inputType is external. NULL else wise.
"data"	(*data.frame*)	Values of the element as defined in defineTreatmentElement.
"admID"	(*integer*)	Administration id as defined in defineTreatmentElement.
"scale"	(*integer*)	information on scaling by covariates if defined in defineTreatmentElement.

If the project was created from a model file, no treatment element is added.

If the project was created by importing a Monolix or a PKanalix project, for each administration type present in the original project (dataset column tagged as administration id), an individual element mlx_adm is created as an external file with the dosing times and amounts from the original dataset in the Monolix or PKanalix project.

See Also

defineTreatmentElement

Examples

```
##### Working example #####

initializeLixoftConnectors("simulx", force = TRUE)
loadProject(paste0(getDemoPath(), "/3.definition/3.1.treatments/treatment_weight_and_genotype_based.smlx"))
getTreatmentElements()
## Not run:
$`14nmolPerKg`
$`14nmolPerKg`$inputType
[1] "manual"

$`14nmolPerKg`$file
NULL

$`14nmolPerKg`$admID
[1] 1

$`14nmolPerKg`$scale
$`14nmolPerKg`$scale$covariate
[1] "Weight"

$`14nmolPerKg`$scale$intercept
[1] 0

$`14nmolPerKg`$scale$scaleDuration
[1] FALSE


$`14nmolPerKg`$data
time amount  tInf washout
1    0     14 0.208   FALSE
2   21     14 0.208   FALSE
3   42     14 0.208   FALSE
4   63     14 0.208   FALSE
5   84     14 0.208   FALSE

## End(Not run)
```

`getTreatmentsInformation`

[Monolix - PKanalix] Get treatments information

Description

Get information about doses present in the loaded dataset.

Usage

```
getTreatmentsInformation()
```

Value

A dataframe whose columns are:

- id and occasion level names (*string*)
- time (*double*)
- amount (*double*)
- [optional] administrationType (*int*)
- [optional] infusionTime (*double*)
- [optional] isArtificial (*bool*): is created from SS or ADDL column
- [optional] isReset (*bool*): IOV case only

Examples

```
{
## Not run:
initializeLixoftConnectors("monolix")
project_name <- file.path(getDemoPath(), "6.PK_models", "6.3.multiple_doses", "ss1_project.mlxtran")
loadProject(project_name)
getTreatmentsInformation()

## End(Not run)
}
```

---

getVariabilityLevels *[Monolix] Get variability levels*

Description

Get a summary of the variability levels (inter-individual and/or intra-individual variability) present in the current project.

Usage

```
getVariabilityLevels()
```

Value

A collection of the variability levels present in the currently loaded project.

Examples

```
## Not run:
getVariabilityLevels()

## End(Not run)
```

importMonolixProject *[Simulx] Import project from Monolix*

Description

[DEPRECATED] This function will be removed in a future version. Use `importProject` instead.

Usage

```
importMonolixProject(projectFile)
```

Arguments

`projectFile`	(*string*) Path to the project file. Can be absolute or relative to the current working directory.

Details

Import all the elements coming from a Monolix project. It imports - the model file, - the population parameters (if estimated, else wise an element is created with all values at 1), - the individual parameters (if estimated, else wise an element is created with all values at 1), - the outputs as an external element, - the treatments (if any) as an external element, - the regressors (if any) as an external element, - the occasion structure (if any) as an external element.

WARNING: R is sensitive between '\' and '/', only '/' can be used.

Examples

```
## Not run:
importfrommonolix("/path/to/project/file.mlxtran") for Linux platform
importfrommonolix("C:/Users/path/to/project/file.mlxtran") for Windows platform

## End(Not run)
```

importProject *[Monolix - PKanalix - Simulx] Import project from Datxplore, Monolix or PKanalix*

Description

Import a Monolix or a PKanalix project into the currently running application initialized in the connectors.
The extensions are .mlxtran for Monolix, .pkx for PKanalix, .smlx for Simulx and .dxp for Datxplore.
WARNING: R is sensitive between '\' and '/', only '/' can be used. Allowed import sources are:

CURRENT SOFTWARE	ALLOWED IMPORTS
Monolix	PKanalix
PKanalix	Monolix, Datxplore
Simulx	Monolix, PKanalix.

Usage

```
importProject(projectFile)
```

Arguments

projectFile	(*character*) Path to the project file. Can be absolute or relative to the current working directory.

Details

At import, a new project is created in a temporary folder with a project setting filesNextToProject = TRUE, which means that file dependencies such as data and model files are copied and kept next to the new project (or in the result folder for Simulx). This new project can be saved to the desired location withsaveProject.

Simulx projects can only be exported, not imported. To export a Simulx project to another application, please load the Simulx project with the Simulx connectors and use exportProject.

Importing a Monolix or a PKanalix project into Simulx automatically creates elements that can be used for simulation, exactly as in the GUI.

To see which elements of some type have been created in the new project, you can use the get..Element functions: getOccasionElements, getPopulationElements, getPopulationElements, getIndividualElements, getCovariateElements, getTreatmentElements, getOutputElements, getRegressorElements.

See Also

saveProject, exportProject

Examples

```
## Not run:
initializeLixoftConnectors(software = "simulx", force = TRUE)
importProject("/path/to/project/file.mlxtran")
importProject("/path/to/project/file.pkx")

initializeLixoftConnectors(software = "monolix", force = TRUE)
importProject("/path/to/project/file.pkx")

initializeLixoftConnectors(software = "pkanalix", force = TRUE)
importProject("/path/to/project/file.mlxtran")

## End(Not run)

# working example to import a Monolix demo project into Simulx. The resulting .smlx file can be opened from Simul

initializeLixoftConnectors(software = "monolix", force = TRUE)
MonolixDemoPath = file.path(getDemoPath(),"1.creating_and_using_models","1.1.libraries_of_models","warfarin
initializeLixoftConnectors(software = "simulx", force = TRUE)
importProject(MonolixDemoPath)
saveProject("importFromMonolix.smlx")
```

initializeLixoftConnectors *Initialize lixoftConnectors API*

Description

Initialize lixoftConnectors API for a given software

Usage

```
initializeLixoftConnectors(software = "monolix", path = "", force = FALSE)
```

Arguments

software	(*character*) [optional] Name of the software to be loaded. By default, "monolix" software is used.
path	(*character*) [optional] Path to installation directory of the Lixoft suite. If lixoftConnectors library is not already loaded and no path is given, the directory written in the lixoft.ini file is used for initialization.
force	(*bool*) [optional] Should software switch security be overpassed or not. Equals FALSE by default.

Value

A boolean equaling TRUE if the initialization has been successful and FALSE if not.

Examples

```
## Not run:
initializeLixoftConnectors(software = "monolix", path = "/path/to/lixoftRuntime/")

## End(Not run)
```

isProjectLoaded *[Monolix - PKanalix - Simulx] Get current project load status*

Description

[Monolix - PKanalix - Simulx] Get current project load status

Usage

```
isProjectLoaded()
```

Value

TRUE if a project is currently loaded, FALSE otherwise

Examples

```
initializeLixoftConnectors("monolix")
project_name <- file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "warfarin
loadProject(project_name)
isProjectLoaded()
initializeLixoftConnectors("pkanalix")
isProjectLoaded()
```

---

lixoftDisplay *Display Lixoft API Structures*

Description

[**Tools**][*Display*]
Display the structures retrieved by the LixoftConnectors library in a user-friendly way.

Usage

```
lixoftDisplay(structure)
```

Arguments

structure	[*miscellanous*] The data structure to be displayed.

Examples

```
## Not run:
settings = getProjectSettings()
lixoftDisplay(settings)

## End(Not run)
```

---

loadProject *[Monolix - PKanalix - Simulx] Load project from file*

Description

Load a project in the currently running application initialized in the connectors.
The extensions are .mlxtran for Monolix, .pkx for PKanalix, and .smlx for Simulx.
WARNING: R is sensitive between '\' and '/', only '/' can be used.

Usage

```
loadProject(projectFile)
```

Arguments

projectFile	(*character*) Path to the project file. Can be absolute or relative to the current working directory.

See Also

saveProject, importProject, exportProject, newProject

Examples

```
## Not run:
loadProject("/path/to/project/file.mlxtran") for Linux platform
loadProject("C:/Users/path/to/project/file.mlxtran") for Windows platform

## End(Not run)
# Load a Monolix project
initializeLixoftConnectors("monolix")
project_name <- file.path(getDemoPath(), "8.case_studies", "hiv_project.mlxtran")
loadProject(project_name)

# Load a PKanalix project
initializeLixoftConnectors("pkanalix")
project_name <- file.path(getDemoPath(), "1.basic_examples", "project_censoring.pkx")
loadProject(project_name)

# Load a Simulx project
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "2.models", "longitudinal.smlx")
loadProject(project_name)
```

---

newProject *[Monolix - PKanalix - Simulx] Create new project*

Description

New projects can be created in the connectors as in PKanalix, Monolix or Simulx GUI. The creation of a new project requires a dataset in PKanalix, a dataset and a model in Monolix, and a model in Simulx.

Usage

```
newProject(modelFile = NULL, data = NULL)
```

Arguments

modelFile	(*character*) Path to the model file. Mandatory for Monolix and Simulx, optional for PKanalix (used only for the CA part). Can be absolute or relative to the current working directory. To use a model from the libraries, you can find the model name with getLibraryModelName and set modelFile = "lib:modelName.txt" with the name obtained. To simulate inter-individual variability in Simulx with a new project, the model file has to include the statistical model, contrary to Monolix and PKanalix for which the model file only contains the structural model. Check here in detail how to write such a model from scratch.
data	(*list*) Structure describing the data. Mandatory for Monolix and PKanalix. • dataFile (*string*): Path to the data file. Can be absolute or relative to the current working directory.

- headerTypes (*array<character>*): A collection of header types. The possible header types are: "ignore", "id", "time", "observation", "amount", "contcov", "catcov", "occ", "evid", "mdv", "obsid", "cens", "limit", "regressor","admid", "rate", "tinf", "ss", "ii", "addl", "date". Notice that these are not exactly the types displayed in the interface, they are shortcuts.
- observationTypes [optional] (*list*): A list giving the type of each observation present in the data file. If there is only one y-type, the corresponding observation name can be omitted. The possible observation types are "continuous", "discrete", and "event".
- nbSSDoses (*int*): Number of doses (if there is a SS column for steady-state).
- mapping [optional](*list*): A list of lists representing a link between observation types and model outputs. Each list contains:
  - data (*string*) Name of observation type
  - prediction (*string*) Prediction name
  - model [Monolix] (*string*) Model observation name (for continuous observations only)

Details

Note: instead of creating a project from scratch, it is also possible in Monolix and PKanalix to load an existing project with loadProject or importProject and change the dataset or the model with setData or setStructuralModel.

See Also

newProject saveProject

Examples

```
# Create a new Monolix project
initializeLixoftConnectors("monolix")
data_file <- file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "data", "warf
newProject(data = list(dataFile = data_file,
               headerTypes = c("id", "time", "amount", "observation", "obsid", "contcov", "catcov", "contcov"
                       observationTypes = list("1" = "continuous", "2" = "continuous"),
                       mapping = list(list(data = "1",
                                           prediction = "Cc",
                                           model = "y1"),
                                      list(data = "2",
                                           prediction = "R",
                                           model = "y2"))),
       modelFile = "lib:oral1_1cpt_IndirectModelInhibitionKin_TlagkaVClR0koutImaxIC50.txt")

# Create a new PKanalix project
initializeLixoftConnectors("pkanalix")
data_file <- file.path(getDemoPath(), "1.basic_examples", "data", "data_BLQ.csv")
newProject(data = list(dataFile = data_file,
                headerTypes = c("id", "time", "amount", "observation", "cens", "catcov")))

# Create a new Simulx project
initializeLixoftConnectors("simulx")
newProject(modelFile = "lib:oral1_1cpt_IndirectModelInhibitionKin_TlagkaVClR0koutImaxIC50.txt")
```

`plotBEConfidenceIntervals` *[PKanalix] Individual NCA parameter vs covariate plot*

Description

[PKanalix] Individual NCA parameter vs covariate plot

Usage

```
plotBEConfidenceIntervals(
  parameters = NULL,
  formulations = NULL,
  settings = list(),
  preferences = NULL,
  data = NULL
)
```

Arguments

`parameters`	vector of bioequivalence parameters to display. (by default the first 4 computed parameters are displayed).
`formulations`	vector of test formulations to display. (by default the first 4 test formulations are displayed).
`settings`	List with the following settings • `median` (*bool*) - If TRUE median is displayed (default TRUE). • `limits` (*bool*) - If TRUE confidence intervals limits are displayed (default TRUE). • `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default TRUE). • `ylab` (*string*) label on y axis (default Ratio). • `ncol` (*int*) number of columns when facet = TRUE (default 4). • `fontsize` (*integer*) Plot text font size. • `units` (*boolean*) Set units in axis labels (default TRUE).
`preferences`	(*optional*) preferences for plot display, run *getPlotPreferences("plotBEConfidenceIntervals")* to check available displays.
`data`	Charts data as dataframe - Output of getChartsData (*getChartsData("plotBESubjectByFormulation", ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one parameter,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)
runNCAEstimation()
runBioequivalenceEstimation()
plotBEConfidenceIntervals()

plotBEConfidenceIntervals(parameters = "Cmax",
                          settings = list(legend = T))

# pre compute dataset
data <- getChartsData(plotName = "plotBEConfidenceIntervals")
plotBEConfidenceIntervals(data = data)

parameters <- c("AUClast", "Cmax")
plotBEConfidenceIntervals(parameters = parameters)
```

---

plotBESequenceByPeriod

[PKanalix] Plot Bioequivalence Sequenced parameters

Description

[PKanalix] Plot Bioequivalence Sequenced parameters

Usage

```
plotBESequenceByPeriod(
  parameters = NULL,
  settings = list(),
  preferences = NULL,
  stratify = list(),
  data = NULL
)
```

Arguments

`parameters`	vector of bioequivalence parameters to display. (by default the first 4 computed parameters are displayed).
`settings`	List with the following settings • `dots` (*bool*) - If TRUE sequenced parameters are displayed as dots (default TRUE). • `lines` (*bool*) - If TRUE sequenced parameters are displayed as lines (default TRUE). • `error` (*bool*) - If TRUE error are displayed as bars (default TRUE). • `formulationColors` (*vector<string>*) List of colors to use for each formulation when there are more than one test formulation. • `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).

- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `fontsize` (*integer*) Plot text font size.
- `units` (*boolean*) Set units in axis labels (default TRUE).
- `xlab` (*string*) Label on x-axis (no label by default) .

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotBESequenceByPeriod")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values.
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter.
  - `interval` - in case of a continuous covariate, a list of filtering intervals.

`data` Charts data as dataframe - Output of getChartsData (*getChartsData("plotBESequenceByPeriod", ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one parameter,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "3.bioequivalence/project_crossover_bioequivalence.pkx")
loadProject(project)

runNCAEstimation()
runBioequivalenceEstimation()

plotBESequenceByPeriod(parameters = "Cmax",
                       settings = list(legend = T))

# stratification
plotBESequenceByPeriod(
  parameters = "AUClast",
  stratify = list(filter = list(name = "AGE", interval = c(25, 30)))
```

```
)
plotBESequenceByPeriod(
  parameters = "AUClast",
  stratify = list(splitGroup = list(name = "AGE", breaks = c(25)))
)
plotBESequenceByPeriod(
  parameters = "AUClast", settings=list(legend=T),
  stratify = list(splitGroup=list(list(name = "AGE", breaks = 25),
                                  list(name = "WT", breaks = 75)))
)

# update settings and preferences
plotBESequenceByPeriod(
  parameters = "Cmax",
  settings = list(legend = T, error = F)
)
preferences <- list(sequence = list(lineType = "dashed"))
plotBESequenceByPeriod(parameter =  "Cmax",
                       preferences = preferences)

# pre compute dataset
data <- getChartsData(plotName = "plotBESequenceByPeriod")
plotBESequenceByPeriod(data = data)

parameters <- c("AUClast", "Cmax")
plotBESequenceByPeriod()
plotBESequenceByPeriod(parameters = parameters)
```

`plotBESubjectByFormulation` *[PKanalix] Plot Bioequivalence Formulation parameters*

Description

[PKanalix] Plot Bioequivalence Formulation parameters

Usage

```
plotBESubjectByFormulation(
  parameters = NULL,
  formulations = NULL,
  settings = list(),
  preferences = NULL,
  stratify = list(),
  data = NULL
)
```

Arguments

`parameters`	vector of bioequivalence parameters to display. (by default the first 4 computed parameters are displayed).

`formulations` list of test formulations to display. (by default the first 4 test formulations are displayed).

`settings` List with the following settings

- `dots` (*bool*) - If TRUE sequenced parameters are displayed as dots (default TRUE).
- `lines` (*bool*) - If TRUE sequenced parameters are displayed as lines (default TRUE).
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `fontsize` (*integer*) Plot text font size.
- `units` (*boolean*) Set units in axis labels (default TRUE).

