Package ‘dbs’

November 22, 2017

Type Package

Title Accessory R functions to support DBSolveOptimum

Version 0.9.1

Date 2017-11-22

Author Evgeny Metelkin,
Aleksey Alekseev,
Aleksey Kalinkin

Maintainer Evgeny Metelkin <Evgeny.Metelkin@gmail.com>

Copyright InSysBio, LLC

Depends R (>= 2.15.0)

Imports mvtnorm, doParallel

Suggests lattice, deSolve, ggplot2, foreach

Description The package provides the extended facilities for DBSolveOptimum users. It has two main purposes: (1) the creation and analysis of DBsolveOptimum inputs and outputs for multiple simulations and statistical analysis, (2) the import/export DBSolveOptimum model and data files from/to different formats.

License MIT

URL https://github.com/insysbio/dbs-package,
http://sourceforge.net/projects/dbsolve/

BugReports https://github.com/insysbio/dbs-package/issues

LazyData true

RoxygenNote 6.0.1

NeedsCompilation no

R topics documented:

`dbs-package` *Accessory functions to support DBSolveOptimum*

Description

The package provides the extended facilities for DBSolveOptimum users. It has two main purposes: (1) the creation and analysis of DBsolveOptimum inputs and outputs for multiple simulations and statistical analysis, (2) the import/export DBSolveOptimum model and data files from/to different formats.

Details

Package:	dbs
Type:	Package
Version:	0.9.1
Date:	2017-11-20
License:	MIT

DBSolveOptimum is a stand-alone software tool for construction and analysis of mathematical models of biological systems. DBSolveOptimum is implemented with new tools for extended data analysis and multiple simulations, which are important for simulation of virtual clinical trials and application of modern modeling techniques, like quantitative systems pharmacology, to problems arising in drug research and development.

DBSolveOptimum is free for academic and industrial use. The latest version of DBSolveOptimum can be downloaded from: `http://sourceforge.net/projects/dbsolve/`

Copyright

Author(s)

Evgeny Metelkin, Aleksey Alekseev, Aleksey Kalinkin

References

N.Gizzatkulov, I.Goryanin, E.Metelkin, E.Mogilevskaya, K.Peskov and O.Demin. DBSolve Optimum: a software package for kinetic modeling which allows dynamic visualization of simulation

results. BMC Systems Biology, 2010, 4 (109): 1-11. PubMed

E.Metelkin, A.Alekseev, G.Lebedeva, O.Demin. DBSolve Optimum r.33: Practical Guide pdf

See Also

`calccb`

`calcop`

`parsetgen`

`import.slv`

`rct.from.slv`

`import.dat`

`C.from.slv`

`C.from.slv` *Create .C code from ruSlv object*

Description

The function creates **deSolve**-compatible C code from .SLV image and save it to a file.

Usage

```
C.from.slv(slv, file = "model.c",
            output = slv$output.on, dbs.compatibility = FALSE)
```

Arguments

`slv`	object of `ruSlv` class, model image.
`file`	filename to save .C code.
`output`	character vector with names of additional output values.
`dbs.compatibility`	logical value to use DBSolve-like method of parameters updates.

See Also

`import.slv`
`deSolve`
`ode`

Examples

```
### create and compile C code for 'example1.slv', Rtools may be reqired
C.from.slv(example1_ruSlv, output="D")
## Not run:
system("R CMD SHLIB model.c") # compilation for .DLL
library(deSolve)
dyn.load(paste0("model", .Platform$dynlib.ext))
res<-ode(y=example1_ruSlv$ode.initials,
         times=seq(0,example1_ruSlv$solver.time.limit,0.1),
          func = "derivs",
          parms=example1_ruSlv$ode.parameters.external,
          dllname = "model",
          initfunc = "initmod",
          nout=length("D"),
          outnames = "D"
  )
dyn.unload(paste0("model", .Platform$dynlib.ext))
plot(res)

## End(Not run)

### plot simulations from SLV model
## Not run:
  filePath<-system.file(package = "dbs", "extdata/example1.slv")
  raw<-read.slv(filePath) # read from example
  compatible.slv(raw) # TRUE
  example1_ruSlv<-import.slv(raw)
  C.from.slv(example1_ruSlv, output="D")
  system("R CMD SHLIB model.c") # compilation for .DLL
  library(deSolve)
  dyn.load(paste0("model", .Platform$dynlib.ext))
  res<-ode(y=example1_ruSlv$ode.initials,
          times=seq(0,example1_ruSlv$solver.time.limit,0.1),
          func = "derivs",
          parms=example1_ruSlv$ode.parameters.external,
          dllname = "model",
          initfunc = "initmod",
          nout=length("D"),
          outnames = "D"
  )
  dyn.unload(paste0("model", .Platform$dynlib.ext))
  plot(res)

## End(Not run)
```

