

Package: uqsa (via r-universe)

October 19, 2026

Type Package

Title Uncertainty Quantification and Global Sensitivity Analysis

Version 0.8.3

Description In the field of systems biology, chemical reaction networks are modeled in various ways, two of those are: (i) stochastic simulations (e.g. Gillespie algorithm) and (ii) ordinary differential equations. In this package we use a simple tabular model description of reaction systems and automatically generate C code for either solver type. We use the ordinary differential equation solvers from the GNU Scientific Library and provide an interface that deals with lists of simulation experiments. Each simulation experiment contains both the data, and instructions for the model to replicate the data. We use approximate Bayesian computation methods (combined with Markov chain Monte Carlo and sequential Monte Carlo, particle filters) as well as classic methods such as Random Walk Metropolis (Gaussian transition kernel) and Simplified Manifold Metropolis adjusted Langevin algorithm for a Bayesian investigation of the model's parameter space. Experiments can be evaluated in a sequence; intermediate probability densities are modeled using the 'VineCopula' package. The package is also intended to be useful in an HPC environment, with some functions that use 'pbdMPI' capabilities.

Language en-US

License GPL (>=3)

Encoding UTF-8

LazyData true

Depends R (>= 4.1)

Imports stats, methods, parallel, Ryacas, VineCopula, MASS, errors, mvtnorm, digest, colorspace, pracma, cli

Suggests ks, remotes, pbdMPI, ggplot2, testthat (>= 3.0.0), knitr, rmarkdown, hexbin

SystemRequirements GSL (≥ 2.7)

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

VignetteBuilder knitr

URL <https://icpm-kth.github.io/uqsa/>

Config/roxygen2/version 8.1.0

RoxygenNote 7.3.3

Config/pak/sysreqs libgsl0-dev

Repository <https://icpm-kth.r-universe.dev>

Date/Publication 2026-10-19 08:53:55 UTC

RemoteUrl <https://github.com/icpm-kth/uqsa>

RemoteRef HEAD

RemoteSha 01123a3a27da571ef1f7e25c93ab052e6bb46ddd

Contents

abc_mcmc	3
ABCSMC	6
as_cme	7
as_ode	8
c_path	9
c_path<-	10
column	10
CRNN	11
dCopulaPrior	12
dNormalPrior	12
dUniformPrior	13
experiments	13
fitCopula	14
formulae	15
generate_code	16
gNormalPrior	16
gsa_binning	17
gsa_saltelli	18
gsl_odeiv2_CRNN	19
gsl_odeiv2_fi	20
high_level_metropolis	22
high_level_smmala	23
kinetic_law_matrix	24
linear_scale	25
logLikelihoodFunc	26
makeObjective	27
mcmc	28
mcmc_init	29

mcmc_mpi	30
method	31
metropolis_update	32
model_from_tsv	33
name_method	34
parse_concise	35
pc_dist	36
rCopulaPrior	36
replace_powers	37
rNormalPrior	38
rUniformPrior	38
saltelli_prior	39
scrnn	40
sensitivity.graph	41
shlib	42
showPosterior	43
simstoch	44
simulator.c	45
small	47
smmala_update	47
so_path	49
so_path<-	50
standard_error_matrix	50
stoichiometric_matrix	51
stoichiometry	52
tune_step_size	52
uncertainty	54
unit.from.string	55
uqsa_example	56
values	57
write_and_compile	57
write_c_code	58
Index	59

abc_mcmc	<i>Performs Approximate Bayesian Computation Sampling of Model Parameters via Markov Chain Monte Carlo</i>
----------	--

Description

ABC replaces the need for an exact likelihood function and uses a distance function instead: the distance between data and simulation. This distance function is very similar to a likelihood function but lacks a statistical justification. Nevertheless, this distance function, like the likelihood function of a deterministic model, performs a simulation of the scientific model, be it fully stochastic or a stochastically embedded, but deterministic in its core.

Usage

```

abc_mcmc(
  objectiveFunction,
  startPar,
  N,
  burnIn = ceiling(sqrt(N)),
  Sigma0 = cov(t(startPar)),
  dprior = NULL,
  deltaSpan = NULL,
  batchSize = NCOL(startPar),
  parAcceptable = function(p) {
    all(is.finite(p))
  },
  verbose = getOption("uqsa.verbose", interactive())
)

```

Arguments

objectiveFunction	function that, given a parameter matrix as input, simulates the model, and outputs the distance between experimental data and data simulated from the model with the parameter provided in input. This has to be a closure that contains the experimental data within itself. The closure must be vectorized over the columns of its matrix argument.
startPar	starting values for the parameter vector, can (and should) be a matrix with n columns, where each column is a valid parameter vector; the number of columns determines the batch size.
N	requested number of batches to return, the sample will be of size batchSize*N (batch size is the number of parallel Markov chains).
burnIn	number of batches where the transition kernel will be adjusted to achieve an acceptance rate of below 10%.
Sigma0	multivariate normal covariance of Markov chain transition kernel, defaults to the covariance of the initial parameters. If startPar is one vector, this matrix must be provided explicitly.
dprior	a function that returns prior probability density values.
deltaSpan	either an initial and final value for the ABC threshold delta, or a fixed value for delta that will never change.
batchSize	the size of each batch, this should be a number that could be sufficient to calculate the covariance of in the given parameter space.
parAcceptable	a function that can reject a parameter vector early based on user-requirements. Has to return a scalar Boolean. Use this to test for inequalities that you find difficult to encode in the prior.
verbose	when TRUE, a progress bar is printed during burn-in and actual sampling.

Details

The distance of the ABC setting is compared to a threshold value *delta*. The threshold doesn't need to be explicitly provided. You can however provide a span of acceptable values in any order, the smaller value will be used as a lower bound, the larger value will be used initially.

The ABC procedure will attempt to converge first, using the initial delta value, and decreasing it slowly using observed distance values.

This function always operates on a bundle of Markov chains. The size of this bundle can be determined through `startPar` (a matrix of column vectors). Each column will be used as the initial point of a Markov chain. The chains will be resampled at each step during the convergence phase, and decouple from one another once the burn-in is complete. When `startPar` is a vector, the `batchSize` will be set to `100*length(startPar)`, if not provided explicitly.

ABC methods (distance function, threshold delta) can be combined with several other methods (like particle filters). Here we use several parallel Markov chains to sample from the approximate posterior.

Since this sampler works in batches, it stores its return value in batches along a 3rd dimension of an array: `ret$draws[, , 1]` is the first iteration of MCMC, `ret$draws[, , N]` the last iteration. One sampled model parameter vector is a column vector: `ret$draws[, 1, 1]` is something that can be passed to the simulator. Because the simulator accepts batches of parameters, this will cause a batch of simulations: `s(ret$draws[, , N])`. This structure is useful when determining the auto-correlation length along the 3rd dimension: `ret$draws[i, j, ]` is auto-correlated along the 3rd dimension for any choice of *i* and *j*. To obtain a classic sample, you can first flatten the third dimension: `dim(ret$draws) <- c(np, batchSize*N)`, and then transpose for functions like `cov`.

Value

a list containing a sample matrix and a vector of observed distances (values that compare to delta for each sample). The sample (draws) is stored as a 3d-array, with these dimensions: n_p *times* *m* *times* *N*, where n_p is the number of model parameters, *m* the batch-size, and *N* the number of MCMC iterations.

Examples

```
f <- uqsa_example("AKAR4")
m <- model_from_tsv(f)
o <- as_ode(m)
ex <- experiments(m,o)
C <- generate_code(o)
c_path(o) <- write_c_code(C)
so_path(o) <- shlib(o)
s <- simulator.c(ex,o,parMap=log10ParMap)
objFunc <- makeObjective(ex,s)
p0 <- log10(values(m$Parameter))
lowerBound <- p0 - 3
upperBound <- p0 + 3
dprior <- dUniformPrior(lowerBound,upperBound)
rprior <- rUniformPrior(lowerBound,upperBound)
```

```

X <- rprior(96)    # increase this number ...
## this always takes more than 5s to run:
if (interactive()){
  abcSample <- abc_mcmc(
    objFunc,
    startPar=t(X),
    N=128,          # ... and this number
    Sigma0=cov(X),
    dprior=dprior
  )
}

```

ABCSMC

Performs Approximate Bayesian Computation as a Particle Filter

Description

Given a set of simulation experiments (list), a model, parameter boundaries, this function will draw a sample of parameters from the posterior probability density of the given problem.

Usage

```

ABCSMC(
  objectiveFunction,
  startPar,
  Sigma = 2 * cov(t(startPar)),
  dprior,
  delta = c(2, 0.5),
  parAcceptable = function(p) {
    all(is.finite(p))
  },
  verbose = getOption("uqsa.verbose", interactive())
)

```

Arguments

objectiveFunction	a function that can simulate the model for a batch of parameter vectors provided as a matrix of columns (batches)
startPar	a matrix that has the same shape as the desired sample, but transposed, this can be a sample from the prior or a pre-conditioned sample that approximates the posterior, e.g.: t(rprior(1000))
Sigma	multivariate normal covariance of Markov chain transition kernel
dprior	a function that returns prior probability density values
delta	ABC acceptance threshold, either a scalar, then it is the initial value of delta, or a pair of values, then it is the starting value and the final value of delta: c(initialDelta,finalDelta)

- parAcceptable** is a rejection-shortcut function; if `parAcceptable(p)` returns `FALSE` for a specific value of `p`, it means that simulations shouldn't even be attempted.
- verbose** a logical value indicating whether log messages should be printed

Details

This is a variant of ABC where the entire batch is simulated with one call to the simulator. `startPar` is the initial batch to be simulated: it is a matrix where columns are different parameter vectors (e.g. prior sample members). In other words: `startPar[,i]` must be a valid argument for the `objectiveFunction`.

The objective function is a closure

The Objective-Function `objectiveFunction(P)` should return a matrix with `n` rows, where `n` is the number of simulation experiments (and thus data-sets), and `m` columns, where `m` is the number of parameterizations `NCOL(P)`.

