Package 'RTMB'

January 10, 2025

Type Package

Title 'R' Bindings for 'TMB'

Version 1.7

Date 2025-01-10

Author Kasper Kristensen [aut, cre] (<https://orcid.org/0000-0003-3425-3762>)

Maintainer Kasper Kristensen <kaskr@dtu.dk>

Description Native 'R' interface to 'TMB' (Template Model Builder) so models can be written entirely in 'R' rather than 'C++'. Automatic differentiation, to any order, is available for a rich subset of 'R' features, including linear algebra for dense and sparse matrices, complex arithmetic, Fast Fourier Transform, probability distributions and special functions. 'RTMB' provides easy access to model fitting and validation following the principles of Kristensen, K., Nielsen, A., Berg, C. W., Skaug, H., & Bell, B. M. (2016) <DOI:10.18637/jss.v070.i05> and Thygesen, U.H., Albertsen, C.M., Berg, C.W. et al. (2017) <DOI:10.1007/s10651-017-0372-4>.

License GPL (>= 2)

Imports Rcpp (>= 1.0.9), Matrix, methods, TMB (>= 1.9.7), MASS

LinkingTo Rcpp, TMB, RcppEigen

VignetteBuilder knitr, rmarkdown

Suggests knitr, rmarkdown, igraph, tinytest, numDeriv

URL https://github.com/kaskr/RTMB

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

NeedsCompilation yes

Encoding UTF-8

Repository CRAN

Date/Publication 2025-01-10 10:30:02 UTC

Contents

`RTMB-package` *RTMB: R bindings for 'TMB'*

Description

The package 'RTMB' provides a native R interface for *a subset of* 'TMB' so you can avoid coding in C++. 'RTMB' only affects the 'TMB' function 'MakeADFun' that builds the objective function. Once 'MakeADFun' has been invoked, everything else is *exactly the same* and *models run as fast* as if coded in C++.

Details

'RTMB' offers a greatly simplified interface to 'TMB'. The TMB objective function can now be written entirely in R rather than C++ (TMB-interface). In addition, we highlight two new simplifications:

1. For the most cases, simulation testing can be carried out *automatically* without the need to add simulation blocks (Simulation).
2. Also, quantile residuals can be obtained without any essential modifications to the objective function (OSA-residuals).

The introduction vignette describes these basic features - see `vignette("RTMB-introduction")`.

In addition to the usual MakeADFun interface, 'RTMB' offers a lower level interface to the AD machinery (`MakeTape`). MakeTape replaces the functionality you would normally get in 'TMB' using C++ functors, such as calculating derivatives inside the objective function.

The advanced vignette covers these topics - see `vignette("RTMB-advanced")`.

Note

'RTMB' relies heavily on the new AD framework 'TMBad' without which this interface would not be possible.

Author(s)

Kasper Kristensen

Maintainer: kaskr@dtu.dk

AD *Convert R object to AD*

Description

Signify that this object should be given an AD interpretation if evaluated in an active AD context. Otherwise, keep object as is.

Usage

```
AD(x, force = FALSE)
```

Arguments

`x`	Object to be converted.
`force`	Logical; Force AD conversion even if no AD context? (for debugging)

Details

`AD` is a generic constructor, converting plain R structures to RTMB objects if in an autodiff context. Otherwise, it does nothing (and adds virtually no computational overhead).

`AD` knows the following R objects:

- Numeric objects from **base**, such as `numeric()`, `matrix()`, `array()`, are converted to class advector with other attributes kept intact.
- Complex objects from **base**, such as `complex()`, are converted to class adcomplex.
- Sparse matrices from **Matrix**, such as `Matrix()`, `Diagonal()`, are converted to adsparse.

`AD` provides a reliable way to avoid problems with method dispatch when mixing operand types. For instance, sub assigning `x[i] <- y` may be problematic when `x` is numeric and `y` is `advector`. A prior statement `x <- AD(x)` solves potential method dispatch issues and can therefore be used as a reliable alternative to ADoverload.

Examples

```
## numeric object to AD
AD(numeric(4), force=TRUE)
## complex object to AD
AD(complex(4), force=TRUE)
## Convert sparse matrices (Matrix package) to AD representation
F <- MakeTape(function(x) {
  M <- AD(Matrix::Matrix(0,4,4))
  M[1,] <- x
  D <- AD(Matrix::Diagonal(4))
  D@x[] <- x
  M + D
}, 0)
F(2)
```

ADapply *AD apply functions*

Description

These **base** apply methods have been modified to keep the AD class attribute (which would otherwise be lost).

Usage

```
## S4 method for signature 'advector'
apply(X, MARGIN, FUN, ..., simplify = TRUE)

## S4 method for signature 'ANY'
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

Arguments

X	As apply
MARGIN	As apply
FUN	As apply
...	As apply
simplify	As sapply
USE.NAMES	As sapply

Value

Object of class "advector" with a dimension attribute.

Functions

- apply(advector): As apply
- sapply(ANY): As sapply

Examples

```
F <- MakeTape(function(x) apply(matrix(x,2,2), 2, sum), numeric(4))
F$jacobian(1:4)
```

ADcomplex *AD complex numbers*

Description

A limited set of complex number operations can be used when constructing AD tapes. The available methods are listed in this help page.

Usage

```
adcomplex(real, imag = rep(advector(0), length(real)))

## S3 method for class 'adcomplex'
Re(z)

## S3 method for class 'adcomplex'
Im(z)

## S4 method for signature 'adcomplex'
show(object)

## S3 method for class 'adcomplex'
dim(x)

## S3 replacement method for class 'adcomplex'
dim(x) <- value

## S3 method for class 'adcomplex'
x[...]

## S3 replacement method for class 'adcomplex'
x[...] <- value

## S3 method for class 'adcomplex'
t(x)

## S3 method for class 'adcomplex'
length(x)

## S3 method for class 'adcomplex'
Conj(z)

## S3 method for class 'adcomplex'
```

```
Mod(z)

## S3 method for class 'adcomplex'
Arg(z)

## S3 method for class 'adcomplex'
x + y

## S3 method for class 'adcomplex'
x - y

## S3 method for class 'adcomplex'
x * y

## S3 method for class 'adcomplex'
x / y

## S3 method for class 'adcomplex'
exp(x)

## S3 method for class 'adcomplex'
log(x, base)

## S3 method for class 'adcomplex'
sqrt(x)

## S4 method for signature 'adcomplex'
fft(z, inverse = FALSE)

## S4 method for signature 'advector'
fft(z, inverse = FALSE)

## S3 method for class 'adcomplex'
rep(x, ...)

## S3 method for class 'adcomplex'
as.vector(x, mode = "any")

## S3 method for class 'adcomplex'
is.matrix(x)

## S3 method for class 'adcomplex'
as.matrix(x, ...)

## S4 method for signature 'adcomplex,ANY'
x %*% y

## S4 method for signature 'adcomplex,ANY'
```

```
solve(a, b)

## S4 method for signature 'adcomplex'
colSums(x)

## S4 method for signature 'adcomplex'
rowSums(x)

## S4 method for signature 'adcomplex,ANY,ANY'
diag(x)

## S4 method for signature 'advector,adcomplex'
Ops(e1, e2)

## S4 method for signature 'adcomplex,advector'
Ops(e1, e2)
```

Arguments

real	Real part
imag	Imaginary part
z	An object of class 'adcomplex'
object	An object of class 'adcomplex'
x	An object of class 'adcomplex'
value	Replacement value
...	As [
y	An object of class 'adcomplex'
base	Not implemented
inverse	As fft
mode	As as.vector
a	matrix
b	matrix, vector or missing
e1	Left operand
e2	Right operand

Value

Object of class "adcomplex".

Functions

- adcomplex(): Construct adcomplex vector
- Re(adcomplex): As complex
- Im(adcomplex): As complex

- `show(adcomplex)`: Print method
- `dim(adcomplex)`: As dim
- `dim(adcomplex) <- value`: As dim
- `[`: As [
- `` `[`(adcomplex) <- value ``: As [<-
- `t(adcomplex)`: As t
- `length(adcomplex)`: As length
- `Conj(adcomplex)`: As complex
- `Mod(adcomplex)`: As complex
- `Arg(adcomplex)`: As complex
- `+`: As complex
- `-`: As complex
- `*`: As complex
- `/`: As complex
- `exp(adcomplex)`: As complex
- `log(adcomplex)`: As complex
- `sqrt(adcomplex)`: As complex
- `fft(adcomplex)`: Fast Fourier Transform equivalent to fft. Notably this is the **multivariate** transform when `x` is an array.
- `fft(advector)`: If real input is supplied it is first converted to complex.
- `rep(adcomplex)`: As rep
- `as.vector(adcomplex)`: Apply for each of real/imag
- `is.matrix(adcomplex)`: Apply for real
- `as.matrix(adcomplex)`: Apply for each of real/imag
- `x %*% y`: Complex matrix multiply
- `solve(a = adcomplex, b = ANY)`: Complex matrix inversion and solve
- `colSums(adcomplex)`: Apply for each of real/imag
- `rowSums(adcomplex)`: Apply for each of real/imag
- `diag(x = adcomplex, nrow = ANY, ncol = ANY)`: Apply for each of real/imag
- `Ops(e1 = advector, e2 = adcomplex)`: Mixed real/complex arithmetic
- `Ops(e1 = adcomplex, e2 = advector)`: Mixed real/complex arithmetic

Examples

