Package: offarray (via r-universe)

October 15, 2024

Title Arrays with non-1 offsets

Description Arrays with non-1 offsets

Author Mark V. Bravington <markb2@summerinsouth.net>

Maintainer Mark V. Bravington <markb2@summerinsouth.net>

Depends R (>= 4.3)

Imports utils, Rcpp (>= 0.12.11), mvbutils (>= 2.9)

LinkingTo Rcpp

Suggests knitr

CommentedOutVignetteBuilder knitr

ByteCompile yes

License GPL (>=2)

Version 2.0.64

Collate offarray.R Cloaders_offarray.R

Repository https://markbravington.r-universe.dev

RemoteUrl https://github.com/markbravington/offarray

RemoteRef HEAD

RemoteSha 5d52487bee9f4a1d15e231db08536e884d4ec40a

Contents

offarray-package *How to use the offarray package*

Description

offarray is like a regular array (or matrix, or vector) but its indices do not have to start at +1. So that, for example:

```
dumbo <- offarray( 1:4, dim=c( 2, 2), first=c( 3, 5))
dumbo
#       [,5] [,6]
#  [3,]    1    3
#  [4,]    2    4
dumbo[4,6] # 4
```

You can of course have character indices (names / dimnames), too. See offarray for more detail. Use autoloop to get *fast* for-loop-functionality; nested for-loops over offarray objects are just toooo slow. There is a vignette for autoloop, but "technical difficulties" mean that for now you can only view it via:

```
RShowDoc( "autoloop", package="offarray") # PDF
# or
RShowDoc( "autoloop", type="html", package="offarray") # HTML
```

I find offarray absolutely indispensible, especially for population dynamics work where datasets do not normally begin at birth-of-Christ, and ages often are only tracked above some non-1 value. If you are a misguided C nut who believes fervently that all array indices *should* all start at 0, you may love it too, albeit for the wrong reasons...

NB indices do still have to be increasing integer sequences with stepsize 1; it's only the starting-point that you can tweak.

Byte compiler quirks are now automatically worked around

I have fixed things in v2.0, so you don't have to read this bit...

Subassignments, eg x[i,j]<-y, used to lead to a problematic interaction between package **offarray** and the default behaviour of R's byte-compiler. Therefore, old versions of offarray (eg v1.1.x) used to turn *off* the byte-compiler automatically immediately before offarray itself was loaded, via a devious trick in package **nobcom**.

It turns out that R's byte-compiler subtly changes the way subassignment (an R nightmare!) is handled, compared to uncompiled execution eg at the command-line. After years of wondering about this in the absence of any information, in 2023 I *eventually* found a rather well-concealed explanation (p73 of "A Byte Code Compiler for R" by Luke Tierney, 23/08/2023). This behaviour is apparently by design.

autoloop *Loopless array calculations*

Description

Suppose you basically want to write some for-loops to fill in an entire (off)array, and you could easily write the loops and the innermost expression in C. But you can't really write the loops in R, because they would take *forever*, and you don't want to resort to C because it's a hideous PITA. Instead, you can do it all in R with one call to `autoloop`, which "does all the loops at once". You just need to say what the loop-indices are (they will also form the dimensions of the result), and what the expression(s)-to-evaluate are (i.e. the "meat" of the "for-loop sandwich"). You can also automatically sum over some indices that aren't needed in the final result. You can return several offarray variables from one call to `autoloop`.

Thus `autoloop` lets you easily and quickly do e.g. generalized dot products, without writing for-loops or contorting yourself to somehow vectorize all the expressions in direct R terms (something that I find excruciatingly difficult); see **Examples**, and the vignette "Fast loops with offarrays". (For now, you have to do that with `RShowDoc("autoloop",package="offarray")`.

`autoloop` is fast, and also (tries to...) give you nice informative error messages when you screw up— e.g., what the offending index values are— which you won't get with base-R. (There is also a debugging facility: see below.) I strongly recommended using `autoloop` as much as possible to replace for-loops, at least with `offarray`. However, you can't always avoid *some* for-looping, and that's fine. One situation is population-dynamics with an annual time step, as per **Examples**. The section **Rules** belwo lists a few restrictions (e.g. no `if`, though you can use `ifelse`), and gives tips on coding style (how to avoid `ifelse` ...); see also the vignette mentioned above.

The `SUMOVER` argument that lets you automatically "contract" the result(s) just before returning, by summing over certain loop variables (generalizing `rowSums` and `colSums`). This lets you do a matrix-multiply in one step:

```
# NB this uses base-R matrices, not offarrays; either would work
M1 <- matrix( 1:6, 2, 3)
M2 <- matrix( 10+1:12, 3, 4)
prod <- autoloop(
  I=1:2, J=1:4,
  SUMOVER=list( K=1:3),
  M1[I,K] * M2[K,J]
) # same as M1 %*% M2
```

You can also do a contraction manually in a second step, by calling `sumover()`, although that's less efficient; sometimes it's unavoidable, though, e.g. when you want the denominator for Bayes' theorem. See (future) vignette.