Value

a list containing a sample matrix and a vector of scores (values of delta for each sample)

Examples

```
library(parallel)
opt <- options(mc.cores=2) # use [detectCores()] here
f <- uqsa_example("AKAR4")
m <- model_from_tsv(f)
ex <- experiments(m,as_ode(m,cla=FALSE))
G <- as_cme(m) # for Gillespie solver
C <- generate_code(G)
c_path(G) <- write_c_code(C)
so_path(G) <- shlib(G)
muX <- m$Parameter$value
sdX <- m$Parameter$stdv
rprior <- rNormalPrior(log(muX^2/(muX^2+sdX^2)),sqrt(log(1+sdX^2/muX^2)))
dprior <- dNormalPrior(log(muX^2/(muX^2+sdX^2)),sqrt(log(1+sdX^2/muX^2)))
s <- simstoch(ex,G,logParMap)
O <- makeObjective(ex,s)
X <- rprior(100)
colnames(X) <- rownames(m$Parameter)
if (interactive()) {
  posterior <- ABCSMC(O,t(X),Sigma=cov(X),dprior=dprior,delta=c(0.4,1.5))
}
options(opt) # restore original options
```

Description

The chemical master equation can be simulated as a Markov jump process (or continuous time Markov chain). One of the stochastic solver algorithms is the Gillespie algorithm. This function return sa data structure that can be used to generate code for the Gillespie solver in this package.

Usage

```
as_cme(m)
```

Arguments

`m` list of data.frames, obtained via `model_from_tsv()`

Details

This function interprets the continuous model `m` as a discrete state model with molecule counts and propensities. For this reason, we need to specify a volume for the simulations to take place in.

The model `m` is assumed to describe a reaction network, as a list of data.frames (as returned by [model_from_tsv](#)). The systems biology information in the file is assumed to be concentrations and rate coefficients, regardless of the interpretation this function will derive from it. This is to make the model format of the TSV file fairly uniform and independent of how we want to solve the derived equations, be it ODE or CME.

Like the ode object, the returned object can also store the paths of files we create for this model, with: `c_path<-`, and `so_path<-`

With the information provided with the rate coefficient units and a volume, this function tries to convert everything to Gillespie rate constants.

Value

a list containing the interpreted model.

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
stochasticModel <- as_cme(m)
print(stochasticModel)
```

as_ode

Interprets a model as an ODE

Description

This function accepts a list generated from a collection of TSV files (or a similar format) and interprets the contents as an ordinary differential equation (ODE).

Usage

```
as_ode(m, cla = requireNamespace("pracma", quietly = TRUE))
```

Arguments

m a list of data.frames, each corresponding to a TSV file or sheet in a spreadsheet.

cla a Boolean value indicating whether conservation law analysis should be performed.

Details

The argument **m** can be obtained via `model_from_tsv()`. It has the components:

- `m$Constant`
- `m$Parameter`
- `m$Input`
- `m$Expression`
- `m$Compound`
- `m$Reaction`
- `m$Experiment`

There can be additional components describing measured data for this model.

Value

a list that contains a summary of this model interpreted as an ODE, crucially, the list contains the element `vf`, the right-hand-side (vector field) of the ODE, this is the main result of this function.

Examples

```
f <- uqsa_example("AKAR4")
m <- model_from_tsv(f)
o <- as_ode(m)
print(names(o))
print(o$vf)
```

c_path

Retrieves information about the model's C code

Description

Returns the location of the model's C code (a file).

Usage

```
c_path(o)
```

Arguments

o the ODE, or CME model

Value

the path where the c code is stored

c_path<-	<i>Adds information about the model's C code</i>
----------	--

Description

Adds the location of the model's C code (a file). The model is typically a list of named numeric and named character vectors, which describe the (interpreted) model.

Usage

```
c_path(o) <- value
```

Arguments

o the ODE , or CME model
value the path to the compiled model

Value

```
modified o, with information about compiled code m <- model_from_tsv(uqsa_example("AKAR4"))
o <- as_ode(m) c_path(o) <- write_c_code(generate_code(o)) so_path(o) <- shlib(o) print(o)
```

column	<i>Retrieves j-th column with names</i>
--------	---

Description

When indexing a matrix or data.frame, rownames are lost. This function will return a column of a matrix, as a vector (dropping rank so to speak), but the vector will retain the rownames of the matrix

Usage

```
column(m, j = 1)
```

Arguments

m a matrix
j a column index

Value

a named vector

Examples

```
m <- model_from_tsv(uqsa_example("AKAP79"))
u <- column(m$Parameter, "unit")
```

CRNN

Creates C code for a chemical reaction neural network

Description

This function creates a very general ODE (c source code), that can be compiled and simulated using the UQSA package.

Usage

```
CRNN(numReactions, initialValues, funcValues, model.name = "CRNN")
```

Arguments

numReactions	the number of reversible mass action law reactions
initialValues	named vector of initial values, names will be used as the names of the reacting compounds.
funcValues	named character vector, can be any valid C expression (one line) of the available state variables (the names can be used literally).
model.name	the prefix of all created model functions: CRNN_vf, CRNN_jac, ...

Details

Example: $A + B \rightleftharpoons C$ numReactions: $n <- 1$ initialValues: $x <- c(A=2, B=3, C=0)$ funcValues: $f <- c("A+B", "log(A))$

The above definition would create a CRNN inspired ODE, where $A+B$ and $\log(A)$ are treated as observable (measurable) values (functions of the state variables).

Note: In addition to the state variables, the function values can also reference the log-parameters of the model as $l[i]$, where i is a 0-based offset to the reaction i ; the backward rate, is stored at position $l[i+numRct]$.

Value

a character vector suitable for writing to a file (.c)

Examples

```
C <- CRNN(4, c(A=1, B=2, C=3), c(out="A+B+C"), model.name="testmodel")
cat(head(C), sep='\n')
```

dCopulaPrior	<i>Creates a prior probability density function</i>
--------------	---

Description

This function accepts the return list of [fitCopula](#) and creates a density function from it.

Usage

```
dCopulaPrior(Copula)
```

Arguments

Copula a list, as returned by [fitCopula](#)

Value

a function that maps parameters (a vector) to probability density values (scalar)

Examples

```
x<-rnorm(300,mean=1,sd=2)
X<-matrix(x,100,3)
C<-fitCopula(X)
d<-dCopulaPrior(C)
print(d(c(1,2,3)))
print(prod(sapply(c(1,2,3),FUN=dnorm,mean=1,sd=2)))
```

dNormalPrior	<i>Creates the density function of a multivariate normal distribution with independent components</i>
--------------	---

Description

The returned density function takes vectors of the same size as mean and sd. It returns the product of the components' one-dimensional normal distribution, with mean "mean" and standard deviation "sd".

Usage

```
dNormalPrior(mean, sd)
```

Arguments

mean mean of the random variables (a vector)
 sd standard deviation of the random variables (same size vector as mean)

Value

a probability density function on vectors with the same length as mean and sd.

Examples

```
dnp<-dNormalPrior(mean=c(0,1,2),sd=c(1,2,3))
dnp(c(0.5,1.5,2.5))
```

dUniformPrior	<i>Creates a uniform density function</i>
---------------	---

Description

The returned density function takes vectors of the same size as ll and ul. It returns the product of the component's one-dimensional uniform distributions.

Usage

```
dUniformPrior(ll, ul)
```

Arguments

ll	lower limit of the random variables (a vector)
ul	upper limit of the random variables (same size vector as ll)

Value

a probability density function on vectors with the same length as ll and ul.

Examples

```
dup<-dUniformPrior(ll=c(0,1,2),ul=c(1,2,3))
dup(c(0.5,1.5,2.5))
```

experiments	<i>Extracts measured data and simulation experiment instructions</i>
-------------	--

Description

This function accepts the model obtained via model_from_tsv or a similar function. It finds the data tables for this model (if any are present), and finds the simulation instructions to reproduce these data sets using the model.

Usage

```
experiments(m, o = NULL, verbose = getOption("uqsa.verbose", interactive()))
```

Arguments

<code>m</code>	the model (with data), as obtained via <code>model_from_tsv()</code> , or similar.
<code>o</code>	the ode derived from <code>m</code> , only necessary if the experiments need to take conservation laws into account
<code>verbose</code>	logical, whether to print diagnostic messages about the experiments

Details

This function requires that the files the model is stored as contains measurements (data) that can be interpreted fairly easily. Each data file needs columns that are named like the observable quantities listed in the Output table.

If the data is very indirectly related to the model, then we don't interpret the data files themselves and the user needs to write a specialized likelihood function to relate the raw data in the files with something that the model does. In such cases, don't use this function.

The simulation experiments returned here, include model input parameters. Whenever conservation law analysis is performed, the conserved constants are set as input parameters, because the conserved amount can differ between experiments. For this reason the Experiment table is interpreted differently in the presence of conservation laws. Otherwise (no conservation laws), the `o` parameter can be omitted.

The instructions must be organized in a table called Experiment(s).

Value

a list of simulation instructions

Examples

```
f <- uqsa_example("AKAR4")
m <- model_from_tsv(f)
o <- as_ode(m)
ex <- experiments(m,o)
print(names(ex))
print(ex[[1]]$input)
```

fitCopula

Makes a Probability Density Estimator from a Sample

Description

Given a sample (from some probability distribution) this function makes a Copula fit to the source distribution using the VineCopula package. This Copula can be used with the prior distribution functions [rCopulaPrior](#) and [dCopulaPrior](#).

Usage

```
fitCopula(X)
```

Arguments

X sample that characterizes the target distribution (rows)

Value

as list: copula, U, Z, and Y where U are marginal probability samples, Z are cumulative density values for U, and Y are the probability density values of U.

Examples

```
rprior <- rNormalPrior(c(1,2,3),c(4,5,6))
X <- rprior(1000)
C <- fitCopula(X)
rCopula <- rCopulaPrior(C)
Z <- rCopula(1000)
print(norm(cov(X) - cov(Z),"2")/norm(X,"2"))
print(abs(sum(colMeans(X) - colMeans(Z))/sum(colMeans(X))))
```

formulae	<i>Finds a column that contains some kind of mathematic expression in a data.frame</i>
----------	--

Description

Given a data.frame that should contain a column that assigns a math expression to a name (in row names), this function returns a named character vector with the expressions. The formula column should be named "formula" (if it exists, only this column will be used). But some other spellings will also work as fallback. As a fallback "value" is acceptable as well, because it makes sense to say "the value of x is 'y/2+1'", even though it is not an atomic value (but an expression).

Usage

```
formulae(df)
```

Arguments

df a data.frame with a "formula" column

Value

character vector with names taken from the row names of df

Examples

```
df <- data.frame(formula=c("exp(x)", "10^x", "2*x + 3"), row.names=c("f1", "f2", 'f3'))
formulae(df)
df <- data.frame(value=c("exp(x)", "10^x", "2*x + 3"), row.names=c("f1", "f2", 'f3'))
formulae(df)
```

generate_code	<i>Constructs Code</i>
---------------	------------------------

Description

Interpret the first argument and generate code in the specified language for the model type.

Usage

```
generate_code(Model, language = "C", LV = 602214076)
```

Arguments

Model	either CME or ODE model
language	either C or R
LV	Avogadro's constant multiplied by the system's volume in litres, only used for CME models

Details

Whenever the model Model is of type "cme", the LV parameter is used to determine the actual number of molecules in the system. Otherwise it is ignored.

Value

a character vector with the code

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
o <- as_ode(m)
C <- generate_code(o)
cat(head(C), sep="\n")
```

gNormalPrior	<i>Creates the gradient function of a multivariate normal distribution with independent components, in log-space</i>
--------------	--

Description

The returned density function takes vectors of the same size as mean and sd. It returns the gradient of the logarithm of the multivariate normal distribution, with mean mean and standard deviation sd.