```
## Tape using complex operations
F <- MakeTape(function(x) {
  x <- as.complex(x)
  y <- exp( x * ( 1 + 2i ) )
  c(Re(y), Im(y))
}, numeric(1))
```

```
F
F(1)
## Complex FFT on the tape
G <- MakeTape(function(x) sum(Re(fft(x))), numeric(3))
G$simplify()
G$print()
```

ADconstruct *AD aware numeric constructors*

Description

These base constructors have been extended to keep the AD class attribute of the data argument.

Usage

```
## S4 method for signature 'advector,ANY,ANY'
diag(x, nrow, ncol)

## S4 method for signature 'advector'
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)

## S4 method for signature 'num.'
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

Arguments

x	As diag
nrow	As matrix
ncol	As matrix
data	As matrix
byrow	As matrix
dimnames	As matrix

Value

Object of class "advector" with a dimension attribute.

Functions

- diag(x = advector, nrow = ANY, ncol = ANY): Equivalent of diag
- matrix(advector): Equivalent of matrix
- matrix(num.): Equivalent of matrix

Examples

```
func <- function(x) {
  M <- matrix(x, 2, 2)
  print(class(M))
  D <- diag(x)
  print(class(D))
  0
}
invisible(func(1:4))            ## 'matrix' 'array'
invisible(MakeTape(func, 1:4))  ## 'advector'
```

`ADjoint` *AD adjoint code from R*

Description

Writing custom AD adjoint derivatives from R

Usage

```
ADjoint(f, df, name = NULL, complex = FALSE)
```

Arguments

`f`	R function representing the function value.
`df`	R function representing the reverse mode derivative.
`name`	Internal name of this atomic.
`complex`	Logical; Assume complex and adcomplex types for all arguments?

Details

Reverse mode derivatives (adjoint code) can be implemented from R using the function `ADjoint`. It takes as input a function of a single argument `f(x)` representing the function value, and another function of *three* arguments `df(x, y, dy)` representing the adjoint derivative wrt `x` defined as `d/dx sum( f(x) * dy )`. Both `y` and `dy` have the same length as `f(x)`. The argument `y` can be assumed equal to `f(x)` to avoid recalculation during the reverse pass. It should be assumed that all arguments `x`, `y`, `dy` are vectors without any attributes *except* for dimensions, which are stored on first evaluation. The latter is convenient when implementing matrix functions (see `logdet` example). Higher order derivatives automatically work provided that `df` is composed by functions that `RTMB` already knows how to differentiate.

Value

A function that allows for numeric and taped evaluation.

Complex case

The argument `complex=TRUE` specifies that the functions `f` and `df` are complex differentiable (holomorphic) and that arguments `x`, `y` and `dy` should be assumed complex (or adcomplex). Recall that complex differentiability is a strong condition excluding many continuous functions e.g. `Re`, `Im`, `Conj` (see example).

Note

`ADjoint` may be useful when you need a special atomic function which is not yet available in `RTMB`, or just to experiment with reverse mode derivatives. However, the approach may cause a *significant overhead* compared to native `RTMB` derivatives. In addition, the approach is *not thread safe*, i.e. calling R functions cannot be done in parallel using OpenMP.

Examples

```
################################################################################
## Lambert W-function defined by W(y*exp(y))=y
W <- function(x) {
  logx <- log(x)
  y <- pmax(logx, 0)
  while (any(abs(logx - log(y) - y) > 1e-9, na.rm = TRUE)) {
      y <- y - (y - exp(logx - y)) / (1 + y)
  }
  y
}
## Derivatives
dW <- function(x, y, dy) {
   dy / (exp(y) * (1. + y))
}
## Define new derivative symbol
LamW <- ADjoint(W, dW)
## Test derivatives
(F <- MakeTape(function(x)sum(LamW(x)), numeric(3)))
F(1:3)
F$print()                ## Note the 'name'
F$jacobian(1:3)          ## gradient
F$jacfun()$jacobian(1:3) ## hessian
################################################################################
## Log determinant
logdet <- ADjoint(
   function(x) determinant(x, log=TRUE)$modulus,
   function(x, y, dy) t(solve(x)) * dy,
   name = "logdet")
(F <- MakeTape(logdet, diag(2)))
## Test derivatives
## Compare with numDeriv::hessian(F, matrix(1:4,2))
F$jacfun()$jacobian(matrix(1:4,2)) ## Hessian
################################################################################
## Holomorphic extension of 'solve'
matinv <- ADjoint(
   solve,
```

```
   function(x,y,dy) -t(y) %*% dy %*% t(y),
   complex=TRUE)
(F <- MakeTape(function(x) Im(matinv(x+AD(1i))), diag(2)))
## Test derivatives
## Compare with numDeriv::jacobian(F, matrix(1:4,2))
F$jacobian(matrix(1:4,2))
```

ADmatrix *AD matrix methods (sparse and dense)*

Description

Matrices (**base** package) and sparse matrices (**Matrix** package) can be used inside the RTMB objective function as part of the calculations. Behind the scenes these R objects are converted to AD representations when needed. AD objects have a temporary lifetime, so you probably won't see them / need to know them. The only important thing is which *methods* work for the objects.

Usage

```
## S3 method for class 'advector'
chol(x, ...)

## S3 method for class 'advector'
determinant(x, logarithm = TRUE, ...)

## S4 method for signature 'adcomplex'
eigen(x, symmetric, only.values = FALSE, EISPACK = FALSE)

## S4 method for signature 'advector'
eigen(x, symmetric, only.values = FALSE, EISPACK = FALSE)

## S4 method for signature 'advector'
svd(x, nu, nv, LINPACK = FALSE)

## S3 method for class 'adsparse'
t(x)

## S3 method for class 'adsparse'
x[...]

## S3 replacement method for class 'adsparse'
x[...] <- value

## S3 method for class 'adsparse'
as.matrix(x, ...)

## S4 method for signature 'adsparse,missing,missing'
```

```
diag(x)

## S4 method for signature 'advector'
expm(x)

## S4 method for signature 'adsparse'
expm(x)

## S4 method for signature 'adsparse'
dim(x)

## S4 method for signature 'anysparse,ad'
x %*% y

## S4 method for signature 'ad,anysparse'
x %*% y

## S4 method for signature 'adsparse,adsparse'
x %*% y

## S4 method for signature 'ad,ad'
x %*% y

## S4 method for signature 'ad,ad.'
tcrossprod(x, y)

## S4 method for signature 'ad,ad.'
crossprod(x, y)

## S4 method for signature 'advector'
cov2cor(V)

## S4 method for signature 'ad,ad.'
solve(a, b)

## S4 method for signature 'num,num.'
solve(a, b)

## S4 method for signature 'anysparse,ad.'
solve(a, b)

## S4 method for signature 'advector'
colSums(x, na.rm, dims)

## S4 method for signature 'advector'
rowSums(x, na.rm, dims)

## S4 method for signature 'adsparse'
```

```
colSums(x, na.rm, dims)

## S4 method for signature 'adsparse'
rowSums(x, na.rm, dims)

## S3 method for class 'advector'
cbind(...)

## S3 method for class 'advector'
rbind(...)
```

Arguments

x	matrix (sparse or dense)
...	As cbind
logarithm	Not used
symmetric	Logical; Is input matrix symmetric (Hermitian) ?
only.values	Ignored
EISPACK	Ignored
nu	Ignored
nv	Ignored
LINPACK	Ignored
value	Replacement value
y	matrix (sparse or dense)
V	Covariance matrix
a	matrix
b	matrix, vector or missing
na.rm	Logical; Remove NAs while taping.
dims	Same as colSums and rowSums.

Value

List (vectors/values) with adcomplex components.

List (vectors/values) with advector components in symmetric case and adcomplex components otherwise.

Object of class advector with a dimension attribute for dense matrix operations; Object of class adsparse for sparse matrix operations.

Functions

- chol(advector): AD matrix cholesky
- determinant(advector): AD log determinant
- eigen(adcomplex): General AD eigen decomposition for complex matrices. Note that argument symmetric is **not** auto-detected so **must** be specified.