`autoloop` is designed for speed and for economy of memory use, especially when you're contracting over temporary indices. It does a whole bunch of calcuations with a single R operation in a vectorized way, but only in chunks of size `.BLOCKSIZE`. The latter's default should be OK, but you might be able to get a speed-up by tweaking it to your particular case.

For convenience, if you are repeatedly using the same large set of indices in multiple `autoloop` calls, you may want to pre-assign them to a `list` called eg `all_my_indices` and use `autoloop( indices=all_my_indices, <expr>)`.

Debugging: The error messages from `autoloop` aren't always perfectly informative; I'm working on that, but sometimes you will just get something that is not obvious. So, what then?

First, I always use my debug package to debug things, because it's Just Better. And, frankly, so should you... so the assumption here is that you are using it. Anyway, when you've gotten an puzzling situation inside a call to `autoloop`, it's not easy to figure out what's going on by simply doing `mtrace(autoloop)`; the execution of (a modified version of) *your* code doesn't happen until deep inside the code of `autoloop` itself. To make things easier, do the following immediately before executing the offending `autoloop` call: `mtrace(eval_autoloop_guts)`. Then it will step (almost) directly into your code, doing a block of loop-indices at once. Note that your code will appear slightly modified (mainly for array accesses); that's deliberate. All your statements will (or should) return vectors, by design. Remember to do `mtrace(eval_autoloop_guts,F)` after you have come out.

It's possible that the same technique will work with `browse` or other inferior debugging tools :)

Rules: What can you do/not-do in an `autoloop` expr? Well, you can think of each statement within your `expr` as an assignment operation in C, nested within for-loops. At the end of each evaulation of `expr`, each returned variable (or the very last assignment, if there is no explicit `return`) is stored into an array of the same name, at the element shown by the indices. Thus, this call in R:

```
Y <- autoloop( i=1:5, j=2:6, {
    divij <- b[ i] / c[j]
    i*j + divij
  })
```

is functionally equivalent to the following C code:

```
for( i=1; i<=5; i++)
  for( j=2; j<=6; j++){
    double divij = b(i) / c(j);
    double RETURNO = i*j + divij;
    // Store the returned value(s); just one in this case
    Y(i,j) = RETURNO;
  };
```

Here's an example with `SUMOVER`, a simple dot-product:

```
Y <- autoloop( i=1:5, SUMOVER=list( j=7:9), x[ i,j] * p[j] )
```

In C, this would be:

```
for( i=1; i<=5; i++){
  double SUM = 0;
  for( j=7; j<=9; j++){
    SUM += x(i,j) * p( j);
  };
  Y( i) = SUM;
};
```

These analogies covers most situations. Under the hood, however, the calculations in R are not scalar; they are automatically vectorized (with some magic so that array accesses work as shown), and the casting of all the results into `offarray` format only happens after all the loops are complete. This has some implications for what you can and can't write inside `expr`. I won't try to give an exhaustive set of rules here, but here's a few pointers:

- Every statement must be a `<-` assignment to a variable that looks like a scalar (no subassignments etc).
- You can't use `if` (nor `for`, `while`, `repeat`, `break`, `next`); you can use `ifelse` instead, but see the vignette for guidance.
- If any functions are called, they should accept vectorized arguments—apart from any arguments that are fixed outside the loop, such as in `dpois( x, y, log=TRUE)`— and should return a vector result of the same length.
- Each statement can refer to index variables, to variables known in the calling environment, and to "scalars" created by earlier statements within `expr`. But you can't do eg `{ Y <- i+j; X <- Y[j]}` because (if it actually meant anything...) it would be treating `Y` as not-a-scalar. `Y` only gets turned into an `offarray` after all the loops are completed.
- References to arrays (matrices, vectors) must specify *all* indices individually; `x[i,j,k]` is good, `x[]` or `x[,j,k]` or `x[cbind(i,j)]` are not.
- Subscripting of arrays/matrices/vectors can use numeric and character indices (the latter only if the relevant `dimnames` exists), but not logical indices: `x[i>j]` won't work, because it is really a function-call but does not necessarily return a result the same length as `i` and `j`.
- You probably can't use `%*%`.

Basically, if you try something fancy and it doesn't work, that means you shouldn't be doing it. Do something else instead!

Details: For those familiar with R's `outer` function: `autoloop` is conceptually similar, but generalized and generally better. Its modus operandi is similar to `outer`, and amounts to almost this:

- use `expand.grid` to make a `data.frame` holding all combinations of indices;
- use `substitute` to ensure array lookups are handled correctly;
- `eval` to apply `expr` to the entire `data.frame` at once, without loops;
- add dimension info etc to each returnee, to turn it into an `offarray`.

The main practical difference to that, is that `partial_expand_dot_grid` (qv) is used to ensure that only a few thousand combinations of indices are handled at once. That minimizes memory problems in large offarrays, especially but not only when `SUMOVER` is used.