Usage

```
gNormalPrior(mean, sd)
```

Arguments

mean	mean of the random variables (a vector)
sd	standard deviation of the random variables (same size vector as mean)

Value

a probability density function on vectors with the same length as mean and sd.

Examples

```
gnp <- gNormalPrior(mean=c(0,1,2),sd=c(1,2,3))
gnp(c(0.5,1.5,2.5))
```

gsa_binning	<i>Performs global sensitivity analysis using the binning method</i>
-------------	--

Description

This function performs a binning based estimation of the global sensitivity of a model's output with respect to the model's parameters. The output can be a prediction of the model's behavior in a scenario of interest (parameters, input, initial values, boundary conditions, scheduled events etc.). The output models a potentially measurable value (the "observable"). The sample-rows and the output rows must correspond (they must be from the same model simulation).

Usage

```
gsa_binning(parSample, outputSample, nBins = "Sturges")
```

Arguments

parSample	a matrix of parameter vectors (rows)
outputSample	a matrix, with rows of outputs (row-index is the sample index)
nBins	number of bins, if unset defaults to the default of the hist function

Value

sensitivity $S[i, j]$ of output $[i]$ with respect to parameter $[j]$

Examples

```
rprior <- rNormalPrior(c(-1,0,1),c(1,2,3))
X <- rprior(10000)
colnames(X) <- LETTERS[seq(3)]
Z <- exp(X[,1,drop=FALSE]+X[,2,drop=FALSE])
colnames(Z) <- "alpha"
GSA <- gsa_binning(X,Z)
print(GSA)
cat("global sensitivity of alpha with respect to B: ",GSA['alpha','B'],"\n")
```

gsa_saltelli	<i>Outputs the global sensitivity scores SI and SIT, calculated by the Sobol-Homma-Saltelli method</i>
--------------	--

Description

M1, M2, and N are matrices prepared by `uqsa::saltelli_prior()`. The parameters (rows) from these matrices need to be simulated (using any method), to obtain `fM1`, `fM2` and `fN`.

Usage

```
gsa_saltelli(fM1, fM2, fN, subtract.mean = TRUE)
```

Arguments

<code>fM1</code>	output (f)unction values for M1, $n_S \times n_O$
<code>fM2</code>	output (f)unction values for M2, $n_S \times n_O$
<code>fN</code>	output (f)unction values for N, $n_S \times n_O \times n_P$
<code>subtract.mean</code>	whether or not to subtract the column-means from all matrices/arrays

Details

These matrices are shaped similarly to M1, M2 and N respectively, but now the parameters are replaced by the effects they have on a observable of interest (the output). It can be the vector valued output at a specific (single) time-point or a scalar output at different time-points.

See Geir Halnes et al. (Halnes, Geir, et al. J. comp. neuroscience 27.3 (2009): 471.

Value

a list with sensitivity indices `$SI` and total sensitivities `$SIT`

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
o <- write_and_compile(as_ode(m))
ex <- experiments(m,o)
s <- simulator.c(ex[1],o)
p0 <- values(m$Parameter)
rprior <- rUniformPrior(p0/2,p0*2)
SP <- saltelli_prior(700,rprior)
fM1 <- t(s(t(SP$M1))[[1]]$func[1,,])
fM2 <- t(s(t(SP$M2))[[1]]$func[1,,])
fN <- lapply(asplit(SP$N,3),\N) t(s(t(N))[[1]]$func[1,,])
fN <- simplify2array(fN)
GSA <- gsa_saltelli(fM1,fM2,fN)
print(names(GSA))
cat(
  "average relative senitivity S(p1) / S(p2): ",
```

```
mean(abs(GSA$SI[,1]/GSA$SI[,2]),na.rm=TRUE)
)
```

gsl_odeiv2_CRNN	<i>Simulates a CRNN ode model</i>
-----------------	-----------------------------------

Description

This function calls a C function which solves an initial value problem, derived from a CRNN.

Usage

```
gsl_odeiv2_CRNN(
  name,
  experiments,
  l,
  nu,
  m,
  abs.tol = 1e-06,
  rel.tol = 1e-05,
  initial.step.size = 0.001,
  method = 0,
  time.out = 1,
  nstep = 0
)
```

Arguments

name	either the name of a file (shared library file) or the name of an ODE model to simulate (a shared library of the same name will be dynamically loaded and needs to be created first). If the name of the model is given, then the so file must have the same name in the current directory or a comment indicates its location.
experiments	a list of N simulation experiments (time, parameters, initial value, events).
l	a matrix of parameters with M columns, in log-space.
nu	a stoichiometry matrix ($N \times R$) where N is the number of state variables and R the number of reactions, all reactions are assumed to be reversible.
m	modifiers – similar to stoichiometry, but indicates whether the species takes part in the reaction without being consumed.
abs.tol	absolute tolerance, real scalar.
rel.tol	relative tolerance, real scalar.
initial.step.size	initial value for the step size; the step size will adapt to a value that observes the tolerances, real scalar.
method	one of the integration methods bundled with GSL (see method and name_method).
time.out	time limit in seconds, checked at every measurement time-point (in the data).
nstep	maximum number of ODE integrator steps, checked at every step, defaults to unlimited (0).

Value

a list of the solution trajectories $y(t;p)$ for all experiments (named like the experiments), as well as the output functions.

Examples

```
f <- uqsa_example("AKAR4")
m <- model_from_tsv(f)
ex <- experiments(m,as_ode(m,cla=FALSE))
nu <- stoichiometric_matrix(m)
l <- matrix(c(log(values(m$Parameter)),0),2,2,dimnames=list(rownames(m$Reaction),c("fwd","bwd")))
C <- CRNN(NCOL(nu),initialValues=values(m$Compound),funcValues=formulae(m$Output))
c.file <- tempfile("AKAR4_",fileext=".c")
cat(C,file=c.file,sep='\n')
so.file <- shlib(c.file)
y <- gsl_odeiv2_CRNN(so.file,ex,l,nu,nu*0)
```

gsl_odeiv2_fi

Simulates an ode model

Description

This function calls a C function which solves an initial value problem, calculates the sensitivity of the solution, log-likelihood value ll , gradient of ll and Fisher-Information.

Usage

```
gsl_odeiv2_fi(
  odeModel,
  experiments,
  p,
  abs.tol = 1e-06,
  rel.tol = 1e-05,
  initial.step.size = 0.001,
  method = 0,
  omit = 0,
  time.out = 1,
  num.steps = 0
)
```

Arguments

odeModel	the name of the ODE model to simulate (a shared library of the same name will be dynamically loaded and needs to be created first). Alternatively this can be the ode object created by as_ode , with a shared library path attached to it.
experiments	a list of N simulation experiments (time, parameters, initial value, events)
p	a matrix of parameters with M columns

<code>abs.tol</code>	absolute tolerance, real scalar
<code>rel.tol</code>	relative tolerance, real scalar
<code>initial.step.size</code>	initial value for the step size; the step size will adapt to a value that observes the tolerances, real scalar
<code>method</code>	integration method (see method and name_method).
<code>omit</code>	an integer that indicates how many of these to omit in this order: fisher information, gradient of the log-likelihood, log-likelihood
<code>time.out</code>	in seconds (early rejection due to long simulation time). This can trigger at measurement times (<code>outputTime</code>).
<code>num.steps</code>	maximum number of steps the integration method is permitted to do; early rejection. This condition can trigger at any point during the integration.

Details

The model is always simulated using a shared library. The path to the shared library can be passed in three different ways:

1. Character vector: `odeModel <- c("AKAR4", "/tmp/path/AKAR4.so")`
2. A comment: `comment(odeModel) <- "/tmp/path/AKAR4.so"`
3. As part of the ode object:

```
odeModel <- as_ode(m)~
so_path(odeModel) <- "/tmp/path/AKAR4.so"
```

The shared library needs to be created first. Either with R CMD SHLIB, [shlib](#), or manually on the system's command line (bash, zsh, etc.).

Value

a list of the solution trajectories $y(t;p)$ for all experiments (named like the experiments), as well as the output functions

Examples

```
requireNamespace("errors")
f <- uqsa_example("AKAR4")
m <- model_from_tsv(f)
o <- as_ode(m)
ex <- experiments(m,o)
C <- generate_code(o)
c_path(o) <- write_c_code(C)
so_path(o) <- shlib(o)
print(o)
y <- gsl_odeiv2_fi(o,ex,values(m$Parameter))
if (interactive()){
  print(length(y))
  print(names(y[[1]]))
}
```

```

oldpar <- par(mfrow=c(length(ex),1))
for (i in seq_along(y)){
  plot(
    errors::as.errors(ex[[i]]$outputTimes),
    ex[[i]]$data,
    xlab="time",
    ylab=rownames(y[[i]]$data)[1],
    main=names(ex)[i],
    ylim=c(100,200)
  )
  lines(ex[[i]]$outputTimes,drop(y[[i]]$func),col='red')
}
par(oldpar)
}

```

`high_level_metropolis` *Determines a likely Metropolis setup based on the information in the model*

Description

This function uses default assumption everywhere and returns a function that will sample from the given model. This function will generate code, compile the code, create an ODE solver for it, infer the sampling space from the scale of the parameters, create all necessary functions to move in parameter space (gradients of likelihood and prior), as well as Fisher Information functions.

Usage