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotBESubjectByFormulation")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` Charts data as dataframe - Output of getChartsData (*getChartsData("plotBESubjectByFormulation", ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one parameter,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

runNCAEstimation()
runBioequivalenceEstimation()

plotBESubjectByFormulation(parameters = "Cmax",
                           settings = list(legend = T))

# stratification
plotBESubjectByFormulation(
  parameters = "AUClast",
  stratify = list(filter = list(name = "AGE", interval = c(25, 30)))
)
plotBESubjectByFormulation(
  parameters = "AUClast",
  stratify = list(splitGroup = list(name = "AGE", breaks = c(25)))
)
plotBESubjectByFormulation(
  parameters = "AUClast",
  stratify = list(colorGroup = list(name = "AGE", breaks = c(25))),
  settings = list(legend = T)
)
plotBESubjectByFormulation(
  parameters = "AUClast",
  stratify = list(colorGroup = list(name = "ID")),
  settings = list(legend = T)
)
plotBESubjectByFormulation(
  parameters = "AUClast", settings=list(legend=T),
  stratify = list(splitGroup=list(list(name = "AGE", breaks = 25),
                                  list(name = "WT", breaks = 75)))
)

# update settings and preferences
plotBESubjectByFormulation(
  parameters = "Cmax",
  settings = list(legend = T, lines = F)
)
preferences <- list(formulationLine = list(lineType = "dashed"))
plotBESubjectByFormulation(parameter =  "Cmax",
                           preferences = preferences)

# pre compute dataset
data <- getChartsData(plotName = "plotBESubjectByFormulation")
plotBESubjectByFormulation(data = data)

parameters <- c("AUClast", "Cmax")
plotBESubjectByFormulation()
plotBESubjectByFormulation(parameters = parameters)
```

`plotBivariateDataViewer` *[Monolix - PKanalix] Generate Bivariate observations plots*

Description

[Monolix - PKanalix] Generate Bivariate observations plots

Usage

```
plotBivariateDataViewer(
  obs1 = NULL,
  obs2 = NULL,
  data = NULL,
  settings = list(),
  stratify = list(),
  preferences = list()
)
```

Arguments

`obs1`	(*string*) Name of the observation to display in x axis (in dataset header). By default the first observation is considered.
`obs2`	(*string*) Name of the observation to display in y axis (in dataset header). By default the second observation is considered.
`data`	List of charts data as dataframe - Output of getChartsData (*getChartsData("plotBivariateDataViewer" ...)*) If data not specified, charts data will be computed inside the function.
`settings`	List with the following settings • `dots` (*bool*) - If TRUE individual observations are displayed as dots (default TRUE). • `lines` (*bool*) - If TRUE individual observations are displayed as lines (default TRUE). • `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • `xlog` (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE). • `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE). • `xlab` (*string*) label on x axis (Name of obs1 by default). • `ylab` (*string*) label on y axis (Name of obs2 by default). • `ncol` (*int*) number of columns when facet = TRUE (default 4). • `xlim` (*c(double, double)*) limits of the x axis. • `ylim` (*c(double, double)*) limits of the y axis. • `fontsize` (*integer*) Plot text font size. • `units` (*boolean*) Set units in axis labels (only available with PKanalix). • `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotBivariateDataViewer")* to check available displays.

Value

A ggplot object

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "warfarinPKPD_project.mlxtran")
loadProject(project)

plotBivariateDataViewer(obs1 = "y1", obs2 = "y2")
plotBivariateDataViewer(settings = list(lines = FALSE))

# stratification
plotBivariateDataViewer(obs1 = "y1", obs2 = "y2", stratify = list(ids = "10"))
plotBivariateDataViewer(stratify = list(splitGroup = list(name = "age", breaks = 25),
                                        filter = list(name = "sex", cat = 1)))
plotBivariateDataViewer(stratify = list(colorGroup = list(name = "wt", breaks = 75)))
plotBivariateDataViewer(stratify = list(splitGroup = list(list(name = "age", breaks = 25),
                                                          list(name = "sex"))))
# update plot settings or preferences
plotBivariateDataViewer(preferences = list(obs = list(color = "#32CD32")))
```

`plotBlqPredictiveCheck` *[Monolix] Plot BLQ predictive checks*

Description

[Monolix] Plot BLQ predictive checks

Usage

```
plotBlqPredictiveCheck(
  obsName = NULL,
  settings = list(),
  preferences = list(),
  data = NULL
)
```

Arguments

`obsName`	(*string*) Name of the observation (in dataset header). By default the first observation is considered.
`settings`	a list of optional plot settings: • `level` (*int*) level for prediction intervals computation (default 90). • `nbPoints` (*int*) Number of points for grid computation (default 200). • `censoredInterval` (*c(double, double)*) Censored interval c(min, max) for censored data. By default, the limit and the censored values are used. • `empirical` (*bool*) If TRUE Empirical data is displayed (default TRUE). • `theoretical` (*bool*) If TRUE theoretical data is displayed (default FALSE): • `predInterval` (*bool*) If TRUE Prediction intervalis displayed (default TRUE). • `outlierAreas` (*bool*) If TRUE Add red areas indicating empirical percentiles that are outside prediction intervals (default TRUE). • `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • `xlog` (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE). • `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE). • `xlab` (*string*) label on x axis (default "Time"). • `ylab` (*string*) label on y axis (default obsName). • `ncol` (*int*) number of columns when facet = TRUE (default 4). • `xlim` (*c(double, double)*) limits of the x axis. • `ylim` (*c(double, double)*) limits of the y axis. • `fontsize` (*integer*) Plot text font size. • `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").
`preferences`	(*optional*) preferences for plot display, run *getPlotPreferences("plotBlqPredictiveCheck")* to check available displays.
`data`	List of charts data as dataframe - Output of getChartsData (*getChartsData("plotBlqPredictiveCheck", ...)*) If data not specified, charts data will be computed inside the function.

Value

a ggplot2 object

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "monolix")

# continuous data
project <- file.path(getDemoPath(), "2.models_for_continuous_outcomes",
                     "2.2.censored_data", "censoring1_project.mlxtran")
loadProject(project)
runScenario()

plotBlqPredictiveCheck(obsName = "Y")
```

`plotCAIndividualFits` *[PKanalix] Generate CA Fit plots*

Description

[PKanalix] Generate CA Fit plots

Usage

```
plotCAIndividualFits(
  obsName = NULL,
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`obsName`	(*string*) Name of the observation (in dataset header). By default the first observation is considered.
`settings`	List with the following settings • `obsDots` (*bool*) - If TRUE individual observations are displayed as dots (default TRUE). • `obsLines` (*bool*) - If TRUE individual observations are displayed as lines (default FALSE). • `cens` (*bool*) - If TRUE censoring data are displayed as intervals (default TRUE). • `indivFits` (*bool*) - If TRUE individual fits are displayed (default TRUE). • `dosingTimes` (*bool*) - Add dosing times as vertical lines (default FALSE).

- `splitOccasions` (*bool*) - If TRUE occasions are displayed on separate plots (default TRUE).
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `xlog` (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE).
- `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default TRUE).
- `xlab` (*string*) label on x axis (default "Time").
- `ylab` (*string*) label on y axis (default "Concentration").
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `xlim` (*c(double, double)*) limits of the x axis.
- `ylim` (*c(double, double)*) limits of the y axis.
- `fontsize` (*integer*) Plot text font size.
- `units` (*boolean*) Set units in axis labels.
- `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotCAIndividualFits")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed),
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotCAIndividualFits", ...)*) If data not specified, charts data will be computed inside the function.

Value

A ggplot object

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

runCAEstimation()

plotCAIndividualFits()

# display
plotCAIndividualFits(stratify = list(ids = c(1, 2, 3)),
                     settings = list(obsDots = T, indivFits = T))
plotCAIndividualFits(stratify = list(ids = c(1, 2, 3)),
                   settings = list(obsDots = F, obsLines = T, cens = T, indivFits = T))

# stratification
plotCAIndividualFits(stratify = list(ids = c(1, 2, 3, 4),
                                     filter = list(name = "Period", cat = 1)))
plotCAIndividualFits(stratify = list(ids = c(1, 2, 3, 4, 5),
                                     colorGroup = list(name = "SEQ"),
                                     colors = c("#5DC088", "#DBA92B")))
plotCAIndividualFits(settings=list(legend=T),
                     stratify = list(ids = c(1, 2, 3, 4, 5),
                                     colorGroup=list(list(name = "AGE", breaks = 25),
                                                     list(name = "Period"))))

# update settings and preferences
plotCAIndividualFits(stratify = list(ids = c(1, 4)),
                     settings = list(ylog = F, scales = "fixed"))
plotCAIndividualFits(settings = list(ncol = 5))
preferences <- list(censObsIntervals = list(opacity = 1, lineWidth = 0.5))
plotCAIndividualFits(stratify = list(ids = c(4, 5, 6)), preferences = preferences)

# pre compute dataset
data <- getChartsData(plot = "plotCAIndividualFits", ids = c(1, 2))
plotCAIndividualFits(data = data)
```

---

plotCAObservationsVsPredictions

[PKanalix] Plot Observation VS Prediction

Description

[PKanalix] Plot Observation VS Prediction

Usage

```
plotCAObservationsVsPredictions(
  obsName = NULL,
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`obsName` (*string*) Name of the observation (in dataset header). By default the first observation is considered.

`settings` List with the following settings

- `useCensored` (*bool*) Choose to use BLQ data (TRUE) or to ignore it (FALSE) to compute the statistics (default TRUE).
- `obs` (*bool*) - If TRUE observations are displayed as dots (default TRUE).
- `cens` (*bool*) - If TRUE censoring data are displayed as red dots (default TRUE).
- `spline` (*bool*) - If TRUE add spline (default FALSE).
- `identityLine` (*bool*) - If TRUE add identity line (default TRUE).
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `xlog` (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE).
- `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `xlim` (*c(double, double)*) limits of the x axis.
- `ylim` (*c(double, double)*) limits of the y axis.
- `fontsize` (*integer*) Plot text font size.
- `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").
- `ylab` (*string*) label on y axis (default "Observations").

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotObservationsVsPredictions")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,

  - interval - in case of a continuous covariate, a list of filtering intervals.
- colors - List of colors to use when colorGroup argument is defined

data List of chazrts data as dataframe - Output of getChartsData (*getChartsData("plotObservationsVsPred ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one prediction type,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "1.basic_examples",
                     "project_censoring.pkx")
loadProject(project)
runCAEstimation()

plotCAObservationsVsPredictions()
plotCAObservationsVsPredictions(settings = list(spline = TRUE))
plotCAObservationsVsPredictions(settings = list(ylog = TRUE, xlog = TRUE))

# stratification
plotCAObservationsVsPredictions(stratify = list(filter = list(name = "STUDY", cat = "102")))
plotCAObservationsVsPredictions(stratify = list(splitGroup = list(name = "STUDY")))
plotCAObservationsVsPredictions(stratify = list(colorGroup = list(name = "STUDY")))

data <- getChartsData(plotName = "plotCAObservationsVsPredictions",
                      colorGroup = list(name = "STUDY"))
plotCAObservationsVsPredictions(data = data)
```

---

plotCAParametersCorrelation

[PKanalix] Correlation between 2 individual CA parameters

Description

[PKanalix] Correlation between 2 individual CA parameters

Usage

```
plotCAParametersCorrelation(
  parametersRows = NULL,
  parametersColumns = NULL,
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`parametersRows` vector with the name of CA parameters to display on rows (by default the first 4 computed parameters are displayed).

`parametersColumns`
vector with the name of CA parameters to display on columns (by default parametersColumns = parametersRows).

`settings` List with the following settings

- `regressionLine` (*bool*) If TRUE, Add regression line in scatterplots (default TRUE).
- `spline` (*bool*) If TRUE, Add xpline in scatterplots (default FALSE).
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `fontsize` (*integer*) Plot text font size.

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotCAParametersCorrelation")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotCAParametersCorrelo ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one element in parametersRows and parametersColumns,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

runCAEstimation()

plotCAParametersCorrelation()
plotCAParametersCorrelation(parametersRows = c("ka", "Cl"))

plotCAParametersCorrelation(parametersRows = "ka", parametersColumns = "Cl")
plotCAParametersCorrelation(parametersRows = "Cl", parametersColumns = "ka")
plotCAParametersCorrelation(parametersRows = "ka", parametersColumns = "Cl",
                            settings = list(spline = TRUE))

# stratification
plotCAParametersCorrelation(parametersRows = "ka", parametersColumns = "Cl",
                      stratify = list(filter = list(name = "AGE", interval = c(25, 30))))
plotCAParametersCorrelation(parametersRows = "ka", parametersColumns = "Cl",
                        stratify = list(splitGroup = list(name = "AGE", breaks = c(25))))
plotCAParametersCorrelation(parametersRows = "ka", parametersColumns = "Cl",
                           stratify = list(colorGroup = list(name = "HT", breaks = 181)))
plotCAParametersCorrelation(
  parametersRows = "ka", parametersColumns = "Cl", settings=list(legend=T),
  stratify = list(splitGroup=list(list(name = "AGE", breaks = 25),
                                  list(name = "Period"))))
)

# update preferences and settings
preferences <- list(obs = list(color = "#51B613"))
plotCAParametersCorrelation(parametersRows = "ka", parametersColumns = "Cl",
                            preferences = preferences)

# pre compute dataset
data <- getChartsData(plotName = "plotCAParametersCorrelation")
plotCAParametersCorrelation(data = data)

plotCAParametersCorrelation(parametersRows = c("Tlag", "ka", "V"))
```

---

plotCAParametersDistribution

[PKanalix] Distribution of the individual CA parameters

Description

[PKanalix] Distribution of the individual CA parameters

Usage

```
plotCAParametersDistribution(
  parameters = NULL,
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`parameters`	vector of ca parameters to display. (by default the first 4 computed ca parameters are displayed).
`settings`	List with the following settings • `plot` Type of plot: probability density distribution ("pdf"), cumulative density distribution ("cdf") (default "pdf"). • `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • `ncol` (*int*) number of columns when facet = TRUE (default 4). • `fontsize` (*integer*) Plot text font size. • `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").
`preferences`	(*optional*) preferences for plot display, run *getPlotPreferences("plotCAParametersDistribution")* to check available displays.
`stratify`	List with the stratification arguments • `ids` - List of ids to display (by default all ids are displayed). • `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields: – `name` : The name of the covariate to use in grouping, – `breaks` : In case of a continuous covariate, a list of break values, – `groups` : [optional] In case of a categorical covariate, define groups of modalities. • `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields: – `name` : The name of the covariate to use in grouping, or the name of the column id, – `breaks` : In case of a continuous covariate, a list of break values, – `groups` : [optional] In case of a categorical covariate, define groups of modalities. • `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields: – `name` - the name of the covariate to filter, – `cat` - in case of a categorical covariate, the name of the category to filter, – `interval` - in case of a continuous covariate, a list of filtering intervals. • `colors` - List of colors to use when colorGroup argument is defined
`data`	List of charts data as dataframe - Output of getChartsData (*getChartsData("plotCAParametersDistrib ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one parameter,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

runCAEstimation()

plotCAParametersDistribution(parameters = "Tlag", settings = list(plot = "pdf"))
plotCAParametersDistribution(parameters = "Cl", settings = list(plot = "cdf"))

# stratification
plotCAParametersDistribution(parameters = "Tlag",
                     stratify = list(filter = list(name = "AGE", interval = c(25, 30))))
plotCAParametersDistribution(parameters = "Cl",
                       stratify = list(splitGroup = list(name = "AGE", breaks = c(25))))
plotCAParametersDistribution(parameters = "Cl", settings = list(plot = "pdf"),
                         stratify = list(colorGroup = list(name = "HT", breaks = 181)))
plotCAParametersDistribution(parameters = "Cl", settings = list(plot = "cdf"),
                          stratify = list(colorGroup = list(name = "HT", breaks = 181),
                             colors = c("#46B4AF", "#B4468A")))
plotCAParametersDistribution(
  parameters = "Cl", settings=list(legend=T),
  stratify = list(splitGroup=list(list(name = "AGE", breaks = 25),
                                  list(name = "Period")))
)

# pre compute dataset
data <- getChartsData(plotName = "plotCAParametersDistribution")
plotCAParametersDistribution(data = data)

parameters <- c("Tlag", "ka", "V")
plotCAParametersDistribution(data = data, parameters = parameters)
plotCAParametersDistribution(parameters = parameters)
plotCAParametersDistribution(parameters = parameters, settings = list(plot = "cdf"))
plotCAParametersDistribution(parameters = parameters, settings = list(plot = "pdf"))
```

---

plotCAParametersVsCovariates

[PKanalix] Individual CA parameter vs covariate plot

Description

[PKanalix] Individual CA parameter vs covariate plot

Usage

```
plotCAParametersVsCovariates(
  parameters = NULL,
  covariates = NULL,
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

parameters	vector of ca parameters to display. (by default the first 4 computed ca parameters are displayed).
covariates	vector of covariates to display. (by default the first 4 covariates are displayed).
settings	List with the following settings • regressionLine (*bool*) If TRUE, Add regression line in scatterplots (default TRUE). • spline (*bool*) If TRUE, Add xpline in scatterplots (default FALSE). • boxplotData (*string*) for categorical covariate, if boxplotData is not NULL, data are added as dots over the boxplot. They can be either "spread" on the box or "aligned" (default NULL) • legend (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • grid (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • ncol (*int*) number of columns when facet = TRUE (default 4). • fontsize (*integer*) Plot text font size.
preferences	(*optional*) preferences for plot display, run *getPlotPreferences("plotCAParametersVsCovariates")* to check available displays.
stratify	List with the stratification arguments • ids - List of ids to display (by default all ids are displayed), • splitGroup - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields: – name : The name of the covariate to use in grouping, – breaks : In case of a continuous covariate, a list of break values, – groups : [optional] In case of a categorical covariate, define groups of modalities. • colorGroup - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields: – name : The name of the covariate to use in grouping, or the name of the column id, – breaks : In case of a continuous covariate, a list of break values, – groups : [optional] In case of a categorical covariate, define groups of modalities. • filter - Filter data (by default no filtering is applied). A list, or a list of list with fields: – name - the name of the covariate to filter, – cat - in case of a categorical covariate, the name of the category to filter,

  - interval - in case of a continuous covariate, a list of filtering intervals.
- colors - List of colors to use when colorGroup argument is defined

data Charts data as dataframe - Output of getChartsData (*getChartsData("plotCAParametersVsCovariates ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one element in covariatesRows and covariatesColumns,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software="pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

runCAEstimation()

plotCAParametersVsCovariates(covariates="AGE", parameters="ka", settings=list(spline=T))
plotCAParametersVsCovariates(covariates="FORM", parameters="Tlag")

# stratification
plotCAParametersVsCovariates(covariates= "HT", parameters="ka",
                             stratify=list(filter=list(name="AGE", interval=c(25, 30))))
plotCAParametersVsCovariates(covariates="WT", parameters="ka",
                              stratify=list(splitGroup=list(name="AGE", breaks=c(25))))
plotCAParametersVsCovariates(covariates="AGE", parameters="ka",
                              stratify=list(colorGroup=list(name="HT", breaks=181)))
plotCAParametersVsCovariates(covariates="SEQ", parameters="ka",
                              stratify=list(colorGroup=list(name="HT", breaks=181),
                                            colors = c("#175C8C", "#ABD3EF")))
plotCAParametersVsCovariates(
  covariates="SEQ", parameters="ka", settings=list(legend=T),
  stratify = list(splitGroup=list(list(name = "AGE", breaks = 25),
                                  list(name = "Period")))
)

# update settings and preferences
plotCAParametersVsCovariates(covariates="SEQ", parameters="Tlag", settings=list(legend=T))
preferences <- list(spline=list(lineType="dashed"))
plotCAParametersVsCovariates(covariates="AGE", parameters="ka",
                              settings=list(regressionLine=F, spline=T),
                              preferences=preferences)

# pre compute dataset
data <- getChartsData(plotName="plotCAParametersVsCovariates")
plotCAParametersVsCovariates(data=data)

parameters <- c("Tlag", "Cl", "V")
covariates <- c("AGE", "WT", "FORM")
plotCAParametersVsCovariates(parameters=parameters, covariates=covariates, data=data)
plotCAParametersVsCovariates(parameters=parameters, covariates=covariates)
```

`plotCovariates` *[Monolix - PKanalix] Generate Covariate plots*

Description

Generate scatterplots between two continuous covariates or bar plot between categorical covariates

Usage

```
plotCovariates(
  covariatesRows = NULL,
  covariatesColumns = NULL,
  data = NULL,
  settings = list(),
  preferences = list(),
  stratify = list()
)
```

Arguments

`covariatesRows` vector with the name of covariates to display on rows (by default the first 4 covariates are displayed).

`covariatesColumns`
vector with the name of covariates to display on columns (by default the first 4 covariates are displayed).

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotCovariates", ...)*) If data not specified, charts data will be computed inside the function.

`settings` List with the following settings

- `regressionLine` (*bool*) If TRUE, Add regression line in scatterplots (default TRUE).
- `spline` (*bool*) If TRUE, Add xpline in scatterplots (default FALSE).
- `histogramColors` (*vector<string>*) List of colors to use in histograms plots.
- `histogramPosition` (*string*) Type of histogram: "stacked", "grouped" or "default" (histograms with categorical covariates in xaxis the plot is grouped else it is stacked), (Default is "default")
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `bins` (*int*) number of bins for the histogram (default 10)
- `fontsize` (*integer*) Plot text font size.

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotCovariates")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).

- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

Value

- A ggplot object if one element in covariatesRows and covariatesColumns,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

# covariate distribution when only one covariate is specified
plotCovariates(covariatesRows = "HT", settings = list(bins = 10))

# scatter plot when both covariates are continuous
plotCovariates(covariatesRows = "HT", covariatesColumns = "AGE", settings = list(spline = TRUE))
plotCovariates(covariatesRows = "HT", covariatesColumns = c("AGE", "FORM"))

# box plot when one covariate is categorical and the othe one is continuous
preferences <- list(boxplot = list(fill = "#2075AE"), boxplotOutlier = list(shape = 3))
plotCovariates(covariatesRows = "FORM", covariatesColumns = "AGE", preferences = preferences)

# histogram when covariate on column is categorical
plotCovariates(covariatesRows = "FORM", covariatesColumns = "SEQ",
               settings = list(histogramColors = c("#5DC088", "#DBA92B")))
plotCovariates(covariatesRows = "AGE", covariatesColumns = "SEQ",
               settings = list(histogramColors = c("#5DC088", "#DBA92B")))
```

```
# stratification
plotCovariates(covariatesRows = "HT", covariatesColumns = "WT", stratify = list(
               splitGroup = list(name = "AGE", breaks = 25),
               filter = list(name = "Period", cat = 1)))
preferences <- list(regressionLine = list(color = "#E5551B"))
plotCovariates(covariatesRows = "AGE", covariatesColumns = "WT", stratify = list(
               colorGroup = list(name = "HT", breaks = 181),
               colors = c("#2BB9DB", "#DD6BD2")), preferences = preferences)
plotCovariates(covariatesRows = "HT", covariatesColumns = "WT",
               stratify = list(splitGroup = list(list(name = "AGE", breaks = 25),
                                                 list(name = "SEQ"))))

# Mulitple covariates
plotCovariates()
plotCovariates(covariatesRows = c("AGE", "SEQ", "HT"), covariatesColumns = c("AGE", "SEQ", "HT"))
plotCovariates(stratify = list(filter = list(name = "AGE", interval = c(20, 30))))
plotCovariates(stratify = list(splitGroup = list(name = "AGE", breaks = c(25))))
plotCovariates(stratify = list(colorGroup = list(name = "AGE", breaks = c(25))))
```

`plotImportanceSampling` *[Monolix] Plot Importance sampling convergence*

Description

[Monolix] Plot Importance sampling convergence

Usage

```
plotImportanceSampling(settings = list(), data = NULL)
```

Arguments

`settings`	a list of optional settings: • `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • `fontsize` (*integer*) Plot text font size.
`data`	dataframe - Output of getChartsData (*getChartsData("plotImportanceSampling", ...)*) If data not specified, charts data will be computed inside the function.