- `eigen(advector)`: AD eigen decomposition for real matrices. The non-symmetric case is redirected to the `adcomplex` method. Note that argument `symmetric` is **not** auto-detected so **must** be specified.
- `svd(advector)`: AD svd decomposition for real matrices.
- `t(adsparse)`: AD sparse matrix transpose. Re-directs to t,CsparseMatrix-method.
- `[`: AD sparse matrix subsetting. Re-directs to [-methods.
- `` `[`(adsparse) <- value ``: AD sparse matrix subset assignment. Re-directs to [<–methods.
- `as.matrix(adsparse)`: Convert AD sparse to dense matrix.
- `diag(x = adsparse, nrow = missing, ncol = missing)`: AD sparse matrix diagonal extract. Re-directs to diag,CsparseMatrix-method.
- `expm(advector)`: AD matrix exponential
- `expm(adsparse)`: AD matrix exponential
- `dim(adsparse)`: AD sparse matrix dimension
- `x %*% y`: AD matrix multiply
- `x %*% y`: AD matrix multiply
- `x %*% y`: AD matrix multiply
- `x %*% y`: AD matrix multiply
- `tcrossprod(x = ad, y = ad.)`: AD matrix multiply
- `crossprod(x = ad, y = ad.)`: AD matrix multiply
- `cov2cor(advector)`: AD matrix cov2cor
- `solve(a = ad, b = ad.)`: AD matrix inversion and solve
- `solve(a = num, b = num.)`: AD matrix inversion and solve
- `solve(a = anysparse, b = ad.)`: Sparse AD matrix solve
- `colSums(advector)`: AD matrix (or array) colsums
- `rowSums(advector)`: AD matrix (or array) rowsums
- `colSums(adsparse)`: AD sparse matrix colsums
- `rowSums(adsparse)`: AD sparse matrix rowsums
- `cbind(advector)`: AD matrix column bind
- `rbind(advector)`: AD matrix row bind

Examples

```
F <- MakeTape(function(x) matrix(1:9,3,3) %*% x, numeric(3))
F$jacobian(1:3)
F <- MakeTape(function(x) Matrix::expm(matrix(x,2,2)), numeric(4))
F$jacobian(1:4)
F <- MakeTape(det, diag(2)) ## Indirectly available via 'determinant'
F$jacobian(matrix(1:4,2))
```

`ADoverload` *Enable extra RTMB convenience methods*

Description

Enable extra RTMB convenience methods

Usage

```
ADoverload(x = c("[<-", "c", "diag<-"))
```

Arguments

`x`	Name of primitive to overload

Details

Work around limitations in R's method dispatch system by overloading some selected primitives, currently:

- Inplace replacement, so you can do `x[i] <- y` when `x` is numeric and `y` is AD.
- Mixed combine, so you can do e.g. `c(x, y)` when `x` numeric and `y` is AD.
- Diagonal assignment, so you can do `diag(x) <- y` when `x` is a numeric matrix and `y` is AD.

In all cases, the result should be AD. The methods are automatically **temporarily** attached to the search path (`search()`) when entering MakeTape or MakeADFun. Alternatively, methods can be overloaded locally inside functions using e.g. `"[<-" <- ADoverload("[<-")`. This is only needed when using RTMB from a package.

Value

Function representing the overload.

Examples

```
MakeTape(function(x) {print(search()); x}, numeric(0))
MakeTape(function(x) c(1,x), 1:3)
MakeTape(function(x) {y <- 1:3; y[2] <- x; y}, 1)
MakeTape(function(x) {y <- matrix(0,3,3); diag(y) <- x; y}, 1:3)
```

ADsparse *AD sparse matrix class*

Description

Sparse matrices in **RTMB** are essentially `dgCMatrix` with an `advector` `x`-slot.

Slots

`x` Non-zeros

`i` row indices (zero based)

`p` col pointers (zero based)

`Dim` Dimension

ADvector *The AD vector and its methods*

Description

An `advector` is a class used behind the scenes to replace normal R numeric objects during automatic differentiation. An `advector` has a 'temporary lifetime' and therefore you do not *see* / *need to know* it as a normal user.

Usage

```
advector(x)

## S3 method for class 'advector'
Ops(e1, e2)

## S3 method for class 'advector'
Math(x, ...)

## S3 method for class 'advector'
as.vector(x, mode = "any")

## S3 method for class 'advector'
as.complex(x, ...)

## S3 method for class 'advector'
aperm(a, perm, ...)

## S3 method for class 'advector'
c(...)
```

```
## S3 method for class 'advector'
x[...]

## S3 replacement method for class 'advector'
x[...] <- value

## S3 method for class 'advector'
x[[...]]

## S3 method for class 'advector'
rep(x, ...)

## S3 method for class 'advector'
is.nan(x)

## S3 method for class 'advector'
is.finite(x)

## S3 method for class 'advector'
is.infinite(x)

## S3 method for class 'advector'
is.na(x)

## S3 method for class 'advector'
sum(x, ..., na.rm = FALSE)

## S3 method for class 'advector'
mean(x, ...)

## S3 method for class 'advector'
prod(x, ..., na.rm = FALSE)

## S3 method for class 'advector'
min(..., na.rm = FALSE)

## S3 method for class 'advector'
max(..., na.rm = FALSE)

## S3 method for class 'advector'
is.numeric(x)

## S3 method for class 'advector'
as.double(x, ...)

## S3 method for class 'advector'
Complex(z)
```

```
## S3 method for class 'advector'
Summary(..., na.rm = FALSE)

## S3 method for class 'advector'
diff(x, lag = 1L, differences = 1L, ...)

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

## S4 method for signature 'num,ad,ad'
ifelse(test, yes, no)

## S4 method for signature 'num,num,num'
ifelse(test, yes, no)
```

Arguments

x	numeric or advector
e1	advector
e2	advector
...	Additional arguments
mode	FIXME might not be handled correctly by as.vector
a	advector with dimension attribute
perm	Permutation as in aperm
value	Replacement value implicitly converted to AD
na.rm	Must be FALSE (default)
z	Complex (not allowed)
lag	As diff
differences	As diff
test	logical vector
yes	advector
no	advector

Details

An AD vector (class='advector') is an atomic R vector of 'codes' that are internally interpretable as 'AD scalars'. A substantial part of R's existing S3 matrix and array functionality can be re-used for AD vectors.

Value

Object of class "advector".

Functions

- `advector()`: Construct a new advector
- `Ops(advector)`: Binary operations
- `Math(advector)`: Unary operations
- `as.vector(advector)`: Makes `array(x)` work.
- `as.complex(advector)`: Convert to ADcomplex. Note that dimensions are dropped for consistency with base R.
- `aperm(advector)`: Equivalent of aperm
- `c(advector)`: Equivalent of c. However note the limitation for mixed types: If `x` is an AD type, `c(x,1)` works while `c(1,x)` does not!
- `[`: Equivalent of [
- `` `[`(advector) <- value ``: Equivalent of [<-
- `[[`: Equivalent of [[
- `rep(advector)`: Equivalent of rep. Makes `outer(x,x,...)` work.
- `is.nan(advector)`: Equivalent of is.nan. Check NaN status of a *constant* `advector` expression. If not constant throw an error.
- `is.finite(advector)`: Equivalent of is.finite. Check finite status of a *constant* `advector` expression. If not constant throw an error.
- `is.infinite(advector)`: Equivalent of is.infinite. Check infinity status of a *constant* `advector` expression. If not constant throw an error.
- `is.na(advector)`: Equivalent of is.na. Check NA status of an `advector`. NAs can only occur directly (as constants) or indirectly as the result of an operation with NA operands. For a tape built with non-NA parameters the NA status of any expression is constant and can therefore safely be used as part of the calculations. (assuming correct propagation of NAs via C-level arithmetic).
- `sum(advector)`: Equivalent of sum. `na.rm=TRUE` is allowed, but note that this feature assumes correct propagation of NAs via C-level arithmetic.
- `mean(advector)`: Equivalent of mean except no arguments beyond `x` are supported.
- `prod(advector)`: Equivalent of prod.
- `min(advector)`: Equivalent of min.
- `max(advector)`: Equivalent of min.
- `is.numeric(advector)`: Makes `cov2cor()` work. FIXME: Any unwanted side-effects with this?
- `as.double(advector)`: Makes `as.numeric()` work.
- `Complex(advector)`: Complex operations are redirected to adcomplex.
- `Summary(advector)`: Unimplemented Summary operations (currently `all any range`) will throw an error.
- `diff(advector)`: Equivalent of diff
- `print(advector)`: Print method
- `ifelse(test = num, yes = ad, no = ad)`: Equivalent of ifelse
- `ifelse(test = num, yes = num, no = num)`: Default method

Examples

```
x <- advector(1:9)
a <- array(x, c(3,3))  ## as an array
outer(x, x, "+") ## Implicit via 'rep'
rev(x)           ## Implicit via '['
```

`Distributions` *Distributions and special functions for which AD is implemented*

Description

The functions listed in this help page are all applicable for AD types. Method dispatching follows a simple rule: *If at least one argument is an AD type then a special AD implementation is selected. In all other cases a default implementation is used* (typically that of the **stats** package). Argument recycling follows the R standard (although wihout any warnings).