```

high_level_metropolis(
  m,
  o = as_ode(m, cla = FALSE),
  ex = experiments(m, o),
  x = values(m$Parameter),
  beta = 1,
  verbose = getOption("uqsa.verbose", interactive())
)

```

Arguments

<code>m</code>	the model's TSV representation read via <code>model_from_tsv</code>
<code>o</code>	(optional) ode representation of <code>m</code>
<code>ex</code>	experiments of <code>m</code> , with simulation instructions for <code>o</code> .
<code>x</code>	initial point of the Markov chain, pre in initialized to have the right attributes.
<code>beta</code>	for parallel tempering, the log-likelihood will have a factor of <code>beta</code> applied to it
<code>verbose</code>	prints extra messages when TRUE

Value

smmala a function of three arguments: p0, N, eps; where p0 is the starting point, N is the desired sample-size, and eps is the step size. This function has an attribute called "init", with a pre-initialized starting point.

Examples

```
m <- model_from_tsv(uqsa_example("AKAP79"))
rwm <- high_level_metropolis(m) # "random walk", metropolis algorithm
p <- rwm %%% "init"           # a valid starting point
N <- 100
if (interactive()){
  smallSample <- rwm(rwm %%% "init",N,1e-6)
  plot(
    smallSample %%% "logLikelihood",
    type="l",
    main=sprintf("%i iterations",N),
    xlab="iterations",
    ylab="log-likelihood"
  )
} else {
  smallSample <- rwm(rwm %%% "init",N/4,1e-6)
}
```

high_level_smmala	<i>Determines a likely SMMALA setup based on the information in the model</i>
-------------------	---

Description

This function uses default assumption everywhere and returns a function that will sample from the given model m. This function will generate code, compile the code, create an ODE solver for it, infer the sampling space from the scale of the parameters, create all necessary functions to move in parameter space (gradients of likelihood and prior), as well as Fisher Information functions.

Usage

```
high_level_smmala(
  m,
  o = as_ode(m, cla = TRUE),
  ex = experiments(m, o),
  x = values(m$Parameter),
  verbose = getOption("uqsa.verbose", interactive())
)
```

Arguments

m	the model's TSV representation read via model_from_tsv
o	(optional) ode representation of m
ex	experiments of m, with simulation instructions for o.
x	initial point of the Markov chain, pre in initialized to have the right attributes.
verbose	prints extra messages when TRUE

Value

smmala a function of three arguments: p0, N, eps; where p0 is the starting point, N is the desired sample-size, and eps is the step size. This function has an attribute called "init", with a pre-initialized starting point.

Examples

```
m <- model_from_tsv(uqsa_example("AKAP79"))
rwm <- high_level_smmala(m) # "random walk", metropolis algorithm
p <- rwm %%% "init"         # a valid starting point
N <- 100
if (interactive()){
  smallSample <- rwm(rwm %%% "init",N,1e-4)
  plot(
    smallSample %%% "logLikelihood",
    type='l',
    main=sprintf("%i iterations",N),
    xlab="iterations",
    ylab="log-likelihood"
  )
} else {
  smallSample <- rwm(rwm %%% "init",1,1e-4)
}
```

kinetic_law_matrix	<i>Represents the kinetic laws of the model as a two column matrix</i>
--------------------	--

Description

This function performs a very simplified split of a kinetic law into a forward part and a backward part, if it isn't pre-split in the file.

Usage

```
kinetic_law_matrix(r)
```

Arguments

r	the reaction table (data.frame)
---	---------------------------------

Details

If the data.frame contains separate forward and backward rates these will be returned instead. Instead of using this function, `m$Reaction[,c("fwd","bwd")]` would accomplish a very similar thing.

Value

a character matrix with a forward and backward column

Examples

```
m <- model_from_tsv(uqsa_example("AKAP79")) # not pre-split
print(colnames(m$Reaction))
k <- kinetic_law_matrix(m$Reaction)
print(k)
```

linear_scale

Interprets a character vector as names of logarithms

Description

The values in `x` are possibly given in a logarithmic space. The parameter `str_scale` gives this logarithmic scale (provided in a language agnostic form), by a human. An empty string causes no transformations. Similarly, providing no scale at all causes no transformations.

Usage

```
linear_scale(x, str_scale = attr(x, "scale"))
```

Arguments

<code>x</code>	values
<code>str_scale</code>	character vector

Details

The words in `str_scale` name a logarithm, e.g. "log10". Currently understood scales:

- log10
- log2, ld
- ln, log

Value

a copy of `x`, transformed into linear space

Examples

```
x <- c(1,2,3,1,1,1)
attr(x,"scale") <- c("log10","log2","log","ln","ld","log5")
print(linear_scale(x))
```

logLikelihoodFunc	<i>Constructs a custom log-likelihood function</i>
-------------------	--

Description

This returns a function $f(\text{parMCMC})$, which maps simulation results to log-likelihood values. The experiments are used implicitly. Here parMCMC is the Markov chain variable (vector), with attached simulations (i.e. initialized).

Usage

```
logLikelihoodFunc(experiments, perExpLLF = NULL, simpleUserLLF = NULL)
```

Arguments

<code>experiments</code>	will be compared to the simulation results
<code>perExpLLF</code>	(optional) a user supplied function with the interface <code>perExpLLF(p, s, e)</code> , where p are the parameters, s are the simulations and e are the experiments (with data). Supply this function if some of your experiments need to be normalized by the other experiments (and other complex cases).
<code>simpleUserLLF</code>	(optional) a user supplied function that is used instead of the default sum of $((y-h)/\text{stdv})^2$ terms. The interface is: <code>simpleUserLLF(y, h, stdv, name=NULL)</code> , where each of them is an N-M-matrix where N is the dimensionality of the model output and M the number of data time-points. Here, y is <code>t(experiments[[i]]\$data)</code> and may contain NA values. This function should also accept an optional <i>name</i> argument (this is the name of the experiment this function is currently called for).

Value

`llf(parMCMC)`, a closure (function) of the mcmc-variable: `parMCMC`; returns a scalar log-likelihood value. Alternatively, the user can define such a function: `parMCMC -> log(Likelihood(parMCMC))`, and use that during sampling. A test simulation of p : `y <- simulate(p)` will reveal which values the simulator produces. These values will be attached to p during sampling, as an attribute. `mcmc_init` will attach the same values for the initial Markov chain state. The log-likelihood function can use these attributes.

Examples

```

m <- model_from_tsv(uqsa_example("AKAR4"))
o <- as_ode(m)
c_path(o) <- write_c_code(generate_code(o))
so_path(o) <- shlib(o)
ex <- experiments(m,o)
s <- simulator.c(ex,o,omit=0)
p <- values(m$Parameter)
attr(p,"simulations") <- s(p)
## this function is fairly flexible and accepts some user settings
llf <- logLikelihoodFunc(ex)
print(llf(p))
## this function uses the values from the solver:
print(ll(p))

```

makeObjective	<i>Creates objective functions from ingredients</i>
---------------	---

Description

the returned objective function has only one argument: the ABC variables that shall be mapped to ODE-model parameters.

Usage

```
makeObjective(experiments, simulate, distance = defaultDistance)
```

Arguments

experiments	a list of simulation experiments
simulate	closure that simulates the model
distance	a function that calculates ABC scores (distance between data and simulations)

Details

The user supplied distance function should accept three arguments: distance(SIM, DATA, STDV), all three matrices. SIM is the model output (simulation), DATA is the measured data, while STDV represents the standard error of that measurement. All three have the same size: $N \times M$, where N is the number of observables (outputs), and M is the number of measurement time-points (length of the time-series).

Value

an objective function

Examples

```
f <- uqsa_example("AKAR4")
m <- model_from_tsv(f)
o <- as_ode(m)
ex <- experiments(m,o)
C <- generate_code(o)
c_path(o) <- write_c_code(C)
so_path(o) <- shlib(o)
s <- simulator.c(ex,o)
objFunc <- makeObjective(ex,s)
print(objFunc(values(m$Parameter)))
```

mcmc

Creates a Markov Chain Monte Carlo sampler

Description

This function creates an MCMC function for a given set of experiments. The Markov chains have no communication between them if more than one is created using this mechanism.

Usage

```
mcmc(update, verbose = getOption("uqsa.verbose", interactive()))
```

Arguments

update	and update function
verbose	prints a progress bar when TRUE

Details

The algorithm is entirely determined by the update function. Any intermediate values that updates requires aside from simulation results have to be attributes of the MCMC variable: `parMCMC`.

The update function: `update(parGiven) -> parUpdate` depends only on the given parameters, all other dependencies have to be either implicit (as a closure) or attributes of `parGiven`.

Value

`M(initPar,N)`, a function of initial starting values and number of Markov chain steps

Examples

```
m <- model_from_tsv(uqsa_example("AKAP79"))
rwm <- high_level_metropolis(m) # "random walk", metropolis algorithm
p <- rwm %%% "init"           # a valid starting point
if (interactive()){
  smallSample <- rwm(rwm %%% "init",500,1e-4)
  pairs(smallSample[,seq(6)])
```

```

} else {
  smallSample <- rwm(rwm %%% "init",10,1e-4)
}

```

mcmc_init	<i>Initializes the Markov chain</i>
-----------	-------------------------------------

Description

This function must append all required attributes to the MCMC variable, for the Markov chain to update correctly.

Usage

```

mcmc_init(
  beta,
  parMCMC,
  simulate,
  logLikelihood = ll,
  dprior = function(x) prod(rnorm(x)),
  gradLogLikelihood = NULL,
  gprior = NULL,
  fisherInformation = NULL
)

```

Arguments

beta	inverse temperature for the Markov chain (parallel tempering)
parMCMC	a plain starting value for the Markov chain
simulate	a closure that maps the MCMC variable to simulation results (the simulation experiments are enclosed in this function).
logLikelihood	a function that maps simulations to logLikelihood values
dprior	density of the prior distribution
gradLogLikelihood	the gradient function of the logLikelihood (optional) – only if the algorithm requires it
gprior	the gradient pf the log-prior (for SMMALA and similar algorithms).
fisherInformation	a function that calculates the Fisher Information matrix

Value

the same starting parameter vector, but with attributes.

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
o <- as_ode(m)
p0 <- values(m$Parameter)
c_path(o) <- write_c_code(generate_code(o))
so_path(o) <- shlib(o)
ex <- experiments(m,o)
s <- simfi(ex,o)
dprior <- dNormalPrior(p0,m$Parameter$stdv)
p <- mcmc_init(1.0,p0,s,dprior=dprior)
print(names(attributes(p))) ## now has attributes necessary for MCMC
```

mcmc_mpi

Creates the MPI version of an MCMC sampler

Description

this version of the MCMC function returns a Markov chain closure that assumes that it is being run in an MPI context: R was launched in an MPI context, e.g. using

```
mpirun -H localhost:8 -N 8 Rscript ...
```

and the pbdMPI package is installed. The chains shall communicate using the provided comm object.