Value

A ggplot object

See Also

getChartsData

Examples

```
initializeLixoftConnectors(software = "monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()
runLogLikelihoodEstimation()

plotImportanceSampling()
```

`plotIndividualFits` *[Monolix] Plot Monolix Individual Fits Only available for Continuous data.*

Description

[Monolix] Plot Monolix Individual Fits Only available for Continuous data.

Usage

```
plotIndividualFits(
  obsName = NULL,
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`obsName`	(*string*) Name of the observation (in dataset header). By default the first observation is considered.
`settings`	List with the following settings • `indivEstimate` (*string*) Calculation of individual estimates: conditional mean ("mean"), conditional mode with EBE's ("mode") (default "mode") . • `obsDots` (*bool*) - If TRUE individual observations are displayed as dots (default TRUE). • `obsLines` (*bool*) - If TRUE individual observations are displayed as lines (default FALSE). • `cens` (*bool*) - If TRUE censored intervals are displayed (default TRUE). • `indivFits` (*bool*) - If TRUE individual fits are displayed (default TRUE). • `popFits` (*bool*) - If TRUE population fits (typical individual) are displayed (default FALSE). • `popCov` (*bool*) - If TRUE population fits (individual covariates) are displayed (default FALSE). • `predMedian` (*bool*) - If TRUE median of individual fits computed based on multiple simulations (default FALSE).

- `predInterval` (*bool*) - If TRUE 90 % prediction interval of individual fits computed based on multiple simulations (default FALSE).
- `splitOccasions` (*bool*) - If TRUE occasions are displayed on separate plots (default TRUE).
- `dosingTimes` (*boolean*) - Add dosing times as vertical lines (default FALSE).
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `xlog` (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE).
- `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE).
- `xlab` (*string*) label on x axis (default "Time").
- `ylab` (*string*) label on y axis (default obsName).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `xlim` (*c(double, double)*) limits of the x axis.
- `ylim` (*c(double, double)*) limits of the y axis.
- `fontsize` (*integer*) Plot text font size.
- `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotIndividualFits")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotIndividualFits", ...)*) If data not specified, charts data will be computed inside the function.

Value

A ggplot object

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()
runConditionalDistributionSampling()
runConditionalModeEstimation()

plotIndividualFits()

plotIndividualFits(settings=list(popFits=T))
plotIndividualFits(settings=list(obsLines=T, obsDots=F, predInterval=T))
plotIndividualFits(settings=list(dosingTimes=T))

# stratification options
plotIndividualFits(stratify=list(ids=c(1, 2, 3, 4)))
plotIndividualFits(stratify=list(filter=list(name="WEIGHT", interval=c(75, 100))))
plotIndividualFits(stratify=list(filter=list(name="SEX", cat ="F")))
plotIndividualFits(stratify=list(colorGroup=list(name="SEX"), colors=c("#5DC088", "#DBA92B")))
plotIndividualFits(
  settings=list(legend=T),
  stratify = list(colorGroup=list(list(name = "SEX"),
                                  list(name = "WEIGHT", breaks = 70)))
)

# settings and preferences options
plotIndividualFits(settings=list(ylog=T, ylim=c(0.8, 11)))
preferences <- list(popFits=list(lineType="solid", legend="Population fits"))
plotIndividualFits(settings=list(popFits=T), preferences=preferences)

data <- getChartsData(plotName="plotIndividualFits",
                      computeSettings=list(indivEstimate="mean"),
                      ids=c(1, 2, 3, 4))
plotIndividualFits(data=data)
```

---

plotMCMC *[Monolix] Plot MCMC convergence*

Description

[Monolix] Plot MCMC convergence

Usage

```
plotMCMC(settings = list(), data = NULL)
```

Arguments

settings	a list of optional settings: • grid (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • ncol (*int*) number of columns when facet = TRUE (default 4).

- `fontsize` (*integer*) Plot text font size.

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotMCMC", ...)*) If data not specified, charts data will be computed inside the function.

Value

A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData

Examples

```
initializeLixoftConnectors(software = "monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()
runConditionalDistributionSampling()

plotMCMC()
```

---

plotNCAIndividualFits *[PKanalix] Generate NCA individual fits (elimination)*

Description

[PKanalix] Generate NCA individual fits (elimination)

Usage

```
plotNCAIndividualFits(
  data = NULL,
  settings = list(),
  preferences = list(),
  stratify = list()
)
```

Arguments

`data` List of charts data as dataframe - Output of getChartsData ((*getChartsData("plotNCAIndividualFits", ...)*) If data not specified, charts data will be computed inside the function.

`settings` List with the following settings

- `obsDots` (*bool*) - If TRUE individual observations are displayed as dots (default TRUE).
- `obsLines` (*bool*) - If TRUE individual observations are displayed as lines (default FALE).
- `cens` (*bool*) - If TRUE censoring data are displayed as dots (default TRUE).

- `obsUnused` (*bool*) - If TRUE and (and if dots is set to TRUE), individual observations not used for lambda z calculation are displayed as dots (default TRUE).
- `obsUnusedColor` - If obsUnused is TRUE, unused data can be colored with the color used for observation (colored), or unused data can be colored in grey (greyed) (default greyed).
- `lambda_z` (*bool*) - If TRUE individual fits are displayed (default TRUE).
- `dosingTimes` (*bool*) - Add dosing times as vertical lines (default FALSE).
- `splitOccasions` (*bool*) - If TRUE occasions are displayed on separate plots (default TRUE).
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `xlog` (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE).
- `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default TRUE).
- `xlab` (*string*) label on x axis (default "Time").
- `ylab` (*string*) label on y axis (default "Concentration").
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `xlim` (*c(double, double)*) limits of the x axis.
- `ylim` (*c(double, double)*) limits of the y axis.
- `fontsize` (*integer*) Plot text font size.
- `units` (*boolean*) Set units in axis labels (default TRUE).
- `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotNCAIndividualFits")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

Value

A ggplot object

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

runNCAEstimation()

plotNCAIndividualFits()

# display
plotNCAIndividualFits(stratify = list(ids = c(1, 2, 3)),
                      settings = list(obsDots = T, lambda_z = T))
plotNCAIndividualFits(stratify = list(ids = c(1, 2, 3)),
                      settings = list(obsDots = F, obsLines = T, lambda_z = T))

# stratification
plotNCAIndividualFits(stratify = list(ids = c(1, 2, 3, 4),
                                      filter = list(name = "Period", cat = 1)))
plotNCAIndividualFits(stratify = list(ids = c(1, 2, 3, 4, 5),
                                      colorGroup = list(name = "SEQ"),
                                      colors = c("#5DC088", "#DBA92B")))
plotNCAIndividualFits(stratify = list(colorGroup = list(list(name = "AGE", breaks = 25),
                                                        list(name = "Period"))))
# update settings and preferences
plotNCAIndividualFits(stratify = list(ids = c(1, 4)),
                      settings = list(ylog = TRUE, scales = "fixed"))
plotNCAIndividualFits(settings = list(ncol = 5))
plotNCAIndividualFits(settings = list(splitOccasions = FALSE, ncol = 5))
preferences <- list(lambda_z = list(color = "pink", lineWidth = 1))
plotNCAIndividualFits(stratify = list(ids = c(4, 5, 6)), preferences = preferences)

# pre compute dataset
data <- getChartsData(plot = "plotNCAIndividualFits", ids = c(1, 2))
plotNCAIndividualFits(data = data)
```

---

plotNCAParametersCorrelation

[PKanalix] Correlation between individual parameters

Description

[PKanalix] Correlation between individual parameters

Usage

```
plotNCAParametersCorrelation(
  parametersRows = NULL,
  parametersColumns = NULL,
```

```
  settings = list(),
  preferences = NULL,
  stratify = list(),
  data = NULL
)
```

Arguments

parametersRows	vector with the name of NCA parameters to display on rows (by default the first 4 computed parameters are displayed).
parametersColumns	vector with the name of NCA parameters to display on columns (by default parametersColumns = parametersRows).
settings	List with the following settings • regressionLine (*bool*) If TRUE, Add regression line in scatterplots (default TRUE). • spline (*bool*) If TRUE, Add xpline in scatterplots (default FALSE). • legend (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • grid (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • ncol (*int*) number of columns when facet = TRUE (default 4). • fontsize (*integer*) Plot text font size.
preferences	(*optional*) preferences for plot display, run *getPlotPreferences("plotNCAParametersCorrelation")* to check available displays.
stratify	List with the stratification arguments • ids - List of ids to display (by default all ids are displayed). • splitGroup - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields: – name : The name of the covariate to use in grouping, – breaks : In case of a continuous covariate, a list of break values, – groups : [optional] In case of a categorical covariate, define groups of modalities. • colorGroup - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields: – name : The name of the covariate to use in grouping, or the name of the column id, – breaks : In case of a continuous covariate, a list of break values, – groups : [optional] In case of a categorical covariate, define groups of modalities. • filter - Filter data (by default no filtering is applied). A list, or a list of list with fields: – name - the name of the covariate to filter, – cat - in case of a categorical covariate, the name of the category to filter, – interval - in case of a continuous covariate, a list of filtering intervals. • colors - List of colors to use when colorGroup argument is defined
data	List of charts data as dataframe - Output of getChartsData (*getChartsData("plotNCAParametersCorre ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one element in parametersRows and parametersColumns,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

runNCAEstimation()

plotNCAParametersCorrelation(settings = list(spline = TRUE))
plotNCAParametersCorrelation(parametersRows = c("AUCINF_obs", "Cl_F_obs"))
plotNCAParametersCorrelation(parametersRows = c("AUCINF_obs", "Cl_F_obs"),
                             parametersColumns = c("AUCINF_obs", "Tmax"))

plotNCAParametersCorrelation(parametersRows = "AUCINF_obs", parametersColumns = "Cl_F_obs",
                             settings = list(spline = TRUE))
plotNCAParametersCorrelation(parametersRows = c("AUCINF_obs", "Tmax"))

# stratification
plotNCAParametersCorrelation(parametersRows = "AUCINF_obs", parametersColumns = "Cl_F_obs",
                      stratify = list(filter = list(name = "AGE", interval = c(25, 30))))
plotNCAParametersCorrelation(parametersRows = "AUCINF_obs", parametersColumns = "Tmax",
                       stratify = list(splitGroup = list(name = "AGE", breaks = c(25))))
plotNCAParametersCorrelation(parametersRows = "AUCINF_obs", parametersColumns = "Tmax",
                         stratify = list(colorGroup = list(name = "HT", breaks = 181)))
plotNCAParametersCorrelation(
  parametersRows = "AUCINF_obs", parametersColumns = "Tmax", settings=list(legend=T),
  stratify = list(splitGroup = list(list(name = "AGE", breaks = 25),
                                    list(name = "HT", breaks = 180)))
)

# update preferences and settings
preferences <- list(obs = list(color = "#51B613"))
plotNCAParametersCorrelation(parametersRows = "AUCINF_obs", parametersColumns = "Tmax",
                             preferences = preferences)

# pre compute dataset
data <- getChartsData(plotName = "plotNCAParametersCorrelation")
plotNCAParametersCorrelation(data = data, settings = list(spline = TRUE))

parameters <- c("Lambda_z", "AUClast", "Clast", "Cmax")
plotNCAParametersCorrelation(parametersRows = parameters)
```

`plotNCAParametersDistribution` *[PKanalix] Distribution of the individual parameters*

Description

[PKanalix] Distribution of the individual parameters

Usage

```
plotNCAParametersDistribution(
  parameters = NULL,
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`parameters`	vector of nca parameters to display. (by default the first 4 computed nca parameters are displayed).
`settings`	List with the following settings • `plot` Type of plot: probability density distribution ("pdf"), cumulative density distribution ("cdf") (default "pdf"). • `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • `ncol` (*int*) number of columns when facet = TRUE (default 4). • `fontsize` (*integer*) Plot text font size. • `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").
`preferences`	(*optional*) preferences for plot display, run *getPlotPreferences("plotNCAParametersDistribution")* to check available displays.
`stratify`	List with the stratification arguments • `ids` - List of ids to display (by default all ids are displayed). • `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields: – `name` : The name of the covariate to use in grouping, – `breaks` : In case of a continuous covariate, a list of break values, – `groups` : [optional] In case of a categorical covariate, define groups of modalities. • `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields: – `name` : The name of the covariate to use in grouping, or the name of the column id, – `breaks` : In case of a continuous covariate, a list of break values, – `groups` : [optional] In case of a categorical covariate, define groups of modalities.

- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotNCAParametersDistri ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one parameter,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

runNCAEstimation()

plotNCAParametersDistribution(parameters = "AUCINF_obs", settings = list(plot = "pdf"))
plotNCAParametersDistribution(parameters = "Lambda_z", settings = list(plot = "cdf"))

# stratification
plotNCAParametersDistribution(parameters = "AUClast",
                      stratify = list(filter = list(name = "AGE", interval = c(25, 30))))
plotNCAParametersDistribution(parameters = "Cmax",
                       stratify = list(splitGroup = list(name = "AGE", breaks = c(25))))
plotNCAParametersDistribution(parameters = "Tmax", settings = list(plot = "pdf"),
                         stratify = list(colorGroup = list(name = "HT", breaks = 181)))
plotNCAParametersDistribution(parameters = "AUCINF_obs", settings = list(plot = "cdf"),
                          stratify = list(colorGroup = list(name = "HT", breaks = 181),
                              colors = c("#46B4AF", "#B4468A")))
plotNCAParametersDistribution(
  parameters = "Tmax", settings=list(legend=T),
  stratify = list(splitGroup = list(list(name = "AGE", breaks = 25),
                                    list(name = "Period")))
)

# pre compute dataset
data <- getChartsData("plotNCAParametersDistribution")
plotNCAParametersDistribution(data = data, parameters = "AUClast")

# display multiple parameters
plotNCAParametersDistribution()
plotNCAParametersDistribution(settings = list(plot = "cdf"))
parameters <- c("Lambda_z", "AUClast", "Clast", "Cmax")
plotNCAParametersDistribution(parameters = parameters)
```

```
plotNCAParametersDistribution(parameters = parameters, settings = list(plot = "cdf"))
plotNCAParametersDistribution(parameters = parameters, settings = list(plot = "pdf"))
```

`plotNCAParametersVsCovariates`
[PKanalix] Individual NCA parameter vs covariate plot

Description

[PKanalix] Individual NCA parameter vs covariate plot

Usage

```
plotNCAParametersVsCovariates(
  parameters = NULL,
  covariates = NULL,
  settings = list(),
  preferences = NULL,
  stratify = list(),
  data = NULL
)
```

Arguments

`parameters`	vector of nca parameters to display. (by default the first 4 computed nca parameters are displayed).
`covariates`	vector of covariates to display. (by default the first 4 covariates are displayed).
`settings`	List with the following settings • `regressionLine` (*bool*) If TRUE, Add regression line in scatterplots (default TRUE). • `spline` (*bool*) If TRUE, Add xpline in scatterplots (default FALSE). • `boxplotData` (*string*) for categorical covariate, if boxplotData is not NULL, data are added as dots over the boxplot. They can be either "spread" on the box or "aligned" (default NULL) • `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • `ncol` (*int*) number of columns when facet = TRUE (default 4). • `fontsize` (*integer*) Plot text font size.
`preferences`	(*optional*) preferences for plot display, run *getPlotPreferences("plotNCAParametersVsCovariates")* to check available displays.
`stratify`	List with the stratification arguments • `ids` - List of ids to display (by default all ids are displayed). • `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields: – `name` : The name of the covariate to use in grouping, – `breaks` : In case of a continuous covariate, a list of break values,

  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` Charts data as dataframe - Output of getChartsData (*getChartsData("plotNCAParametersVsCovariate ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one element in covariatesRows and covariatesColumns,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

runNCAEstimation()

plotNCAParametersVsCovariates(covariates="AGE", parameters="AUClast", settings=list(spline=T))
plotNCAParametersVsCovariates(covariates="FORM", parameters="Cl_F_obs")

# stratification
plotNCAParametersVsCovariates(
  covariates="HT", parameters="AUClast",
  stratify=list(filter=list(name="AGE", interval=c(25, 30)))
)
plotNCAParametersVsCovariates(
  covariates="WT", parameters="AUClast",
  stratify=list(splitGroup=list(name="AGE", breaks=c(25)))
)
plotNCAParametersVsCovariates(
  covariates="AGE", parameters="AUClast",
  stratify=list(colorGroup=list(name="HT", breaks=181))
)
plotNCAParametersVsCovariates(
```

```
  covariates="SEQ", parameters="AUClast",
  stratify=list(colorGroup=list(name="HT", breaks=181),
                colors=c("#175C8C", "#ABD3EF"))
)
plotNCAParametersVsCovariates(
  covariates="SEQ", parameters="AUClast", settings=list(legend=T),
  stratify = list(colorGroup = list(list(name = "AGE", breaks = 25),
                                    list(name = "Period")))
)
# update settings and preferences
plotNCAParametersVsCovariates(
  covariates="SEQ", parameters="Tmax",
  settings=list(legend=T)
)
preferences <- list(spline=list(lineType="dashed"))
plotNCAParametersVsCovariates(covariates="AGE", parameter="Tmax",
                              settings=list(regressionLine=F, spline=T),
                              preferences=preferences)

# pre compute dataset
data <- getChartsData(plotName="plotNCAParametersVsCovariates")
plotNCAParametersVsCovariates(data=data)

parameters <- c("Lambda_z", "AUClast", "Clast", "Cmax")
covariates <- c("AGE", "WT", "FORM")
plotNCAParametersVsCovariates()
plotNCAParametersVsCovariates(parameters=parameters, covariates=covariates)
```

`plotNpc` *[Monolix] Plot Numerical predictive checks*

Description

[Monolix] Plot Numerical predictive checks

Usage

```
plotNpc(
  obsName = NULL,
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`obsName`	(*string*) Name of the observation (in dataset header). By default the first observation is considered.
`settings`	a list of optional settings: • `level` (*int*) level for prediction intervals computation (default 90). • `nbPoints` (*int*) Number of points for cdf grid computation (default 100).

