Package ‘TMB’

May 28, 2014

Type Package

Title General random effect model builder tool inspired by ADMB.

Version 1.0

Date 2013-09-03

Author Kasper Kristensen

Maintainer Kasper Kristensen <kaskr@imm.dtu.dk>

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

Depends R (>= 3.0.0),Matrix(>= 1.0-12)

LinkingTo Matrix

Collate 'asmle.R' 'examples.R' 'options.R' 'TMB.R' 'zzz.R' 'config.R' 'benchmark.R' 'gdbsource.R' 'sdreport.R'

R topics documented:

benchmark 2
compile 2
gdbsource 3
MakeADFun 4
newton 6
openmp 8
precompile 8
Rinterface 9
runExample 9
sdreport 10
template 11

Index **13**

benchmark *Benchmark parallel templates*

Description

Benchmark parallel templates

Usage

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

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.

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

```
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)
```

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),
  libtmb = TRUE, ...)
```

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)?
`...`	Passed as Makeconf variables.

Details

TMB relies on R's built in functionality to create shared libraries independent on 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 `...` argument, which will overwrite any previous selections.

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

Description

Source R-script through gdb to get backtrace.

Usage

```
gdbsource(file, interactive = FALSE)
```

Arguments

`file`	Your R script
`interactive`	Run interactive gdb session?

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 i.e. `compile(cppfile,"-O0 -g")` in order to provide correct line numbers.

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"), random = 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, ...)
```

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`.
`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 ontrolling 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.
`checkParameterOrder`	Optional check for correct parameter order.
`regexp`	Match random effects by regular expressions?
`...`	

Details

A call to `MakeADFun` will return an object that, based on the users DLL code (specified through `DLL`), contain functions to calculate the objective function and its gradient. The object contain 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 R's `optim` function. 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 all of the functions `fn` and `gr` automatically performs 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`. By default, we use `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 mgc of every gradient evaluation ?

Value

List with components (fn,gr, etc) suitable for an optim call.

`newton` *Generalized newton optimizer.*

Description

Generalized newton optimizer used for the inner optimization problem.

Usage

```
newton(par, fn, gr, he, trace = newtonOption("trace"),
  maxit = newtonOption("maxit"),
  tol = newtonOption("tol"), alpha = 1,
  smartsearch = newtonOption("smartsearch"),
  mgcmax = newtonOption("mgcmax"), 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 gradient 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.
`...`	

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 functions 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.

`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.

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

Description

Precompile the TMB library

Usage

```
precompile(...)
```

Arguments

`...`	Passed to `compile`.

Details

The precompilation should only be run once, typically right after installaion of TMB. Note that the precompilation requires write access to the TMB package folder. Two versions of the library - with/without the openmp flag - will be generated. After this, compilation times of templates should be reduced.

`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, ...)
```

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.
`...`	Passed to `compile`.

`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 and fixed effects parameters specified through the ADREPORT() macro from the user template.

Usage

```
sdreport(obj, par.fixed = NULL, hessian.fixed = NULL,
  getJointPrecision = FALSE)
```

Arguments

`obj`	Object returned by `MakeADFun`
`par.fixed`	Optional. Fixed effect parameter estimate (will be known to `obj` when an optimization has been carried out).
`hessian.fixed`	Optional. Hessian wrt. fixed effects (will be calculated from `obj` if missing).
`getJointPrecision`	Optional. Return full joint precision matrix of random and fixed effects?

Details

First, the Hessian wrt. the fixed effect parameter vector (θ) is calculated. The fixed effects 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`).

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

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

For random effect models a generalized delta-method is used. First the joint covariance of random and fixed effects 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 fixed effects and the second term represents the variance of the conditional mean wrt. the fixed effects. Now the delta method can be applied on a general non-linear function $\phi(u, \theta)$ of random effects u and fixed effects 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`).

Value

Object of class `sdreport`

Examples

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

`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.

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_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(n3,n2,n1))
v+v,v-v,v*v,v/v	Pointwise binary operations
m*v	Matrix-vector multiply
a(i)	R equivalent of a[,,i]
a(i)(j)	R equivalent of a[,j,i]
a(i)(j)[k]	R equivalent of a[k,j,i]
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 avaliable as c++ templates with syntax close to R's distributions:

Function header	**Distribution**
dnbinom(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()
```

Index

benchmark, 2

compile, 2

gdbsource, 3

MakeADFun, 4

newton, 6

openmp, 8

precompile, 8

Rinterface, 9
runExample, 9

sdreport, 10

template, 11