The trick for handling array lookups, is to replace `x[i,j]` with something close to `x[cbind(i,j)]`, which works automatically with R's matrix-subscripting-of-arrays mechanism (something which is too-little-known; it's buried deep in `?Extract`). There is an equivalent in `offarray`, via `MATSUB`. The details in `autoloop` are a little bit more complicated, but that's the idea.

Usage

```
## See examples; this formal version of USAGE makes no sense !
## Normally eg:
## autoloop( AGE=<something>, YEAR=<something>,
##   { <multi-part expression involving AGE and YEAR> }
```

```
## )
autoloop(expr, ..., indices=list(), SUMOVER=list(),
  single_object= NA, offarray_wanted= TRUE,
  .BLOCKSIZE=3000, .REPORTSEC=0)
sumover( x, mm, drop=FALSE)
eval_autoloop_guts( expr, envir) ## NEVER call this yourself!!
```

Arguments

`expr`	Expression to evaluate— think of it as the body of a function whose arguments are the dotty-args. It should return either one vector argument, or a named list of vector arguments.
`...`	Named subscript ranges, eg `A=1:5`, `B=c( "CAT", "DOG")`
`indices`	Instead of/as well as `...`, you can supply a bunch of subscripts all-in-one, as a `list`. This is handy when you are repeatedly using similar *sets* of indices; see **Examples**.
`SUMOVER`	named `list` of *additional* subscript ranges, included in the loops for evaluation but summed-over before the result is returned. Don't duplicate them with `....`
`single_object`	Normally you can ignore this, and just let `autoloop` deduce whether `expr` is returning just one (unnamed) object, or a (named) list containing several objects. However, if `expr` does return a single object, then it is always safe to set `single_object=TRUE`, and you should definitely do so if `expr` returns a single object that is (or might be) a named list.
`offarray_wanted`	whether the returned object(s) should be `offarray`, or just regular array/matrix/vector(s).
`x`	(`sumover`) an `offarray`, or at any rate something with named dimnames.
`mm`	(`sumover`) which dimensions to sum over; either a character vector with elements in `names(dimnames( x))`, or a numeric vector.
`drop`	(`sumover`) as per standard R usage, but default is FALSE here because that's better. NB that `drop` will always preserve `offarray`-ness (unless *all* dimensions are contracted), so the `dim` attribute will not necessarily disappear even if it "could".
`.BLOCKSIZE`	how many "cases" (different "rows" aka different sets of values from the subscript ranges) to do at once. Avoids memory barfing. Too small (eg 1e3) seems to slow things down; too big could cause barfing. Default should be fine.
`.REPORTSEC`	how many seconds to let pass before reporting on progress. 0 or silly value means don't report.
`expr, envir`	not your business, since you won't be calling `eval_autoloop_guts` yourself...

Value

`autoloop` returns one or more `offarray`s with dimensions ranging over the input subscripts (except the ones in `SUMOVER`). The result is always an `offarray` even if all first elements are 1. If `expr` returns a list (which must be named), then a list of `offarray`s with the same names will be returned; `extract.named` then conveniently creates all corresponding arrays in the caller's environment. `sumover` returns an `offarray` (or an `array` if the input was not an `offarray`), or a scalar iff *all* dimensions are summed over.

