Package 'TMB'

March 5, 2018

Type Package

Title Template Model Builder: A General Random Effect Tool Inspired by 'ADMB'

Version 1.7.12

Date 2017-12-10

Maintainer Kasper Kristensen <kaskr@dtu.dk>

Author Kasper Kristensen [aut, cre, cph],
Brad Bell [cph],
Hans Skaug [ctb],
Arni Magnusson [ctb],
Casper Berg [ctb],
Anders Nielsen [ctb],
Martin Maechler [ctb],
Theo Michelot [ctb],
Mollie Brooks [ctb],
Alex Forrence [ctb],
Christoffer Moesgaard Albertsen [ctb],
Cole Monnahan [ctb]

Description With this tool, a user should be able to quickly implement complex random effect models through simple C++ templates. The package combines 'CppAD' (C++ automatic differentiation), 'Eigen' (templated matrix-vector library) and 'CHOLMOD' (sparse matrix routines available from R) to obtain an efficient implementation of the applied Laplace approximation with exact derivatives. Key features are: Automatic sparseness detection, parallelism through 'BLAS' and parallel user templates.

License GPL-2

URL http://tmb-project.org

BugReports https://github.com/kaskr/adcomp/issues

Depends R (>= 3.0.0)

Imports graphics,
methods,
stats,

utils,
Matrix (>= 1.0-12)

LinkingTo Matrix

Suggests numDeriv,
parallel

RoxygenNote 5.0.1

R topics documented:

`as.list.sdreport` *Convert estimates to original list format.*

Description

Get estimated parameters or standard errors in the same shape as the original parameter list.

Usage

```
## S3 method for class 'sdreport'
as.list(x, what = "", ...)
```

Arguments

x	Output from sdreport.
what	Select what to convert.
...	Passed to summary.sdreport.

Details

This function converts the selected column what of summary(x, select = c("fixed", "random"), ...) to the same format as the original parameter list (re-ordered as the template parameter order). The argument what is partially matched among the column names of the summary table. The actual match is added as an attribute to the output.

Value

List of same shape as original parameter list.

Examples

```
## Not run:
example(sdreport)
as.list(rep, "Est")
as.list(rep, "Std")
as.list(rep, "Pr", p.value=TRUE)

## End(Not run)
```

---

benchmark *Benchmark parallel templates*

Description

Benchmark parallel templates

Plot result of parallel benchmark

Usage

```
benchmark(obj, n = 10, expr = NULL, cores = NULL)

## S3 method for class 'parallelBenchmark'
plot(x, type = "b", ..., show = c("speedup",
  "time"), legendpos = "topleft")
```

Arguments

obj	Object from MakeADFun
n	Number of replicates to obtain reliable results.
expr	Optional expression to benchmark instead of default.
cores	Optional vector of cores.
x	Object to plot
type	Plot type
...	Further plot arguments
show	Plot relative speedup or relative time?
legendpos	Position of legend

Details

By default this function will perform timings of the most critical parts of an AD model, specifically

1. Objective function of evaluated template.
2. Gradient of evaluated template.
3. Sparse hessian of evaluated template.
4. Cholesky factorization of sparse hessian.

(for pure fixed effect models only the first two). Expressions to time can be overwritten by the user (expr). A plot method is available for Parallel benchmarks.

Examples

```
## Not run:
runExample("linreg_parallel",thisR=TRUE)  ## Create obj
ben <- benchmark(obj,n=100,cores=1:4)
plot(ben)
ben <- benchmark(obj,n=10,cores=1:4,expr=expression(do.call("optim",obj)))
plot(ben)

## End(Not run)
```

---

checkConsistency *Check consistency and Laplace accuracy*

Description

Check consistency of various parts of a TMB implementation. Requires that user has implemented simulation code for the data and optionally random effects. (*Beta version; may change without notice*)

Usage

```
checkConsistency(obj, par = NULL, hessian = FALSE, n = 100)
```

Arguments

obj	Object from MakeADFun
par	Parameter vector (θ) for simulation. If unspecified use the best encountered parameter of the object.
hessian	Calculate the hessian matrix for each replicate ?
n	Number of simulations

Details

This function checks that the simulation code of random effects and data is consistent with the implemented negative log-likelihood function. It also checks whether the approximate *marginal* score function is central indicating whether the Laplace approximation is suitable for parameter estimation.

Denote by u the random effects, θ the parameters and by x the data. The main assumption is that the user has implemented the joint negative log likelihood $f_\theta(u, x)$ satisfying

$$\int \int \exp(-f_\theta(u, x))\, du\, dx = 1$$

It follows that the joint and marginal score functions are central:

1. $E_{u,x}\left[\nabla_\theta f_\theta(u, x)\right] = 0$
2. $E_x\left[\nabla_\theta - \log\left(\int \exp(-f_\theta(u, x))\, du\right)\right] = 0$

For each replicate of u and x joint and marginal gradients are calculated. Appropriate centrality tests are carried out by summary.checkConsistency. An asymptotic χ^2 test is used to verify the first identity. Power of this test increases with the number of simulations n. The second identity holds *approximately* when replacing the marginal likelihood with its Laplace approximation. A formal test would thus fail eventually for large n. Rather, the gradient bias is transformed to parameter scale (using the estimated information matrix) to provide an estimate of parameter bias caused by the Laplace approximation.