`calccb` *Calculation of confidence bands*

Description

The function calculates pointwice confidence bands based on Monte-Carlo simulations in DB-SolveOptimum. The lower and upper confidence band calculated as lower and upper quantile for interpolated particular x point.

Usage

```
calccb(input, x.col, x.seq, y.col, factor.col = c(),
         q.seq = c(0.025, 0.5, 0.975), nos.col = "nos",
        par_calc = FALSE, cpu.cores = 4, silent = FALSE, include.nos = c(), ...)
```

Arguments

`input`	`data.frame` passed from DBSolveOptimum output
`x.col`	number or name of column in `input` corresponded to free variable (i.e. time).
`x.seq`	numerical vector of points to interpolate values in `x.col`.
`y.col`	vector of column numbers or names in `input` corresponded to simulated variables (model output).
`factor.col`	vector of column numbers or names in `input` corresponded to condition parameters (model input).
`q.seq`	sequence of probabilities for calculation of lower and upper quantile. The default vector `c(0.025, 0.5, 0.975)` corresponds to calculation of median value and 0.95 confidence band.
`nos.col`	number or name of column in `input` corresponded to enumeration of random parameter set.
`par_calc`	logical value to use parallel calculation for acceleration. It requires `parallel`, `foreach`, `iterators` packages.
`cpu.cores`	the number of CPU cores to use if `par_calc=TRUE`.
`silent`	logical value to suppress the messages during calculations.
`include.nos`	vector of number of samples to analyze the approximation in the chosed number of sample.
`...`	other arguments passed to `quantile`

Value

The returned value is `data.frame` class object. The columns describe:

`names(x.col)`	free variable values passed from argument `x.col`.
`var_id`	names of simulated variables as passed from `y.col`.
`quant_`	columns represent calculated quntiles for interpolated points.
`names(factor.col)`	condition variable values passed from argument `factor.col`.
`group`	unique identifier for combination of `factor.col`.

The value has the additional attributes:

`col.def`	definition of columns, type of data in columns.
`col.title`	titles for columns. May be usefull for visualization.
`var.title`	titles for simulated variables. May be usefull for visualization.
`group.title`	titles for condition groups. May be usefull for visualization.
`approx_nos_`	column(or columns) represents interpolated points for chosed number of sample (presented only if `include.nos` has values in).

See Also

`foreach`
`quantile`
`approxfun`

Examples

```
### calculation of confidence bands based on example4.slv
## Not run: example4_parset_bs.cond_res<-read.delim("example4_parset_bs.cond_res.txt") # read from DBSolve ou
example4_cb<-calccb(input=example4_parset_bs.cond_res,
                    x.col="t",
                    x.seq=seq(0,96,by=0.5),
                    y.col=c("C0","C1"),
                    factor.col = c("Dose","T"))
## Not run: write.delim(example4_cb, "example4_cb.txt") # save results

### plot all results with lattice
library(lattice)
xyplot(quant_0.025+quant_0.5+quant_0.975~t|var_id+group,
       data=example4_cb,
       type="l",
       lty=c(2,1,2),
       xlab="Time, h",
       ylab="Concentration of drug, ng/ml",
       main="All CB simulations")

###You can also plot all results using ggplot2:
library(ggplot2)
ggplot(example4_cb,aes(t,quant_0.025))+
        geom_line(linetype="dashed", color="blue")+
        geom_line(aes(t,quant_0.5),color="black",linetype="solid")+
        geom_line(aes(t,quant_0.975),color="blue",linetype="dashed")+
        facet_wrap(~var_id+group)+
        ggtitle("All CB simulations")+
        scale_x_continuous(name="Time,h")+
        scale_y_continuous(name="Concentration of drug, ng/ml")


### plot dbsolve output results
example4_cb<-calccb(input=example4_parset.cond_res,
                    x.col="t",
                    x.seq=seq(0,96,by=0.5),
                    y.col=c("C0","C1"),
                    factor.col = c("Dose","T"),
                    par_calc = FALSE)
```

---

calcop *Calculate interpolation for optimal values*

Description

The function interpolates simulations in DBSolveOptimum for the series of conditions and create `mod.frame` object.