Examples

```
library( mvbutils) # for extract.named, cq, etc; already imported
# Normal offarray creation:
directo <- offarray( 0, dimseq=list( A=1:3, B=cq( CAT, DOG)))
# Or you can do it with autoloop!
quicko <- autoloop( A=1:3, B=cq( CAT, DOG), 0)
# Simplissimo (though using return(), which is often unnecessary):
cicada <- autoloop( A=1:5, Y=1:3, {
  cohort <- Y-A # an intermediate statement
return( cohort %% 17)
})
# Offarray:
cohort <- autoloop( A=0:3, Y=2000:2001, {
  Y-A
})
# Two variables at once:
extract.named( autoloop( A=0:2, Y=2000:2001, {
  cohort <- Y-A
  weird <- Y %% A
returnList( cohort, weird)
}))
# 'indices' feature. Should be same as 'cohort' above.
A_and_Y <- list( A=0:2, Y=2000:2001)
alt_cohort <- autoloop( indices=A_and_Y, {
  Y-A
})
# Lookup:
B <- outer( 1:5, 1:3, '*')
weird2 <- autoloop( A=2:4, Y=1:2, {
return( B[ A, Y]) # ie ALL combos of A & Y
})
# Dot-product: C[i,k] := A[i,j] %[j]% B[j,k] in vecless
A <- matrix( 1:12, 3, 4)
B <- matrix( 1:8, 4, 2)
A %*% B
C <- autoloop( I=1:3, K=1:2, SUMOVER=list( J=1:4), {
  A[ I, J] * B[ J, K]
})
# Errors
yy <- offarray( 0, dimseq=list( 3:4, 5:6))
try( autoloop( i=3:5, j=5:6, yy[ i,j])) # OOB
try( autoloop( i=3:5, j=5:6, yy[ rep( i, 2), rep( j, 2)])) # weird length
try( autoloop( i=3:5, j=5:6, yy[ i, rep( j, 3)])) # one weird length
try( autoloop( i=3:4, j=5:6, yy[ i,j+'word'])) # WHAT were you THINKING?
try( autoloop( i=3:4, j=5:6, yy[ i])) # wrong number of indices
try( autoloop( i=3:4, j=5:6,
    yy[ expression(),environment()])) # general craziness
try( autoloop( i=3:4, j=5:6, meaning_of_life))
# Old-style:
CCold <- autoloop( I=1:3, J=1:4, K=1:2, {
  A[ I, J] * B[ J, K]
})
```

```
Cold <- sumover( CCold, 'J')
# autoloop() inside for-loop (eg for population dynamics)
# Also, idiomatic
YMIN <- 2010
YMAX <- 2014
YEARS <- 2010:2014
AMIN <- 2
APLUS <- 5
AGES <- AMIN:APLUS
N <- offarray( 0, dimseq=list( Y=YEARS, A=AGES))
# Some random margins...
N[ YMIN,] <- runif( length( AGES)) # !!!
N[ , AMIN] <- runif( length( YEARS)) # !!!
# ... and mortality rates...
Z <- 0*N + runif( length( N)) # !!!
# Fill in the pop dyn, each year depending on previous
for( y in (YMIN+1) %upto% YMAX){
  nextN <- autoloop( A=(AMIN+1) %upto% APLUS,
    N[ y-1, A-1] * exp( -Z[ y-1, A-1]) +
    # ... maybe add plus-group, or 0 if not
    (A==APLUS) * (
      N[ y-1, A] * exp( -Z[ y-1, A])
    )
  )
  N[ y, (AMIN+1) %upto% APLUS] <- nextN
}
```

`offarray` *Arrays with non-1 offsets*

Description

An `offarray` is an array where (some) indices may not start at 1. Easiest example is

```
dumbo <- offarray( 1:6, dimseq=list( X=4:6, Y=c( 'aleph', 'beth'))
print( dumbo)
# now e.g. dumbo[ 5, 'aleph'] will give 2
```

Things should "just work" as you expect. However, compared to base-R and to the **Oarray** package which inspired `offarray`, there are a few things that might trip you up. The comprehensive list below looks much worse than it actually is; after the first 3, you might do better to simply read the EXAMPLES.

- For speedy loop operations on `offarray` objects, you should definitely use `autoloop` rather than writing nested for-loops; it's *much* faster and *no* harder.
- from v2.0 onwards, `offarray` and `autoloop` code *should* be compatible with package **RTMB**, with one tiiiiny modification. You will also need to load the package **offartmb**.

- To pass an `offarray` to some existing R array/matrix routine, you *may* need to `as.array` it first to switch to 1-based indexing (basically if that R code is gonna subset it explicitly), or just `c` it the dimensions are irrelevant. That depends on the other code, and it's up to you to figure out. (One example is `matplot`; another, which I might eventually try to fix, is `apply`, as per **Examples**.)

Behind the scenes, each `offarray` has an attribute "offset", a vector which holds the lowest index value for that dimension. You shouldn't need to access the offset directly, andbut there are helper routines to help you with relevant stuff. See **Utility functions** below.

As in base-R, you can also use character subscripts in an `offarray`, for any dimension that has non-empty `dimnames`.

- If you subscript a dimension-that-has-dimnames with a number (as opposed to a character), then it will behave just like a normal R dimension, with negative number meaning "drop this element" etc. (Internally, the offset for such a dimension is stored as NA. You can also set an NA offset for any dimension, to make it behave (almost) exactly like a regular R subscript; it's not quite the same as setting the offset to 1, as shown in **Examples**.)

Now we get into more detailed stuff:

- Negative offsets are interpreted just like positive ones, so `x[-1]` means "give me the minus-one-th element of `x`". (It's not that weird; there are plenty of algorithms where indices can be negative, e.g. Wynn's epsilon method which naturally start at index -1). In base-R, on the other hand, `x[-1]` means "discard the first element of `x`". To get the base-R behaviour in one specific dimension, you can set its offset to NA.
- Subset-extraction keeps (and adjusts if necessary) each non-NA offset, provided that its corresponding subset is increasing and consecutive. Thus, in a 1D case, `x[2]` and `x[2][2]` and `x[2][2][2]` will always return the same thing, whereas base-R would give NAs after the first; and after `y <- x[2]`, the offset of `y` will be 2.
- Subsetting with a *non* consecutive-increasing index, eg `x[c(5,5),]` or `x[5:4,,]`, will set the corresponding offset to NA. This will trigger a warning unless the offset was NA already, or you have explicitly told R to do it via `NOOF`; see **Examples**.
- As in base-R, the idiom `x[]` still returns all of `x`, as an `offarray`. And an "explicitly missing" subset, such as the first index in `x[,5:6]`, means "all of that dimension".
- By default, dimensions are **not** dropped, so that `x[5,3,4:6]` will still return a 3D offarray. Use `x[ SLICE=5, SLICE=3, 4:6]` to drop those two dimensions, to give you 1D 3-element `offarray`. You can set the additional argument `drop=TRUE` to auto-slice, which is convenient if there's several such indices.
- Vector (1D) subsets are available that work like R's normal "treat the whole array as a 1-indexed vector, then extract the specified elements". They are indicated by `x[ VECSUB=1:5] <- 17` or `x[ VECSUB=x>0]`. The return value is always just a vector, of the same type as `x`. Note that, if `x` is a vector (of class `offarray`), then `x[1:5]` treats the `1:5` relative to the offset (so if the offset is -3, `x[1:5]` would take the 5th–9th elements), whereas `x[VECSUB=1:5]` always takes the first five elements.
- Matrix-subsetting works (and respects any non-NA offsets), but you need to tell R you want it, via `x[ MATSUB=mymat]`. If some of your dimensions are character-index rather than numeric-index, you can use a `data.frame` or even `list` (one element per dimension) instead of a numeric `matrix`, again with `x[ MATSUB=my_list_or_data.frame]`. The return value is always just a vector, of the same type as `x`.

- Logical subsets (which must be the same length as the corresponding dimension) are first mapped to numeric ones, then the above rules are applied.
- Character-indexed dimensions, ie with non-NULL `dimnames` (of mode `character`, of course) can be mixed with normal integer-indexed ones, just like in base R. You can subset from character-indexed dimensions using either characters (safer and clearer) or logicals or integers; if integers are used, they always are treated as starting from 1, ie you can't set a non-1 offset for character-indexed dimensions. The offset for such dimensions will be `NA`; as noted above, you can also specify offset `NA` for non-character-indexed dimensions, so that they are subsettable just like base-R.
- There are some weird subsetting options allowed in base-R, that produce IMO inconsistent and unuseful results. I have generally banned these in `offarray`, to protect you from yourself! One example is subset-values that are NA or 0 or out-of-range; in `offarray`, they produce errors. (Except, you can of course use 0 as a perfectly legimimate index provided that the corresponding offset is zero or negative.)
- If any subset is non-empty and zero-length, the return value is a zero-length vector of the same type as `x`, eg from `x[rep(F,length(x))]` or `x[integer(0)]`.
- Subset-replacement, eg `x[5:6,i2>0]`, follows the same rules, *mutatis mutandum*.
- `apply` generally works, but if the result is 1D, its vector

Constructing and deconstructing an offarray: You can construct an `offarray` directly from numeric (or other atomic) values, much like you would a normal array. The easiest way is usually by setting the `dimseq` argument (which then bypasses all other arguments except the values `x`), like so:

```
test <- offarray( 1:6, dimseq=list( X=3:5, Y=c( 'A', 'b')))
```

which means the first dimension ("X"), is numeric-indexed with offset 3, and the second ("Y") is character-indexed.

If you just want a dimension to behave like base-R (ie starting at 1 but with negative subscripts meaning "drop"), but not to have `dimnames`, then the following will work (by magically setting the offset for that dimension to NA):

```
test <- offarray( 1:6, dimseq=list( X=3:5, Y=character( 2)))
```

Instead of `dimseq`, you can specify `dim/first/last` directly; any 2 of the 3 should do, and they will be sanity-checked. However, if any dimension is supposed to be character-indexed, you'd also need to pass `dimnames=list(<something>)` too— so, just using `dimseq=...` is usually easier.

For a 1D offarray, you can use a vector `dimseq` instead of a list of length 1. For example:

```
eco <- offarray( runif( 4), dimseq=0:3)
# same as
veco <- offarray( runif( 4), dimseq=list( 0:3))
```

To "deconstruct", `as.array` has a method for `offarray` objects. Unless `make_dimnames==FALSE`, it turns all the numeric dimensions with non-NA offsets into *character* dimnames that incorporate the offset— so you can see what the original indices were. I prefer this to `unclass` for technical reasons, but `unclass` should also work.

You can also convert directly to an `offarray` from a `table` (ie something returned by `table()`) or from a `data.frame`. Technically, those two cases are S3 methods for the generic "constructor"

function `offarray()`. Note that `table` might not give you the full ranges you expect, if there are no observations in categories where you might expect some. The safest solution is to add the `dimseq` argument explicitly; see **Examples**.