Value

List with gradient simulations (joint and marginal)

See Also

summary.checkConsistency, print.checkConsistency

Examples

```
## Not run:
runExample("simple")
chk <- checkConsistency(obj)
chk
## Get more details
s <- summary(chk)
s$marginal$p.value  ## Laplace exact for Gaussian models
## End(Not run)
```

`compile` *Compile a C++ template to DLL suitable for MakeADFun.*

Description

Compile a C++ template into a shared object file. OpenMP flag is set if the template is detected to be parallel.

Usage

```
compile(file, flags = "", safebounds = TRUE, safeunload = TRUE,
  openmp = isParallelTemplate(file[1]), libtmb = TRUE, libinit = TRUE,
  tracesweep = FALSE, ...)
```

Arguments

`file`	C++ file.
`flags`	Character with compile flags.
`safebounds`	Turn on preprocessor flag for bound checking?
`safeunload`	Turn on preprocessor flag for safe DLL unloading?
`openmp`	Turn on openmp flag? Auto detected for parallel templates.
`libtmb`	Use precompiled TMB library if available (to speed up compilation)?
`libinit`	Turn on preprocessor flag to register native routines?
`tracesweep`	Turn on preprocessor flag to trace AD sweeps? (Silently disables `libtmb`)
`...`	Passed as Makeconf variables.

Details

TMB relies on R's built in functionality to create shared libraries independent of the platform. A template is compiled by `compile("template.cpp")`, which will call R's makefile with appropriate preprocessor flags. Compiler and compiler flags can be stored in a configuration file. In order of precedence either via the file pointed at by R_MAKEVARS_USER or the file ~/.R/Makevars if it exists. Additional configuration variables can be set with the `flags` and `...` arguments, which will override any previous selections.

See Also

`precompile`

config *Get or set internal configuration variables*

Description

Get or set internal configuration variables of user's DLL.

Usage

```
config(..., DLL = getUserDLL())
```

Arguments

...	Variables to set
DLL	Name of user's DLL. Auto-detected if missing.

Details

A model compiled with the TMB C++ library has several configuration variables set by default. The variables can be read and modified using this function. The meaning of the variables can be found in the Doxygen documentation.

Value

List with current configuration

Examples

```
## Not run:
## Load library
dyn.load(dynlib("mymodel"))
## Read the current settings
config(DLL="mymodel")
## Reduce memory peak of a parallel model by creating tapes in serial
config(tape.parallel=0, DLL="mymodel")
obj <- MakeADFun(..., DLL="mymodel")

## End(Not run)
```

confint.tmbprofile *Profile based confidence intervals.*

Description

Calculate confidence interval from a likelihood profile.

Usage

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

Arguments

object	Output from tmbprofile.
parm	Not used
level	Confidence level.
...	Not used

Value

Lower and upper limit as a matrix.

dynlib *Add dynlib extension*

Description

Add the platform dependent dynlib extension. In order for examples to work across platforms DLLs should be loaded by `dyn.load(dynlib("name"))`.

Usage

```
dynlib(name)
```

Arguments

name	Library name without extension

Value

Character

gdbsource *Source R-script through gdb to get backtrace.*

Description

Source R-script through gdb to get backtrace.

If gdbsource is run non-interactively (the default) only the relevant information will be printed. Note that this will only work if the cpp file and the R file share the same base name.

Usage

```
gdbsource(file, interactive = FALSE)

## S3 method for class 'backtrace'
print(x, ...)
```

Arguments

file	Your R script
interactive	Run interactive gdb session?
x	Backtrace from gdbsource
...	Not used

Details

This function is useful for debugging templates. If a script aborts e.g. due to an out-of-bound index operation it should be fast to locate the line that caused the problem by running gdbsource(file). Alternatively, If more detailed debugging is required, then gdbsource(file,TRUE) will provide the full backtrace followed by an interactive gdb session where the individual frames can be inspected. Note that templates should be compiled without optimization and with debug information in order to provide correct line numbers:

- On Linux/OS X use compile(cppfile,"-O0 -g").
- On Windows use compile(cppfile,"-O1 -g",DLLFLAGS="") (lower optimization level will cause errors).

Value

Object of class backtrace

MakeADFun *Construct objective functions with derivatives based on a compiled C++ template.*

Description

Construct objective functions with derivatives based on the users C++ template.