- `useCensored` (*bool*) Choose to use BLQ data (TRUE) or to ignore it (FALSE) to compute the VPC (default TRUE). For continuous data only.
- `censoring` (*string*) BLQ data can be simulated ('simulated'), or can be equal to the limit of quantification ('loq') (default 'simulated'). For continuous data only.
- `empirical` (*bool*) - If TRUE, Empirical data is displayed (default TRUE): empirical percentiles for continuous data; empirical probability for discrete data; empirical curve for event data
- `theoretical` (*bool*) - If TRUE, median is displayed (default FALSE): median of predicted percentiles
- `predInterval` (*bool*) - If TRUE, Prediction interval is displayed (default TRUE).
- `outlierAreas` (*bool*) -If TRUE Add red areas indicating empirical percentiles that are outside prediction intervals (default TRUE).
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `xlog` (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE).
- `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE).
- `xlab` (*string*) label on x axis (default "Time").
- `ylab` (*string*) label on y axis (default obsName).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `xlim` (*c(double, double)*) limits of the x axis.
- `ylim` (*c(double, double)*) limits of the y axis.
- `fontsize` (*integer*) Plot text font size.
- `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotNpc")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values.
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter.
  - `interval` - in case of a continuous covariate, a list of filtering intervals.

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotNpc", ...)*) If data not specified, charts data will be computed inside the function.

Value

a ggplot2 object

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "monolix")
# continuous data
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()

plotNpc(obsName = "CONC")
```

---

plotObservationsVsPredictions

[Monolix] Plot Observation VS Prediction

Description

[Monolix] Plot Observation VS Prediction

Usage

```
plotObservationsVsPredictions(
  obsName = NULL,
  predictions = c("indiv"),
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

obsName	(*string*) Name of the observation (in dataset header). By default the first observation is considered.
predictions	(*string*) LIst of predictions to display: population prediction ("pop"), individual prediction ("indiv") (default c("indiv")).
settings	List with the following settings • `indivEstimate` (*string*) Calculation of individual estimates: conditional mean ("mean"), conditional mode with EBE's ("mode"), conditional distribution ("simulated") (default "mode").

- `useCensored` (*bool*) Choose to use BLQ data (TRUE) or to ignore it (FALSE) to compute the statistics (default TRUE).
- `censoring` (*string*) BLQ data can be simulated ('simulated'), or can be equal to the limit of quantification ('loq') (default 'simulated').
- `obs` (*bool*) - If TRUE observations are displayed as dots (default TRUE).
- `cens` (*bool*) - If TRUE censoring data are displayed as red dots (default TRUE).
- `spline` (*bool*) - If TRUE add spline (default FALSE).
- `identityLine` (*bool*) - If TRUE add identity line (default TRUE).
- `predInterval` (*bool*) - If TRUE add 90% prediction interval (default FALSE).
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `xlog` (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE).
- `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `xlim` (*c(double, double)*) limits of the x axis.
- `ylim` (*c(double, double)*) limits of the y axis.
- `fontsize` (*integer*) Plot text font size.
- `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").
- `ylab` (*string*) label on y axis (default "Observations").

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotObservationsVsPredictions")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of cahrts data as dataframe - Output of getChartsData (*getChartsData("plotObservationsVsPredic ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one prediction type,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()
runConditionalDistributionSampling()
runConditionalModeEstimation()

plotObservationsVsPredictions()
plotObservationsVsPredictions(predictions = "pop")
plotObservationsVsPredictions(prediction = "indiv", settings = list(indivEstimate = "simulated"))
plotObservationsVsPredictions(settings = list(indivEstimate = "mean", spline = TRUE))
plotObservationsVsPredictions(settings = list(indivEstimate = "mode", predInterval = TRUE))

# stratification
plotObservationsVsPredictions(stratify = list(filter = list(name = "SEX", cat = "F")))
plotObservationsVsPredictions(settings = list(ylog = TRUE, xlog = TRUE))
plotObservationsVsPredictions(stratify = list(splitGroup = list(name = "WEIGHT", breaks = c(75))))
plotObservationsVsPredictions(stratify = list(colorGroup = list(name = "WEIGHT", breaks = c(75))))
plotObservationsVsPredictions(
  settings=list(legend=T),
  stratify = list(colorGroup=list(list(name = "SEX"),
                                  list(name = "WEIGHT", breaks = 70)))
)

data <- getChartsData(plotName = "plotObservationsVsPredictions",
                      computeSettings = list(indivEstimate = "simulated"),
                      colorGroup = list(name = "WEIGHT", breaks = c(75)))
plotObservationsVsPredictions(data = data)

# display multiple predictions
plotObservationsVsPredictions(predictions = c("pop", "indiv"))
```

---

plotObservedData *[Monolix - PKanalix] Generate Observation plots*

Description

[Monolix - PKanalix] Generate Observation plots

Usage

```
plotObservedData(
  obsName = NULL,
  data = NULL,
  settings = list(),
  stratify = list(),
  preferences = list()
)
```

Arguments

obsName	(*string*) Name of the observation (in dataset header). By default the first observation is considered.
data	List of charts data as dataframe - Output of getChartsData (*getChartsData("plotObservedData", ...)*) If data not specified, charts data will be computed inside the function.
settings	List with the following settings [CONTINUOUS - DISCRETE] Settings specific to continuous and discrete data

- dots (*bool*) - If TRUE individual observations are displayed as dots (default TRUE).
- lines (*bool*) - If TRUE individual observations are displayed as lines (default TRUE).
- mean (*bool*) - If TRUE mean of observations is displayed (default FALSE).
- error (*bool*) If TRUE error bar is is displayed (default FALSE).
- meanMethod (*string*) - When mean is set to TRUE, display arithmetic mean ("arithmetic") or geometric mean ("geometric"). Default value is "arithmetic".
- errorMethod (*string*) - When error is set to TRUE, display standard deviation ("standardDeviation") or standard error ("standardError"). Default value is "standardDeviation".
- useCensored (*bool*) Choose to use censored data to compute mean and error (TRUE) or to ignore it (FALSE) (default FALSE).
- binLimits (*bool*) - Add bins limits as vertical lines (default FALSE).
- binsSettings a list of settings for time axis binning for observation statistics computation:
  - criteria (*string*) - Bining criteria, one of 'equalwidth', 'equalsize', or 'leastsquare' methods. (default leastsquare).
  - is.fixedNbBins (*bool*) - If TRUE define a fixed number of bins, else define a range for automatic selection (default FALSE).
  - nbBins (*int*) - Define a fixed number of bins (default 10).
  - binRange (*vector(int, int)*) - Define a range for the number of bins (default c(5, 100)).
  - nbBinData (*vector(int, int)*) - Define a range for the number of data points per bin (default c(10, 200) for Monolix and c(3, 200) for PKanalix).

[DISCRETE] Settings specific to discrete data

- plot (*string*) Type of plot: "continuous" (default), "stacked" and "grouped".
- histogramColors (*vector<string>*) List of colors to use in histograms plots.

[EVENT] Settings specific to event data

- `eventPlot` - Display Survival function ("survivalFunction") or mean number of events per subject ("averageEventNumber") (default "survivalFunction").

Other settings

- `cens` (*boolean*) - If TRUE censored data are displayed as dots, in addition to survival function (default TRUE).
- `dosingTimes` (*boolean*) - Add dosing times as vertical lines (default FALSE). For project with dose information only
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) - Add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `xlog` (*bool*) - Add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE).
- `ylog` (*bool*) - Add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE).
- `xlab` (*string*) - Label on x axis (default "Time").
- `ylab` (*string*) - Label on y axis (default obsName).
- `ncol` (*int*) - Number of columns when facet = TRUE (default 4).
- `xlim` (*c(double, double)*) - Limits of the x axis.
- `ylim` (*c(double, double)*) - Limits of the y axis.
- `fontsize` (*integer*) - Plot text font size.
- `units` (*boolean*) - Set units in axis labels (only available with PKanalix).
- `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotObservedData")* to check available displays.

Value

A ggplot object

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "pkanalix")
project <- file.path(getDemoPath(), "2.case_studies/project_Theo_extravasc_SD.pkx")
loadProject(project)

plotObservedData()
plotObservedData(settings = list(binLimits = TRUE))
plotObservedData(settings = list(dosingTimes = TRUE))
plotObservedData(settings = list(meanMethod = "geometric", mean = TRUE))
plotObservedData(settings = list(mean = TRUE, error = TRUE, dots = FALSE, lines = TRUE))

# stratification
plotObservedData(stratify = list(splitGroup = list(name = "AGE", breaks = 25),
                                 filter = list(name = "Period", cat = 1)))
plotObservedData(stratify = list(colorGroup = list(name = "HT", breaks = 181)))
plotObservedData(stratify = list(splitGroup = list(list(name = "AGE", breaks = 25),
                                                   list(name = "Period"))))
# update plot theme or preferences
plotObservedData(settings = list(xlab = "Time", ylab = "Plasma Concentration"))
plotObservedData(preferences = list(obs = list(color = "#32CD32"),
                                    observationStatistics = list(lineType = "dashed")))
```

---

plotParametersDistribution

[Monolix] Distribution of the individual parameters computed by Monolix

---

Description

[Monolix] Distribution of the individual parameters computed by Monolix

Usage

```
plotParametersDistribution(
  parameters = NULL,
  plot = "pdf",
  settings = list(),
  preferences = NULL,
  stratify = list(),
  data = NULL
)
```

Arguments

`parameters` vector of parameters to display. (by default the first 4 computed parameters are displayed).

`plot` (*string*) Type of plot: probability density distribution ("pdf"), cumulative density distribution ("cdf") (default "pdf")

`settings` a list of optional plot settings:

- `indivEstimate` Calculation of individual estimates: conditional mean ("mean"), conditional mode with EBE's ("mode"), conditional distribution ("simulated") (default "simulated").
- `empirical` (*bool*) If TRUE, plot empirical density distribution (default TRUE).
- `theoretical` (*bool*) If TRUE, plot theoretical density distribution (default TRUE).
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `fontsize` (*integer*) Plot text font size.

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotParametersDistribution")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotParametersDistributio ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one parameter,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software="monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()
runConditionalDistributionSampling()

plotParametersDistribution(parameters="ka")
plotParametersDistribution(parameters="Cl", plot="pdf")
plotParametersDistribution(parameters="ka", plot="cdf")
plotParametersDistribution(parameters="ka", plot="cdf",
                           settings=list(indivEstimate="simulated"))
plotParametersDistribution(parameters="Cl", plot="pdf",
                           settings=list(theoretical=F))

# stratification
plotParametersDistribution(stratify=list(filter=list(name="WEIGHT", interval=c(0, 75))))
plotParametersDistribution(parameters="Cl", stratify=list(splitGroup=list(name="SEX")))
colorGroup <- list(name="WEIGHT", breaks=c(75))
plotParametersDistribution(parameters= "Cl", plot="pdf",
                  stratify=list(colorGroup=colorGroup, colors=c("#46B4AF", "#B4468A")))
plotParametersDistribution(parameters="Cl", plot="cdf",
                  stratify=list(colorGroup=colorGroup, colors=c("#46B4AF", "#B4468A")))

# update preferences
preferences = list(theoretical=list(color="#B4468A", lineType="solid", lineWidth=0.8))
plotParametersDistribution(parameters="ka", plot="cdf", preferences=preferences)


# pre compute dataset
data <- getChartsData(plotName="plotParametersDistribution",
                      computeSettings=list(indivEstimate="simulated"))
plotParametersDistribution(data=data)

# multiple plots
plotParametersDistribution(parameters=c("ka", "Cl"))
plotParametersDistribution(plot="pdf")
plotParametersDistribution(plot="cdf")
plotParametersDistribution(plot="cdf", settings=list(indivEstimate="simulated"))
plotParametersDistribution(plot = "pdf", settings=list(theoretical=F))
```

`plotParametersVsCovariates`

[Monolix] Individual monolix parameter vs covariate plot

Description

[Monolix] Individual monolix parameter vs covariate plot

Usage

```
plotParametersVsCovariates(
  parameters = NULL,
  covariates = NULL,
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

parameters	vector of parameters to display. (by default the first 4 computed parameters are displayed).
covariates	vector of covariates to display. (by default the first 4 computed covariates are displayed).
settings	List with the following settings • indivEstimate Calculation of individual estimates: conditional mean ("mean"), conditional mode with EBE's ("mode"), conditional distribution ("simulated") (default "simulated"). • parameterType (*string*) display random effect vs covariates ("randomEffect"), or transformed individual parameters vs covariates ("indivParameter") (default "indivParameter"). • boxplotData (*string*) for categorical covariate, if boxplotData is not NULL, data are added as dots over the boxplot. They can be either "spread" on the box or "aligned" (default NULL) • regressionLine (*bool*) If TRUE, Add regression line in scatterplots (default TRUE). • spline (*bool*) If TRUE, Add xpline in scatterplots (default FALSE). • legend (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • grid (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • ncol (*int*) number of columns when facet = TRUE (default 4). • fontsize (*integer*) Plot text font size.
preferences	(*optional*) preferences for plot display, run *getPlotPreferences("plotParametersVsCovariates")* to check available displays.
stratify	List with the stratification arguments • ids - List of ids to display (by default all ids are displayed). • splitGroup - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields: – name : The name of the covariate to use in grouping, – breaks : In case of a continuous covariate, a list of break values, – groups : [optional] In case of a categorical covariate, define groups of modalities. • colorGroup - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:

  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotParametersVsCovaria ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one covariate and one parameter in argument,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software="monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()
runConditionalDistributionSampling()

# Individual parameters
plotParametersVsCovariates(covariates="SEX", parameters="Cl")
plotParametersVsCovariates(covariates="WEIGHT", parameters="V", settings=list(spline=T))
plotParametersVsCovariates(covariates="WEIGHT", parameters="V",
                           settings=list(indivEstimate="simulated"))

# Random effects
plotParametersVsCovariates(covariates="SEX", parameters="V",
                           settings=list(parameterType="randomEffect"))
plotParametersVsCovariates(covariates="WEIGHT", parameters="V",
                  settings=list(indivEstimate="simulated", parameterType="randomEffect"))

# Stratification
plotParametersVsCovariates(covariates="SEX", parameters="ka",
                         stratify=list(filter=list(name="WEIGHT", interval=c(0, 75))))
plotParametersVsCovariates(covariates="WEIGHT", parameters="ka",
                           stratify=list(splitGroup=list(name="SEX")))
plotParametersVsCovariates(covariates="SEX", parameters="Cl",
                           stratify=list(colorGroup=list(name="WEIGHT", breaks=75)))
plotParametersVsCovariates(covariates="WEIGHT", parameters="V",
```

```
                            stratify=list(colorGroup=list(name="SEX")))
plotParametersVsCovariates(covariates="WEIGHT", parameters="V",
                            stratify = list(colorGroup = list(list(name = "SEX"),
                                                    list(name="WEIGHT", breaks=70))))

# pre process dataset
data <- getChartsData(plotName="plotParametersVsCovariates",
                      computeSettings=list(indivEstimate="simulated"))
plotParametersVsCovariates(data=data)

# multiple plots
plotParametersVsCovariates()
plotParametersVsCovariates(covariates="WEIGHT")
plotParametersVsCovariates(settings=list(indivEstimate="simulated"))
plotParametersVsCovariates(settings=list(parameterType="randomEffect"))
plotParametersVsCovariates(settings=list(parameterType="randomEffect", indivEstimate="simulated"))
plotParametersVsCovariates(stratify=list(colorGroup=list(name="WEIGHT", breaks=75)))
plotParametersVsCovariates(stratify=list(colorGroup=list(name="SEX")))
```

`plotPredictionDistribution`

[Monolix] Plot distribution of the predictions

Description

Note that computation settings are not available for this connector in 2021 version: Number of bands is set to 9 and Level is set to 90

Usage

```
plotPredictionDistribution(
  obsName = NULL,
  settings = list(),
  preferences = list(),
  data = NULL
)
```

Arguments

`obsName`	(*string*) Name of the observation (in dataset header). By default the first observation is considered.
`settings`	a list of optional settings • `perc` (*bool*) - If TRUE display 9 Bands for each percentile (default TRUE). • `median` (*bool*) - If TRUE display Median (default TRUE). • `obs` (*bool*) - If TRUE display observations as dots (default FALSE). • `cens` (*bool*) - If TRUE display censored observations as dots (default FALSE). • `binLimits` (*bool*) If TRUE display limits of bins (default FALSE). For discrete data only. • `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).

- `xlog` (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE).
- `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE).
- `xlab` (*string*) label on x axis (default "Time").
- `ylab` (*string*) label on y axis (default obsName).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `xlim` (*c(double, double)*) limits of the x axis.
- `ylim` (*c(double, double)*) limits of the y axis.
- `fontsize` (*integer*) Plot text font size.
- `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotPredictionDistribution")* to check available displays.

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotPredictionDistributio ...)*) If data not specified, charts data will be computed inside the function.

Details

Note that stratification options are not available for this connector in 2021 version:

Value

a ggplot2 object

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "monolix")

# continuous data
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()

plotPredictionDistribution()
```

`plotRandomEffectsCorrelation` *[Monolix] Correlations between random effect*

Description

[Monolix] Correlations between random effect

Usage

```
plotRandomEffectsCorrelation(
  parametersRows = NULL,
  parametersColumns = NULL,
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`parametersRows`	vector with the name of parameters to display on rows (by default the first 4 computed parameters are displayed).
`parametersColumns`	vector with the name of parameters to display on columns (by default parametersColumns = parametersRows).
`settings`	List with the following settings • `indivEstimate` Calculation of individual estimates: conditional mean ("mean"), conditional mode with EBE's ("mode"), conditional distribution ("simulated") (default "simulated"). • `variabilityLevel` (*string*) In case of IOV and if the conditional distribution is computed plot is displayed for one given level of variability (default NULL) If NULL, the variability level is ID + Occasions Run getVariabilityLevels() to see the available levels of variability • `regressionLine` (*bool*) If TRUE, Add regression line in scatterplots (default TRUE). • `spline` (*bool*) If TRUE, Add xpline in scatterplots (default FALSE). • `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE). • `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • `ncol` (*int*) number of columns when facet = TRUE (default 4). • `fontsize` (*integer*) Plot text font size.
`preferences`	(*optional*) preferences for plot display, run *getPlotPreferences("plotRandomEffectsCorrelation")* to check available displays.
`stratify`	List with the stratification arguments • `ids` - List of ids to display (by default all ids are displayed). • `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields: – `name` : The name of the covariate to use in grouping,

  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotRandomEffectsCorrel ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one element in parametersRows and parametersColumns,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()
runConditionalDistributionSampling()

plotRandomEffectsCorrelation()

plotRandomEffectsCorrelation(parametersRows = "ka", parametersColumns = "V",
                             settings = list(indivEstimate = "simulated"))
plotRandomEffectsCorrelation(parametersRows = "ka", parametersColumns = "V",
                             settings = list(spline = TRUE))

plotRandomEffectsCorrelation(parametersRows = c("ka", "V"))

# stratification
plotRandomEffectsCorrelation(parametersRows = "ka", parametersColumns = "V",
                             stratify = list(filter = list(name = "SEX", cat = "M")))
plotRandomEffectsCorrelation(parametersRows = "ka", parametersColumns = "V",
                             stratify = list(
```

```
                                colorGroup = list(name = "WEIGHT", breaks = 75),
                                colors = c("#46B4AF", "#B4468A")))
  plotRandomEffectsCorrelation(parametersRows = "ka", parametersColumns = "V",
                               stratify = list(splitGroup = list(name = "SEX")))

  plotRandomEffectsCorrelation(parametersRows = "ka", parametersColumns = "V",
                               stratify = list(splitGroup = list(list(name = "SEX"),
                                                   list(name="WEIGHT", breaks=70))))

  # pre compute dataset
  data <- getChartsData(plotName = "plotRandomEffectsCorrelation",
                        computeSettings = list(indivEstimate = "simulated"))
  plotRandomEffectsCorrelation(data = data)

  plotRandomEffectsCorrelation(settings = list(indivEstimate = "mean"))
```