Usage

```
mcmc_mpi(
  update,
  comm,
  swapDelay = 0,
  swapFunc = pbdMPI_bcast_reduce_temperatures
)
```

Arguments

update	an update function
comm	an mpi comm which this function will use for send/receive operations
swapDelay	swaps will be attempted every 2*swapDelay+1 iterations deprecated
swapFunc	can be a custom function that does the MPI communication and decides whether or nopt to swap temperatures

Details

This function is intended for use within a parallel tempering approach and MPI. For trivial parallelization (many chains), this is not at all required, only a random number seed for each worker.

It is possible to supply a custom swap function, with the interface:

```
swapFunc <- function(i, B, LL, H, r, comm, cs)
```

where `i` is the current iteration (for round robin rank choices), `B` is the current beta value, `LL` the current log-likelihood (scalar) and `H` the current step-size (scalar); `r`, `comm`, and `cs` are the MPI rank, `comm`, and `comm-size`. The swap function returns a list: `list(B=,LL=,H=)` with the updated values (after swapping) or the old values if the swap was rejected.

Value

an mcmc closure `m(parMCMC,N,eps)` that implicitly uses the supplied update function

Examples

```
## works only in an MPI context (R session started with `mpirun Rscript ...`)
## similar to mcmc without _mpi prefix
## Not run:
  ## prepare the update functions
  pt_mcmc <- mcmc_mpi(update, comm, swapDelay=0, swapFunc=pb(MPI_bcast_reduce_temperatures))

## End(Not run)
```

method	<i>Finds the method id given a named method</i>
--------	---

Description

Given a ODE solver name (from the GSL solver module `odeiv2`), return an integer offset $\{0..10\}$. This integer can be passed as the "method" argument for all ODE simulator functions ([simulator.c](#), [simfi](#))

Usage

```
method(name)
```

Arguments

name	character scalar, name of the method
------	--------------------------------------

Value

an integer that is acceptable to [simfi](#) and [simulator.c](#)

metropolis_update	<i>Performs an MCMC update using the Metropolis rule</i>
-------------------	--

Description

During Markov chain Monte Carlo a given parameter needs to be updated, the model needs to be simulated at the updated point.

Usage

```
metropolis_update(
  simulate,
  logLikelihood = ll,
  dprior = function(x) prod(dnorm(x)),
  Sigma = NULL,
  parAcceptable = function(p) {
    all(is.finite(p))
  }
)
```

Arguments

<code>simulate</code>	a function that simulates the model
<code>logLikelihood</code>	a function that returns the log-likelihood value given the parameter value, with simulations attached to the parameter as an attribute (probably a closure)
<code>dprior</code>	a function that returns the prior density of the given parameter vector
<code>Sigma</code>	the transition kernel's covariance matrix.
<code>parAcceptable</code>	a function that can be used to reject a proposal based on the values of the parameters alone (shortcut to rejection, sans simulation)

Details

Using the simulations, and an acceptance rule, the proposed update is either accepted or rejected.

This function returns a closure `metropolis`, with only `parMCMC` as it's sole argument: `parProposal <- metropolis(parGiven)`

An optional argument to this function is `parAcceptable`, during sampling, when `metropolis` is called as the update function, and `parAcceptable(parProposal)` returns `FALSE`, then `metropolis` shortcuts to `return(parGiven)` without performing simulations.

This function can be used to weed out parameter combinations that would result in obviously non-sensical simulations without wasting CPU-time.

The return value is a function (closure) that operates on mcmc variables, these variables are numeric vectors with some necessary attributes. The attributes record this vector's simulation results, and other derived quantities.

Value

a closure with the arguments: (parGiven, eps=1e-4), where parGiven is the current position of the Markov chain (a numeric vector, with attributes), and eps the current step size.

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
o <- as_ode(m)
c_path(o) <- write_c_code(generate_code(o))
so_path(o) <- shlib(o)
ex <- experiments(m,o)
s <- simulator.c(ex,o,omit=0)
dprior <- dNormalPrior(values(m$Parameter),m$Parameter$stdv)
p <- mcmc_init(1.0,values(m$Parameter),s,ll,dprior)
UP <- metropolis_update(s,ll,dprior=dprior,Sigma=diag(m$Parameter$stdv)^2)
p2 <- UP(p)
## updated value:
print(p2)
print(sum(abs(p2-p)))
```

model_from_tsv

Loads the content from a series of TSV files

Description

The argument can be either a series of tsv file-names, or a directory with tsv files. If it is a directory, all tsv files therein will be used.

Usage

```
model_from_tsv(src = ".")
```

Arguments

src either a vector of files, or a directory with tsv files

Value

a list of data.frames, one per file, named like the files.

name_method	<i>Reverse look-up of method name from key</i>
-------------	--

Description

These are the methods in the gsl library (documented in the official documentation), but in reverse order, as they are approximately ordered by complexity, with more complex methods usually being better (but slower).

Usage

```
name_method(key = seq(0, 10))
```

Arguments

key an integer from 0 to 10 (this is used as an offset in c, for 11 items)

Details

It is therefore a reasonable approach to try methods from the more complex end of the list first and try the next method if the solutions are too slow. But we need to check the accuracy/stability of the result. The mapping between method names and keys:

```
msbdf: 0
msadams: 1
bsimp: 2
rk4imp: 3
rk2imp: 4
rk1imp: 5
rk8pd: 6
rkck: 7
rkf45: 8
rk4: 9
rk2: 10
```

The returned value is an integer index.

Value

a string representation of the integration method.

Examples

```
print(name_method())
```

parse_concise	<i>Reads Concise Error Notation</i>
---------------	-------------------------------------

Description

Convert a vector of strings of the form: `c("1.2(3)E-4","1.2(3)E-2")` to a matrix with two rows:

1. values,
2. uncertainties.

Usage

```
parse_concise(
  v,
  use.errors = requireNamespace("errors", quietly = TRUE),
  na = c(NA, NA)
)
```

Arguments

<code>v</code>	a character vector of numbers in concise error notation
<code>use.errors</code>	if TRUE, the errors package will be used to return an object of type "errors" (from that package). Otherwise, the errors will be attached as an attribute (also called "errors" to be consistent with the errors package)
<code>na</code>	a two element vector which will replace NA values, e.g. <code>c(NA,NA)</code> ; <code>na=c(0,Inf)</code> means <i>infinite uncertainty</i> for missing values

Details

If the errors package is available, then an *errors* object is returned instead (uncertainties are an attribute). In that case the dimensions of `v` are preserved on output. You can override this choice using the second argument `use.errors`.

Concise error notation means that a floating point number is followed by an integer in parentheses which indicates the uncertainty of the last digits of the value:

$$1.2345(12) = 1.2345 \pm 0.0012$$

.

If the errors package is installed, then it will be used to represent the return value.

Value

either a numeric object with class *errors* (with the same dimensions as `v`), or a numeric matrix of values and uncertainties (2 rows), dimensions of original object are lost

Examples

```
x <- parse_concise(c("1.23(4)", "0.51099895069(16)", "1.25663706127(20)e-6", "1.3±1.6", "5;1"))
print(as.data.frame(x))
```

pc_dist	<i>Plots a sample in parallel coordinates</i>
---------	---

Description

This function makes a plot that is quite similar to parallel coordinates. It includes information about the prior as error-bars, centered around the prior's median.

Usage

```
pc_dist(posterior, prior, color = rgb(0.5, 0.5, 0.5, 0.05), ...)
```

Arguments

posterior	a matrix, with N rows (sample-members), and M columns (different model parameters). The columns must be named.
prior	a data.frame with at least \$median, and \$stdv columns. This data.frame may also include the fields: color, and colorOutline to change the prior error-bars.
color	the color of the sample lines, should have some transparency.
...	parameters are passed to matplot.

Value

produces a plot

Examples

```
rprior <- rNormalPrior(c(-1,0,1),c(1,2,3))
A <- matrix(rnorm(9),3,3)
A <- (A + t(A))^2/norm(A)^2
X <- rprior(1000)
Z <- X %*% A
colnames(Z) <- letters[seq(3)]
pr <- data.frame(median=apply(X,2,median),stdv=apply(X,2,sd))
pc_dist(Z,pr)
```

rCopulaPrior	<i>Returns a function that generates random values from the copula model</i>
--------------	--

Description

The returned function generates n random vectors, as rows of a matrix.

Usage

```
rCopulaPrior(Copula)
```

Arguments

Copula the return value of fitCopula()

Value

a matrix of random values

Examples

```
rprior <- rNormalPrior(c(-1,0,1),c(1,2,3))
C <- fitCopula(rprior(1000))
D <- rCopulaPrior(C)
print(cov(D(100)))
print(D(10))
```

replace_powers

Interprets caret notation of arithmetic powers for C

Description

This function takes a string argument with human readable math (e.g. R code), and replaces the power operator z^n with C-compatible function calls: `pow(x,n)`, it counts parentheses to determine the base and exponent automatically.

Usage

```
replace_powers(v)
```

Arguments

v a character vector

Details

This functions assumes that gsl functions can be used, the GNU Scientific Library includes powers of small integers. These functions may be faster than always calling `pow` from `math.h`.

This is necessary because in C the `^` operator means something else (exclusive bitwise xor for integers). No attempt will be made to cast the numbers to float or double.

Value

a string where all occurrences of `^` have been replaced by function calls like `pow()`

Examples