Usage

```
MakeADFun(data, parameters, map = list(), type = c("ADFun", "Fun",
  "ADGrad"[!is.null(random) || !is.null(profile)]), random = NULL,
  profile = NULL, random.start = expression(last.par.best[random]),
  hessian = FALSE, method = "BFGS", inner.method = "newton",
  inner.control = list(maxit = 1000), MCcontrol = list(doMC = FALSE, seed =
  123, n = 100), ADreport = FALSE, atomic = TRUE,
  LaplaceNonZeroGradient = FALSE, DLL = getUserDLL(),
  checkParameterOrder = TRUE, regexp = FALSE, silent = FALSE, ...)
```

Arguments

data	List of data objects (vectors,matrices,arrays,factors,sparse matrices) required by the user template (order does not matter and un-used components are allowed).
parameters	List of all parameter objects required by the user template (both random and fixed effects).
map	List defining how to optionally collect and fix parameters - see details.
type	Character vector defining which operation stacks are generated from the users template - see details.
random	Character vector defining the random effect parameters. See also regexp.
profile	Parameters to profile out of the likelihood (this subset will be appended to random with Laplace approximation disabled).
random.start	Expression defining the strategy for choosing random effect initial values as function of previous function evaluations - see details.
hessian	Calculate Hessian at optimum?
method	Outer optimization method.
inner.method	Inner optimization method (see function "newton").
inner.control	List controlling inner optimization.
MCcontrol	List controlling importance sampler (turned off by default).
ADreport	Calculate derivatives of macro ADREPORT(vector) instead of objective_function return value?
atomic	Allow tape to contain atomic functions?
LaplaceNonZeroGradient	Allow Taylor expansion around non-stationary point?

DLL	Name of shared object file compiled by user (without the conventional extension, '.so', '.dll', ...).
checkParameterOrder	Optional check for correct parameter order.
regexp	Match random effects by regular expressions?
silent	Disable all tracing information?
...	Currently unused.

Details

A call to MakeADFun will return an object that, based on the users DLL code (specified through DLL), contains functions to calculate the objective function and its gradient. The object contains the following components:

- par A default parameter.
- fn The likelihood function.
- gr The gradient function.
- report A function to report all variables reported with the REPORT() macro in the user template.
- env Environment with access to all parts of the structure.

and is thus ready for a call to an R optimizer, such as nlminb or optim. Data (data) and parameters (parameters) are directly read by the user template via the macros beginning with DATA_ and PARAMETER_. The order of the PARAMETER_ macros defines the order of parameters in the final objective function. There are no restrictions on the order of random parameters, fixed parameters or data in the template.

Optionally, a simple mechanism for collecting and fixing parameters from R is available through the map argument. A map is a named list of factors with the following properties:

- names(map) is a subset of names(parameters).
- For a parameter "p" length(map$p) equals length(parameters$p).
- Parameter entries with NAs in the factor are fixed.
- Parameter entries with equal factor level are collected to a common value.

More advanced parameter mapping, such as collecting parameters between different vectors etc., must be implemented from the template.

Random effects are specified via the argument random: A component of the parameter list is marked as random if its name is matched by any of the characters of the vector random (Regular expression match is performed if regexp=TRUE). If some parameters are specified as random effects, these will be integrated out of the objective function via the Laplace approximation. In this situation the functions fn and gr automatically perform an optimization of random effects for each function evaluation. This is referred to as the 'inner optimization'. Strategies for choosing initial values of the inner optimization can be controlled via the argument random.start. The default is expression(last.par.best[random]) where last.par.best is an internal full parameter vector corresponding to the currently best likelihood. An alternative choice could be expression(last.par[random]) i.e. the random effect optimum of the most recent - not necessarily best - likelihood evaluation. Further control of the inner optimization can be obtained by

the argument inner.control which is a list of control parameters for the inner optimizer newton. Depending of the inner optimization problem type the following settings are recommended:

1. Quasi-convex: smartsearch=TRUE (the default).
2. Strictly-convex: smartsearch=FALSE and maxit=20.
3. Quadratic: smartsearch=FALSE and maxit=1.

Technically, the user template is processed several times by inserting different types as template parameter, selected by argument type:

- "ADFun" Run through the template with AD-types and produce a stack of operations representing the objective function.
- "Fun" Run through the template with ordinary double-types.
- "ADGrad" Run through the template with nested AD-types and produce a stack of operations representing the objective function gradient.

Each of these are represented by external pointers to C++ structures available in the environment env.

Further objects in the environment env:

- validpar Function defining the valid parameter region (by default no restrictions). If an invalid parameter is inserted fn immediately return NaN.
- parList Function to get the full parameter vector of random and fixed effects in a convenient list format.
- random An index vector of random effect positions in the full parameter vector.
- last.par Full parameter of the latest likelihood evaluation.
- last.par.best Full parameter of the best likelihood evaluation.
- tracepar Trace every likelihood evaluation ?
- tracemgc Trace maximum gradient component of every gradient evaluation ?
- silent Pass 'silent=TRUE' to all try-calls ?