Usage

```
## S4 method for signature 'ad,ad.,logical.'
dexp(x, rate = 1, log = FALSE)

## S4 method for signature 'num,num.,logical.'
dexp(x, rate = 1, log = FALSE)

## S4 method for signature 'osa,ANY,ANY'
dexp(x, rate = 1, log = FALSE)

## S4 method for signature 'simref,ANY,ANY'
dexp(x, rate = 1, log = FALSE)

## S4 method for signature 'ad,ad,ad.,logical.'
dweibull(x, shape, scale = 1, log = FALSE)

## S4 method for signature 'num,num,num.,logical.'
dweibull(x, shape, scale = 1, log = FALSE)

## S4 method for signature 'osa,ANY,ANY,ANY'
dweibull(x, shape, scale = 1, log = FALSE)

## S4 method for signature 'simref,ANY,ANY,ANY'
dweibull(x, shape, scale = 1, log = FALSE)

## S4 method for signature 'ad,ad,ad,logical.'
dbinom(x, size, prob, log = FALSE)

## S4 method for signature 'num,num,num,logical.'
dbinom(x, size, prob, log = FALSE)
```

```
## S4 method for signature 'osa,ANY,ANY,ANY'
dbinom(x, size, prob, log = FALSE)

## S4 method for signature 'simref,ANY,ANY,ANY'
dbinom(x, size, prob, log = FALSE)

## S4 method for signature 'ad,ad,ad,missing,logical.'
dbeta(x, shape1, shape2, log)

## S4 method for signature 'num,num,num,missing,logical.'
dbeta(x, shape1, shape2, log)

## S4 method for signature 'osa,ANY,ANY,ANY,ANY'
dbeta(x, shape1, shape2, log)

## S4 method for signature 'simref,ANY,ANY,ANY,ANY'
dbeta(x, shape1, shape2, log)

## S4 method for signature 'ad,ad,ad,missing,logical.'
df(x, df1, df2, log)

## S4 method for signature 'num,num,num,missing,logical.'
df(x, df1, df2, log)

## S4 method for signature 'osa,ANY,ANY,ANY,ANY'
df(x, df1, df2, log)

## S4 method for signature 'simref,ANY,ANY,ANY,ANY'
df(x, df1, df2, log)

## S4 method for signature 'ad,ad.,ad.,logical.'
dlogis(x, location = 0, scale = 1, log = FALSE)

## S4 method for signature 'num,num.,num.,logical.'
dlogis(x, location = 0, scale = 1, log = FALSE)

## S4 method for signature 'osa,ANY,ANY,ANY'
dlogis(x, location = 0, scale = 1, log = FALSE)

## S4 method for signature 'simref,ANY,ANY,ANY'
dlogis(x, location = 0, scale = 1, log = FALSE)

## S4 method for signature 'ad,ad,missing,logical.'
dt(x, df, log)

## S4 method for signature 'num,num,missing,logical.'
dt(x, df, log)
```

```
## S4 method for signature 'osa,ANY,ANY,ANY'
dt(x, df, log)

## S4 method for signature 'simref,ANY,ANY,ANY'
dt(x, df, log)

## S4 method for signature 'ad,ad,ad,missing,logical.'
dnbinom(x, size, prob, log)

## S4 method for signature 'num,num,num,missing,logical.'
dnbinom(x, size, prob, log)

## S4 method for signature 'osa,ANY,ANY,ANY,ANY'
dnbinom(x, size, prob, log)

## S4 method for signature 'simref,ANY,ANY,ANY,ANY'
dnbinom(x, size, prob, log)

## S4 method for signature 'ad,ad,logical.'
dpois(x, lambda, log = FALSE)

## S4 method for signature 'num,num,logical.'
dpois(x, lambda, log = FALSE)

## S4 method for signature 'osa,ANY,ANY'
dpois(x, lambda, log = FALSE)

## S4 method for signature 'simref,ANY,ANY'
dpois(x, lambda, log = FALSE)

## S4 method for signature 'ad,ad,missing,ad.,logical.'
dgamma(x, shape, scale, log)

## S4 method for signature 'num,num,missing,num.,logical.'
dgamma(x, shape, scale, log)

## S4 method for signature 'osa,ANY,ANY,ANY,ANY'
dgamma(x, shape, scale, log)

## S4 method for signature 'simref,ANY,ANY,ANY,ANY'
dgamma(x, shape, scale, log)

## S4 method for signature 'ad,ad.,ad.,missing,missing'
pnorm(q, mean, sd)

## S4 method for signature 'num,num.,num.,missing,missing'
pnorm(q, mean, sd)
```

```
## S4 method for signature 'ad,ad,missing,ad.,missing,missing'
pgamma(q, shape, scale)

## S4 method for signature 'num,num,missing,num.,missing,missing'
pgamma(q, shape, scale)

## S4 method for signature 'ad,ad,missing,missing'
ppois(q, lambda)

## S4 method for signature 'num,num,missing,missing'
ppois(q, lambda)

## S4 method for signature 'ad,ad.,missing,missing'
pexp(q, rate)

## S4 method for signature 'num,num.,missing,missing'
pexp(q, rate)

## S4 method for signature 'ad,ad,ad.,missing,missing'
pweibull(q, shape, scale)

## S4 method for signature 'num,num,num.,missing,missing'
pweibull(q, shape, scale)

## S4 method for signature 'ad,ad,ad,missing,missing,missing'
pbeta(q, shape1, shape2)

## S4 method for signature 'num,num,num,missing,missing,missing'
pbeta(q, shape1, shape2)

## S4 method for signature 'ad,ad.,ad.,missing,missing'
qnorm(p, mean, sd)

## S4 method for signature 'num,num.,num.,missing,missing'
qnorm(p, mean, sd)

## S4 method for signature 'ad,ad,missing,ad.,missing,missing'
qgamma(p, shape, scale)

## S4 method for signature 'num,num,missing,num.,missing,missing'
qgamma(p, shape, scale)

## S4 method for signature 'ad,ad.,missing,missing'
qexp(p, rate)

## S4 method for signature 'num,num.,missing,missing'
qexp(p, rate)
```

```
## S4 method for signature 'ad,ad,ad.,missing,missing'
qweibull(p, shape, scale)

## S4 method for signature 'num,num,num.,missing,missing'
qweibull(p, shape, scale)

## S4 method for signature 'ad,ad,ad,missing,missing,missing'
qbeta(p, shape1, shape2)

## S4 method for signature 'num,num,num,missing,missing,missing'
qbeta(p, shape1, shape2)

## S4 method for signature 'ad,ad,missing'
besselK(x, nu)

## S4 method for signature 'num,num,missing'
besselK(x, nu)

## S4 method for signature 'ad,ad,missing'
besselI(x, nu)

## S4 method for signature 'num,num,missing'
besselI(x, nu)

## S4 method for signature 'ad,ad'
besselJ(x, nu)

## S4 method for signature 'num,num'
besselJ(x, nu)

## S4 method for signature 'ad,ad'
besselY(x, nu)

## S4 method for signature 'num,num'
besselY(x, nu)

dbinom_robust(x, size, logit_p, log = FALSE)

dsn(x, alpha, log = FALSE)

dSHASHo(x, mu, sigma, nu, tau, log = FALSE)

dtweedie(x, mu, phi, p, log = FALSE)

dnbinom_robust(x, log_mu, log_var_minus_mu, log = FALSE)

dnbinom2(x, mu, var, log = FALSE)
```

```
dlgamma(x, shape, scale, log = FALSE)

logspace_add(logx, logy)

logspace_sub(logx, logy)

## S4 method for signature 'ad,ad.,ad.,logical.'
dnorm(x, mean = 0, sd = 1, log = FALSE)

## S4 method for signature 'num,num.,num.,logical.'
dnorm(x, mean = 0, sd = 1, log = FALSE)

## S4 method for signature 'osa,ANY,ANY,ANY'
dnorm(x, mean = 0, sd = 1, log = FALSE)

## S4 method for signature 'simref,ANY,ANY,ANY'
dnorm(x, mean = 0, sd = 1, log = FALSE)

## S4 method for signature 'ANY,ANY,ANY,ANY'
dlnorm(x, meanlog = 0, sdlog = 1, log = FALSE)

## S4 method for signature 'osa,ANY,ANY,ANY'
dlnorm(x, meanlog = 0, sdlog = 1, log = FALSE)

## S4 method for signature 'num,num.,num.,logical.'
dlnorm(x, meanlog = 0, sdlog = 1, log = FALSE)

## S4 method for signature 'advector,missing,missing,missing,missing'
plogis(q)

## S4 method for signature 'advector,missing,missing,missing,missing'
qlogis(p)

dcompois(x, mode, nu, log = FALSE)

dcompois2(x, mean, nu, log = FALSE)

## S4 method for signature 'ad,ad,ad,missing,missing'
pbinom(q, size, prob)

## S4 method for signature 'num,num,num,missing,missing'
pbinom(q, size, prob)

## S4 method for signature 'ad,ad.,ad,logical.'
dmultinom(x, size = NULL, prob, log = FALSE)

## S4 method for signature 'num,num.,num,logical.'
```

```
dmultinom(x, size = NULL, prob, log = FALSE)

## S4 method for signature 'osa,ANY,ANY,ANY'
dmultinom(x, size = NULL, prob, log = FALSE)

## S4 method for signature 'simref,ANY,ANY,ANY'
dmultinom(x, size = NULL, prob, log = FALSE)

## S4 method for signature 'ANY,ANY,ANY,ANY'
dmultinom(x, size = NULL, prob, log = FALSE)
```