Usage

```
calcop(input, x.col, x.seq, y.col, factor.col = c())
```

Arguments

`input`	`data.frame` passed from DBSolveOptimum output
`x.col`	number or name of column in `input` corresponded to free variable (i.e. time).
`x.seq`	numerical vector of points to interpolate values in `x.col`.
`y.col`	vector of column numbers or names in `input` corresponded to simulated variables (model output).
`factor.col`	vector of column numbers or names in `input` corresponded to condition parameters (model input).

Value

The returned value is `mod.frame` class object which is extension of `data.frame` class with the additional attributed. The columns describe:

`names(x.col)`	free variable values passed from argument `x.col`.
`var_id`	names of simulated variables as passed from `y.col`.
`simulation`	column represents simulation value for interpolated points.
`names(factor.col)`	condition variable values passed from argument `factor.col`.
`group`	unique identifier for combination of `factor.col`.

The value has the additional attributes:

`col.def`	definition of columns, type of data in columns.
`col.title`	titles for columns. May be usefull for visualization.
`var.title`	titles for simulated variables. May be usefull for visualization.
`group.title`	titles for condition groups. May be usefull for visualization.

See Also

`calccb`
`approxfun`

Examples

```
### calculation based on example4.slv
## Not run: example4_output_op<-read.delim("dbs_output_op.txt") # read from output
example4_op<-calcop(input=example4_output_op,
                    x.col="t",
                    x.seq=seq(0,96,by=0.5),
                    y.col=c("C0","C1"),
                    factor.col = c("Dose","T"))
## Not run: write.delim(example4_op, "example4_op.txt") # save results

### plot all results with lattice
library(lattice)
xyplot(simulation~t|var_id+group,
       data=example4_op,
       type="l",
       lty=1,
       xlab="Time, h",
       ylab="Concentration of drug, ng/ml",
       main="All CB simulations")
```

`clean.comments` *Clean C-style text*

Description

Function takes character vector of C-style lines (as passed from readLines), delete comments, spaces, empty lines, multiple semicolomns and line breaks.

Usage

```
clean.comments(input)
```

Arguments

`input`	character vector of C-style text lines, vector can be passed from readLines function.

Value

Character vector of cleaned C-style text. It can be saved to file by cat(..., sep = "\n") function.

Examples

```
## Not run: cTextExample
(cln<-clean.comments(cTextExample))
cat(cln, file="cln.txt", sep = "\n") # save text to file
```

`hessian2stat` *Functions for calculating statistical characteristics from hessian or parameter set*

Description

The functions calculate statistical characteristics based on normal distribution or transformed normal distribution.

Usage

```
  hessian2stat(hessian, optim, transform="", level=0.95, ...)

  parset2stat(parset, transform="", level=0.95, norm.test=shapiro.test, ...)
```

Arguments

`hessian`	Hessian matrix of second derivatives (-2logL vs parameters). It must be positive-definite.
`optim`	Optimal values of parameters (best-fit values).
`transform`	String vector for what distribution is used on parameter. For default all parameters are set to normal distribution.

`level`	confidence level for calculation of confidence intervals.
`...`	other arguments passed to `solve`.
`parset`	data.frame with parameter set (mandatory for `parset2stat`)
`norm.test`	function to perform normality test

Value

The output for this function is covariance `matrix` calculated based on assymptotic approach, and it takes into account the transformation of parameter space.

Examples

```
### calculate statistical characteristics based on calculated hessian, see 'example4.slv' from DBSolve manual
## Not run: example4_parset_bs<-read.delim("example4_hessian.txt") # read hessian from file
optimal<-c(kcat=7.130016e-01, Vd=5.205980e+00, Km=5.240306e+00, kabs=2.014304e+00)
hessian2stat(hessian=as.matrix(example4_hessian), optim=optimal, transform="log")

### calculate statistical characteristics based on parameter set, see 'example4.slv' from DBSolve manual
## Not run: example4_parset_bs<-read.delim("example4_parset_bs.txt") # read hessian from file
parset2stat(parset=example4_parset_bs[,1:4])
```

`nan.plot` *Plot all points including infinite*

Description

The functions to plot all points of the dataset even they are out of the `xlim` and `ylim` range.