```
print(replace_powers(c("2^3.1", "10^-6", "x^2", "(1+(1+x))^(n-0.5)")))
```

rNormalPrior

Returns a random vector generator

Description

The return value is a function that generates random vectors of the same size as mean and sd from a multivariate normal distribution with independent components with mean "mean" and standard deviation "sd". The random vectors are returned as n rows of a matrix, where n is the only argument of the returned function.

Usage

```
rNormalPrior(mean, sd)
```

Arguments

mean	mean of the random variables (a vector)
sd	standard deviation of the random variables (same size vector as mean)

Value

an independent multivariate normal random vector generating function: rprior(n), where n is the requested number of vectors (rows)

Examples

```
rnp<-rNormalPrior(mean=c(0,1,2),sd=c(1,2,3))
rnp(12)
```

rUniformPrior

Returns a random vector generator

Description

The return value is a function that generates random vectors of the same size as ll and ul from a uniform distribution within the limits defined by ul and ll. The random vectors are returned as n rows of a matrix, where n is the only argument of the returned function.

Usage

```
rUniformPrior(ll, ul)
```

Arguments

ll lower limit of the random variables (a vector)
 ul upper limit of the random variables (same size vector as ll)

Value

a uniform random vector generating function: runiform(n), where n is the requested number of vectors (rows)

Examples

```
rup<-rUniformPrior(ll=c(0,1,2),ul=c(1,2,3))
rup(12)
```

saltelli_prior	<i>Creates prior samples for the Sobol-Homma-Saltelli global sensitivity analysis</i>
----------------	---

Description

Each parameter vector has length nPars, The sample consists of two random (nSamples x nPars) matrices M1, M2 and a third (nSamples x nPars x nPars) array N. N consists of nPars copies of M2, except that in each M2-matrix one column has been replaced by the corresponding column of M1. M1 and M2 consists of random numbers from a normal distribution.

Usage

```
saltelli_prior(nSamples, rprior)
```

Arguments

nSamples number of rows to return
 rprior a function that samples from the prior distribution

Details

These matrices provide prior distribution samples to be further processed by the simulator, similar to this:

```
sim <- simulator.c(experiments,modelName)
fM1 <- t(sim(t(M1))[[1]]$state[,ti,])        # or similar
```

For details see: Halnes, Geir, et al. J. comp. neuroscience 27.3 (2009): 471.

Value

a list with the components M1, M2 (both matrices) and N (a 3D-array).

Examples

```
rprior <- rNormalPrior(c(-1,0,1),c(1,2,3))
SP <- saltelli_prior(1000,rprior)
print(names(SP))
```

scrnn

Returns a closure around gsl_odeiv2_CRNN()

Description

the returned value is a function of a variable `p` that encodes the CRNN in some way. Three user supplied functions are used to extract the three components of a CRNN:

Usage

```
scrnn(
  experiments,
  modelName,
  parMap = function(p) p$l,
  stoichiometry = function(p) p$nu,
  modifiers = function(p) p$m,
  method = 0,
  time.out = 1
)
```

Arguments

<code>experiments</code>	list of experiments (inputs are ignored).
<code>modelName</code>	scalar string, can indicate a shared library with an attached comment attribute.
<code>parMap</code>	(function) extracts kinetic rate coefficients from its argument.
<code>stoichiometry</code>	(function) extracts the stoichiometry matrix from its argument.
<code>modifiers</code>	(function) extracts the modifier matrix from its argument.
<code>method</code>	(integer) integration method key (0:10) corresponds to these GSL methods: ms-bdf, msadams, bsimp, rk4imp, rk2imp, rk1imp, rk8pd, rkck, rkf45, rk4, rk2
<code>time.out</code>	time limit for solution in seconds

Details

- kinetic rate coefficients (in log-space): `l <- parMap(p)`
- stoichiometric matrix: `nu <- stoichiometry(p)`
- modifier matrix: `m <- modifiers(p)`

these three components (one numeric vector, and two matrices) are passed to the simulation procedure. The vector `l` can be a matrix with `M` columns. In that case, one simulation per column is performed. The stoichiometry and modifiers remain unchanged throughout.

Value

closure that maps one argument (p) to simulation results (y).

Examples

```
f <- uqsa_example("AKAR4")
m <- model_from_tsv(f)
ex <- experiments(m)
nu <- stoichiometric_matrix(m)
l <- matrix(
  c(log(values(m$Parameter)), -1e6),
  2, 2,
  byrow=TRUE,
  dimnames=list(rownames(m$Reaction), c("fwd", "bwd")))
)
C <- CRNN(
  NCOL(nu),
  initialValues=values(m$Compound),
  funcValues=formulae(m$Output)
)
c.file <- tempfile("AKAR4", fileext=".c")
cat(C, file=c.file, sep='\n')
modelName <- "CRNN"
comment(modelName) <- shlib(c.file)
s <- scrnn(ex, modelName)
p <- list(l=l, nu=nu, m=nu*0)
y <- s(p)
if (interactive()){
  plot(ex, y)
}
```

sensitivity.graph

Plot the result of global sensitivity analysis

Description

Produce a cumulative shaded area plot for the sensitivity matrix. This function is intended for use with many observables, e.g. the state of the model at several given times. The x-axis of the plot is meant to be continuous. This will not produce a bar-chart, but a graph that shows how sensitivities change between fairly similar observables.

Usage

```
sensitivity.graph(
  u,
  S,
  color = hcl.colors(dim(S)[2]),
  line.color = hcl.colors(dim(S)[2] + 1),
  do.sort = TRUE,
```

```

    decreasing = FALSE,
    ...
)

```

Arguments

<code>u</code>	the values of the x-axis for the plot, if named, the names are put at the tick-marks
<code>S</code>	the sensitivity matrix as returned by <code>globalSensitivity()</code> , <code>S[i,j]</code> is with respect to model output <code>i</code> and parameter <code>j</code>
<code>color</code>	the list of colors to use for the shaded areas, e.g.: <code>rainbow(24)</code>
<code>line.color</code>	the color of the lines drawn between the shaded areas
<code>do.sort</code>	the parameter sensitivities are sorted according to the mean over all outputs, the parameter with the most sensitivity is plotted first, at the bottom
<code>decreasing</code>	direction of sort, the first item in the sorted list (the parameter) will be plotted first, and thus at the bottom of the plot
<code>...</code>	passed on to plot

Value

nothing

Examples

```

rprior <- rNormalPrior(c(-1,0,1),c(1,2,3))
X <- rprior(10000)
colnames(X) <- LETTERS[seq(3)]
Z <- exp(
  cbind(
    rowSums(X),
    rowMeans(X),
    exp(X[,1])
  )
)
colnames(Z) <- c("sum", "mean", "exp1")
GSA <- gsa_binning(X,Z)
print(GSA)
sensitivity.graph(c(sum=1,mean=2,exp1=3),GSA)

```

shlib

Compiles C code to shared library

Description

Calls R CMD SHLIB to create the model's shared library.

Usage

```
shlib(file, verbose = getOption("uqsa.verbose", default = interactive()))
```

Arguments

file	the c file that is to be compiled, OR an ODE/CME object with a c.file defined and recorded in it.
verbose	print decision outcomes about the compiler options

Details

The first argument can be a raw character scalar with just the path of the c code to be compiled, or alternatively an object that has this information stored within it. The models returned by [as_cme](#) and [as_ode](#) can both carry this information, attach it via `c_path<-`.

Value

the path of the created shared library

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
o <- as_ode(m)
C <- generate_code(o)
c_path(o) <- write_c_code(C)
so_path(o) <- shlib(o)
print(o)
if (file.exists(so_path(o))) cat("shared library exists.\n")
```

showPosterior	<i>Makes a pairs plot for a sample using density estimation methods</i>
---------------	---

Description

This function will display the difference between the posterior and prior by plotting the posterior as shaded density plots and the prior as contour lines of level sets. If the two are identical, the lines will be invisible as they blend into the density plot. Otherwise the contour lines will show up as a distinct feature.

Usage

```
showPosterior(posterior, prior, ...)
```

Arguments

posterior	a matrix, each row is a sample member
prior	a matrix of the same size as the posterior
...	passed to <code>graphics::pairs()</code>

Value

pairs plot object

Examples

```
rprior <- rNormalPrior(c(-1,0,1),c(1,2,3))
A <- matrix(rnorm(9),3,3)
A <- (A + t(A))^2/norm(A)^2
X <- rprior(30)
Z <- X %**% A
colnames(Z) <- letters[seq(3)]
colnames(X) <- letters[seq(3)]
## make a plot:
if (interactive()) showPosterior(Z,X) # this can take a while
```

simstoch

Simulates a Stochastic Model

Description

Simulate a stochastic model generated with `uqsa::generateGillespieModel()`, using the solver in this package.

Usage

```
simstoch(ex, cmeModel, parMap = identity, time.out = 1, nstep = 0)
```

Arguments

<code>ex</code>	list of experiments, same as for the deterministic solvers.
<code>cmeModel</code>	Either the <code>cmeModel</code> from as_cme , with a shared library path stored inside, or the path to the <code>so</code> file
<code>parMap</code>	map from MCMC variable (or ABC variable) to model-parameters.
<code>time.out</code>	in seconds
<code>nstep</code>	number of reactions for early exit, defaults to unlimited (0)

Details

This will simulate all experimental conditions included in the list of experiments, including applying the inputs: `u <- experiments[[i]]$input` - the input will be copied to the end of the model's internal parameter vector.

Like for deterministic models, we assume that there is a vector of unknown parameter (a Markov chain variable, a vector of optimization variables) and also known parameters (aka the input parameters). The model itself does not distinguish between the two, but one is the same between the experiments and one is different between different experiments: `modelParam <- c(mcmcParam, inputParam)`

The path to the shared library, is required to contain at least one slash in it, e.g.: `"/.model.so"`, `"/tmp/Rsdljhsjdhf/model.so"` But, not just `"model.so"`, otherwise the shared library is interpreted as a system library by `dlopen()` (it will not be found).