Arguments

x	observation vector
rate	parameter
log	Logical; Return log density/probability?
shape	parameter
scale	parameter
size	parameter
prob	parameter
shape1	parameter
shape2	parameter
df1	parameter
df2	parameter
location	parameter
df	parameter
lambda	parameter
q	vector of quantiles
mean	parameter
sd	parameter
p	parameter
nu	parameter
logit_p	parameter
alpha	parameter
mu	parameter
sigma	parameter
tau	parameter
phi	parameter
log_mu	parameter
log_var_minus_mu	parameter

var	parameter
logx	Log-space input
logy	Log-space input
meanlog	Parameter; Mean on log scale.
sdlog	Parameter; SD on log scale.
mode	parameter

Details

Specific documentation of the functions and arguments should be looked up elsewhere:

- All S4 methods behave as the corresponding functions in the **stats** package. However, some arguements may not be implemented in the AD case (e.g. `lower-tail`).
- Other funtions behave as the corresponding TMB versions for which documentation should be looked up online.

Value

In autodiff contexts an object of class `"advector"` is returned; Otherwise a standard numeric vector.

Functions

- `dexp(x = ad, rate = ad., log = logical.)`: AD implementation of dexp
- `dexp(x = num, rate = num., log = logical.)`: Default method
- `dexp(x = osa, rate = ANY, log = ANY)`: OSA implementation
- `dexp(x = simref, rate = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `dweibull(x = ad, shape = ad, scale = ad., log = logical.)`: AD implementation of dweibull
- `dweibull(x = num, shape = num, scale = num., log = logical.)`: Default method
- `dweibull(x = osa, shape = ANY, scale = ANY, log = ANY)`: OSA implementation
- `dweibull(x = simref, shape = ANY, scale = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `dbinom(x = ad, size = ad, prob = ad, log = logical.)`: AD implementation of dbinom
- `dbinom(x = num, size = num, prob = num, log = logical.)`: Default method
- `dbinom(x = osa, size = ANY, prob = ANY, log = ANY)`: OSA implementation
- `dbinom(x = simref, size = ANY, prob = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `dbeta(x = ad, shape1 = ad, shape2 = ad, ncp = missing, log = logical.)`: AD implementation of dbeta
- `dbeta(x = num, shape1 = num, shape2 = num, ncp = missing, log = logical.)`: Default method
- `dbeta(x = osa, shape1 = ANY, shape2 = ANY, ncp = ANY, log = ANY)`: OSA implementation
- `dbeta(x = simref, shape1 = ANY, shape2 = ANY, ncp = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.

- `df(x = ad, df1 = ad, df2 = ad, ncp = missing, log = logical.)`: AD implementation of df
- `df(x = num, df1 = num, df2 = num, ncp = missing, log = logical.)`: Default method
- `df(x = osa, df1 = ANY, df2 = ANY, ncp = ANY, log = ANY)`: OSA implementation
- `df(x = simref, df1 = ANY, df2 = ANY, ncp = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `dlogis(x = ad, location = ad., scale = ad., log = logical.)`: AD implementation of dlogis
- `dlogis(x = num, location = num., scale = num., log = logical.)`: Default method
- `dlogis(x = osa, location = ANY, scale = ANY, log = ANY)`: OSA implementation
- `dlogis(x = simref, location = ANY, scale = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `dt(x = ad, df = ad, ncp = missing, log = logical.)`: AD implementation of dt
- `dt(x = num, df = num, ncp = missing, log = logical.)`: Default method
- `dt(x = osa, df = ANY, ncp = ANY, log = ANY)`: OSA implementation
- `dt(x = simref, df = ANY, ncp = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `dnbinom(x = ad, size = ad, prob = ad, mu = missing, log = logical.)`: AD implementation of dnbinom
- `dnbinom(x = num, size = num, prob = num, mu = missing, log = logical.)`: Default method
- `dnbinom(x = osa, size = ANY, prob = ANY, mu = ANY, log = ANY)`: OSA implementation
- `dnbinom(x = simref, size = ANY, prob = ANY, mu = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `dpois(x = ad, lambda = ad, log = logical.)`: AD implementation of dpois
- `dpois(x = num, lambda = num, log = logical.)`: Default method
- `dpois(x = osa, lambda = ANY, log = ANY)`: OSA implementation
- `dpois(x = simref, lambda = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `dgamma(x = ad, shape = ad, rate = missing, scale = ad., log = logical.)`: AD implementation of dgamma
- `dgamma(x = num, shape = num, rate = missing, scale = num., log = logical.)`: Default method
- `dgamma(x = osa, shape = ANY, rate = ANY, scale = ANY, log = ANY)`: OSA implementation
- `dgamma(x = simref, shape = ANY, rate = ANY, scale = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `pnorm(q = ad, mean = ad., sd = ad., lower.tail = missing, log.p = missing)`: AD implementation of pnorm
- `pnorm(q = num, mean = num., sd = num., lower.tail = missing, log.p = missing)`: Default method
- `pgamma( q = ad, shape = ad, rate = missing, scale = ad., lower.tail = missing, log.p = missing )`: AD implementation of pgamma

- `pgamma( q = num, shape = num, rate = missing, scale = num., lower.tail = missing, log.p = missing )`: Default method
- `ppois(q = ad, lambda = ad, lower.tail = missing, log.p = missing)`: AD implementation of ppois
- `ppois(q = num, lambda = num, lower.tail = missing, log.p = missing)`: Default method
- `pexp(q = ad, rate = ad., lower.tail = missing, log.p = missing)`: AD implementation of pexp
- `pexp(q = num, rate = num., lower.tail = missing, log.p = missing)`: Default method
- `pweibull( q = ad, shape = ad, scale = ad., lower.tail = missing, log.p = missing )`: AD implementation of pweibull
- `pweibull( q = num, shape = num, scale = num., lower.tail = missing, log.p = missing )`: Default method
- `pbeta( q = ad, shape1 = ad, shape2 = ad, ncp = missing, lower.tail = missing, log.p = missing )`: AD implementation of pbeta
- `pbeta( q = num, shape1 = num, shape2 = num, ncp = missing, lower.tail = missing, log.p = missing )`: Default method
- `qnorm(p = ad, mean = ad., sd = ad., lower.tail = missing, log.p = missing)`: AD implementation of qnorm
- `qnorm(p = num, mean = num., sd = num., lower.tail = missing, log.p = missing)`: Default method
- `qgamma( p = ad, shape = ad, rate = missing, scale = ad., lower.tail = missing, log.p = missing )`: AD implementation of qgamma
- `qgamma( p = num, shape = num, rate = missing, scale = num., lower.tail = missing, log.p = missing )`: Default method
- `qexp(p = ad, rate = ad., lower.tail = missing, log.p = missing)`: AD implementation of qexp
- `qexp(p = num, rate = num., lower.tail = missing, log.p = missing)`: Default method
- `qweibull( p = ad, shape = ad, scale = ad., lower.tail = missing, log.p = missing )`: AD implementation of qweibull
- `qweibull( p = num, shape = num, scale = num., lower.tail = missing, log.p = missing )`: Default method
- `qbeta( p = ad, shape1 = ad, shape2 = ad, ncp = missing, lower.tail = missing, log.p = missing )`: AD implementation of qbeta
- `qbeta( p = num, shape1 = num, shape2 = num, ncp = missing, lower.tail = missing, log.p = missing )`: Default method
- `besselK(x = ad, nu = ad, expon.scaled = missing)`: AD implementation of besselK
- `besselK(x = num, nu = num, expon.scaled = missing)`: Default method
- `besselI(x = ad, nu = ad, expon.scaled = missing)`: AD implementation of besselI
- `besselI(x = num, nu = num, expon.scaled = missing)`: Default method
- `besselJ(x = ad, nu = ad)`: AD implementation of besselJ
- `besselJ(x = num, nu = num)`: Default method