A high level of tracing information will be output by default when evaluating the objective function and gradient. This is useful while developing a model, but may eventually become annoying. Disable all tracing by passing silent=TRUE to the MakeADFun call.

Value

List with components (fn, gr, etc) suitable for calling an R optimizer, such as nlminb or optim.

`newton` *Generalized newton optimizer.*

Description

Generalized newton optimizer used for the inner optimization problem.

Usage

```
newton(par, fn, gr, he, trace = 1, maxit = 100, tol = 1e-08, alpha = 1,
  smartsearch = TRUE, mgcmax = 1e+60, super = TRUE, silent = TRUE,
  ustep = 1, power = 0.5, u0 = 1e-04, grad.tol = tol, step.tol = tol,
  tol10 = 0.001, env = environment(), ...)
```

Arguments

`par`	Initial parameter.
`fn`	Objective function.
`gr`	Gradient function.
`he`	Sparse hessian function.
`trace`	Print tracing information?
`maxit`	Maximum number of iterations.
`tol`	Convergence tolerance.
`alpha`	Newton stepsize in the fixed stepsize case.
`smartsearch`	Turn on adaptive stepsize algorithm for non-convex problems?
`mgcmax`	Refuse to optimize if the maximum gradient component is too steep.
`super`	Supernodal Cholesky?
`silent`	Be silent?
`ustep`	Adaptive stepsize initial guess between 0 and 1.
`power`	Parameter controlling adaptive stepsize.
`u0`	Parameter controlling adaptive stepsize.
`grad.tol`	Gradient convergence tolerance.
`step.tol`	Stepsize convergence tolerance.
`tol10`	Try to exit if last 10 iterations not improved more than this.
`env`	Environment for cached Cholesky factor.
`...`	Currently unused.

Details

If `smartsearch=FALSE` this function performs an ordinary newton optimization on the function `fn` using an exact sparse hessian function. A fixed stepsize may be controlled by `alpha` so that the iterations are given by:

$$u_{n+1} = u_n - \alpha f''(u_n)^{-1} f'(u_n)$$

If `smartsearch=TRUE` the hessian is allowed to become negative definite preventing ordinary newton iterations. In this situation the newton iterations are performed on a modified objective function defined by adding a quadratic penalty around the expansion point u_0:

$$f_t(u) = f(u) + \frac{t}{2}\|u - u_0\|^2$$

This function's hessian ($f''(u) + tI$) is positive definite for t sufficiently large. The value t is updated at every iteration: If the hessian is positive definite t is decreased, otherwise increased. Detailed control of the update process can be obtained with the arguments `ustep`, `power` and `u0`.

Value

List with solution similar to `optim` output.

See Also

`newtonOption`

`newtonOption` *Set newton options for a model object.*

Description

Inner-problem options can be set for a model object using this function.

Usage

```
newtonOption(obj, ...)
```

Arguments

`obj`	Object from `MakeADFun` for which to change settings.
`...`	Parameters for the `newton` optimizer to set.

Value

List of updated parameters.

normalize *Normalize process likelihood using the Laplace approximation.*

Description

If the random effect likelihood contribution of a model has been implemented without proper normalization (i.e. lacks the normalizing constant), then this function can perform the adjustment automatically. In order for this to work, the model must include a flag that disables the data term so that the un-normalized random effect (negative log) density is returned from the model template. Automatic process normalization may be useful if either the normalizing constant is difficult to implement, or if its calulation involves so many operations that it becomes infeasible to include in the AD machinery.

Usage

```
normalize(obj, flag, value = 0)
```

Arguments

obj	Model object from MakeADFun without proper normalization of the random effect likelihood.
flag	Flag to disable the data term from the model.
value	Value of 'flag' that signifies to not include the data term.

Value

Modified model object that can be passed to an optimizer.

oneStepPredict *Calculate one-step-ahead (OSA) residuals for a latent variable model.*

Description

Calculate one-step-ahead (OSA) residuals for a latent variable model. (*Beta version; may change without notice*)

Usage

```
oneStepPredict(obj, observation.name = NULL, data.term.indicator = NULL,
  method = c("oneStepGaussianOffMode", "fullGaussian", "oneStepGeneric",
  "oneStepGaussian", "cdf"), subset = NULL, conditional = NULL,
  discrete = NULL, discreteSupport = NULL, range = c(-Inf, Inf),
  seed = 123, parallel = FALSE, trace = TRUE, ...)
```