`plotResidualsDistribution`

[Monolix] Generate Distribution of the residuals

Description

[Monolix] Generate Distribution of the residuals

Usage

```
plotResidualsDistribution(
  obsName = NULL,
  residuals = c("indiv", "npde"),
  plots = c("pdf", "cdf"),
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`obsName`	(*string*) Name of the observation (in dataset header). By default the first observation is considered.
`residuals`	(*string*) List of residuals to display: population residuals ("pop"), individual residuals ("indiv"), normalized prediction distribution error ("npde") (default c("indiv", "npde")).
`plots`	Type of plots: probability density distribution ("pdf"), cumulative density distribution ("cdf") (default c("pdf", "cdf")).
`settings`	List with the following settings • `indivEstimate` (*string*) Calculation of individual estimates: conditional mean ("mean"), conditional mode with EBE's ("mode"), conditional distribution ("simulated") (default "simulated").

- `useCensored` (*bool*) Choose to use BLQ data (TRUE) or to ignore it (FALSE) to compute the statistics (default TRUE). For continuous data only.
- `censoring` (*string*) BLQ data can be simulated ('simulated'), or can be equal to the limit of quantification ('loq') (default 'simulated'). For continuous data only.
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `fontsize` (*integer*) Plot text font size.

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotResidualsDistribution")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotResidualsDistribution ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one prediction type,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software="monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()
runConditionalDistributionSampling()

plotResidualsDistribution()
plotResidualsDistribution(residuals="indiv", settings=list(indivEstimate="simulated"))
plotResidualsDistribution(residuals="indiv", settings=list(indivEstimate="mode"))
plotResidualsDistribution(residuals="pop", plots="pdf")
plotResidualsDistribution(residuals="npde", plots="cdf")
plotResidualsDistribution(stratify=list(filter=list(name="SEX", cat="F")))
plotResidualsDistribution(stratify=list(splitGroup=list(name="WEIGHT", breaks=c(75))))
plotResidualsDistribution(
  residuals="indiv", settings=list(legend=T),
  stratify = list(splitGroup=list(list(name = "SEX"),
                                  list(name = "WEIGHT", breaks = 70)))
)

data <- getChartsData(plotName="plotResidualsDistribution",
                      computeSettings=list(indivEstimate="simulated"))
plotResidualsDistribution(data=data)

plotResidualsDistribution()
plotResidualsDistribution(residuals=c("indiv", "npde"), settings=list(indivEstimate="simulated"))
plotResidualsDistribution(residuals=c("pop", "indiv"), settings=list(indivEstimate="mode"))
plotResidualsDistribution(plots=c("pdf", "cdf"))
plotResidualsDistribution(plots=c("cdf"))
plotResidualsDistribution(residuals="npde")
```

---

plotResidualsScatterPlot

[Monolix] Generate Scatter plots of the residuals

Description

Note that 'prediction interval' setting is not available in 2021 version for this connector.

Usage

```
plotResidualsScatterPlot(
  obsName = NULL,
  residuals = c("indiv"),
  xaxis = c("time", "prediction"),
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`obsName` (*string*) Name of the observation (in dataset header). By default the first observation is considered.

`residuals` (*string*) List of residuals to display: population residuals ("pop"), individual residuals ("indiv"), normalized prediction distribution error ("npde") (default c("indiv")).

`xaxis` (*string*) List of x-axis to display: time ("time"), prediction ("prediction") (default c("time", "prediction") for continuous data, c("time") for discrete data).

`settings` List with the following settings

- `indivEstimate` (*string*) Calculation of individual estimates: conditional mean ("mean"), conditional mode with EBE's ("mode"), conditional distribution ("simulated") (default "mode"). For continuous data only
- `level` (*int*) level for prediction intervals computation (default 90).
- `higherPercentile` (*int*) Higher percentile for empirical and predicted percentiles computation (default 90).
- `useCensored` (*bool*) Choose to use BLQ data (TRUE) or to ignore it (FALSE) to compute the statistics (default TRUE). For continuous data only.
- `censoring` (*string*) BLQ data can be simulated ('simulated'), or can be equal to the limit of quantification ('loq') (default 'simulated'). For continuous data only.
- `binsSettings` a list of settings for bins:
  - `criteria` (*string*) Bining criteria, one of 'equalwidth', 'equalsize', or 'leastsquare' methods. (default leastsquare).
  - `is.fixedNbBins` (*bool*) If TRUE define a fixed number of bins, else define a range for automatic selection (default TRUE).
  - `nbBins` (*int*) Define a fixed number of bins (default 10).
  - `binRange` (*vector(int, int)*) Define a range for the number of bins (default c(5, 100)).
  - `nbBinData` (*vector(int, int)*) Define a range for the number of data points per bin (default c(10, 200)).
- `residuals` (*bool*) - If TRUE display residuals (default TRUE).
- `cens` (*bool*) - If TRUE display censored data (default TUE).
- `empPercentiles` (*bool*) - If TRUE display empirical percentiles (default FALSE).
- `predPercentiles` (*bool*) - If TRUE display predicted percentiles (default FALSE).
- `spline` (*bool*) - If TRUE display spline (default FALSE).
- `binLimits` (*bool*) - If TRUE Add bins limits as vertical lines (default FALSE).
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `xlog` (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE).
- `ylog` (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `xlim` (*c(double, double)*) limits of the x axis.
- `ylim` (*c(double, double)*) limits of the y axis.

- `fontsize` (*integer*) Plot text font size.
- `scales` (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotResidualsScatterPlot")* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of cahrts data as dataframe - Output of getChartsData (*getChartsData("plotResidualsScatterPlot" ...)*) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one prediction type,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software="monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()
runConditionalDistributionSampling()
runConditionalModeEstimation()
```

```
plotResidualsScatterPlot()
plotResidualsScatterPlot(residuals="indiv", settings=list(indivEstimate="simulated"))
plotResidualsScatterPlot(residuals="indiv", settings=list(indivEstimate="mode"))
plotResidualsScatterPlot(xaxis="prediction", residuals="pop")
plotResidualsScatterPlot(xaxis="time", residuals="pop")
plotResidualsScatterPlot(residuals="npde")
plotResidualsScatterPlot(settings=list(spline=T))
plotResidualsScatterPlot(settings=list(empPercentiles=T, level=90,
                                       binsSettings=list(is.fixedNbBins=T, nbBins=5),
                                       binLimits=T))

# Stratification
plotResidualsScatterPlot(stratify=list(filter=list(name="SEX", cat="F")))
plotResidualsScatterPlot(stratify=list(splitGroup=list(name="WEIGHT", breaks=c(75))))
plotResidualsScatterPlot(stratify=list(colorGroup=list(name="WEIGHT", breaks=c(75))))

data <- getChartsData(plotName="plotResidualsScatterPlot",
                      computeSettings=list(indivEstimate="simulated"))
plotResidualsScatterPlot(data=data)

plotResidualsScatterPlot(residuals=c("indiv", "pop"),
                         settings=list(indivEstimate="simulated"))
plotResidualsScatterPlot(residuals="indiv", xaxis=c("prediction"),
                         settings=list(indivEstimate="mode"))
plotResidualsScatterPlot(xaxis=c("prediction"), residuals=c("indiv", "pop"))
plotResidualsScatterPlot(residuals="npde")
```

`plotSaem` *[Monolix] Plot SAEM convergence*

Description

[Monolix] Plot SAEM convergence

Usage

```
plotSaem(settings = list(), data = NULL)
```

Arguments

`settings`	a list of optional settings: • `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE). • `ncol` (*int*) number of columns when facet = TRUE (default 4). • `fontsize` (*integer*) Plot text font size.
`data`	dataframe - Output of getChartsData (*getChartsData("plotSaem", ...)*) If data not specified, charts data will be computed inside the function.

Value

A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData

Examples

```
initializeLixoftConnectors(software = "monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)

runPopulationParameterEstimation()

plotSaem()
```

`plotStandardizedRandomEffectsDistribution`
[Monolix] Distribution of the standardized random effects

Description

[Monolix] Distribution of the standardized random effects

Usage

```
plotStandardizedRandomEffectsDistribution(
  parameters = NULL,
  plot = "boxplot",
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

`parameters`	vector of parameters to display. (by default the first 4 computed parameters are displayed).
`plot`	Type of plot: probability density distribution ("pdf"), cumulative density distribution ("cdf"), boxplot ("boxplot") (default "boxplot").
`settings`	a list of optional plot settings: • `indivEstimate` Calculation of individual estimates: conditional mean ("mean"), conditional mode with EBE's ("mode"), conditional distribution ("simulated") (default "mode"). • `variabilityLevel` (*bool*) In case of IOV and if the conditional distribution is computedplot is displayed for one given level of variability (default NULL) If NULL, the variability level is ID + Occasions Run getVariabilityLevels() to see the available levels of variability • `empirical` (*bool*) If TRUE, plot empirical density distribution(default TRUE). To define when plot is "pdf" or "cdf" • `theoretical` (*bool*) If TRUE, plot theoretical density distribution(default TRUE). To define when plot is "pdf" or "cdf" • `median` (*bool*) If TRUE, add median line (default TRUE). To define when plot is "boxplot"

- `quartile` (*bool*) If TRUE, add quartile line (default TRUE). To define when plot is "boxplot"
- `legend` (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- `grid` (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- `ncol` (*int*) number of columns when facet = TRUE (default 4).
- `fontsize` (*integer*) Plot text font size.

`preferences` (*optional*) preferences for plot display, run *getPlotPreferences("plotStandardizedRandomEffectsDistr* to check available displays.

`stratify` List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotStandardizedRandom* ...)) If data not specified, charts data will be computed inside the function.

Value

- A ggplot object if one parameter,
- A TableGrob object if multiple plots (output of grid.arrange)

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software="monolix")
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)
```

```
runPopulationParameterEstimation()
runConditionalDistributionSampling()
runConditionalModeEstimation()

# Random effect distribution as boxplot
plotStandardizedRandomEffectsDistribution(parameters="ka", plot="boxplot")
plotStandardizedRandomEffectsDistribution(parameters="ka", plot="boxplot",
                                          settings=list(indivEstimate="mode"))
plotStandardizedRandomEffectsDistribution(parameters="ka", plot="boxplot",
                                          settings=list(quartile=F))

# Random effect distribution as pdf
plotStandardizedRandomEffectsDistribution(parameters="ka", plot="pdf")
plotStandardizedRandomEffectsDistribution(parameters="ka", plot="pdf",
                                          settings=list(empirical=F))
plotStandardizedRandomEffectsDistribution(parameters="ka", plot="pdf",
                                          settings=list(theoretical=F))

# Random effect distribution as cdf
plotStandardizedRandomEffectsDistribution(parameters="ka", plot="cdf")
plotStandardizedRandomEffectsDistribution(parameters="ka", plot="cdf",
                                          settings=list(indivEstimate="simulated"))
plotStandardizedRandomEffectsDistribution(parameters="ka", plot="cdf",
                                          settings=list(theoretical=F))

# stratification
plotStandardizedRandomEffectsDistribution(
  stratify=list(filter=list(name="WEIGHT", interval=c(0, 75)))
)
plotStandardizedRandomEffectsDistribution(parameters="Cl",
                                          stratify=list(splitGroup=list(name="SEX")))
colorGroup <- list(name="WEIGHT", breaks=c(75))
plotStandardizedRandomEffectsDistribution(
  parameters="Cl", plot="pdf",
  stratify=list(colorGroup=colorGroup, colors=c("#46B4AF", "#B4468A"))
)
plotStandardizedRandomEffectsDistribution(
  parameters="Cl", plot="cdf",
  stratify=list(colorGroup=colorGroup, colors=c("#46B4AF", "#B4468A"))
)
plotStandardizedRandomEffectsDistribution(
  parameters="Cl", settings=list(plot="boxplot"),
  stratify=list(colorGroup=colorGroup, colors=c("#46B4AF", "#B4468A"))
)

data <- getChartsData(plotName="plotStandardizedRandomEffectsDistribution",
                      computeSettings=list(indivEstimate="simulated"))
plotStandardizedRandomEffectsDistribution(data=data)

plotStandardizedRandomEffectsDistribution(parameters=c("ka", "Cl"))
plotStandardizedRandomEffectsDistribution(plot="boxplot")
plotStandardizedRandomEffectsDistribution(plot="pdf")
plotStandardizedRandomEffectsDistribution(plot="cdf")
plotStandardizedRandomEffectsDistribution(plot="pdf", settings=list(theoretical=F))
```

plotVpc *[Monolix] Plot Visual predictive checks*

Description

[Monolix] Plot Visual predictive checks

Usage

```
plotVpc(
  obsName = NULL,
  eventPlot = "survivalFunction",
  settings = list(),
  preferences = list(),
  stratify = list(),
  data = NULL
)
```

Arguments

obsName	(*string*) Name of the observation (in dataset header). By default the first observation is considered.
eventPlot	(*string*) Display Survival function ("survivalFunction") or average number of event ("averageEventNumber) (default "survivalFunction"). For event data only.
settings	a list of optional settings: • `level` (*int*) level for prediction intervals computation (default 90), • `higherPercentile` (*int*) Higher percentile for empirical and predicted percentiles computation (default 90). For continuous data only. • `useCorrpred` (*bool*) if TRUE, pcVPC are computed using Uppsala prediction correction (default FALSE). For continuous data only. • `useCensored` (*bool*) Choose to use BLQ data (TRUE) or to ignore it (FALSE) (default TRUE). For continuous data only. • `censoring` (*string*) BLQ data can be simulated ('simulated'), or can be equal to the limit of quantification ('loq') (default 'simulated'). For continuous data only. • `timeAfterLastDose` (*bool*) display vpc only after last dose (default FALSE) For data with dose information only. • `nbDataPoints` (*int*) Number of data point in event time grid (default 100) For event data only. • `xBinsSettings` a list of optional settings for time axis binning For continuous and discrete data only – `criteria` (*string*) Bining criteria, one of 'equalwidth', 'equalsize', or 'leastsquare' methods. (default leastsquare). – `is.fixedNbBins` (*bool*) If TRUE define a fixed number of bins, else define a range for automatic selection (default FALSE). – `nbBins` (*int*) Define a fixed number of bins (default 10). – `binRange` (*vector(int, int)*) Define a range for the number of bins (default c(5, 100)).

  - nbBinData (*vector(int, int)*) Define a range for the number of data points per bin (default c(10, 200)).
- yBinsSettings a list of optional settings for y axis binning. For countable discrete data only
  - criteria (*string*) Bining criteria, one of 'equalwidth', 'equalsize', or 'leastsquare' methods. (default leastsquare).
  - is.fixedNbBins (*bool*) If TRUE define a fixed number of bins, else define a range for automatic selection (default TRUE).
  - nbBins (*int*) Define a fixed number of bins (default 10).
  - binRange (*vector(int, int)*) Define a range for the number of bins (default c(5, 100)).
  - nbBinData (*vector(int, int)*) Define a range for the number of data points per bin (default c(10, 200)).
- obs (*bool*) - If TRUE, Observed data is displayed as dots (defaul FALSE).
- cens (*bool*) - If TRUE, Censored data is displayed as dots (defaul FALSE).
- empirical (*bool*) - If TRUE, Empirical data is displayed (default TRUE): empirical percentiles for continuous data; empirical probability for discrete data; empirical curve for event data
- theoretical (*bool*) - If TRUE, median is displayed (default FALSE): median of predicted percentiles for continuous data, median of predicted probability for discrete data, median of KM curves for event data
- predInterval (*bool*) - If TRUE, Prediction interval is displayed (default TRUE).
- linearInterpolation (*bool*) - If TRUE set piece wise display for prediction intervals, else show bins as rectangular (default TRUE).
- outlierDots (*bool*) - If TRUE, Add red dots indicating empirical percentiles that are outside prediction intervals (default TRUE).
- outlierAreas (*bool*) - If TRUE Add red areas indicating empirical percentiles that are outside prediction intervals (default TRUE).
- binLimits (*bool*) - Add/remove vertical lines on the scatter plots to indicate the bins (default FALSE).
- legend (*bool*) add (TRUE) / remove (FALSE) plot legend (default FALSE).
- grid (*bool*) add (TRUE) / remove (FALSE) plot grid (default TRUE).
- xlog (*bool*) add (TRUE) / remove (FALSE) log scaling on x axis (default FALSE).
- ylog (*bool*) add (TRUE) / remove (FALSE) log scaling on y axis (default FALSE).
- xlab (*string*) label on x axis (default "Time").
- ylab (*string*) label on y axis (default obsName).
- ncol (*int*) number of columns when facet = TRUE (default 4).
- xlim (*c(double, double)*) limits of the x axis.
- ylim (*c(double, double)*) limits of the y axis.
- fontsize (*integer*) Plot text font size.
- scales (*string*) Should scales be fixed ("fixed"), free ("free", the default), or free in one dimension ("free_x", "free_y") (default "free").

preferences (*optional*) preferences for plot display, run *getPlotPreferences("plotVpc")* to check available displays.

stratify List with the stratification arguments

- `ids` - List of ids to display (by default all ids are displayed).
- `splitGroup` - Split plots by groups of covariates (by default no split is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `colorGroup` - Color plots by groups of covariates (by default no color is applied). A list, or a list of list with fields:
  - `name` : The name of the covariate to use in grouping, or the name of the column id,
  - `breaks` : In case of a continuous covariate, a list of break values,
  - `groups` : [optional] In case of a categorical covariate, define groups of modalities.
- `filter` - Filter data (by default no filtering is applied). A list, or a list of list with fields:
  - `name` - the name of the covariate to filter,
  - `cat` - in case of a categorical covariate, the name of the category to filter,
  - `interval` - in case of a continuous covariate, a list of filtering intervals.
- `colors` - List of colors to use when colorGroup argument is defined

`data` List of charts data as dataframe - Output of getChartsData (*getChartsData("plotVpc", ...)*) If data not specified, charts data will be computed inside the function.