- `besselY(x = ad, nu = ad)`: AD implementation of besselY
- `besselY(x = num, nu = num)`: Default method
- `dbinom_robust()`: AD implementation
- `dsn()`: AD implementation
- `dSHASHo()`: AD implementation
- `dtweedie()`: AD implementation
- `dnbinom_robust()`: AD implementation
- `dnbinom2()`: AD implementation
- `dlgamma()`: AD implementation
- `logspace_add()`: AD implementation
- `logspace_sub()`: AD implementation
- `dnorm(x = ad, mean = ad., sd = ad., log = logical.)`: AD implementation of dnorm
- `dnorm(x = num, mean = num., sd = num., log = logical.)`: Default method
- `dnorm(x = osa, mean = ANY, sd = ANY, log = ANY)`: OSA implementation
- `dnorm(x = simref, mean = ANY, sd = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `dlnorm(x = ANY, meanlog = ANY, sdlog = ANY, log = ANY)`: AD implementation of dlnorm.
- `dlnorm(x = osa, meanlog = ANY, sdlog = ANY, log = ANY)`: OSA implementation.
- `dlnorm(x = num, meanlog = num., sdlog = num., log = logical.)`: Default method.
- `plogis( q = advector, location = missing, scale = missing, lower.tail = missing, log.p = missing )`: Minimal AD implementation of plogis
- `qlogis( p = advector, location = missing, scale = missing, lower.tail = missing, log.p = missing )`: Minimal AD implementation of qlogis
- `dcompois()`: Conway-Maxwell-Poisson. Calculate density.
- `dcompois2()`: Conway-Maxwell-Poisson. Calculate density parameterized via the mean.
- `pbinom(q = ad, size = ad, prob = ad, lower.tail = missing, log.p = missing)`: AD implementation of pbinom
- `pbinom(q = num, size = num, prob = num, lower.tail = missing, log.p = missing)`: Default method
- `dmultinom(x = ad, size = ad., prob = ad, log = logical.)`: AD implementation of dmultinom
- `dmultinom(x = num, size = num., prob = num, log = logical.)`: Default method
- `dmultinom(x = osa, size = ANY, prob = ANY, log = ANY)`: OSA implementation
- `dmultinom(x = simref, size = ANY, prob = ANY, log = ANY)`: Simulation implementation. Modifies `x` and returns zero.
- `dmultinom(x = ANY, size = ANY, prob = ANY, log = ANY)`: Default implementation that checks for invalid usage.

Examples

```
MakeTape( function(x) pnorm(x), x=numeric(5))$jacobian(1:5)
```

expAv *Matrix exponential of sparse matrix multiplied by a vector.*

Description

Calculates `expm(A) %*% v` using plain series summation. The number of terms is determined adaptively when `uniformization=TRUE`. The uniformization method essentially pushes the spectrum of the operator inside a zero centered disc, within which a uniform error bound is available. If `A` is a generator matrix (i.e. `expm(A)` is a probability matrix) and if `v` is a probability vector, then the relative error of the result is bounded by `tol`.

Usage

```
expAv(A, v, transpose = FALSE, uniformization = TRUE, tol = 1e-08, ...)
```

Arguments

`A`	Sparse matrix (usually a generator)
`v`	Vector (or matrix)
`transpose`	Calculate `expm(t(A)) %*% v` ? (faster due to the way sparse matrices are stored)
`uniformization`	Use uniformization method?
`tol`	Accuracy if A is a generator matrix and v a probability vector.
`...`	Extra configuration parameters

Details

Additional supported arguments via `...` currently include:

- `Nmax` Use no more than this number of terms even if the spcified accuracy cannot be met.
- `warn` Give warning if number of terms is truncated by `Nmax`.
- `trace` Trace the number of terms when it adaptively changes.

Value

Vector (or matrix)

References

Grassmann, W. K. (1977). Transient solutions in Markovian queueing systems. *Computers & Operations Research*, 4(1), 47–53.

Sherlock, C. (2021). Direct statistical inference for finite Markov jump processes via the matrix exponential. *Computational Statistics*, 36(4), 2863–2887.

Examples

```
set.seed(1); A <- Matrix::rsparsematrix(5, 5, .5)
expAv(A, 1:5) ## Matrix::expm(A) %*% 1:5
F <- MakeTape(function(x) expAv(A*x, 1:5), 1)
F(1)
F(2) ## More terms needed => trigger retaping
```

`Interpolation` *Interpolation*

Description

Some interpolation methods are available to be used as part of 'RTMB' objective functions.

Usage

```
interpol1Dfun(z, xlim = c(1, length(z)), ...)

interpol2Dfun(z, xlim = c(1, nrow(z)), ylim = c(1, ncol(z)), ...)

## S4 method for signature 'ANY,advector,ANY,missing'
splinefun(x, y, method = c("fmm", "periodic", "natural"))

## S4 method for signature 'advector,missing,ANY,missing'
splinefun(x, method = c("fmm", "periodic", "natural"))
```

Arguments

`z`	Matrix to be interpolated
`xlim`	Domain of x
`...`	Configuration parameters
`ylim`	Domain of y
`x`	spline x coordinates
`y`	spline y coordinates
`method`	Same as for the stats version, however only the three first are available.

Details

`interpol1Dfun` and `interpol2Dfun` are kernel smoothers useful in the case where you need a 3rd order *smooth* representation of a *data* vector or matrix. A typical use case is when a high-resolution map needs to be accessed along a random effect trajectory. Both 1D and 2D cases accept an 'interpolation radius' parameter (default R=2) controlling the degree of smoothness. Note, that only the value R=1 will match the data exactly, while higher radius trades accuracy for smoothness. Note also that these smoothers do not attempt to extrapolate: The returned value will be `NaN` outside the valid range (`xlim` / `ylim`).

`splinefun` imitates the corresponding `stats` function. The AD implementation (in contrast to `interpol1Dfun`) works for parameter dependent y-coordinates.

Value

function of x.

function of x and y.

Functions

- `interpol1Dfun()`: Construct a kernel smoothed representation of a vector.
- `interpol2Dfun()`: Construct a kernel smoothed representation of a matrix.
- `splinefun(x = ANY, y = advector, method = ANY, ties = missing)`: Construct a spline function.
- `splinefun(x = advector, y = missing, method = ANY, ties = missing)`: Construct a spline function.

Examples

```
## ======= interpol1D
## R=1 => exact match of observations
f <- interpol1Dfun(sin(1:10), R=1)
layout(t(1:2))
plot(sin(1:10))
plot(f, 1, 10, add=TRUE)
title("R=1")
F <- MakeTape(f, 0)
F3 <- F$jacfun()$jacfun()$jacfun()
plot(Vectorize(F3), 1, 10)
title("3rd derivative")
## ======= interpol2D
## R=1 => exact match of observations
f <- interpol2Dfun(volcano, xlim=c(0,1), ylim=c(0,1), R=1)
f(0,0) == volcano[1,1]   ## Top-left corner
f(1,1) == volcano[87,61] ## Bottom-right corner
## R=2 => trades accuracy for smoothness
f <- interpol2Dfun(volcano, xlim=c(0,1), ylim=c(0,1), R=2)
f(0,0) - volcano[1,1]    ## Error Top-left corner
F <- MakeTape(function(x) f(x[1],x[2]), c(.5,.5))
## ======= splinefun
T <- MakeTape(function(x){
   S <- splinefun(sin(x))
   S(4:6)
}, 1:10)
```

`MVgauss` *Multivariate Gaussian densities*

Description

Multivariate Gaussian densities

Usage

```
dmvnorm(x, mu = 0, Sigma, log = FALSE, scale = 1)

dgmrf(x, mu = 0, Q, log = FALSE, scale = 1)

dautoreg(x, mu = 0, phi, log = FALSE, scale = 1)

dseparable(...)

unstructured(k)
```

Arguments

x	Density evaluation point
mu	Mean parameter vector
Sigma	Covariance matrix
log	Logical; Return log density?
scale	Extra scale parameter - see section 'Scaling'.
Q	Sparse precision matrix
phi	Autoregressive parameters
...	Log densities
k	Dimension

Details

Multivariate normal density evaluation is done using `dmvnorm()`. This is meant for dense covariance matrices. If *many evaluations* are needed for the *same covariance matrix* please note that you can pass matrix arguments: When `x` is a matrix the density is applied to each row of `x` and the return value will be a vector (length = `nrow(x)`) of densities.