Arguments

`obj`	Output from `MakeADFun`.
`observation.name`	Character naming the observation in the template.
`data.term.indicator`	Character naming an indicator data variable in the template (not required by all methods - see details).
`method`	Method to calculate OSA (see details).
`subset`	Index vector of observations that will be added one by one during OSA. By default `1:length(observations)` (with `conditional` subtracted).
`conditional`	Index vector of observations that are fixed during OSA. By default the empty set.
`discrete`	Are observations discrete? (assumed FALSE by default)
`discreteSupport`	Possible outcomes of discrete distribution (`method="oneStepGeneric"` only).
`range`	Possible range of the observations.
`seed`	Randomization seed (discrete case only). If `NULL` the RNG seed is untouched by this routine.
`parallel`	Run in parallel using the `parallel` package?
`trace`	Trace progress?
`...`	Control parameters for OSA method

Details

Given a TMB latent variable model this function calculates OSA standardized residuals that can be used for goodness-of-fit assessment. The approach is based on a factorization of the joint distribution of the *observations* $X_1, ..., X_n$ into successive conditional distributions. Denote by

$$F_n(x_n) = P(X_n \leq x_n | X_1 = x_1, ..., X_{n-1} = x_{n-1})$$

the one-step-ahead CDF, and by

$$p_n(x_n) = P(X_n = x_n | X_1 = x_1, ..., X_{n-1} = x_{n-1})$$

the corresponding point probabilities (zero for continuous distributions). In case of continuous observations the sequence

$$\Phi^{-1}(F_1(X_1)), ..., \Phi^{-1}(F_n(X_n))$$

will be iid standard normal. These are referred to as the OSA residuals. In case of discrete observations draw (unit) uniform variables $U_1, ..., U_n$ and construct the randomized OSA residuals

$$\Phi^{-1}(F_1(X_1) - U_1 p_1(X_1)), ..., \Phi^{-1}(F_n(X_n) - U_n p_n(X_n))$$

These are also iid standard normal.

The user must specify one of the following methods to calculate the residuals:

method="fullGaussian" This method assumes that the joint distribution of data *and* random effects is Gaussian (or well approximated by a Gaussian). It does not require any changes to the user template. However, if used in conjunction with `subset` and/or `conditional` a `data.term.indicator` is required - see the next method.

method="oneStepGeneric" This method calculates the one-step conditional probability density as a ratio of Laplace approximations. The approximation is integrated (and re-normalized for improved accuracy) using 1D numerical quadrature to obtain the one-step CDF evaluated at each data point. The method works in the continuous case as well as the discrete case (`discrete=TRUE`).

It requires a specification of a `data.term.indicator` explained in the following. Suppose the template for the observations given the random effects (u) looks like

```
DATA_VECTOR(x);
...
nll -= dnorm(x(i), u(i), sd(i), true);
...
```

Then this template can be augmented with a `data.term.indicator = "keep"` by changing the template to

```
DATA_VECTOR(x);
DATA_VECTOR_INDICATOR(keep, x);
...
nll -= keep(i) * dnorm(x(i), u(i), sd(i), true);
...
```

The new data vector (`keep`) need not be passed from R. It automatically becomes a copy of `x` filled with ones.

method="oneStepGaussian" This is a special case of the generic method where the one step conditional distribution is approximated by a Gaussian (and can therefore be handled more efficiently).

method="oneStepGaussianOffMode" This is an approximation of the "oneStepGaussian" method that avoids locating the mode of the one-step conditional density.

method="cdf" The generic method can be slow due to the many function evaluations used during the 1D integration (or summation in the discrete case). The present method can speed up this process but requires more changes to the user template. The above template must be expanded with information about how to calculate the negative log of the lower and upper CDF:

```
DATA_VECTOR(x);
DATA_VECTOR_INDICATOR(keep, x);
...
nll -= keep(i) * dnorm(x(i), u(i), 0.0, true);
nll -= keep.cdf_lower(i) * log( pnorm(x(i), u(i), 0.0) );
nll -= keep.cdf_upper(i) * log( 1.0 - pnorm(x(i), u(i), 0.0) );
...
```

The specialized members `keep.cdf_lower` and `keep.cdf_upper` automatically become copies of `x` filled with zeros.

Value

data.frame with OSA *standardized* residuals in column residual. Depending on the method the output may also include OSA expected observation in column mean.

Examples

```
######################## Gaussian case
runExample("simple")
osa.simple <- oneStepPredict(obj, observation.name = "x", method="fullGaussian")
qqnorm(osa.simple$residual); abline(0,1)

## Not run:
######################## Poisson case (First 100 observations)
runExample("ar1xar1")
osa.ar1xar1 <- oneStepPredict(obj, "N", "keep", method="cdf", discrete=TRUE, subset=1:100)
qqnorm(osa.ar1xar1$residual); abline(0,1)

## End(Not run)
```

---

openmp *Control number of openmp threads.*

Description

Control number of openmp threads.

Usage

```
openmp(n = NULL)
```

Arguments

n	Requested number of threads, or NULL to just read the current value.

Value

Number of threads.

plot.tmbprofile *Plot likelihood profile.*

Description

Plot (negative log) likelihood profile with confidence interval added.