The number of reactions can be limited by `nstep` (micro time-steps forward). The maximum can be set by performing a good simulation and reading out the number of steps taken in that reference simulation: `y[[i]]$numSteps`.

Value

a closure that simulates the model in `model.so`

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
cme <- as_cme(m)
C <- generate_code(cme)
c_path(cme) <- write_c_code(C)
so_path(cme) <- shlib(cme)
ex <- experiments(m)
p0 <- values(m$Parameter)
s <- simstoch(ex,cme)
res <- s(p0)
require(errors)
plot(as.errors(ex[[1]]$outputTimes),ex[[1]]$data,xlab="time",ylab="AKAR4p",main=names(ex)[1])
lines(ex[[1]]$outputTimes,res[[1]]$func,type="s",lwd=2,col="red3")
```

simulator.c

Creates a closure that simulates the model

Description

This function will use the [parallel::mclapply](#) to do the simulations simultaneously. Set `options(mc.cores=detectCores())` or a similar sensible value: `options(mc.cores=length(experiments))`

Usage

```
simulator.c(
  experiments,
  modelName,
  parMap = identity,
  noise = FALSE,
  omit = 3,
  method = 0,
  time.out = 1,
  num.steps = 0
)
```

Arguments

<code>experiments</code>	a list of experiments to simulate: initial values, inputs, time vectors, initial times
<code>modelName</code>	a string (with optional comment indicating an <code>.so</code> file) which points out the model to simulate if <code>modelName</code> is a <code>cme</code> object, the simulation will be done stochastically
<code>parMap</code>	the model will be called with <code>parMap(parABC)</code> ; so any parameter transformation can happen there.

noise	boolean variable. If noise=TRUE, Gaussian noise is added to the output of the simulations. The standard deviation of the Gaussian noise is equal to the measurement error. If noise=FALSE the output is the deterministic solution of the ODE system. noise and sensitivity calculations are mutually exclusive.
omit	omit=0 returns all optional return values from the simulator, omit=1 will not calculate the fisher information (and thus not return it), omit=2 will omit the gradient of the log-likelihood, and omit=3 will omit the likelihood calculations altogether. Omission is cumulative: omit=3 omits all the previously mentioned optional quantities.
method	an integer offset, integration method (for ODE models), see method and name_method
time.out	in seconds, for early stops.
num.steps	maximum number of steps taken by the integrator (in the case of ODEs), or maximum number of total reaction-steps performed by the Gillespie algorithm (over time) for stochastic models.

Details

It returns a closure around: - experiments, - the model, and - parameter mapping

The returned function depends only on the parameter vector (or matrix if more than one simulation per experiment is desired). The parameter vector this simulator accepts is probably derived from the sampling space of a Bayesian method θ , so in the list of arguments, it is called parABC or (parMCMC would also have been a valid choice). These sampling parameters can be mapped to values the simulator can use via parMap. parModel <- parMap(parABC), where the ODE model is expected to work with parModel. The model can be specified by name (with a comment indicating a file location)

Some return values are optional and omitting them saves time.

Value

a closure that returns the model's output for a given parameter vector

Examples

```
requireNamespace("errors")
f <- uqsa_example("AKAR4")
m <- model_from_tsv(f)
o <- as_ode(m)
ex <- experiments(m,o)
C <- generate_code(o)
c_path(o) <- write_c_code(C)
so_path(o) <- shlib(o)
s <- simulator.c(ex,o)
y <- s(values(m$Parameter))
```

small	<i>Reduces the sample to its effective size</i>
-------	---

Description

When plotting, we want to show only a few representative lines or points derived from a sample. This function will determine the auto-correlation length very roughly and use that number to thin out the sample to a minimal size that still represents the original sample well.

Usage

```
small(
  S,
  L = attr(S, "logLikelihood"),
  verbose = getOption("uqsa.verbose", default = interactive())
)
```

Arguments

S	an MCMC sample
L	the log-likelihood values of S
verbose	when TRUE the acf plot option is set to TRUE, and the found auto-correlation length is printed.

Value

a smaller version of S

Examples

```
S <- matrix(rnorm(300),100,3)
## the next line fakes auto-correlation:
attr(S,"logLikelihood") <- cos(seq(0,1,length.out=100)) + rnorm(100,sd=0.05)
print(dim(S))
print(dim(small(S)))
```

smmala_update	<i>Performs an SMMALA update</i>
---------------	----------------------------------

Description

During Markov chain Monte Carlo a given parameter needs to be updated, the model needs to be simulated at the updated point.

Usage

```

smmala_update(
  simulate,
  logLikelihood = ll,
  dprior = function(x) prod(dnorm(x)),
  gradLogLikelihood = gllf(log10ParMapJac),
  gprior = function(x) (-x),
  fisherInformation = fi(log10ParMapJac),
  fisherInformationPrior = 0,
  parAcceptable = function(p) all(is.finite(p))
)

```

Arguments

<code>simulate</code>	a function that simulates the model
<code>logLikelihood</code>	a function that returns the log-likelihood value given the parameter value, with simulations attached to the parameter as an attribute (probably a closure)
<code>dprior</code>	a function that returns the prior density of the given parameter vector
<code>gradLogLikelihood</code>	any function that calculates or estimates the gradient of the log-likelihood function, for the chosen parameter mapping. Function must take one argument (the MCMC variable)
<code>gprior</code>	a function that returns the gradient of the log-prior distribution.
<code>fisherInformation</code>	a function that estimates the Fisher Information for a given MCMC variable (<code>parMCMC</code>).
<code>fisherInformationPrior</code>	a constant <code>fisherInformation</code> of the prior distribution (or rather, the precision of the prior)
<code>parAcceptable</code>	a function that can be used to reject a proposal based on the values of the parameters alone (shortcut to rejection, sans simulation)

Details

Using the simulations, and an acceptance rule, the proposed update is either accepted or rejected.

This function returns a closure `smmala`, with only `parMCMC` as its sole argument: `parProposal <- smmala(parGiven)`

An optional argument to this function is `parAcceptable`, during sampling, when `metropolis` is called as the update function, and `parAcceptable(parProposal)` returns `FALSE`, then `metropolis` shortcuts to `return(parGiven)` without performing simulations.

This function can be used to weed out parameter combinations that would result in obviously non-sensical simulations without wasting CPU-time.

The argument `fisherInformationPrior` is really the precision of the prior (a constant matrix). Its role is additive to the `fisherInformation` and is used to regularize the *final* Fisher Information Matrix (makes it invertible).

Value

a closure with the arguments: (parGiven, eps=1e-4), where parGiven is the current position of the Markov chain (a numeric vector, with attributes), and eps the current step size.

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
o <- as_ode(m)
c_path(o) <- write_c_code(generate_code(o))
so_path(o) <- shlib(o)
ex <- experiments(m,o)
s <- simulator.c(ex,o,omit=0)
### without parameter transformations
gll <- gllf()
FI <- fi()
dprior <- dNormalPrior(values(m$Parameter),m$Parameter$stdv)
gprior <- dNormalPrior(values(m$Parameter),m$Parameter$stdv)
p <- mcmc_init(1.0,values(m$Parameter),s,ll,dprior,gll,gprior,FI)
UP <- smmala_update(s,ll,dprior=dprior,gll,gprior=gprior,FI,solve(diag(m$Parameter$stdv)))
p2 <- UP(p)
## updated value:
print(p2)
print(sum(abs(p2-p)))
```

so_path

*Retrieves information about compiled code***Description**

Returns the path of the shared library (.so file). The model is typically a list of named arrays and matrices.

Usage

```
so_path(o)
```

Arguments

o the ODE, or CME model

Value

```
modified o, with information about compiled code
m <- model_from_tsv(uqsa_example("AKAR4"))
o <- as_ode(m) c_path(o) <- write_c_code(generate_code(o)) so_path(o) <- shlib(o) print(so_path(o))
```

<code>so_path<-</code>	<i>Adds information about compiled code</i>
---------------------------	---

Description

Adds the path of the shared library (.so file) to the ODE model.

Usage

```
so_path(o) <- value
```

Arguments

<code>o</code>	the ODE (list of named arrays and matrices), or CME model
<code>value</code>	the path to the compiled model

Value

modified `o`, with information about compiled code

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
o <- as_ode(m)
c_path(o) <- write_c_code(generate_code(o))
so_path(o) <- shlib(o)
print(o)
```

<code>standard_error_matrix</code>	<i>Standard Error Matrix from an errors object</i>
------------------------------------	--

Description

If a matrix has an `errors` attribute, it is usually a vector. This function returns the values of this attribute as a matrix (it preserves the dimensions of the host matrix).

Usage

```
standard_error_matrix(M)
```

Arguments

<code>M</code>	a matrix with errors (uncertainties)
----------------	--------------------------------------

Value

A matrix similar to `E`, with standard error values

Examples

```
M <- matrix(seq(12),3,4,dimnames=list(letters[seq(3)],LETTERS[seq(4)]))
errors::errors(M) <- abs(M*0.1 + 0.1)
E <- standard_error_matrix(M)
print(E)
```

stoichiometric_matrix *Infers the stoichiometric matrix of a reaction network*

Description

Given a model, described in tabular form (*m* is a list of data-frames). The stoichiometric matrix is the linear map between the model's flux vector and the ODE's right-hand-side vector field. If the flux vector is `rr <- flux(t,x,p)`, which maps the state variables *x* and parameters *p* to the reaction rate *rr* of each reaction. The stoichiometric matrix *nu* (ν), will map the reaction rates to the rate of change of the state variables: $dx/dt := nu \%*\% flux(t,x,p)$.

Usage

```
stoichiometric_matrix(m, compound.names = rownames(m$Compound))
```

Arguments

m list of data frames with at least the 'Reaction' table, and the 'Compound' table
compound.names all names of the reacting compounds

Details

The matrix is usually sparse, but not extremely big. This function attaches a sparse version of the same information as attributes to the return-value, as two lists, for convenience.

Value

the stoichiometric matrix, with some additional attributes.

Examples

```
the_reaction <- "A + B <=> C"
m <- list(
  Reaction=data.frame(reactants=c("A+B"),products=c("C"))
)
nu <- stoichiometric_matrix(m,c("A","B","C"))
```

stoichiometry	<i>Returns a list of named stoichiometric vectors</i>
---------------	---

Description

Given an already split list of entries such as `c("3 A","B")`, this function returns a numeric vector `c(3,1)` with names `c("A","B")`.

Usage