The function `dgmrf()` is essentially identical to `dmvnorm()` with the only difference that `dgmrf()` is specified via the *precision* matrix (inverse covariance) assuming that this matrix is *sparse*.

Autoregressive density evaluation is implemented for all orders via `dautoreg()` (including the simplest AR1). We note that this variant is for a *stationary*, *mean zero* and *variance one* process. FIXME: Provide parameterization via partial correlations.

Separable extension can be constructed for an unlimited number of inputs. Each input must be a function returning a *gaussian mean zero* **log** density. The output of `dseparable` is another **log** density which can be evaluated for array arguments. For example `dseparable(f1,f2,f3)` takes as input a 3D array `x`. `f1` acts in 1st array dimension of `x`, `f2` in 2nd dimension and so on. In addition to `x`, parameters `mu` and `scale` can be supplied - see below.

Value

Vector of densities.

Functions

- `dmvnorm()`: Multivariate normal distribution. OSA-residuals can be used for argument `x`.
- `dgmrf()`: Multivariate normal distribution. OSA is *not* implemented.
- `dautoreg()`: Gaussian stationary mean zero AR(k) density
- `dseparable()`: Separable extension of Gaussian log-densities
- `unstructured()`: Helper to generate an unstructured correlation matrix to use with `dmvnorm`

Scaling

All the densities accept a `scale` argument which replaces `SCALE` and `VECSCALE` functionality of TMB. Scaling is applied elementwise on the residual `x-mu`. This works as expected when `scale` is a *scalar* or a *vector* object of the same length as `x`. In addition, `dmvnorm` and `dgmrf` can be scaled by a vector of length equal to the covariance/precision dimension. In this case the `scale` parameter is recycled by row to meet the special row-wise vectorization of these densities.

Unstructured correlation

Replacement of `UNSTRUCTURED_CORR` functionality of TMB. Constuct object using `us <- unstructured(k)`. Now `us` has two methods: `x <- us$parms()` gives the parameter vector used as input to the objective function, and `us$corr(x)` turns the parameter vector into an unstructured correlation matrix.

Examples

```
func <- function(x, sd, parm, phi) {
   ## IID N(0, sd^2)
   f1 <- function(x)sum(dnorm(x, sd=sd, log=TRUE))
   Sigma <- diag(2) + parm
   ## MVNORM(0, Sigma)
   f2 <- function(x)dmvnorm(x, Sigma=Sigma, log=TRUE)
   ## AR(2) process
   f3 <- function(x)dautoreg(x, phi=phi, log=TRUE)
   ## Separable extension (implicit log=TRUE)
   -dseparable(f1, f2, f3)(x)
}
parameters <- list(x = array(0, c(10, 2, 10)), sd=2, parm=1, phi=c(.9, -.2))
obj <- MakeADFun(function(p)do.call(func, p), parameters, random="x")
## Check that density integrates to 1
obj$fn()
## Check that integral is independent of the outer parameters
obj$gr()
## Check that we can simulate from this density
s <- obj$simulate()
```

`OSA-residuals` *Recursive quantile residuals*

Description

OSA residuals are computed using the function `oneStepPredict`. For this to work, you need to mark the observation inside the objective function using the OBS function. Thereafter, residual calculation is as simple as `oneStepPredict(obj)`. However, you probably want specify a `method` to use.

Usage

```
oneStepPredict(
  obj,
  observation.name = names(obj$env$obs)[1],
  data.term.indicator = "_RTMB_keep_",
  ...
)

## S3 method for class 'osa'
x[...]

## S3 method for class 'osa'
length(x)

## S3 method for class 'osa'
dim(x)

## S3 method for class 'osa'
is.array(x)

## S3 method for class 'osa'
is.matrix(x)
```

Arguments

`obj`	TMB model object (output from `MakeADFun`)
`observation.name`	Auto detected - use the default
`data.term.indicator`	Auto detected - use the default
`...`	Passed to `TMB::`oneStepPredict - **please carefully read the documentation**, especially the `method` argument.
`x`	Object of class 'osa'

Value

`data.frame` with standardized residuals; Same as oneStepPredict.

Functions

- `oneStepPredict()`: Calculate the residuals. See documentation of `TMB::`oneStepPredict.
- `[`: Subset observations marked for OSA calculation. This function makes sure that when you subset an observation of class `"osa"` such as `obs <- new("osa", x=advector(matrix(1:10,2)), keep = cbind(rep(TRUE,10),FALSE,FALSE))` the 'keep' attribute will be adjusted accordingly `obs[,1:2]`
- `length(osa)`: Equivalent of length
- `dim(osa)`: Equivalent of dim
- `is.array(osa)`: Equivalent of is.array
- `is.matrix(osa)`: Equivalent of is.matrix

Examples

```
set.seed(1)
rw <- cumsum(.5*rnorm(20))
obs <- rpois(20, lambda=exp(rw))
func <- function(p) {
  obs <- OBS(obs) ## Mark 'obs' for OSA calculation on request
  ans <- 0
  jump <- c(p$rw[1], diff(p$rw))
  ans <- ans - sum(dnorm(jump, sd=p$sd, log=TRUE))
  ans <- ans - sum(dpois(obs, lambda=exp(p$rw), log=TRUE))
  ans
}
obj <- MakeADFun(func,
                 parameters=list(rw=rep(0,20), sd=1),
                 random="rw")
nlminb(obj$par, obj$fn, obj$gr)
res <- oneStepPredict(obj,
                      method="oneStepGeneric",
                      discrete=TRUE,
                      range=c(0,Inf))$residual
```

Simulation *Simulation*

Description

An RTMB objective function can be run in 'simulation mode' where standard likelihood evaluation is replaced by corresponding random number generation. This facilitates automatic simulation under some restrictions. Simulations can be obtained directly from the model object by `obj$simulate()` or used indirectly via checkConsistency.

Usage

```
simref(n)

## S3 replacement method for class 'simref'
dim(x) <- value

## S3 method for class 'simref'
length(x)

## S3 method for class 'simref'
dim(x)

## S3 method for class 'simref'
is.array(x)

## S3 method for class 'simref'
is.matrix(x)

## S3 method for class 'simref'
as.array(x, ...)

## S3 method for class 'simref'
is.na(x)

## S3 method for class 'simref'
x[...]

## S3 replacement method for class 'simref'
x[...] <- value

## S3 method for class 'simref'
Ops(e1, e2)

## S3 method for class 'simref'
Math(x, ...)

## S3 method for class 'simref'
t(x)

## S3 method for class 'simref'
diff(x, lag = 1L, differences = 1L, ...)

## S3 method for class 'simref'
Summary(..., na.rm = FALSE)
```

Arguments

n	Length

x	Object of class 'simref'
value	Replacement (numeric)
...	Extra arguments
e1	First argument
e2	Second argument
lag	As diff
differences	As diff
na.rm	Ignored

Details

In simulation mode all log density evaluation, involving either random effects or observations, is interpreted as probability assignment.

direct vs indirect Assignments can be 'direct' as for example

```
dnorm(u, log=TRUE) ## u ~ N(0, 1)
```

or 'indirect' as in

```
dnorm(2*(u+1), log=TRUE) ## u ~ N(-1, .25)
```

Indirect assignment works for a limited set of easily invertible functions - see `methods(class="simref")`.

Simulation order Note that probability assignments are sequential: All information required to draw a new variable must already be simulated. Vectorized assignment implicitly occurs element-wise from left to right. For example the assignment

```
dnorm(diff(u), log=TRUE)
```

is not valid without a prior assignment of `u[1]`, e.g.

```
dnorm(u[1], log=TRUE)
```

Supported distributions Assignment must use supported density functions. I.e.

```
dpois(N, exp(u), log=TRUE)
```

cannot be replaced by

```
N * u - exp(u)
```

The latter will have no effect in simulation mode (the simulation will be `NA`).

Return value Note that when in simulation mode, the density functions all return zero. The actual simulation is written to the input argument by reference. This is very unlike standard R semantics.

Value

An object with write access to store the simulation.

Functions

- `simref()`: Construct `simref`
- `dim(simref) <- value`: Equivalent of dim<-
- `length(simref)`: Equivalent of length

- `dim(simref)`: Equivalent of dim
- `is.array(simref)`: Equivalent of is.array
- `is.matrix(simref)`: Equivalent of is.matrix
- `as.array(simref)`: Equivalent of as.array
- `is.na(simref)`: Equivalent of is.na
- `[`: Equivalent of [
- `` `[`(simref) <- value ``: Equivalent of [<-
- `Ops(simref)`: Equivalent of Ops
- `Math(simref)`: Equivalent of Math
- `t(simref)`: Equivalent of t
- `diff(simref)`: Equivalent of diff
- `Summary(simref)`: Summary operations are not invertible and will throw an error.