Usage

```
## S3 method for class 'tmbprofile'
plot(x, type = "l", level = 0.95, ...)
```

Arguments

x	Output from tmbprofile.
type	Plot type.
level	Add horizontal and vertical lines depicting this confidence level (NULL disables the lines).
...	Additional plot arguments.

precompile *Precompile the TMB library in order to speed up compilation of templates.*

Description

Precompile the TMB library

Usage

```
precompile(all = TRUE, clean = FALSE, trace = TRUE, ...)
```

Arguments

all	Precompile all or just the core parts of TMB ?
clean	Remove precompiled libraries ?
trace	Trace precompilation process ?
...	Not used.

Details

Precompilation can be used to speed up compilation of templates. It is only necessary to run `precompile()` once, typically right after installation of TMB. The function *prepares* TMB for precompilation, while the actual pre-compilation takes place the first time you compile a model after running `precompile()`.

Note that the precompilation requires write access to the TMB package folder. Three versions of the library will be prepared: Normal, parallel and a debugable version.

Precompilation works the same way on all platforms. The only known side-effect of precompilation is that it increases the file size of the generated binaries.

Examples

```
## Not run:
## Prepare precompilation
precompile()
## Perform precompilation by running a model
runExample(all = TRUE)

## End(Not run)
```

---

print.checkConsistency

Print output from checkConsistency

Description

Print diagnostics output from checkConsistency

Usage

```
## S3 method for class 'checkConsistency'
print(x, ...)
```

Arguments

x	Output from checkConsistency
...	Not used

print.sdreport *Print brief model summary*

Description

Print parameter estimates and give convergence diagnostic based on gradient and Hessian.

Usage

```
## S3 method for class 'sdreport'
print(x, ...)
```

Arguments

x	Output from sdreport
...	Not used

Rinterface *Create minimal R-code corresponding to a cpp template.*

Description

Create a skeleton of required R-code once the cpp template is ready.

Usage

```
Rinterface(file)
```

Arguments

file	cpp template file.

Examples

```
file <- system.file("examples/simple.cpp", package = "TMB")
Rinterface(file)
```

runExample *Run one of the test examples.*

Description

Compile and run a test example (`runExample()` shows all available examples).

Usage

```
runExample(name = NULL, all = FALSE, thisR = TRUE, clean = FALSE,
  exfolder = NULL, dontrun = FALSE, subarch = TRUE, ...)
```

Arguments

`name`	Character name of example.
`all`	Run all the test examples?
`thisR`	Run inside this R?
`clean`	Cleanup before compile?
`exfolder`	Alternative folder with examples.
`dontrun`	Build only (don't run) and remove temporary object files ?
`subarch`	Build in sub-architecture specific folder ?
`...`	Passed to `compile`.

runSymbolicAnalysis *Run symbolic analysis on sparse Hessian*

Description

Aggressively tries to reduce fill-in of sparse Cholesky factor by running a full suite of ordering algorithms. NOTE: requires a specialized installation of the package. More information is available at the package URL.

Usage

```
runSymbolicAnalysis(obj)
```

Arguments

`obj`	Output from `MakeADFun`

sdreport *General sdreport function.*

Description

After optimization of an AD model, sdreport is used to calculate standard deviations of all model parameters, including non linear functions of random effects and parameters specified through the ADREPORT() macro from the user template.

Usage

```
sdreport(obj, par.fixed = NULL, hessian.fixed = NULL,
  getJointPrecision = FALSE, bias.correct = FALSE,
  bias.correct.control = list(sd = FALSE, split = NULL, nsplit = NULL),
  ignore.parm.uncertainty = FALSE, getReportCovariance = TRUE,
  skip.delta.method = FALSE)
```

Arguments

obj	Object returned by MakeADFun
par.fixed	Optional. Parameter estimate (will be known to obj when an optimization has been carried out).
hessian.fixed	Optional. Hessian wrt. parameters (will be calculated from obj if missing).
getJointPrecision	Optional. Return full joint precision matrix of random effects and parameters?
bias.correct	logical indicating if bias correction should be applied
bias.correct.control	a list of bias correction options; currently sd, split and nsplit are used - see details.
ignore.parm.uncertainty	Optional. Ignore estimation variance of parameters?
getReportCovariance	Get full covariance matrix of ADREPORTed variables?
skip.delta.method	Skip the delta method? (FALSE by default)

Details

First, the Hessian wrt. the parameter vector (θ) is calculated. The parameter covariance matrix is approximated by

$$V(\hat{\theta}) = -\nabla^2 l(\hat{\theta})^{-1}$$

where l denotes the log likelihood function (i.e. -obj$fn). If ignore.parm.uncertainty=TRUE then the Hessian calculation is omitted and a zero-matrix is used in place of $V(\hat{\theta})$.