Value

a ggplot2 object

See Also

getChartsData getPlotPreferences

Examples

```
initializeLixoftConnectors(software = "monolix")
# continuous data
project <- file.path(getDemoPath(), "1.creating_and_using_models",
                     "1.1.libraries_of_models", "theophylline_project.mlxtran")
loadProject(project)
runPopulationParameterEstimation()

data <- getChartsData("plotVpc")
p <- plotVpc(data = data, obsName = "CONC",
             settings = list(outlierDots = FALSE, grid = FALSE,
                             ylab = "Concentration", xlab = "time (in hour)"))

# categorical data
project <- file.path(getDemoPath(), "3.models_for_noncontinuous_outcomes",
                     "3.1.categorical_data_model", "categorical1_project.mlxtran")
loadProject(project)
runPopulationParameterEstimation()
data <- getChartsData(plotName = "plotVpc")
```

```
p <- plotVpc(data = data, obsName = "level",
             settings = list(theoretical = TRUE, outlierDots = FALSE))

# countable data
project <- file.path(getDemoPath(), "3.models_for_noncontinuous_outcomes",
                     "3.2.count_data_model", "count1a_project.mlxtran")
loadProject(project)
runPopulationParameterEstimation()
data <- getChartsData(plotName = "plotVpc")
p <- plotVpc(data = data, obsName = "Y")

# time to event data
project <- file.path(getDemoPath(), "3.models_for_noncontinuous_outcomes",
                     "3.3.time_to_event_data_model", "tte1_project.mlxtran")
loadProject(project)
runPopulationParameterEstimation()
data <- getChartsData(plotName = "plotVpc")
plotVpc(data = data, obsName = "Event", eventPlot = "survivalFunction")
```

`removeCovariate` *[Monolix] Remove covariate*

Description

Remove some of the transformed covariates (discrete and continuous) and/or latent covariates. Call `getCovariateInformation` to know which covariates can be removed.

Usage

```
removeCovariate(...)
```

Arguments

`...`	A list of covariate names.

See Also

`getCovariateInformation` `addContinuousTransformedCovariate` `addCategoricalTransformedCovariate` `addMixture`

Examples

```
## Not run:
removeCovariate("tWt","lcat1")

## End(Not run)
```

removeFilter *[Monolix - PKanalix] Remove filter*

Description

Remove the last filter applied on the current data set.

Usage

```
removeFilter()
```

See Also

applyFilter selectData

Examples

```
## Not run:
removeFilter()

## End(Not run)
```

removeGroup *[Simulx] Remove simulation group*

Description

Remove a simulation group.

Usage

```
removeGroup(group)
```

Arguments

group	(*string*) Name of the group to remove.

Details

Simulation groups are used for simulation as in Simulx GUI. At the creation of a Simulx project, a first group is created by default with the name "simulationGroup1". Use getGroups to check which groups have already been defined, and which elements are set in each group. To add a simulation group, use addGroup. To remove a simulation group, use removeGroup. To add or change a group element, use setGroupElement. To define new elements, use one of the define...Element functions.

See Also

getGroups, addGroup

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "simulationGroups_treatment.smlx")
loadProject(project_name)
removeGroup("medium_dose")
```

`removeGroupElement` *[Simulx] Remove element from simulation group*

Description

Remove an element from a simulation group.

Usage

```
removeGroupElement(group, element)
```

Arguments

`group`	(*character*) Group name
`element`	(*character*) Element to remove

Details

Simulation groups are used for simulation as in Simulx GUI. At the creation of a Simulx project, a first group is created by default with the name "simulationGroup1". Use getGroups to check which groups have already been defined, and which elements are set in each group. To add a simulation group, use addGroup. To remove a simulation group, use removeGroup. To add or change a group element, use setGroupElement. To define new elements, use one of the define...Element functions.

Note: Removing an output element used in an outcome will delete the corresponding outcome and remove the outcome from the endpoints using it.

See Also

`setGroupElement`

Examples

```
# Remove an output element from all groups
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "simulationGroups_treatment.smlx")
loadProject(project_name)
removeGroupElement(group = "low_dose", element = "regularY1")
removeGroupElement(group = "medium_dose", element = "regularY1")
removeGroupElement(group = "high_dose", element = "regularY1")
```

renameAdditionalCovariate *[Monolix - PKanalix] Rename additional covariate*

Description

Rename an existing additional covariate.

Usage

```
renameAdditionalCovariate(oldName, newName)
```

Arguments

oldName	(*string*) current name of the covariate to rename
newName	(*string*) new name.

See Also

addAdditionalCovariate

Examples

```
## Not run:
renameAdditionalCovariate(oldName = "observationNumberPerIndividual_y1", newName = "nbObsForY1")

## End(Not run)
```

renameFilter *[Monolix - PKanalix] Rename filter*

Description

Rename an existing filtered data set.

Usage

```
renameFilter(newName, oldName = "")
```

Arguments

newName	(*string*) new name.
oldName	(*string*) [optional] current name of the filtered data set to rename (current one by default)

See Also

createFilter editFilter

Examples

```
## Not run:
renameFilter("newFilter")\cr
renameFilter(oldName = "filter", newName = "newFilter")

## End(Not run)
```

`renameGroup` *[Simulx] Rename simulation group*

Description

Rename a simulation group.

Usage

```
renameGroup(currentGroupName, newGroupName)
```

Arguments

`currentGroupName`	(*string*) Name of the current group name.
`newGroupName`	(*string*) Name of the new group name.

Details

Note: At the creation of a Simulx project, a first group is created by default with the name "simulationGroup1". It is possible to rename this group.

See Also

`addGroup` `getGroups`

Examples

```
 initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "3.definition", "3.1.treatments", "treatment_manual.smlx")
 loadProject(project_name)
 renameGroup("simulationGroup1", "two_doses")
```

resetPlotPreferences *Reset plot preferences to go back to default preferences*

Description

Reset plot preferences to go back to default preferences

Usage

```
resetPlotPreferences()
```

See Also

getPlotPreferences setPlotPreferences

Examples

```
## Not run:
  getPlotPreferences()$obs[c("color", "legend")]
  update = list(obs = list(color = "green", legend = "Observation"))
  setPlotPreferences(update = update)
  getPlotPreferences()$obs[c("color", "legend")]
  resetPlotPreferences()
  getPlotPreferences()$obs[c("color", "legend")]

## End(Not run)
```

runAssessment *[Monolix] Run assessment*

Description

Run assessment. To change the initialization before a run, use getAssessmentSettings to receive all the settings. See example.

Usage

```
runAssessment(settings = NULL)
```

Arguments

settings	(*list<settings>*) [optional] Settings to initialize the assessment algorithm. If not provided, current settings are used. See getAssessmentSettings.

See Also

getAssessmentSettings getAssessmentResults

Examples

```
## Not run:
runAssessment()
set = getAssessmentSettings()
runAssessment(set)

## End(Not run)
```

`runBioequivalenceEstimation` *[PKanalix] Estimate the bioequivalence.*

Description

Estimate the bioequivalence for the selected parameters.

Usage

```
runBioequivalenceEstimation()
```

Examples

```
## Not run:
runNCAEstimation()

## End(Not run)
```

`runCAEstimation` *[PKanalix] Estimate the individual parameters using compartmental analysis.*

Description

Estimate the CA parameters for each individual of the project.

Usage

```
runCAEstimation()
```

Examples

```
## Not run:
runCAEstimation()

## End(Not run)
```

runConditionalDistributionSampling

[Monolix] Sampling from the conditional distribution

Description

Estimate the individual parameters using conditional distribution sampling algorithm. The associated method keyword is "conditionalMean".

Usage

```
runConditionalDistributionSampling()
```

Examples

```
## Not run:
runConditionalDistributionSampling()

## End(Not run)
```

runConditionalModeEstimation

[Monolix] Estimation of the conditional modes (EBEs)

Description

Estimate the individual parameters using the conditional mode estimation algorithm (EBEs). The associated method keyword is "conditionalMode".

Usage

```
runConditionalModeEstimation()
```

Examples

```
## Not run:
runConditionalModeEstimation()

## End(Not run)
```

runEndpoints *[Simulx] Run endpoints task*

Description

Run the endpoints task.

Usage

```
runEndpoints()
```

See Also

getEndpointsResults

Examples

```
## Not run:
  runEndpoints()

## End(Not run)
```

runEstimation *[PKanalix] Run both non compartmental and compartmental analysis.*

Description

Run the NCA analysis and the CA analysis if the structural model for the CA calculation is defined.

Usage

```
runEstimation()
```

Examples

```
## Not run:
runEstimation()

## End(Not run)
```

runLogLikelihoodEstimation *[Monolix] Log-Likelihood estimation*

Description

Run the log-Likelihood estimation algorithm. By default, this task is not processed in the background of the R session. Existing methods:

Method	*Identifier*
Log-Likelihood estimation by linearization	linearization = T
Log-Likelihood estimation by Importance Sampling (default)	linearization = F

The Log-likelihood outputs(-2LL, AIC, BIC) are available using `getEstimatedLogLikelihood` function

Usage

```
runLogLikelihoodEstimation(linearization = FALSE)
```

Arguments

`linearization`	option (*boolean*)[optional] method to be used. When no method is given, the importance sampling is used by default.

Examples

```
## Not run:
runLogLikelihoodEstimation(linearization = T)

## End(Not run)
```

runModelBuilding *[Monolix] Run model building*

Description

Run model building. To change the initialization before a run, use `getModelBuildingSettings` to receive all the settings. See example.

Usage

```
runModelBuilding(...)
```

Arguments

`...`	(*list<settings>*) Settings to initialize the model buildign algorithm. See `getModelBuildingSettings`

See Also

getModelBuildingSettings getModelBuildingResults

Examples

```
## Not run:
runModelBuilding()
set = getModelBuildingSettings()
runModelBuilding(settings = set)

## End(Not run)
```

---

runNCAEstimation *[PKanalix] Estimate the individual parameters using non compartmental analysis.*

Description

Estimate the NCA parameters for each individual of the project.

Usage

```
runNCAEstimation()
```

Examples

```
## Not run:
runNCAEstimation()

## End(Not run)
```

---

runPopulationParameterEstimation *[Monolix] Population parameter estimation*

Description

Estimate the population parameters with the SAEM method. The associated method keyword is "saem". The initial values of the population parameters can be accessed by calling getPopulationParameterInformati and customized with setPopulationParameterInformation.
The estimated population parameters are available using getEstimatedPopulationParameters function.

Usage

```
runPopulationParameterEstimation()
```

Examples

```
## Not run:
runPopulationParameterEstimation()

## End(Not run)
```

runScenario *[Monolix - PKanalix - Simulx] Run scenario*

Description

Run the scenario that has been set with setScenario.

Usage

```
runScenario()
```

Details

A scenario is a list of tasks to be run. Setting the scenario is equivalent to selecting tasks in Monolix, PKanalix or Simulx GUI that will be performed when clicking on RUN.

Note: every task can also be run separately with a specific function, such as runSimulation in Simulx, runEstimation in Monolix. The CA task in PKanalix cannot be part of a scenario, it must be run with runCAEstimation.

See Also

setScenario getScenario

Examples

```
## Not run:
runScenario()

## End(Not run)
```

runSimulation *[Simulx] Run simulation*

Description

Run the simulation task.

Usage

```
runSimulation()
```

Details

As when clicking on SIMULATION in Simulx GUI, simulated values are saved as text files in a result folder which is located next to the project file when calling saveProject.

To get the sampled parameters and simulated outputs as data frames, use getSimulationResults.

To post-process the results by computing outcomes and endpoints, use runEndpoints.

To run both the simulation and endpoint tasks, use setScenario and runScenario.

To get some output in the console showing the status of the simulation, use setConsoleMode.

See Also

getSimulationResults

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "1.overview", "importFromMonolix_clinicalTrial.smlx")
loadProject(project_name)
setConsoleMode("basic")
runSimulation()
```

---

runStandardErrorEstimation

[Monolix] Standard error estimation

Description

Estimate the Fisher Information Matrix and the standard errors of the population parameters. By default, this task is not processed in the background of the R session. Existing methods:

Method	*Identifier*
Estimate the FIM by Stochastic Approximation	linearization = F (default)
Estimate the FIM by Linearization	linearization = T

The Fisher Information Matrix is available using getCorrelationOfEstimates function, while the standard errors are avalaible using getEstimatedStandardErrors function.

Usage

```
runStandardErrorEstimation(linearization = FALSE)
```

Arguments

linearization	option (*boolean*)[optional] method to be used. When no method is given, the stochastic approximation is used by default.

Examples

```
## Not run:
runStandardErrorEstimation(linearization = T)

## End(Not run)
```

saveProject *[Monolix - PKanalix - Simulx] Save current project*

Description

Save the current project as a file that can be reloaded in the connectors or in the GUI.

Usage

```
saveProject(projectFile = "")
```

Arguments

projectFile	[optional](*character*) Path where to save a copy of the current mlxtran model. Can be absolute or relative to the current working directory. If no path is given, the file used to build the current configuration is updated.

Details

The extensions are .mlxtran for Monolix, .pkx for PKanalix, and .smlx for Simulx.
WARNING: R is sensitive between '\' and '/', only '/' can be used.

If the project setting "userfilesnexttoproject" is set to TRUE with setProjectSettings, all file dependencies such as model, data or external files are saved next to the project for Monolix and PKanalix, and in the result folder for Simulx.

See Also

newProject loadProject

Examples

```
## Not run:
[PKanalix only]
saveProject("/path/to/project/file.pkx") # save a copy of the model
[Monolix only]
saveProject("/path/to/project/file.mlxtran") # save a copy of the model
[Simulx only]
saveProject("/path/to/project/file.smlx") # save a copy of the model
[Monolix - PKanalix - Simulx]
saveProject() # update current model

## End(Not run)
# Load, change and save a PKanalix project under a new name
initializeLixoftConnectors("pkanalix")
project_name <- file.path(getDemoPath(), "1.basic_examples", "project_censoring.pkx")
loadProject(project_name)
setNCASettings(blqMethodAfterTmax = "missing")
saveProject("~/changed_project.pkx")

# Load, change and save a Monolix project under a new name
initializeLixoftConnectors("monolix")
project_name <- file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "warfarin
loadProject(project_name)
```

```
addContinuousTransformedCovariate(tWt = "3*exp(wt)")
saveProject("~/changed_project.mlxtran")

# Load, change and save a Simulx project under a new name
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "2.models", "longitudinal.smlx")
loadProject(project_name)
defineTreatmentElement(name = "trt", element = list(data = data.frame(time = 0, amount = 100)))
saveProject("~/changed_project.smlx")
```

`selectData` *[Monolix - PKanalix] Select data set*

Description

Select the new current data set within the previously defined ones (original and filters).

Usage

```
selectData(name)
```

Arguments

`name`	(*string*) data set name.

See Also

`getAvailableData`

Examples

```
## Not run:
selectData(name = "filter1")

## End(Not run)
```

`setAddLines` *[Simulx] Add lines to the model*

Description

Additional equations can be added to the model file as in Simulx GUI. It is useful in case of import from Monolix or PKanalix, in order to add equations to the model, eg to compute an additional variable, without modifying the model file used for estimation.

Usage

```
setAddLines(lines)
```

Arguments

lines	(*string*) Additional lines to define.

Details

All variables defined in the add lines will be available as an output.

See Also

getAddLines

Examples

```
# Calculate AUC
initializeLixoftConnectors("monolix")
monolix_project <- file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "theo
initializeLixoftConnectors("simulx")
importProject(monolix_project)
setAddLines("ddt_AUC = Cc")

# Calculate partial AUC
initializeLixoftConnectors("monolix")
monolix_project <- file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "warf
initializeLixoftConnectors("simulx")
importProject(monolix_project)
setAddLines(c("if t<24", "deriv = Cc", "else", "deriv = 0", "end", "ddt_AUC = deriv"))
```

---

setAutocorrelation *[Monolix] Set auto-correlation*

Description

Add or remove auto-correlation from the error model used on some of the observation models.
Call getObservationInformation to get a list of the observation models present in the current project.

Usage

```
setAutocorrelation(...)
```

Arguments

...	Sequence of comma-separated pairs {(*string*)"observationModel",(*boolean*)hasAutoCorrelation}.

See Also

getContinuousObservationModel

Examples

```
## Not run:
setAutocorrelation(Conc = TRUE)
setAutocorrelation(Conc = TRUE, Effect = FALSE)

## End(Not run)
```

`setBioequivalenceSettings`

[PKanalix] Set the value of one or several of the settings associated to the bioequivalence estimation

---

Description

Set the value of one or several of the settings associated to the bioequivalence estimation. Associated settings are:

"level"	(*int*)
"bioequivalenceLimits"	(*vector*)
"computedBioequivalenceParameters"	(*data.frame*) Parameters to consider for the bioequivalence analysis and if they sh
"linearModelFactors"	(*list*)
"degreesFreedom"	(*string*)

Usage

```
setBioequivalenceSettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

`getBioequivalenceSettings`

Examples

```
## Not run:
setBioequivalenceSettings(level = 90, bioequivalencelimits = c(85, 115)) # set the settings whose name has been
setBioequivalenceSettings(computedbioequivalenceparameters = data.frame(parameters = c("Cmax", "Tmax"), logtr
setBioequivalenceSettings(linearmodelfactors = list(id="SUBJ", period="OCC", formulation="FORM", reference="

## End(Not run)
```

setCAInitialValues *[PKanalix] Set the initial values of individual parameters for the compartmental analysis*

Description

[PKanalix] Set the initial values of individual parameters for the compartmental analysis

Usage

```
setCAInitialValues(initialValues)
```

Arguments

`initialValues` a list of lists. For each parameter, a list specifies:

"value"	(*double*)	Initial value to use. Must be in the limits in case of bounded constraint.
"constraint"	(*string*)	Possible values are "none", "positive" or "bounded".
"limits"	(*vector of doubles*)	[optional] Limits in case of bounded constraint.

See Also

`getCAInitialValues`

Examples

```
## Not run:
 setCAInitialValues(list(Cl=list(value=0.4, constraint = "none"), V=list(value=0.5, constraint="positive"),

## End(Not run)
```

setCAResultsStratification *[PKanalix] Set CA results stratification*

Description

Set the stratification used to compute NCA parameters stratistics table. Stratification is defined by:

- stratification covariate groups which are shared by both NCA and CA results
- a stratification state which is specific to each task results

Usage

```
setCAResultsStratification(
  split = NULL,
  filter = NULL,
  groups = NULL,
  state = NULL
)
```

Arguments

split	(*vector<string>*) Ordered list of splitted covariates
filter	(*list< pair<string, vector<int» >*) List of paired containing a covariate name and the indexes of associated kept groups
groups	Stratification groups list
state	Stratification state

Details

For each covariate, stratification groups can be defined as a list with:

name	*string*	covariate name
definition	*vector<double>*(continuous) || list<vector<string»(categorical)	group separations (continuous) || modality s

A stratification state is represented as a list with:

split	*vector<string>*	ordered list of splitted covariates
filter	*list< pair<string, vector<int» >*	list of paired containing a covariate name and the indexes of associated kept gr

See Also

getCAResultsStratification

Examples

```
## Not run:
setCAResultsStratification(split = "SEX")
setCAResultsStratification(split = c("SEX", "WEIGHT"))

setCAResultsStratification(filter = list("SEX", 1))
setCAResultsStratification(filter = list(list("SEX", 1), list("WEIGHT", c(1,3))))

setCAResultsStratification(split = "WEIGHT", filter = list(list("TRT", c(1,2))),
groups = list(list(name = "WEIGHT", definition = c(65,5, 72)), list(name = "TRT", definition = list(c("a","b"),

s = getCAResultsStratification()
setCAResultsStratification(state = s$state, groups = s$groups)

## End(Not run)
```

---

setCASettings *[PKanalix] Set the settings associated to the compartmental analysis*

Description

Set the settings associated to the compartmental analysis. Associated settings names are:

"weightingCA"	(*string*)	Type of weighting objective function. Possible methods are "uniform", "Yobs", "Ypred", "Yp
"pool"	(*logical*)	If FALSE, fit with individual parameters or with the same parameters for all individuals if TR
"blqMethod"	(*string*)	Method by which the BLQ data should be replaced. Possible methods are "zero", "LOQ", "L

Usage

```
setCASettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

getCASettings

Examples

```
## Not run:
setCASettings(weightingCA = "uniform", blqMethod = "zero") # set the settings whose name has been passed in argu

## End(Not run)
```

`setConditionalDistributionSamplingSettings`
[Monolix] Set conditional distribution sampling settings

Description

Set the value of one or several of the conditional distribution sampling settings. Associated settings are:

"ratio"	(*0< double <1*)	Width of the confidence interval.
"enableMaxIterations"	(*bool*)	Enable maximum of iterations.
"nbMinIterations"	(*int >=1*)	Minimum number of iterations.
"nbMaxIterations"	(*int >=1*)	Maximum number of iterations.
"nbSimulatedParameters"	(*int >=1*)	Number of replicates.