Examples

```
s <- simref(4)
s2 <- 2 * s[1:2] + 1
s2[] <- 7
s ## 3 3 NA NA
## Random walk
func <- function(p) {
  u <- p$u
  ans <- -dnorm(u[1], log=TRUE) ## u[1] ~ N(0,1)
  ans <- ans - sum(dnorm(diff(u), log=TRUE)) ## u[i]-u[i-1] ~ N(0,1)
}
obj <- MakeADFun(func, list(u=numeric(20)), random="u")
obj$simulate()
```

Tape *The AD tape*

Description

The AD tape as an R function

Usage

```
MakeTape(f, x)

## S3 method for class 'Tape'
x$name

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

```
TapeConfig(
  comparison = c("NA", "forbid", "tape", "allow"),
  atomic = c("NA", "enable", "disable"),
  vectorize = c("NA", "disable", "enable")
)

DataEval(f, x)

GetTape(obj, name = c("ADFun", "ADGrad", "ADHess"), warn = TRUE)
```

Arguments

f	R function
x	numeric vector
name	Name of a tape method
...	Ignored
comparison	Set behaviour of AD comparison (">","==", etc).
atomic	Set behaviour of AD BLAS operations (notably matrix multiply).
vectorize	Enable/disable AD vectorized 'Ops' and 'Math'.
obj	Output from MakeADFun
warn	Give warning if obj was created using another DLL?

Details

A 'Tape' is a representation of a function that accepts *fixed size* numeric input and returns *fixed size* numeric output. The tape can be constructed using `F <- MakeTape(f, x)` where `f` is a standard *differentiable* R function (or more precisely: One using only functions that are documented to work for AD types). Having constructed a tape F, a number of methods are available:

Evaluation:

- Normal function evaluation 'F(x)' for numeric input.
- AD evaluation 'F(x)' as part of other tapes.
- Jacobian calculations using 'F$jacobian(x)'.

Transformation:

- Get new tape representing the Jacobian using `F$jacfun()`.
- Get new tape representing the sparse Jacobian using `F$jacfun(sparse=TRUE)`.
- Get new tape representing the Laplace approximation using `F$laplace(indices)`.
- Get new tape representing the Saddle Point approximation using `F$laplace(indices,SPA=TRUE)`.
- Get new tape representing the optimum (minimum) wrt `indices` by `F$newton(indices)`.
- Get a 'shared pointer' representation of a tape using `F$atomic()`.
- Get tape of a single node by `F$node(index)` (mainly useful for derivative debugging).

Modification:

- Simplify internal representation of a tape using `F$simplify()`.

Extract tape information:

- Get internal parameter vector by `F$par()`.
- Get computational graph by `F$graph()`.
- Print the tape by `F$print()`.
- Get internal arrays as a `data.frame` by `F$data.frame()`.

Value

Object of class `"Tape"`.

Methods (by generic)

- `$`: Get a tape method.
- `print(Tape)`: Print method

Functions

- `MakeTape()`: Generate a 'Tape' of an R function.
- `TapeConfig()`: Global configuration parameters of the tape (experts only!) **comparison** By default, AD comparison gives an error (`comparison="forbid"`). This is the safe and recommended behaviour, because comparison is a non-differentiable operation. If you are building a tape that requires indicator functions e.g. `f(x)*(x<0)+g(x)*(x>=0)` then use `comparison="tape"` to add the indicators to the tape. A final option `comparison="allow"` exists for testing/illustration purposes. Do not use.
- `DataEval()`: Move a chunk of data from R to the tape by evaluating a normal R function (replaces TMB functionality 'DATA_UPDATE').
- `GetTape()`: Extract tapes from a model object created by `MakeADFun`.

Examples

```
F <- MakeTape(prod, numeric(3))
show(F)
F$print()
H <- F$jacfun()$jacfun() ## Hessian tape
show(H)
#### Handy way to plot the graph of F
if (requireNamespace("igraph")) {
   G <- igraph::graph_from_adjacency_matrix(F$graph())
   plot(G, vertex.size=17, layout=igraph::layout_as_tree)
}
## Taped access of an element of 'rivers' dataset
F <- MakeTape(function(i) DataEval( function(i) rivers[i] , i), 1 )
F(1)
F(2)
```

TMB-interface *Interface to TMB*

Description

Interface to TMB

Usage

```
MakeADFun(
  func,
  parameters,
  random = NULL,
  profile = NULL,
  integrate = NULL,
  intern = FALSE,
  map = list(),
  ADreport = FALSE,
  silent = FALSE,
  ridge.correct = FALSE,
  ...
)

sdreport(obj, ...)

ADREPORT(x)

REPORT(x)

getAll(..., warn = TRUE)

OBS(x)

checkConsistency(obj, fast = TRUE, ...)
```

Arguments

func	Function taking a parameter list (or parameter vector) as input.
parameters	Parameter list (or parameter vector) used by func.
random	As MakeADFun.
profile	As MakeADFun.
integrate	As MakeADFun.
intern	As MakeADFun.
map	As MakeADFun.
ADreport	As MakeADFun.

silent	As MakeADFun.
ridge.correct	Experimental
...	Passed to TMB
obj	TMB model object (output from MakeADFun)
x	Observation object
warn	Give a warning if overwriting an existing object?
fast	Pass observation.name to TMB ?

Details

MakeADFun builds a TMB model object mostly compatible with the **TMB** package and with an almost identical interface. The main difference in **RTMB** is that the objective function **and** the data is now given via a single argument func. Because func can be a *closure*, there is no need for an explicit data argument to MakeADFun (see examples).

Value

TMB model object.

Functions

- MakeADFun(): Interface to MakeADFun.
- sdreport(): Interface to sdreport.
- ADREPORT(): Can be used inside the objective function to report quantities for which uncertainties will be calculated by sdreport.
- REPORT(): Can be used inside the objective function to report quantities via the model object using obj$report().
- getAll(): Can be used to assign all parameter or data objects from a list inside the objective function.
- OBS(): Mark the observation to be used by either oneStepPredict or by obj$simulate. If your objective function is using an observation x, you simply need to run x <- OBS(x) *inside the objective function*. This will (1) allow oneStepPredict to change the class of x to "osa" (OSA-residuals) or (2) allow obj$simulate to change the class of x to "simref" (Simulation) on request.
- checkConsistency(): Interface to checkConsistency.

Examples

```
## Objective with data from the user workspace
data(rivers)
f <- function(p) { -sum(dnorm(rivers, p$mu, p$sd, log=TRUE)) }
obj <- MakeADFun(f, list(mu=0, sd=1), silent=TRUE)
opt <- nlminb(obj$par, obj$fn, obj$gr)
sdreport(obj)
## Same objective with an explicit data argument
f <- function(p, data) { -sum(dnorm(data, p$mu, p$sd, log=TRUE)) }
```

```
cmb <- function(f, d) function(p) f(p, d) ## Helper to make closure
obj <- MakeADFun(cmb(f, rivers), list(mu=0, sd=1), silent=TRUE)
## 'REML trick'
obj2 <- MakeADFun(cmb(f, rivers), list(mu=0, sd=1), random="mu", silent=TRUE)
opt2 <- nlminb(obj2$par, obj2$fn, obj2$gr)
sdreport(obj2) ## Compare with sd(rivers)
## Single argument vector function with numeric 'parameters'
fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
obj <- MakeADFun(fr, numeric(2), silent=TRUE)
nlminb(c(-1.2, 1), obj$fn, obj$gr, obj$he)
```

`%~%` *Distributional assignment operator*

Description

Distributional assignment operator

Usage

```
x %~% distr
```

Arguments

`x`	LHS; Random effect or data for which distribution assignment applies
`distr`	RHS; Distribution expression

Details

Provides a slightly simplified syntax *inspired by*, but *not* compatible with, other probabilistic programming languages (e.g. BUGS/JAGS):

- `x %~% distribution(...)` is syntactic sugar for `.nll <- .nll - sum(distribution(x,...,log=TRUE))`
- The variable `.nll` is automatically initialized to `0` and returned on exit.

Value

The updated value of the hidden variable `.nll`.

Note

If the shorter name ~ is preferred, it can be locally overloaded using `"~" <- RTMB::"%~%"`.

Examples

```
f <- function(parms) {
  getAll(parms)
  x %~% dnorm(mu, 1)
  y %~% dpois(exp(x))
}
p <- list(mu=0, x=numeric(10))
y <- 1:10
obj <- MakeADFun(f, p, random="x")
```

Index