For non-random effect models the standard delta-method is used to calculate the covariance matrix of transformed parameters. Let $\phi(\theta)$ denote some non-linear function of θ. Then

$$V(\phi(\hat{\theta})) \approx \nabla\phi V(\hat{\theta})\nabla\phi'$$

The covariance matrix of reported variables $V(\phi(\hat{\theta}))$ is returned by default. This can cause high memory usage if many variables are ADREPORTed. Use `getReportCovariance=FALSE` to only return standard errors. In case standard deviations are not required one can completely skip the delta method using `skip.delta.method=TRUE`.

For random effect models a generalized delta-method is used. First the joint covariance of random effects and parameters is estimated by

$$V\begin{pmatrix}\hat{u}\\ \hat{\theta}\end{pmatrix} \approx \begin{pmatrix}H_{uu}^{-1} & 0\\ 0 & 0\end{pmatrix} + JV(\hat{\theta})J'$$

where H_{uu} denotes random effect block of the full joint Hessian of `obj$env$f` and J denotes the Jacobian of $\begin{pmatrix}\hat{u}(\theta)\\ \theta\end{pmatrix}$ wrt. θ. Here, the first term represents the expected conditional variance given the parameters and the second term represents the variance of the conditional mean wrt. the parameters.

Now the delta method can be applied on a general non-linear function $\phi(u,\theta)$ of random effects u and parameters θ:

$$V(\phi(\hat{u},\hat{\theta})) \approx \nabla\phi V\begin{pmatrix}\hat{u}\\ \hat{\theta}\end{pmatrix}\nabla\phi'$$

The full joint covariance is not returned by default, because it may require large amounts of memory. It may be obtained by specifying `getJointPrecision=TRUE`, in which case $V\begin{pmatrix}\hat{u}\\ \hat{\theta}\end{pmatrix}^{-1}$ will be part of the output. This matrix must be manually inverted using `solve(jointPrecision)` in order to get the joint covariance matrix. Note, that the parameter order will follow the original order (i.e. `obj$env$par`).

Using $\phi(\hat{u},\theta)$ as estimator of $\phi(u,\theta)$ may result in substantial bias. This may be the case if either ϕ is non-linear or if the distribution of u given x (data) is sufficiently non-symmetric. A generic correction is enabled with `bias.correct=TRUE`. It is based on the identity

$$E_\theta[\phi(u,\theta)|x] = \partial_\varepsilon\left(\log\int \exp(-f(u,\theta)+\varepsilon\phi(u,\theta))\,du\right)_{|\varepsilon=0}$$

stating that the conditional expectation can be written as a marginal likelihood gradient wrt. a nuisance parameter ε. The marginal likelihood is replaced by its Laplace approximation.

If `bias.correct.control$sd=TRUE` the variance of the estimator is calculated using

$$V_\theta[\phi(u,\theta)|x] = \partial_\varepsilon^2\left(\log\int \exp(-f(u,\theta)+\varepsilon\phi(u,\theta))\,du\right)_{|\varepsilon=0}$$

A further correction is added to this variance to account for the effect of replacing θ by the MLE $\hat{\theta}$ (unless `ignore.parm.uncertainty=TRUE`).

Bias correction can be be performed in chunks in order to reduce memory usage or in order to only bias correct a subset of variables. First option is to pass a list of indices as `bias.correct.control$split`.

E.g. a list `list(1:2,3:4)` calculates the first four ADREPORTed variables in two chunks. Second option is to pass the number of chunks as `bias.correct.control$nsplit` in which case all ADREPORTed variables are bias corrected in the specified number of chunks. Also note that `skip.delta.method` may be necessary when bias correcting a large number of variables.

Value

Object of class `sdreport`

See Also

`summary.sdreport`, `print.sdreport`, `as.list.sdreport`

Examples

```
## Not run:
runExample("linreg_parallel", thisR = TRUE) ## Non-random effect example
sdreport(obj)
## End(Not run)

runExample("simple", thisR = TRUE)          ## Random effect example
rep <- sdreport(obj)
summary(rep, "random")                      ## Only random effects
summary(rep, "fixed", p.value = TRUE)       ## Only non-random effects
summary(rep, "report")                      ## Only report

## Bias correction
rep <- sdreport(obj, bias.correct = TRUE)
summary(rep, "report")                      ## Include bias correction
```

---

summary.checkConsistency

Summarize output from `checkConsistency`

Description

Summarize output from `checkConsistency`

Usage

```
## S3 method for class 'checkConsistency'
summary(object, ...)
```

Arguments

`object`	Output from `checkConsistency`
`...`	Not used

Value

List of diagnostics

`summary.sdreport` *summary tables of model parameters*

Description

Extract parameters, random effects and reported variables along with uncertainties and optionally Chi-square statistics. Bias corrected quantities are added as additional columns if available.