Usage

```
setConditionalDistributionSamplingSettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

getConditionalDistributionSamplingSettings

Examples

```
## Not run:
setConditionalDistributionSamplingSettings(ratio = 0.05, nbMinIterations = 50)

## End(Not run)
```

`setConditionalModeEstimationSettings`
[Monolix] Set conditional mode estimation settings

Description

Set the value of one or several of the conditional mode estimation settings. Associated settings are:

"nbOptimizationIterationsMode"	(*int >=1*)	Maximum number of iterations.
"optimizationToleranceMode"	(*double >0*)	Optimization tolerance.

Usage

```
setConditionalModeEstimationSettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

getConditionalModeEstimationSettings

Examples

```
## Not run:
setConditionalModeEstimationSettings(nbOptimizationIterationsMode = 20,
                                     optimizationToleranceMode = 0.1)

## End(Not run)
```

setConsoleMode *[Monolix - Pkanalix - Simulx] Set console mode*

Description

Set console mode to choose the volume of output in the console after running estimation tasks (verbosity level):

"none"	no output
"basic"	for each algorithm, display current iteration then associated results at algorithm end
"complete"	display all iterations then associated results at algorithm end

Usage

```
setConsoleMode(mode)
```

Arguments

mode	(*string*) Accepted values are: "none" [default], "basic", "complete"

See Also

getConsoleMode

setCorrelationBlocks *[Monolix] Set correlation block structure*

Description

Define the correlation block structure associated to some of the variability levels of the current project. Call getVariabilityLevels to get a list of the variability levels and getIndividualParameterModel to get a list of the available individual parameters within the current project.

Usage

```
setCorrelationBlocks(...)
```

Arguments

...	A list of comma-separated pairs {variabilityLevel = vector< (*array*<*string*>)parameterNames} > }.

See Also

getVariabilityLevels getIndividualParameterModel

Examples

```
## Not run:
setCorrelationBlocks(id = list( c("ka","V","Tlag") ), iov1 = list( c("ka","Cl"), c("Tlag","V") ) )

## End(Not run)
```

setCovariateModel *[Monolix] Set covariate model*

Description

Set which are the covariates influencing individual parameters present in the project. Call getIndividualParameterMod
to get a list of the individual parameters present within the current project. and getCovariateInformation
to know which are the available covariates for a given level of variability and a given individual parameter.

Usage

```
setCovariateModel(...)
```

Arguments

...	A list of comma-separated pairs {parameterName = { covariateName = (*bool*)isInfluent, ...} }

See Also

getCovariateInformation

Examples

```
## Not run:
setCovariateModel( ka = c( Wt = FALSE, tWt = TRUE, lcat2 = TRUE),
                   Cl = c( SEX = TRUE )
                   )

## End(Not run)
```

setData *[Monolix - PKanalix] Set project data*

Description

Set project data giving a data file and specifying headers and observations types.

Usage

```
setData(dataFile, headerTypes, observationTypes, nbSSDoses = NULL)
```

Arguments

dataFile	(*character*): Path to the data file. Can be absolute or relative to the current working directory.
headerTypes	(*array<character>*): A collection of header types. The possible header types are: "ignore", "id", "time", "observation", "amount", "contcov", "catcov", "occ", "evid", "mdv", "obsid", "cens", "limit", "regressor","admid", "rate", "tinf", "ss", "ii", "addl", "date". Notice that these are not the types displayed in the interface, these one are shortcuts.
observationTypes	[optional] (*list*): A list giving the type of each observation present in the data file. If there is only one y-type, the corresponding observation name can be omitted. The possible observation types are "continuous", "discrete", and "event".
nbSSDoses	[optional](*int*): Number of doses (if there is a SS column).

See Also

getData

Examples

```
## Not run:
setData(dataFile = "/path/to/data/file.txt",
        headerTypes = c("IGNORE", "OBSERVATION"), observationTypes = "continuous")
setData(dataFile = "/path/to/data/file.txt",
        headerTypes = c("IGNORE", "OBSERVATION", "YTYPE"),
       observationTypes = list(Concentration = "continuous", Level = "discrete"))

## End(Not run)
```

---

setDataSettings *[PKanalix] Set the value of one or several of the data settings associated to the non compartmental analysis*

Description

Set the value of one or several of the data settings associated to the non compartmental analysis. Associated settings names are:

"urinevolume"	(*string*)	regressor name used as urine volume.
"datatype"	(*list*)	list("obsId" = *string*("plasma" or "urine"). The type of data associated with each *obsId*. Defaul
"units"	(*list*)	list with the units associated to "dose" ("pg","ng","ug","mg","g","pmol","nmol","umol","mmol
"scalings"	(*list*)	list with the scaling factor associated to "concentration", "dose", "time" and "urinevolume".
"enableunits"	(*bool*)	are units enabled or not.

Usage

```
setDataSettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

getDataSettings

Examples

```
## Not run:
setDataSettings("datatype" = list("Y" ="plasma")) # set the settings whose name has been passed in argument
setDataSettings("units"=list(dose="ng",time="h",volume="mL", grading=""))
setDataSettings("scalings"=list(dose=0.001, time=24))

## End(Not run)
```

---

setErrorModel *[Monolix] Set error model*

Description

Set the error model type to be used with some of the observation models. Call getObservationInformation to get a list of the observation models present in the current project.

Usage

```
setErrorModel(...)
```

Arguments

...	A list of comma-separated pairs {observationModel = (*string*)errorModelType}.

Details

Available error model types are :

"constant"	obs = pred + a*err
"proportional"	obs = pred + (b*pred)*err
"combined1"	obs = pred + (b*pred^c + a)*err
"combined2"	obs = pred + sqrt(a^2 + (b^2)*pred^(2c))*err

Error model parameters will be initialized to 1 by default. Call setPopulationParameterInformation to modify their initial value.
The value of the exponent parameter is fixed by default when using the "combined1" and "combined2" models.
Use setPopulationParameterInformation to enable its estimation.

See Also

getContinuousObservationModel setPopulationParameterInformation

Examples

```
## Not run:
setErrorModel(Conc = "constant", Effect = "combined1")

## End(Not run)
```

`setGeneralSettings` *[Monolix] Set common settings for algorithms*

Description

Set the value of one or several of the common settings for Monolix algorithms. Associated settings are:

"autoChains"	(*bool*)	Automatically adjusted the number of chains to have at least a minimum number of sul
"nbChains"	(*int >0*)	Number of chains to be used if "autoChains" is set to FALSE.
"minIndivForChains"	(*int >0*)	Minimum number of individuals by chain.

Usage

```
setGeneralSettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

getGeneralSettings

Examples

```
## Not run:
setGeneralSettings(autoChains = FALSE, nbchains = 10)

## End(Not run)
```

`setGroupComparisonSettings`
[Simulx] Set group comparison settings

Description

Set settings related to the comparison of endpoints across groups.

Usage

```
setGroupComparisonSettings(referenceGroup = NULL, enable = TRUE)
```

Arguments

referenceGroup	(*string*) (optional) Group to use as reference.
enable	(*bool*) (optional) Enable group comparison, TRUE by default.

Details

Endpoints summarize the outcome values over all individuals, for each simulation group and each replicate. Endpoints are defined with defineEndpoint and can be compared across groups as in Simulx GUI.

setGroupComparisonSettings enables to specify if endpoints should be compared across simulation groups and which group to use as a reference. Group comparison is performed during the Endpoints task.

See Also

getGroupComparisonSettings

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "6.outcome_endpoints", "6.2.group_comparison", "groupComp_PKPD_medi
loadProject(project_name)
# Change the reference group
setGroupComparisonSettings(referenceGroup = "OD_300mgPerDay_", enable = TRUE)
# Turn off group comparison
setGroupComparisonSettings(enable = FALSE)
```

---

setGroupElement *[Simulx] Set elements to a simulation group*

Description

Set the new element of a specific group. If an element of the same type is already set, setGroupElement will replace it. For treatments and outputs, it is possible to set several elements at the same time by using a vector.

Usage

```
setGroupElement(group, elements)
```

Arguments

group	(*character*) Group name (when creating a new Simulx project, the default group name is "simulationGroup1").
elements	(*character*) Vector of elements that are already defined

Details

Simulation groups are used for simulation as in Simulx GUI. The same rules apply as in the GUI to set group elements. For example, a covariate element can be set only if the parameter element is a population parameter.

At the creation of a Simulx project, a first group is created by default with the name "simulationGroup1". Use getGroups to check which groups have already been defined, and which elements are set in each group. To add a simulation group, use addGroup. To remove a simulation group, use removeGroup. To add or change a group element, use setGroupElement. To remove an element from a group, use removeGroupElement. To define new elements, use one of the define...Element functions.

See Also

`getGroups`

Examples

```
initializeLixoftConnectors("monolix")
monolix_project <- file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "warf
initializeLixoftConnectors("simulx")
importProject(monolix_project)
setGroupElement(group = "simulationGroup1", elements = c("mlx_EBEs", "mlx_Cc"))
```

---

setGroupRemaining *[Simulx] Set remaining parameters for a simulation group*

Description

Set the values of the remaining parameters (typically the error model parameters) for a group.

Usage

```
setGroupRemaining(group, remaining)
```

Arguments

`group`	(*character*) Group name
`remaining`	(*vector*) list of the remaining variables

Details

Remaining parameters are all parameters that appear in the structural model (in the input line of [LONGITUDINAL]) and are neither individual parameters not regressors. They are typically error model parameters.

If an individual parameters element is selected for simulation, and the model includes remaining parameters, it is possible to set their values with setGroupRemaining.

It typically enables to make a simulation with measurement noise, with an individual element. These error model parameters will impact the simulation only if a noisy observation (from the DEFINITION section of the [LONGITUDINAL] block) is set as output element (instead of a smooth prediction in OUTPUT or variable in EQUATION).

If a population parameters element is selected, it is not possible to set remaining parameters because these parameters are already part of the population element.

See Also

getGroupRemaining

Examples

```
initializeLixoftConnectors("monolix")
monolix_project <- file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "warf
initializeLixoftConnectors("simulx")
importProject(monolix_project)
setGroupElement(group = "simulationGroup1", elements = "mlx_EBEs")
setGroupRemaining(group = "simulationGroup1", remaining = list(a = 0.2, b = 0.05))
```

---

setGroupSize *[Simulx] Set simulation group size*

Description

Define the size of a simulation group.

Usage

```
setGroupSize(group, size)
```

Arguments

group	(*string*) Name of the group where the size will be changed.
size	(*int*) Size of the new group.

Details

Group size is the number of individuals (ie sets of individual parameters and covariate values) that will be sampled by Simulx for a given group. It should not be mixed up with the number of replicates that can be set with setNbReplicates.

To get the size of a group, please use getGroups.

All options of the Simulx scenario are the same as in Simulx GUI. Check the online doc of Simulx to get more guidance on how to use them.

See Also

getGroups

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "1.overview", "importFromMonolix_clinicalTrial.smlx")
loadProject(project_name)
setGroupSize("gr_BID_N30", 40)
setGroupSize("gr_OD_N30", 40)
```

setIndividualLogitLimits

[Monolix] Set individual parameter distribution limits

Description

Set the minimum and the maximum values between the individual parameter can be used. Used only if the distribution of the parameter is "logitNormal", else wise it will not be taken into account

Usage

```
setIndividualLogitLimits(...)
```

Arguments

...	A list of comma-separated pairs {individualParameter = [(double)min,(double)max] }

See Also

getIndividualParameterModel

Examples

```
## Not run:
setIndividualLogitLimits( V = c(0, 1), ka = c(-1, 2) )

## End(Not run)
```

setIndividualParameterDistribution

[Monolix] Set individual parameter distribution

Description

Set the distribution of the estimated parameters. Available distributions are "normal", "logNormal" and "logitNormal".
Call getIndividualParameterModel to get a list of the available individual parameters within the current project.

Usage

```
setIndividualParameterDistribution(...)
```

Arguments

...	A list of comma-separated pairs {parameterName = (*string*)"distribution"}.

See Also

getIndividualParameterModel

Examples

```
## Not run:
setIndividualParameterDistribution(V = "logNormal")
setIndividualParameterDistribution(Cl = "normal", V = "logNormal")

## End(Not run)
```

---

setIndividualParameterModel

[Monolix] Set individual parameter model

Description

Set the information concerning the individual parameter model. The editable informations are:

- distribution: (*string*) distribution of the parameter values. The distribution type can be "normal", "logNormal", or "logitNormal".
- limits: a list giving the distribution limits for each parameter following a "logitNormal" distribution
- variability: a list giving, for each variability level, if individual parameters have variability or not
- covariateModel: a list giving, for each individual parameter, if the related covariates are used or not.
- correlationBlocks : a list giving, for each variability level, the blocks of the correlation matrix of the random effects. A block is represented by a vector of individual parameter names.

Usage

```
setIndividualParameterModel(...)
```

Arguments

...	A list of comma-separated pairs {[info] = [value]}.

---

setIndividualParameterVariability

[Monolix] Individual variability management

Description

Add or remove inter-individual and/or intra-individual variability from some of the individual parameters present in the project.
Call `getIndividualParameterModel` to get a list of the available parameters within the current project.

Usage

```
setIndividualParameterVariability(...)
```

Arguments

...	A list of comma-separated pairs {variabilityLevel = {individualParameterName = (*bool*)hasVariability} }.

See Also

getIndividualParameterModel

Examples

```
## Not run:
setIndividualParameterVariability(ka = TRUE, V = FALSE)
setIndividualParameterVariability(id = list(ka = TRUE), iov1 = list(ka = FALSE))

## End(Not run)
```

---

setInitialEstimatesToLastEstimates

[Monolix] Initialize population parameters with the last estimated ones

---

Description

Set the initial value of all the population parameters present within the current project to the ones previously estimated. These the values will be used in the population parameter estimation algorithm during the next scenario run.
WARNING: If there is any set after a run, it will not be possible to set the initial values as the structure of the project has changed since last results.

Usage

```
setInitialEstimatesToLastEstimates(fixedEffectsOnly = FALSE)
```

Arguments

fixedEffectsOnly	(*bool*) If this boolean is set to TRUE, only the fixed effects are initialized to their last estimated values. Otherwise, individual variances and error model parameters are re-initialized too. Equals FALSE by default.

See Also

getEstimatedPopulationParameters getPopulationParameterInformation

Examples

```
## Not run:
setInitialEstimatesToLastEstimates() # fixedEffectsOnly = FALSE by default
setInitialEstimatesToLastEstimates(TRUE)

## End(Not run)
```

setLogLikelihoodEstimationSettings *[Monolix] Set loglikelihood estimation settings*

Description

Set the value of the loglikelihood estimation settings. Associated settings are:

"nbFixedIterations"	(*int >0*)	Monte Carlo size for the loglikelihood evaluation.
"samplingMethod"	(*string*)	Should the loglikelihood estimation use a given number of freedom d
"nbFreedomDegrees"	(*int >0*)	Degree of freedom of the Student t-distribution. *Used only if "samplin*
"freedomDegreesSampling"	(*vector<int(>0)>*)	Sequence of freedom degrees to be tested. *Used only if "samplingMet*

Usage

```
setLogLikelihoodEstimationSettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

getLogLikelihoodEstimationSettings

Examples

```
## Not run:
setLogLikelihoodEstimationSettings(nbFixedIterations = 20000)

## End(Not run)
```

setMapping *[Monolix - PKanalix] Set mapping*

Description

Set mapping between data and model.

Usage

```
setMapping(mapping)
```

Arguments

mapping	(*list<list>*) A list of lists representing a link between the data and the model. Each list contains: • data (*string*) Data name • prediction (*string*) Prediction name • model [Monolix] (*string*) Model observation name (for continuous observations only)

See Also

getMapping

Examples

```
## Not run:
[Monolix] setMapping(list(list(data = "1", prediction = "Cc", model = "concentration"), list(data = "2", predic
[PKanalix] setMapping(list(list(data = "1", prediction = "Cc"), list(data = "2", prediction = "Level")))

## End(Not run)
```

---

setMCMCSettings *[Monolix] Set settings associated to the MCMC algorithm*

Description

Set the value of one or several of the MCMC algorithm specific settings of the current project. Associated settings are:

"strategy"	(*vector<int>[3]*)	Number of calls for each one of the three MCMC kernels.
"acceptanceRatio"	(*double*)	Target acceptance ratio.

Usage

```
setMCMCSettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

getMCMCSettings

Examples

```
## Not run:
setMCMCSettings(strategy = c(2,1,2))

## End(Not run)
```

setNbReplicates *[Simulx] Set number of replicates*

Description

Define the number of replicates of the simulation.

Usage

```
setNbReplicates(nb)
```

Arguments

nb	(*int*) Number of replicates.

Details

The number of replicates is the number of times that Simulx will simulate a given study. It should not be mixed up with the group size defined by setGroupSize.

To simulate one study, Simulx samples a number of individuals for each group, defined by setGroupSize (let's say NidsPerGroup). To simulate replicate studies, it will sample for each replicate NidsPerGroup other individuals for each group (it is like changing the seed).

If the parameter element is an individual element or a population parameter defined with a vector, replicates will always sample individuals using the same population parameters. In this case, they are useful to check the effect of changing the seed, to get for example the uncertainty of an endpoint due to limited sampling.

If the parameter element is a population element defined with a table containing several lines, or an imported element such as mlx_PopUncertainSA or mlx_TypicalUncertainSA, each replicate will use a different population parameter to simulate the study. In this case, it is possible to see the effect of changing the population parameters on the prediction (in addition to uncertainty due to limited sampling).

All options of the Simulx scenario are the same as in Simulx GUI. Check the online doc of Simulx to get more guidance on how to use them.