The almost-inverse to `offarray( <data.frame>)` is `as.data.frame` which has a method for `offarray` objects; see **Examples**. Technically: the output is a `data.frame` with one column per dimension of the input, plus a column for the contents, which will be called "response" unless you set the `name_of_response` argument. The other columns will have names "D1", "D2", etc, unless either (i) the input has a *named* `dimnames` attribute, in which the names will be used, or (ii) the argument "add_names" is set to a character vector naming the dimensions. If the input has any `dimnames`, then its non-NULL elements will be used in place of 1,2,3,... etc for the entries in the corresponding columns. Character `dimnames` are kept as `character` columns in the output, *never* converted to horrible factors— regardless of `stringsAsFactors`.

`offarray` offers partial support for naming your dimnames. For example, if your first dimension is Sex and your second dimension is Age, you can create your `offarray` like this:

```
myoff <- offarray( ....., dimnames=list( SEX=c( 'F', 'M'), AGE=NULL))
```

the NULL signifying that Age is numeric. `autoloop` will always do this, and `sumover` expects names-of-dimnames. However, I'm not sure whether absolutely every operation on `offarray` object will propagate names-of-dimnames correctly.

If you don't name the dimnames, then `offarray` will try to figure them out from any names supplied in the vector arguments `first`, `last`, and/or `dim`; it will also check that any such names are consistent with each other, and with names-of-dimnames if the latter is supplied. For example:

```
myoff <- offarray( ..., first=c( SEX=1, 0), last=c( 2, AGE=20),
    dimnames=list( c( 'F', 'M'), NULL)
```

will work, and will produce an `offarray` with names-of-dimnames `SEX` and `AGE` respectively. But this next one won't work, because there's a name clash:

```
myoff <- offarray( 1:42, first=c( SEX=1, 0), last=c( GENDER=2, AGE=20), dimnames=list( c( 'F', 'M'),
```

Calling `dim` on an offarray will often return a *named* vector, related to the previous paragraph. This has pros, eg being able to write `dim(myoff)["SEX"]` or `dimseq(x)$SEX`, but potentially also cons, eg due to R's habit of augmenting names when things are joined via `c()`. Time will tell whether it was a wise decision.

Utility functions: `firstel` (and `firstel<-`), `lastel`, `dimrange`, and `dimseq` are convenience functions for getting the range of an array index, without you having to faff around with offsets directly. They work with regular arrays/matrices, as well as `offarray` objects. Note `dimseq(x)[[i]]` returns `seq_len(dim(x)[i])` if `dimnames(x)[i]` is NULL and `attr(x,"offset")[i]` is NA. Watch out if you're going to put the results back into another call to `offarray`— see `which` and EXAMPLES. To set the offset manually, you can do eg `firstel(x,3)<-5` or `firstel(x)<-c(3,6)` (set both offsets of a 2D `offarray`) or `firstel(x,4)<-NA` (to set base-R-style subscripting for the 4th dimension). To set a non-NA offset for some dimension, there must be no `dimnames` for that dimension (since character-indexed dimensions must always have an NA offset), so eg `dimnames(x)[[3]]<-NULL;firstel(x,3)<-10` would be the safe if weird way.

`slindoff` is the analog of `slice.index` for offarrays, returning values in the "coordinate system" of the `offarray`. I deliberately haven't tried to make `slice.index` generic since— as far as reasonably possible— "standard" operations on offarrays, i.e. ops that don't need to know about offarray-ness, should just work.

whichoff is like which(x, arr.ind=TRUE, useNames=FALSE) but returns a matrix with x-like dimensions (ie the offsets are incorporated). Thus you can use the result to matrix-subset x, ie via x[MATSUB=<result of whichoff>]). Note that "for technical reasons" you usually need to supply the logical operation as a second argument, rather than doing it "inline" as you would with which; see **Examples**.

Specific methods for offarray class are currently: subset (eg X[...]); subset-replacement (eg X[...] <- Y); as.array; print; t (transpose); aperm; str (which calls strguts.offarray); rev; head and tail (which preserve offarray-ness for 1D and 2D offarrays, but act otherwise like normal head and tail); dim<- & dimnames<- which avoid problems caused by base-R's treatment of dim; and as.data.frame. Also, pmax and pmin are modified to be offarray-aware.

See also autoloop for a convenient loop-free way to vectorize some computations on offarrays (and on standard arrays).

Features:

- row and col don't work properly even with 2D offarrays.
- Even 1D offarray objects, which are like vectors "really", still have a dim attribute.