Usage

```
## S3 method for class 'sdreport'
summary(object, select = c("all", "fixed", "random",
  "report"), p.value = FALSE, ...)
```

Arguments

`object`	Output from `sdreport`
`select`	Parameter classes to select. Can be any subset of `"fixed"` ($\hat{\theta}$), `"random"` ($\hat{u}$) or `"report"` ($\phi(\hat{u}, \hat{\theta})$) using notation as `sdreport`.
`p.value`	Add column with approximate p-values
`...`	Not used

Value

matrix

`template` *Create cpp template to get started.*

Description

Create a cpp template to get started.

Usage

```
template(file = NULL)
```

Arguments

`file`	Optional name of cpp file.

Details

This function generates a C++ template with a header and include statement. Here is a brief overview of the C++ syntax used to code the objective function. For a full reference see the Doxygen documentation (more information at the package URL).

Macros to read data and declare parameters:

Template Syntax	C++ type	R type
DATA_VECTOR(name)	vector<Type>	vector
DATA_MATRIX(name)	matrix<Type>	matrix
DATA_SCALAR(name)	Type	numeric(1)
DATA_INTEGER(name)	int	integer(1)
DATA_FACTOR(name)	vector<int>	factor
DATA_IVECTOR(name)	vector<int>	integer
DATA_SPARSE_MATRIX(name)	Eigen::SparseMatrix<Type>	dgTMatrix
DATA_ARRAY(name)	array<Type>	array
PARAMETER_MATRIX(name)	matrix<Type>	matrix
PARAMETER_VECTOR(name)	vector<Type>	vector
PARAMETER_ARRAY(name)	array<Type>	array
PARAMETER(name)	Type	numeric(1)

Basic calculations:

Template Syntax	Explanation
REPORT(x)	Report x back to R
ADREPORT(x)	Report x back to R with derivatives
vector<Type> v(n1);	R equivalent of v=numeric(n1)
matrix<Type> m(n1,n2);	R equivalent of m=matrix(0,n1,n2)
array<Type> a(n1,n2,n3);	R equivalent of a=array(0,c(n1,n2,n3))
v+v,v-v,v*v,v/v	Pointwise binary operations
m*v	Matrix-vector multiply
a.col(i)	R equivalent of a[,,i]
a.col(i).col(j)	R equivalent of a[,j,i]
a(i,j,k)	R equivalent of a[i,j,k]
exp(v)	Pointwise math
m(i,j)	R equivalent of m[i,j]
v.sum()	R equivalent of sum(v)
m.transpose()	R equivalent of t(m)

Some distributions are available as C++ templates with syntax close to R's distributions:

Function header	Distribution
dnbinom2(x,mu,var,int give_log=0)	Negative binomial with mean and variance
dpois(x,lambda,int give_log=0)	Poisson distribution as in R
dlgamma(y,shape,scale,int give_log=0)	log-gamma distribution
dnorm(x,mean,sd,int give_log=0)	Normal distribution as in R

Examples

```
template()
```

`tmbprofile` *Adaptive likelihood profiling.*

Description

Calculate 1D likelihood profiles wrt. single parameters or more generally, wrt. arbitrary linear combinations of parameters (e.g. contrasts).

Usage

```
tmbprofile(obj, name, lincomb, h = 1e-04, ytol = 2, ystep = 0.1,
  maxit = ceiling(5 * ytol/ystep), parm.range = c(-Inf, Inf),
  slice = FALSE, trace = TRUE, ...)
```

Arguments

`obj`	Object from `MakeADFun` that has been optimized.
`name`	Name or index of a parameter to profile.
`lincomb`	Optional linear combination of parameters to profile. By default a unit vector corresponding to `name`.
`h`	Initial adaptive stepsize on parameter axis.
`ytol`	Adjusts the range of the likelihood values.
`ystep`	Adjusts the resolution of the likelihood profile.
`maxit`	Max number of iterations for adaptive algorithm.
`parm.range`	Valid parameter range.
`slice`	Do slicing rather than profiling?
`trace`	Trace progress?
`...`	Unused

Details

Given a linear combination

$$t = \sum_{i=1}^{n} v_i \theta_i$$

of the parameter vector θ, this function calculates the likelihood profile of t. By default v is a unit vector determined from `name`. Alternatively the linear combination may be given directly (`lincomb`).

Value

data.frame with parameter and function values.

See Also

plot.tmbprofile, confint.tmbprofile

Examples

```
## Not run:
runExample("simple",thisR=TRUE)
## Parameter names for this model:
## beta   beta   logsdu   logsd0

## Profile wrt. sigma0:
prof <- tmbprofile(obj,"logsd0")
plot(prof)
confint(prof)

## Profile the difference between the beta parameters (name is optional):
prof2 <- tmbprofile(obj,name="beta1 - beta2",lincomb = c(1,-1,0,0))
plot(prof2)
confint(prof2)

## End(Not run)
```

Index