Usage

```
nan.plot(x, y, xlim = range(x, finite = TRUE), ylim = range(y, finite = TRUE),
         log = c("", "x", "y", "xy"), ...,
         force.bound = FALSE, delta = 0.1)

nan.points(x, y, ...)
```

Arguments

`x`	numeric vector to plot on x-axis.
`y`	numeric vector to plot on y-axis.
`xlim`	the x limits (x1, x2) for main region.
`ylim`	the y limits (y1, y2) for main region.
`log`	a character string which contains "x" if the x axis is to be logarithmic, "y" if the y axis is to be logarithmic and "xy" or "yx" if both axes are to be logarithmic.
`...`	other graphical parameters passed to `plot.default` or `points`
`force.bound`	logical value to forcefully create the extended region. `F` means the extended region is created if some of points is out of the main region.
`delta`	the relative size of region based on main region size.

Details

If someone use the default `plot` and `points` method the points out of `xlim` and `ylim` bacame invisible. In some cases it is not OK. The presented functions creates the exteded region and plot all points including infinete values there.

The function work only for numerical vectors but not for data.frame, matrix, etc.

Note

These are experimental functions so the troubles are possible.

See Also

`plot.default`
`points`

Examples

```
### comparison of exponential plots
x<-seq(0,20, 0.1)
y<-5*exp(x)
plot(x,y) # default plot without limits
plot(x,y, ylim=c(0,100)) # default plot with y limits
nan.plot(x,y, ylim=c(0,100))
```

`parsetgen` *Functions for DBSolveOptimum to generate parameters and conditions using multivariate normal distribution*

Description

The function `parsetgen` generates random dataset.

The function `parsetgen.cond` add to the dataset from `parsetgen` conditions columns and nos column

Usage

```
parsetgen(cov, mu, transform="", samples=1024)
parsetgen.cond(parset, cond=data.frame(), max.samples=nrow(parset), uniq.nos=FALSE)
```

Arguments

`cov`	covariance matrix with dimension nxn and whose element in i,j positions is the covariance between the i and j elements.
`mu`	named vector length of n with named parameters and their expectations
`transform`	character vector containing the distribution for each parameter. It can be "" for normal distribution and "log" for log-normal distribution
`samples`	number of the samples that we want to get in the ouput dataset (1024 as default)
`parset`	output data.frame from parsetgen function or users data.frame

`cond`	conditional data.frame. If empty, the result of parsetgen.cond is input data.frame with nos column
`max.samples`	number of resulted samples for each condition. If nrow(parset)<max.samples, that provide an error. If nrow(parset)>max.samples, that cuts the data frame by number of samples
`uniq.nos`	number of unique parameter set. If FALSE, then nos will be with repeating, and if TRUE - nos will have unique number.

Value

The output is `data.frame` with the columns named as names in your `expect` vector, and for `parsetgen.cond` it will be `data.frame` with the "nos" column, and names(parset) columns and names(cond) column

See Also

`rmvnorm`

Examples

```
### create parameter set for Monte-Carlo simulation using covariance matrix (example4)
example4_parset<-parsetgen(example4_stat$muCov, example4_stat$mu, example4_stat$transform)
example4_parset.cond<-parsetgen.cond(
  example4_parset,
  cond=data.frame(Dose=c(1,5,10,1,5,10), T=c(12,12,12,24,24,24))
  )
write.delim(example4_parset.cond, "example4_parset.cond.txt")


### create parameter set for Monte-Carlo simulation using parameter set based on bootstrapping(example4)
example4_parset_bs.cond<-parsetgen.cond(
  example4_parset_bs[,c(1,2,3,4,6)],
  cond=data.frame(Dose=c(1,5,10,1,5,10), T=c(12,12,12,24,24,24))
  )
write.delim(example4_parset_bs.cond, "example4_parset_bs.cond.txt")

### Making some parameter set with three expectation and cov as vector
  expect<-c(kcat=0.5, Vd=23.4, Km=12.4) #making expect vector
  cov<-c(23.5, 37.9, 23.5) #making vector for coviance matrix
  transform<-c("log", "", "log") #the transform vector with distributions
  output<-parsetgen(cov, expect, transform)

  #Making some dataset,with three expectation and cov as vector
  expect<-c(kcat=0.5,Vd=23.4,Km=12.4) #making expect vector
  cov<-matrix(c(23.5,0,1, 0,37.9,0, 1,0,23.5), ncol=3) #coviance matrix
  transform<-c("log","","log") #the transform vector with distributions
  output<-parsetgen(cov, expect, transform)

  #Using the parsetgen.cond function. Suppose we have output from parsetgen function
  cond<-data.frame(cond1=c(1,2.5,1), cond2=c(0,0,1), cond3=c(0,0,0))
  output1<-parsetgen.cond(output, cond=cond, max.samples=1024,uniq.nos=FALSE)
```