Usage

```
# Don't read this--- look at EXAMPLES and ARGUMENTS
offarray( x, ...) # generic
## Default S3 method:
offarray( x= NA, first = rep(1L, length(dim)),
  last = first + dim - 1L, dim = NULL, dimnames = NULL,
  dimseq, ...) # S3 method for default
## S3 method for class 'table'
offarray( x, template=NULL, dimseq=NULL,
    warn_trim=getOption( "offarray_table_warn_trim", FALSE),
    ...) # S3 method for table
## S3 method for class 'data.frame'
offarray( x, data.col, ...) # S3 method for data.frame
## S3 method for class 'offarray'
as.data.frame( x, row.names=NULL, optional= NULL, add_names = NULL,
    name_of_response = "response", ...) # S3 method for offarray
## S3 method for class 'offarray'
as.array( x, ..., make_dimnames=TRUE) # S3 method for offarray
firstel( x, which=NULL)
firstel( x, i) <- value
lastel( x, which=NULL)
dimrange( x, which=NULL)
dimseq( x, which=NULL, drop=TRUE)
slindoff( offa, MARGIN)
whichoff( x, expr=.)
strguts.offarray( x)
```

Arguments

x	(offarray.default) elements of the array, as per matrix or array; or a table or data.frame, in which case the converter method will be called. In all other

functions documented here, x should be an existing offarray.

first, last, dim — set any 2 out of these 3— obvious meaning, as per array. If any of these has names, then these names will also be applied to dimnames, and will print on output. For character-indexed dimensions (ie with non-null dimnames), first must be 1 and last must be the number of elements.

dimnames — Set this only if you want one or more indices to be primarily "character-addressed" (eg sex being "F" or "M"). It's OK to have some null ones, eg dimnames=list(Year=NULL, Sex=c("F", "M")).

dimseq — You can use this to set up an offarray directly— it's often the easiest way. It should always be a list.See dimseq details for more info; you could also get it from calling dimseq on another offarray.

data.col — (offarray.data.frame) which column should become the contents of the offarray

template — (offarray.table) template can be an existing offarray, in which the result will have the same dimensions. This gets round the issue that table won't return index-values that don't occur in that particular dataset, but which might in other similar datasets. Alternatively, dimseq lets you specify the output dimensions directly, just as you would for offarray itself; I think you can use dimseq=dimseq(mytemp) instead of template=mytemp.

warn_trim — (offarray.table) if template or dimseq is supplied and x has some dimensions which don't fully fit into the desired output, x will be trimmed to fit; this parameter controls whether a warning is issued (and you can use options(warn=2) to force a crash in that case). In some applications, auto-trimming is good, but in others it's bad; you can control the general behaviour via options(offarray_table_warn_trim=TRUE/F rather than at every call to offarray<.table>.

add_names — (as.data.frame.offarray) how to set the column names of the results. Should be logical or character, or the default of NULL. NULL uses information in dimnames, including names(dimnames) if those exist; FALSE means the elements will be called 1, 2, 3 etc (as per slice.index), and the columns "D1", "D2", etc, which is also where things end up if NULL doesn't give anything useful. Set to your own character vector if dimnames doesn't help and you want to clearly label the columns.

name_of_response — (as.data.frame.offarray) what to call the output column that holds the array *contents* (as opposed to its dimensions).

row.names — (as.data.frame.offarray) as per data.frame()

optional, ... — ignored; present only for compatibility with generic S3 definitions.

make_dimnames — (as.array.offarray) whether to create extra dimnames for indices that used to have offsets.

which, i — if only certain dimensions are of interest to firstel/lastel/dimrange/dimseq. Default is to return results for all dimensions. firstel(x, "Sex") strips the names attribute of the result, and is preferable to firstel(x)["Sex"] if the assignee might be used again in offarray— see **Examples**.

value — (in firstel<-) first index value for that dimension; will be coerced to integer, or NA to use base-R indexing for that dimension (subtly different to value=1).

drop	only in dimseq and if length(which)==1, should a vector rather than list be returned?
offa	(slindoff) an offarray
MARGIN	(slindoff) 1 for row, 2 for column, etc. Or, a string which matches into names(dimnames(offa)). Scalar only.
expr	(whichoff) an R "test" expression containing a ., which will be evaluated for you after substituting the . with the first argument x. It should return a logical result. If you just passed in the evaluated test like you would with which, you'd lose the offsets; see **Examples**.

Details

The first element of each index is stored in the offset attribute. This will be set to NA for indices with explicit dimnames, which is useful. It can also be NA for a dimension without dimnames where base-R indexing applies; you can do that via firstel(x,3)<-NA, or at creation by offarray(...,dimseq=list(character(5)),...). Sometimes the latter helps, but my advice is to avoid mixing offarray-style and base-R-style indices within the same object.

Value

offarray returns, yes, an offarray; and as.data.frame returns a data.frame... firstel and lastel return vectors, named if the dimnames are too. dimrange returns a two-column matrices, with rows named according to dimnames. dimseq returns a list containing the indices each dimension, as character or integer; see also which and drop.'strguts.offarray' returns a string showing the dimension names and ranges— for use eg in a print method for something with an offarray component. slindoff returns an offarray of the same shape; the attributes preserved are dim, dimnames, offset, and class.