See Also

getNbReplicates

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "replicates.smlx")
loadProject(project_name)
setNbReplicates(nb = 20)
```

`setNCAResultsStratification` *[PKanalix] Set NCA results stratification*

Description

Set the stratification used to compute NCA parameters stratistics table. Stratification is defined by:

- stratification covariate groups which are shared by both NCA and CA results
- a stratification state which is specific to each task results

Usage

```
setNCAResultsStratification(
  split = NULL,
  filter = NULL,
  groups = NULL,
  state = NULL
)
```

Arguments

`split`	(*vector<string>*) Ordered list of splitted covariates
`filter`	(*list< pair<string, vector<int» >*) List of paired containing a covariate name and the indexes of associated kept groups
`groups`	Stratification groups list
`state`	Stratification state

Details

For each covariate, stratification groups can be defined as a list with:

name	*string*	covariate name
definition	*vector<double>*(continuous) || list<vector<string»(categorical)	group separations (continuous) || modality s

A stratification state is represented as a list with:

split	*vector<string>*	ordered list of splitted covariates
filter	*list< pair<string, vector<int» >*	list of paired containing a covariate name and the indexes of associated kept gr

Note: For acceptance criteria filtering, it is possible to give only the criterion name instead of a pair.

See Also

`getNCAResultsStratification`

Examples

```
## Not run:
setNCAResultsStratification(split = "SEX")
setNCAResultsStratification(split = c("SEX", "WEIGHT"))

setNCAResultsStratification(filter = "Span")
setNCAResultsStratification(filter = list("Span", list("SEX", 1)))

setNCAResultsStratification(split = "WEIGHT", filter = list(list("TRT", c(1,2))),
groups = list(list(name = "WEIGHT", definition = c(65,5, 72)), list(name = "TRT", definition = list(c("a","b"),

s = getNCAResultsStratification()
setNCAResultsStratification(state = s$state, groups = s$groups)

## End(Not run)
```

`setNCASettings` *[PKanalix] Set the value of one or several of the settings associated to the non compartmental analysis*

Description

Set the value of one or several of the settings associated to the non compartmental analysis. Associated settings are:

"obsidtouse"	(*string*)	The observation id from data section to use for computations.
"administrationType"	(*list*)	list(key = "admId", value = *string*("intravenous" or "extravascular")). *admId*
"integralMethod"	(*string*)	Method for AUC and AUMC calculation and interpolation. Possible method
"partialAucTime"	(*list*)	The first element of the list is a bolean describing if this setting is used. The
"blqMethodBeforeTmax"	(*string*)	Method by which the BLQ data before Tmax should be replaced. Possible m
"blqMethodAfterTmax"	(*string*)	Method by which the BLQ data after Tmax should be replaced. Possible me
"ajdr2AcceptanceCriteria"	(*list*)	The first element of the list is a boolean describing if this setting is used. Th
"extrapAucAcceptanceCriteria"	(*list*)	The first element of the list is a boolean describing if this setting is used. Th
"spanAcceptanceCriteria"	(*list*)	The first element of the list is a boolean describing if this setting is used. Th
"lambdaRule"	(*string*)	Main rule for the lambda_Z estimation. Possible rules are "R2", "interval",
"timeInterval"	(*vector*)	Time interval for the lambda_Z estimation when "*lambdaRule*" = "interval"
"timeValuesPerId"	(*list*)	list("idName" = idTimes,...): *idTimes* Observation times to use for the calcu
"nbPoints"	(*integer*)	Number of points for the lambda_Z estimation when "*lambdaRule*" = "poin
"maxNbOfPoints"	(*list*)	The first element of the list is a boolean describing if this setting is used. Th

"startTimeNotBefore"	(*list*)	The first element of the list is a boolean describing if this setting is used. Th
"weightingNca"	(*string*)	Weighting method used for the regression that estimates lambda_Z. Possibl
"computedNCAParameters"	(*vector*)	All the parameters to compute during the analysis. Possible parameters:
		parameters related to the calculation of lambda_z: "Rsq", "Rsq_adjusted", "C

Usage

- `setNCASettings(...)`

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

`getNCASettings`

Examples

```
## Not run:
setNCASettings(integralMethod = "LinLogTrapLinLogInterp", weightingnca = "uniform") # set the settings whose n
setNCASettings(administrationType = list("1"="extravascular")) # set the administration id "1" to extravascula
setNCASettings(startTimeNotBefore = list(TRUE, 15)) # set the estimation of the lambda_z with points with time c
setNCASettings(timeValuesPerId = list('1'=c(4, 6, 8, 30), '4'=c(8, 12, 18, 24, 30))) # set the points to use for
setNCASettings(timeValuesPerId = NULL) # set the points to use for the lambda_z to the default rule

## End(Not run)
```

---

setObservationDistribution

[Monolix] Set observation model distribution

Description

Set the distribution in the Gaussian space of some of the observation models. Available distribution types are "normal", "logNormal", or "logitNormal". Call `getObservationInformation` to get a list of the available observation models within the current project.

Usage

```
setObservationDistribution(...)
```

Arguments

...	A list of comma-separated pairs {observationModel = (*string*)"distribution"}.

See Also

`getContinuousObservationModel`

Examples

```
## Not run:
setObservationDistribution(Conc = "normal")
setObservationDistribution(Conc = "normal", Effect = "logNormal")

## End(Not run)
```

---

setObservationLimits *[Monolix] Set observation model distribution limits*

Description

Set the minimum and the maximum values between which some of the observations can be found. Used only if the distribution of the error model is "logitNormal", else wise it will not be taken into account

Usage

```
setObservationLimits(...)
```

Arguments

...	A list of comma-separated pairs {observationModel = [(double)min,(double)max] }

See Also

getContinuousObservationModel getObservationInformation

Examples

```
## Not run:
setObservationLimits( Conc = c(-Inf,Inf), Effect = c(0,Inf) )

## End(Not run)
```

---

setPlotPreferences *Set preferences to customize plots When preferences are Set, the updated preferences will used in all the plots*

Description

Set preferences to customize plots When preferences are Set, the updated preferences will used in all the plots

Usage

```
setPlotPreferences(update = NULL)
```

Arguments

update	list containing the plot elements to be updated.

Details

This function creates a theme that customizes how a plot looks, i.e. legend, colors fills, transparencies, linetypes an sizes, etc. For each curve, list of available customizations:

- color: color (when lines or points)
- fill: color (when surfaces)
- opacity: color transparency
- radius: size of points
- shape: shape of points
- lineType: linetype
- lineWidth: line size
- legend: name of the legend (if NULL, no legend is displayed for the element)

See Also

getPlotPreferences resetPlotPreferences

Examples

```
## Not run:
  getPlotPreferences()$obs[c("color", "legend")]
  update = list(obs = list(color = "green", legend = "Observation"))
  setPlotPreferences(update = update)
  getPlotPreferences()$obs[c("color", "legend")]

## End(Not run)
```

---

setPopulationParameterEstimationSettings

[Monolix] Set population parameter estimation settings

Description

Set the value of one or several of the population parameter estimation settings. Associated settings are:

"nbBurningIterations"	(*int >=0*)	Number of iterations in the burn-in phase.
"nbExploratoryIterations"	(*int >=0*)	If "exploratoryAutoStop" is set to FALSE, it is the number of iteration
"exploratoryAutoStop"	(*bool*)	Should the exploratory step automatically stop.
"exploratoryInterval"	(*int >0*)	Minimum number of interation in the exploratory phase. *Used only if*
"exploratoryAlpha"	(*0<= double <=1*)	Convergence memory in the exploratory phase. *Used only if "explorat*
"nbSmoothingIterations"	(*int >=0*)	If "smoothingAutoStop" is set to FALSE, it is the number of iterations
"smoothingAutoStop"	(*bool*)	Should the smoothing step automatically stop.
"smoothingInterval"	(*int >0*)	Minimum number of interation in the smoothing phase. *Used only if "*
"smoothingAlpha"	(*0.5< double <=1*)	Convergence memory in the smoothing phase. *Used only if "smoothin*

"smoothingRatio"	(*0< double <1*)	Width of the confidence interval. *Used only if "smoothingAutoStop" is*
"simulatedAnnealing"	(*bool*)	Should annealing be simulated.
"tauOmega"	(*double >0*)	Proportional rate on variance. *Used only if "simulatedAnnealing" is T*
"tauErrorModel"	(*double >0*)	Proportional rate on error model. *Used only if "simulatedAnnealing" i*
"variability"	(*string*)	Estimation method for parameters without variability: "firstStage" | "c
"nbOptimizationIterations"	(*int >=1*)	Number of optimization iterations.
"optimizationTolerance"	(*double >0*)	Tolerance for optimization.

Usage

```
setPopulationParameterEstimationSettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = SettingValue}.

See Also

getPopulationParameterEstimationSettings

Examples

```
## Not run:
setPopulationParameterEstimationSettings(exploratoryAutoStop = TRUE, tauOmega = 0.95)

## End(Not run)
```

---

setPopulationParameterInformation

[Monolix] Population parameters initialization and estimation method

---

Description

Set the initial value, the estimation method and, if relevant, the MAP parameters of one or several of the population parameters present within the current project (fixed effects + individual variances + error model parameters). Available methods are:

- "FIXED": Fixed
- "MLE": Maximum Likelihood Estimation
- "MAP": Maximum A Posteriori

Call getPopulationParameterInformation to get a list of the initializable population parameters present within the current project.

Usage

```
setPopulationParameterInformation(...)
```

Arguments

...	A list of comma-separated pairs {paramName = list(initialValue = (*double*), method = (*string*)"method"}. In case of "MAP" method, the user can specify the associated typical value and standard deviation values by using an additional list elements {paramName = list(priorValue = (*double*)1, priorSD = (*double*)2)}. By default, the prior value corresponds to the the population parameter and the prior standard deviation is set to 1.

See Also

getPopulationParameterInformation

Examples

```
## Not run:
setPopulationParameterInformation(Cl_pop = list(initialValue = 0.5, method = "FIXED"),
                                  V_pop  = list(initialValue = 1),
                                ka_pop = list(method = "MAP", priorValue = 1, priorSD = 0.1))

## End(Not run)
```

---

setPreferences *[Monolix - PKanalix - Simulx] Set preferences*

Description

Set the value of one or several of the project preferences. Prefenreces are:

"relativepath"	(*bool*)	Use relative path for save/load operations.
"threads"	(*int >0*)	Number of threads.
"temporarydirectory"	(*string*)	Path to the directory used to save temporary files.
"timestamping"	(*bool*)	Create an archive containing result files after each run.
"delimiter"	(*string*)	Character use as delimiter in exported result files.
"exportchartsdata"	(*bool*)	Should charts data be exported.
"exportchartsdatasets"	(*bool*)	[Monolix] Should charts datasets be exported if possible.
"exportvpcsimulations"	(*bool*)	[Monoliw] Should vpc simulations be exported if possible.
"exportsimulationfiles"	(*bool*)	[Simulx] Should simulation results files be exported.
"headeraliases"	(*list("header" = vector<string>)*)	For each header, the list of the recognized aliases.
"ncaparameters"	(*vector<string>*)	[PKanalix] Defaulty computed NCA parameters.
"units"	(*list("type" = string)*	[PKanalix] Time, amount and/or volume units.

Usage

```
setPreferences(...)
```

Arguments

...	A collection of comma-separated pairs {preferenceName = settingValue}.

See Also

getPreferences

Examples

```
## Not run:
setPreferences(exportCharts = FALSE, delimiter = ",")

## End(Not run)
```

`setProjectSettings` *[Monolix - PKanalix - Simulx] Set project settings*

Description

Set the value of one or several of the settings of the project. Associated settings for Monolix projects are:

"directory"	(*string*)	Path to the folder where simulation results will be saved. It shou
"exportResults"	(*bool*)	Should results be exported.
"seed"	(*0< int <2147483647*)	Seed used by random generators.
"grid"	(*int*)	Number of points for the continuous simulation grid.
"nbSimulations"	(*int*)	Number of simulations.
"dataandmodelnexttoproject"	(*bool*)	Should data and model files be saved next to project.

Associated settings for PKanalix projects are:

"directory"	(*string*)	Path to the folder where simulation results will be saved. It should be a v
"dataNextToProject"	(*bool*)	Should data and model (in case of CA) files be saved next to project.
"seed"	(*0< int <2147483647*)	Seed used by random generators.

Associated settings for Simulx projects are:

"directory"	(*string*)	Path to the folder where simulation results will be saved. It should be
"seed"	(*0< int <2147483647*)	Seed used by random generators.
"userfilesnexttoproject"	(*bool*)	Should user files be saved next to project.

Usage

```
setProjectSettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

`getProjectSettings`

Examples

```
## Not run:
setProjectSettings(directory = "/path/to/export/directory", seed = 12345)

## End(Not run)
```

setResultsStratificationGroups

[Monolix - PKanalix - Simulx] Set results stratification groups

Description

Set the stratification covariate groups used to compute statistics over individual parameters. These groups are shared by all the task results.

Usage

```
setResultsStratificationGroups(groups)
```

Arguments

groups	Stratification groups list

Details

For each covariate, stratification groups can be defined as a list with:

name	*string*	covariate name
definition	*vector*<*double*>(continuous) || list<vector<string»(categorical)	group separations (continuous) || modality s

See Also

@seealso getResultsStratificationGroups

Examples

```
## Not run:
setResultsStratificationGroups(list(list(name = "WEIGHT", definition = c(65,5, 72)), list(name = "TRT", defini

## End(Not run)
```

setSameIndividualsAmongGroups

[Simulx] Set same individuals among groups

Description

Define if the same individuals will be simulated among all groups.

Usage

```
setSameIndividualsAmongGroups(value)
```

Arguments

value	(*boolean*) Boolean to define if the same individuals will be the same for all groups.

Details

setSameIndividualsAmongGroups(value = TRUE) allows to have the same individual parameters in all groups. It is available if the following elements (required for the sampling) are the same for all groups: size of groups, parameters (population or individual) and covariates.

The main goal is to make the comparison between groups easier. In particular, it is used to compare different treatments on the same individuals - subjects with the same individual parameters.

Selecting same individuals among groups ensures that the differences between groups are only due to the treatment itself. To obtain the same conclusion without this option enabled, simulation should be performed on a very large number of individuals to averaged out the individual differences.

All options of the Simulx scenario are the same as in Simulx GUI. Check the online doc of Simulx to get more guidance on how to use them.

See Also

getSameIndividualsAmongGroups

Examples

```
# create two groups with different treatments and same individuals
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "4.exploration", "PKPD_exploration.smlx")
loadProject(project_name)
addGroup("simulationGroup2")
setGroupElement("simulationGroup2", elements = "Dose_4000")
setSameIndividualsAmongGroups(value = TRUE)
```

---

setSamplingMethod *[Simulx] Set sampling method*

Description

Define which sampling method is used for the simulation. The possibilities are:

- keepOrder (default): individual values are taken in the same order as they appear in a table.
- withReplacement: individual values are sampled from a table with replacement.
- withoutReplacement: individual values are sampled from a table without replacement. This option is available only if tables contain at least the same number of individual values as a group size.

All of the above sampling methods are general and apply to all tables in a simulation scenario.

Usage

```
setSamplingMethod(method)
```

Arguments

method	(*character*) keepOrder, withReplacement, withoutReplacement

Details

All options of the Simulx scenario are the same as in Simulx GUI. Check the online doc of Simulx to get more guidance on how to use them.

See Also

getSamplingMethod

Examples

```
initializeLixoftConnectors("simulx")
project_name <- file.path(getDemoPath(), "5.simulation", "samplingOptions.smlx")
loadProject(project_name)
setSamplingMethod(method = "withReplacement")
```

---

setScenario *[Monolix - PKanalix - Simulx] Set scenario*

Description

Clear the current scenario and build a new one from a given list of tasks.

Usage

```
setScenario(...)
```

Arguments

...	A list of tasks as previously defined

Details

A scenario is a list of tasks to be run by runScenario. Setting the scenario is equivalent to selecting tasks in Monolix, PKanalix or Simulx GUI that will be performed when clicking on RUN.

For Monolix, setScenario requires a given list of tasks, the linearization option and the list of plots. Every task in the list should be associated to a boolean.
NOTE: by default the boolean is false, thus, the user can only state what will run during the scenario.
NOTE: Within a MONOLIX scenario, the order according to which the different algorithms are run is fixed:

Algorithm	Algorithm Keyword
Population Parameter Estimation	"populationParameterEstimation"
Conditional Mode Estimation (EBEs)	"conditionalModeEstimation"
Sampling from the Conditional Distribution	"conditionalDistributionSampling"
Standard Error and Fisher Information Matrix Estimation	"standardErrorEstimation"
LogLikelihood Estimation	"logLikelihoodEstimation"
Plots	"plots"

For PKanalix, setScenario requires a given list of tasks. Every task in the list should be associated to a boolean.
NOTE: By default the boolean is false, thus, the user can only state what will run during the scenario.
NOTE: Within a PKanalix scenario, the order according to which the different algorithms are run is fixed:

Algorithm	Algorithm keyword
Non Compartmental Analysis	"nca"
Bioequivalence estimation	"be"

For Simulx, setScenario requires a given list of tasks. Every task in the list should be associated to a boolean.
NOTE: By default the boolean is false, thus, the user can only state what will run during the scenario.
NOTE: Within a Simulx scenario, the order according to which the different algorithms are run is fixed:

Algorithm	Algorithm keyword
Simulation	"simulation"
Outcomes and endpoints	"endpoints"

Note: every task can also be run separately with a specific function, such as runSimulation in Simulx, runEstimation in Monolix. The CA task in PKanalix cannot be part of a scenario, it must be run with runCAEstimation.

See Also

getScenario.

Examples

```
## Not run:
[MONOLIX]
scenario = getScenario()
scenario$tasks = c(populationParameterEstimation = T, conditionalModeEstimation = T, conditionalDistributionS
setScenario(scenario)

[PKANALIX]
scenario = getScenario()
scenario = c(nca = T, be = F)
setScenario(scenario)

[SIMULX]
scenario = getScenario()
scenario = c(simulation = T, endpoints = F)
setScenario(scenario)

## End(Not run)
```

setSharedIds *[Simulx] Set element types sharing individuals*

Description

Select the element types that will share the same individuals in the simulation.

Usage

```
setSharedIds(sharedIds)
```

Arguments

sharedIds	(*vector<string>*) List of element types. The available types are: covariate, output, treatment, regressor, population, individual

Details

If several elements are defined with tables of individual values and set to some simulation groups, the option "shared ids" allows to create an intersection of ids present in these tables. After that, ids from this intersection are sampled to create the data for simulation.

All options of the Simulx scenario are the same as in Simulx GUI. Check the online doc of Simulx to get more guidance on how to use them.

See Also

getSharedIds

Examples

```
initializeLixoftConnectors("monolix")
monolix_project <- file.path(getDemoPath(), "1.creating_and_using_models", "1.1.libraries_of_models", "warf
loadProject(monolix_project)
runScenario()
initializeLixoftConnectors("simulx")
importProject(monolix_project)
setGroupElement(group = "simulationGroup1", elements = c("mlx_EBEs", "mlx_Cc"))
setSharedIds(sharedIds = c("individual", "treatment"))

# to remove shared IDs
setSharedIds(sharedIds = c())
```

setStandardErrorEstimationSettings *[Monolix] Set standard error estimation settings*

Description

Set the value of one or several of the standard error estimation settings. Associated settings are:

"minIterations"	(*int >=1*)	Minimum number of iterations.
"maxIterations"	(*int >=1*)	Maximum number of iterations.

Usage

```
setStandardErrorEstimationSettings(...)
```

Arguments

...	A collection of comma-separated pairs {settingName = settingValue}.

See Also

getStandardErrorEstimationSettings

Examples

```
## Not run:
setStandardErrorEstimationSettings(minIterations = 20, maxIterations = 250)

## End(Not run)
```

---

setStructuralModel *[Monolix - PKanalix] Set structural model file*

Description

Set the structural model.

Usage

```
setStructuralModel(modelFile)
```

Arguments

modelFile	(*character*) Path to the model file. Can be absolute or relative to the current working directory.

Details

To use a model from the libraries, you can find the model name with getLibraryModelName and set modelFile = "lib:modelName.txt" with the name obtained.

See Also

getStructuralModel

Examples

```
## Not run:
setStructuralModel("/path/to/model/file.txt")
setStructuralModel("'lib:oral1_2cpt_kaClV1QV2.txt'")

# working example to set a model from the library:

initializeLixoftConnectors("monolix",force = TRUE)
loadProject(file.path(getDemoPath(),"1.creating_and_using_models","1.1.libraries_of_models","warfarinPK_pro
#check model currently loaded:
getStructuralModel()
#get the name for a model from the library with 2 compartments:
LibModel2cpt = getLibraryModelName(library = "pk", filters = list(administration = "oral", delay = "lagTime", a
#check model content:
getLibraryModelContent(LibModel2cpt)
#set this new model in the project:
setStructuralModel(LibModel2cpt)
# check that the project has now the new model instead of the previous one:
getStructuralModel()

## End(Not run)
```

Index