rct.from.slv *Create .RCT from ruSlv*

Description

The function analyzes ruSlv stoicheometry matrix and creates reaction list in .RCT format.

Usage

```
rct.from.slv(y)
```

Arguments

`y`	the object of class `ruSlv`, model image.

Details

The function uses the stoicheometry matrix, names of rates and metabolites only. The true structure of differential equation is not taken into account.

Value

character vector representing list of reactions which can be saved using `cat(..., sep="\n")`.

See Also

`import.slv`
`cat`

Examples

```
### create .RCT file from 'example4.slv'
rct<-rct.from.slv(example4_ruSlv)
cat(rct, file="example.rct", sep="\n")
```

read.dat *Import .DAT files*

Description

The set of functions to import experimental dataset from DBSolveOptimum format .DAT file.
`read.dat` reads .DAT file and perform initial parsing.
`import.dat` creates `ruData` object from `read.dat` output.

Usage

```
read.dat(file)

import.dat(dat)
```

Arguments

`file`	filename of .DAT file.
`dat`	list object of format ruData.raw which is output of `read.dat` function.

Value

The returned value of `import.dat` is an object of class `ruData` which mode is `list` and structure corresponds to `ruList`. Second level is lists each of which has the following components:

`data_id`	character identifier of the dataset
`data`	experimental data of `data.frame` class
`conditions`	data.frame describing conditions
`solver`	character identifier of solver type: ode, explicit, implicit
`error.type`	description of error model, currently possible values are: "additive T", "additive F"

See Also

`write.list`

Examples

```
### read and save data from 'example4.dat'
filePath<-system.file(package = "dbs", "extdata/example4.dat")
dat_raw<-read.dat(filePath)
example4_ruData<-import.dat(dat_raw)
write.list(example4_ruData, "example4_ruData.txt")
```

---

read.list *Read, write and check object of ruList format*

Description

The common functions for manipulating objects of structure `ruList`: reading from file, writing to file of format `ruList.txt` and checking `ruList` object for appropriateness.

Usage

```
read.list(file)

write.list(x, file="")

check.list(x)
```

Arguments

`x`	the object of format `ruList`
`file`	a character string naming a file

Details

ruList and ruList.txt is the internal dbs-package format for representing complex objects.

The structure of ruList format can be described as three-level nested object: (1) list with any number of elements of 2-d level, (2) list with any number of elements of 3-d level, (3) objects of classes: data.frame, matrix, numeric, character, integer, logical, mod.frame. Any level may have attributes of classes: data.frame, matrix, numeric, character, integer, logical.

The ruList.txt is a human readable representation of ruList object saved to .TXT file.

Value

read.list returns the object of ruList format

check.list returns logical TRUE if the object x has appropriate structure

Note

The current version of check.list does not check attributes of 3-d level.

The current version of write.list does not check x argument for consistency. Be carefull.

See Also

list

Examples

```
### write, read and check ruList
models<-list(example4_ruSlv)
write.list(models, "models.txt")
models1<-read.list("models.txt")
check.list(models1) # output: TRUE
all.equal(models1, models) # output: 'names for target but not for current'
```

---

read.slv *Import .SLV files to R*

Description

A set of functions to import model files of DBSolveOptimum (.SLV) to R-environment as the object of class ruSlv.
read.slv function reads .SLV file and creates list of format ruSlv.raw
compatible.slv function checks compatibility of ruSlv.raw passed from read.slv for compatibility with current version of import.slv.
import.slv function analyzes ruSlv.raw passed from read.slv and creates the object of class ruSlv.