```
stoichiometry(formulaList)
```

Arguments

`formulaList` reaction formulae, either as a pre-split list or character vector

Value

named numeric vector of stoichiometric coefficients

Examples

```
m <- model_from_tsv(uqsa_example("AKAP79"))
r <- stoichiometry(m$Reaction$reactants)
print(head(r))
print(tail(r))
```

tune_step_size	<i>Finds a good step-size for a given MCMC algorithm</i>
----------------	--

Description

Given a closure `MCMC(p,N,eps)`, where `p` is the initial Markov-chain position, `N` a sample-size, and `eps` a step-size, this function finds a good value for `eps`.

Usage

```
tune_step_size(
  MCMC,
  parMCMC = attr(MCMC, "init"),
  target_acceptance = 0.25,
  iter.max = 6,
  h = 1e-04,
  N = 100,
  verbose = getOption("uqsa.verbose", interactive())
)
```

Arguments

MCMC	a Markov chain Monte Carlo closure (function)
parMCMC	initial position of the Markov chain, has to be initialized with mcmc_init .
target_acceptance	a scalar value for the desired acceptance rate, some algorithms are most efficient with 20% to 30% acceptance, some work well with a very high acceptance.
iter.max	maximum number of iterations until the function has to return.
h	initial guess for the MCMC step size
N	size of test-samples for acceptance rate estimate
verbose	when TRUE, this function prints a progress bar, 'a:' reports current acceptance rate, and 'h:' reports the current step-size.

Details

It will take 100 sample points repeatedly, until an acceptance of `target_acceptance` is reached (defaults to 25%). The step-size is decreased if acceptance is very low and increased when it is too high.

When verbose This function will do at most

Value

optimal step size

Examples

```

opt <- options(mc.cores=2)
m <- model_from_tsv(uqsa_example("AKAP79"))
rwm <- high_level_metropolis(m) # "random walk", metropolis algorithm
p <- rwm %%% "init"           # a valid starting point
N <- 100
if (interactive()){
  h <- tune_step_size(rwm,p)
  smallSample <- rwm(rwm %%% "init",N,h)
  print(h)
  plot(
    smallSample %%% "logLikelihood",
    type="l",
    main=sprintf("step size: %g",h),
    xlab="iterations",
    ylab="log-likelihood"
  )
} else {
  h <- tune_step_size(rwm,p,N=20,iter.max=1)
}
options(opt)

```

`uncertainty`*Finds the uncertainty of values in a data.frame*

Description

given a data.frame, this function will look for a column that contains some kind of standard error and retrieve it. The returned numeric vector will be named. This function is not intended for data, for data, the `values` function will retrieve both the value and the standard error if it was specified.

Usage

```
uncertainty(df)
```

Arguments

`df` a data frame with a "value" column

Details

This function is for the case that the table specifies a distribution with a mean and an range (of some sort). The type of uncertainty found will be attached as a comment to the returned value: "sd" standard deviation for normal distribution, "se" standard error (for a normal prior), and "range" for a uniform prior. Other priors are not recognized yet.

The distinction between standard-error and standard-deviation doesn't matter much here: either the value is some kind of mean and the *uncertainty* is the standard-error or standard-deviation of the mean, or it is a raw data-point (not averaged) and we know the standard deviation (noise) of the device that measured it, then *uncertainty* is the standard deviation of the noise distribution. In either case, the value will be taken at face value and the uncertainty is used as sigma in the default log-likelihood function.

Any entry of prior.distribution other than "uniform", will start a search for some kind of standard deviation or standard error (or sigma). As more priors are added, this function will look for the parameters of those distributions.

This function makes many assumptions specifically that all variables in the table have the same type of prior distribution (but not identically distributed).

Value

a named numeric vector

unit.from.string	<i>Interprets a unit string</i>
------------------	---------------------------------

Description

This function will try its best to interpret strings like "liter/(nmol ms)" rules: 1. only one slash is allowed 2. M can be mega or mol/l: writing M for molarity will treat molarity as it's own unit kind; writing "molarity" will be translated into two SI units (mol and litre) 3. prefixes and units can be words or single letters 4. everything after a slash is the denominator 5. u is an accepted replacement for *mu* (unicode Greek mu or unicode micro symbol) 6. no parentheses (ignored): "(m/s)*kg" will be misinterpreted

Usage

```
unit.from.string(unit.str)
```

Arguments

`unit.str` a string that contains a human readable unit

Details

this returns a data.frame with components as in the sbml standard: kind, multiplier, scale and exponent since there is only one slash, parentheses do nothing everything after a slash is the denominator, so: l/mol s is the same as (l)/(mol s) Remark: not all units are understood.

Value

data.frame with an interpretation of the unit (multiplier is unused here, but may be used later to deal with units such as hours (kind=second, multiplier=60)

Examples

```
print(unit.from.string("m/s"))
print(unit.from.string("micromolarity"))
print(unit.from.string("µM"))
```

uqsa_example

Loads an example model for this package

Description

This function finds the path to an example model, given by name. In the SBtab format, model and data travel together (in different tables, but the same documents).

Usage

```
uqsa_example(
  modelName = NULL,
  full.names = TRUE,
  pattern = "[.]tsv$",
  f = NULL
)
```

Arguments

modelName	name of model, e.g.: "AKAR4", "AKAP79", "CaMKII"; if empty, this function lists all available examples.
full.names	return full paths to files - defaults to TRUE
pattern	pattern to find specific files; if NULL, this function returns the directory of the example
f	file ending, search for file endings in f, alternative to pattern

Details

By default this function returns the names of the tsv files belonging to the named model. If no modelName is provided it returns possible names (contents of the top-level example directory).

Value

The location of the examples in the current environment if called with no arguments, the paths to the model files if a modelName was provided or the full path to the example if the file pattern *pattern* is unset

Examples

```
uqsa_example()
uqsa_example("AKAR4", full.names=FALSE)
uqsa_example("AKAP79", f='R', full.names=FALSE)
uqsa_example("AKAP79", pat="^run.*R$")
```

values	<i>Finds the values in a data.frame</i>
--------	---

Description

given a data.frame, this function will look for a column that contains some kind of value and retrieve it. The returned numeric vector will be named.

Usage

```
values(df)
```

Arguments

df a data frame with a "value" column

Details

If the values contain a standard error, the returned value is of class *errors*.

Value

a named numeric vector

write_and_compile	<i>Writes code to a file and compiles</i>
-------------------	---

Description

This function accepts an ode model, or cme model, generates code, compiles it to a shared library, and returns a changed object. possibly changed by the user. It writes the contents to a c file named 'modelName_gvf.c'. This file is compiled to './modelName.so' using normal command line tools, not R CMD SHLIB

Usage

```
write_and_compile(M)
```

Arguments

M ode or cme Model for which code is generated and written to a file

Details

This entire function can be replaced with a call to cat() and then compiling the written file in the system's shell.

Value

a copy of `o` with file paths added to it

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
o <- write_and_compile(as_ode(m))
print(o)
```

write_c_code	<i>Writes C code to a file</i>
--------------	--------------------------------

Description

This function does not compile the code, it only writes it to a file in a temporary location (`tempdir`). By default, the name of the file will contain the hash of the entire code.

Usage

```
write_c_code(C, model.name = comment(C), file = NULL)
```

Arguments

<code>C</code>	the code to write, as a character array.
<code>model.name</code>	a string with no special characters, will be used in the file name
<code>file</code>	override the default file name (based on hashing)

Details

If instead of a character vector, an `ode` or `cme` object is passed, this function will generate code from it with default options.

Value

the path of the written file

Examples

```
m <- model_from_tsv(uqsa_example("AKAR4"))
o <- as_ode(m)
C <- generate_code(o)
c_path(o) <- write_c_code(C)
print(o)
if (file.exists(c_path(o))) cat("c file exists.\n")
```

Index

* ODE

gsl_odeiv2_CRNN, [19](#)
gsl_odeiv2_fi, [20](#)

abc_mcmc, [3](#)
ABCSMC, [6](#)
as_cme, [7](#), [43](#), [44](#)
as_ode, [8](#), [20](#), [43](#)

c_path, [9](#)
c_path<-, [8](#), [10](#), [43](#)
column, [10](#)
CRNN, [11](#)

dCopulaPrior, [12](#), [14](#)
deprecated, [30](#)
dNormalPrior, [12](#)
dUniformPrior, [13](#)

experiments, [13](#)

fitCopula, [12](#), [14](#)
formulae, [15](#)

generate_code, [16](#)
gNormalPrior, [16](#)
gsa_binning, [17](#)
gsa_saltelli, [18](#)
gsl_odeiv2_CRNN, [19](#)
gsl_odeiv2_fi, [20](#)

high_level_metropolis, [22](#)
high_level_smmala, [23](#)

kinetic_law_matrix, [24](#)

linear_scale, [25](#)
logLikelihoodFunc, [26](#)

makeObjective, [27](#)
mcmc, [28](#)

mcmc_init, [26](#), [29](#), [53](#)
mcmc_mpi, [30](#)
method, [19](#), [21](#), [31](#), [46](#)
metropolis_update, [32](#)
model_from_tsv, [8](#), [24](#), [33](#)

name_method, [19](#), [21](#), [34](#), [46](#)

parallel::mclapply, [45](#)
parse_concise, [35](#)
pc_dist, [36](#)

rCopulaPrior, [14](#), [36](#)
replace_powers, [37](#)
rNormalPrior, [38](#)
rUniformPrior, [38](#)

saltelli_prior, [39](#)
scrnn, [40](#)
sensitivity.graph, [41](#)
shlib, [21](#), [42](#)
showPosterior, [43](#)
simfi, [31](#)
simstoch, [44](#)
simulator.c, [31](#), [45](#)
small, [47](#)

smmala_update, [47](#)
so_path, [49](#)
so_path<-, [8](#), [50](#)
standard_error_matrix, [50](#)
stoichiometric_matrix, [51](#)
stoichiometry, [52](#)

tune_step_size, [52](#)

uncertainty, [54](#)
unit.from.string, [55](#)
uqsa_example, [56](#)

values, [54](#), [57](#)

write_and_compile, [57](#)
write_c_code, [58](#)