Usage

```
read.slv(file)

compatible.slv(x)

import.slv(x)
```

Arguments

`file`	valid .SLV file saved from DBSolveOptimum.
`x`	object of format `ruSlv.raw` to use for model import.

Details

In many cases DBSolveOptimum features are not enough to manipulate the model structure and to simulate specific conditions. This set of functions transforms all the structure of .SLV file to the R-environment for easy manipulation.

`read.slv` function reads model code from file and perform initial parsing based on SLV version described in file '`slv25tab.csv`'.

`import.slv` analyzes different parts of `read.slv` output and create an object of class `ruSlv` which is an image of initial .SLV file. `compatible.slv` is developed to check the structure of `ruSlv.raw` object for further parsing using `import.slv`. It is part of `import.slv` function so it is not necessary to use it separately.

Value

`read.slv`	returns `list` of the format `ruSlv.raw`
`compatible.slv`	returns `TRUE` if `x` output is compatible with current version of `import.slv` and `FALSE` otherwise
`import.slv`	returns the object of mode `list` and `ruSlv` class attribute

Note

The current version of dbs-package officially supports only .SLV version 25. Try `compatible.slv` to check.
Known restrictions:

1. Cannot read 'events'
2. Cannot read 'fit conditions'

See Also

`rct.from.slv`
`C.from.slv`

Examples

```
### import 'example4.slv'
filePath<-system.file(package = "dbs", "extdata/example4.slv")
raw<-read.slv(filePath) # read from example
compatible.slv(raw) # TRUE
example4_ruSlv<-import.slv(raw)

### import 'example1.slv'
filePath<-system.file(package = "dbs", "extdata/example1.slv")
raw<-read.slv(filePath) # read from example
compatible.slv(raw) # TRUE
example1_ruSlv<-import.slv(raw)
```

`replacePow` *Replacing ^ to Pow*

Description

Function `replacePow(e)` used in `import.slv` to replace ^ to pow, for convert to C model file without errors.

Usage

```
replacePow(e)
```

Arguments

`e`	expression with ^ and sqrt as expression type.

Details

For using function `replacePow(e)`, You need to parse your text and after using function deparse to obtain the text.

See Also

`C.from.slv`
`read.slv`

Examples

```
#Example to using the replacePow function

text = "Cr1=fdr^x+x^y+sqrt(25^(x+3))-sqrt(sqrt(13-x)+(4*x-5)^(2+y))"
e=parse(text=text) #Its neccesary to use parse and deparse functions
print(deparse(replacePow(e)[[1]])) #printing text with pow
```

`replacePowlist` *Function for replacing ^'s to pow inside the list*

Description

The function replacing ^'s to pow based on parsing the string vector

Usage

```
replacePowlist(l,x)
```

Arguments

`l`	list with vectors containing exrpessions to replace
`x`	names of the vector inside list

Value

The output is the list with replaced ^'s.

See Also

`read.slv`
`import.slv`

Examples

```
##Example of using the function

temp1<-list(one=c("eee^3","2^rrr"))
replacePowlist(temp1,1)
```

`signup` *Up- and down- rounding*

Description

`signup` rounds the values in its first argument to the specified number of significant digits upwards.

`signdown` rounds the values in its first argument to the specified number of significant digits downwards.

Usage

```
signup(x, digits = 6)

signdown(x, digits = 6)
```

Arguments

`x`	a numeric vector.
`digits`	integer indicating the number of significant digits to be used.

See Also

`signif`

Examples

```
signup(c(1.111, 1.2345e5, 9.8765e-5), 3)

signdown(c(1.111, 1.2345e5, 9.8765e-5), 3)
```

`write.delim` *Data output with tab delimiters*

Description

prints its required argument x (after converting it to a data frame if it is not one nor a matrix) to a file with tab delimiter without quotes and row names

Usage

```
write.delim(x, file = "", ...)
```

Arguments

`x`	the object to be written, preferably a matrix or data frame. If not, it is attempted to coerce x to a data frame.
`file`	a character string naming a file
`...`	other arguments passed to write.table

Details

The function is equivalent to
`write.table(x = x, file = file, quote = F, sep = "\t", row.names = F, ...)`

See Also

`write.table`

Examples

```
### create and write data.frame
df<-data.frame(number=1:5, words=c("one", "two", "three", "four", "five"))
write.delim(df, "df.txt")
```

Index