Examples

```
## Not fully checked!!
# Manual construction
test <- offarray( 1:6, first=c( X=3, Y=4), last=c( 5, 5))
test
#        Y
#  X      [,4] [,5]
#    [3,]    1 4
#    [4,]    2 5
#    [5,]    3 6
#
# Easier:
test2 <- offarray( 1:6, dimseq=list( X=3:5, Y=4:5))
# Easiest!
test3 <- offarray( 6:1, dimseq=dimseq( test))
## From a table:
set.seed( 1)
df <- data.frame( X=sample(  LETTERS[1:3], 5, TRUE), Y=sample( 8:10, 5, TRUE))
tab <- with( df, table( X, Y))
offo <- offarray( tab)
# reordered version of 'df', unless 'df' has duplicated rows
```

```
df2 <- as.data.frame( offo, name_of_response='count')
# and back from data.frame to offarray
offo2 <- offarray( df2, 'count')
identical( offo, offo2)
## With explicit dimnames (offset disregarded for those indices)
# Next will fail because dimnamed dims must start at 1
try( test <- offarray( 1:6, first=c( X=3, Y=4), last=c( 5, 5), dimnames=list( NULL, c( 'A', 'b'))))
# This is OK:
test <- offarray( 1:6, first=c( X=3, Y=1), last=c( 5, 2), dimnames=list( NULL, c( 'A', 'b')))
# Or:
test <- offarray( 1:6, dimseq=list( X=3:5, Y=c( 'A', 'b')))
t( test)
#     X
#  Y   [,3] [,4] [,5]
#    A    1    2    3
#    b    4    5    6
#
test[3,1]
#        Y
#  X      A
#    [3,] 1
#
# NB offset changes automatically if subsetting is consecutive:
test[4,]
#        Y
#  X      A b
#    [4,] 2 5
test[4,][4,] ## SAME!
#        Y
#  X      A b
#    [4,] 2 5
test[ SLICE=4,] # first dim dropped, as requested
#  Y
#  A b
#  2 5
#
test[3,1, nooff=TRUE] # loses offarrayness; NB row labels in printout
#        Y
#  X      A
#    [1,] 1
test[c(3,3), 1, nooff=TRUE]
#       Y
#  X      A
#    [1,] 1
#    [2,] 1
test[ NOOFF=c( 3,3),1 ] # same
try( test[c(3,3), 1]) # omitted NOOFF --- warning
# 1 vs NA offset
test1 <- offarray( 1:4, first=1, dim=4)
test1[2][2]
#   [2]
#     2
firstel( test1, 1) <- NA
```

```
# test1 looks exactly the same, but...
try( test1[2][2]) # OOB
test[ MATSUB=rbind( c( 3, 2), c( 3, 1), c( 5, 2))] # 4 1 6
test[ MATSUB=list( c( 3,5,5), c( 'b', 'A', 'b') )] # 4 3 6
test[ 3, 'A'] <- 999
test[ VECSUB=2] <- 17
## Disregarding that "4"--- NB using try() just to avoid RCMD CHECK nannyism :/
try( test[3,4])
# Error in structure(1:6, .Dim = structure(c(3L, 2L), .Names = c("X", "Y" :  subscript out of bounds
## Without try(), you'd see this:
# Error in 1:6[1, 4, drop = FALSE] : subscript out of bounds
## Y-offset is really 1
test[3,1]
#      Y
# X      b
#   [3,] 4
## still an array
## ... but Y is probably meant to be referred to "by character"
test[3,'b']
#      Y
# X      b
#   [3,] 4
## Dropping single-element dimensions:
test[ 3, SLICE='b']
#  X
#  [3,]
#     4
## ie from a 2D to a 1D offarray
test[ SLICE=3, SLICE='b']
#  [1] 4
## ie full drop
firstel( test)
#  X Y
#  3 1
firstel( test, 'X')
#  [1] 3
## no name
dimrange( test)
#    [,1] [,2]
#  X    3    5
#  Y    1    2
dimseq( test, 1)
# [1] 3 4, 5
## Collapsors: see also autoloop()
# sumover() always returns an offarray...
sumover( test, 'Y')
sumover( test, 2)
sumover( test, 2)[5] # fine
# ... but apply() doesn't quite work
apply( test, 1, sum) # NB index definition is oppposite for apply!
# ... not actually an offarray
try( apply( test, 1, sum)[5])
# ... to apply() something other than sum() (for which, use sumover)
```

```
# ... and to keep offarrayness, do this:
ds <- dimseq( test)
apptest <- apply( test, 1, sum)
apptest <- offarray( apptest, dimseq=ds[ 1])
## Converters
flubbo <- table( X=c( 1, 1, 3), Y=c( 'A', 'B', 'A'))
offarray( flubbo)
## That's OK: it figures out that X=2 is also wanted
#        Y
#  X      A B
#    [1,] 1 1
#    [2,] 0 0
#    [3,] 1 0
## But if no observations at the end of a range...
flubbo2 <- table( X=c( 1, 1), Y=c( 'A', 'B'))
offarray( flubbo2)
## ... X=2 and X=3 are missing (of course)
## So let's make sure it includes them:
offarray( flubbo2, dimseq=list( X=1:3, Y=c( 'A', 'B')) )
#        Y
#  X      A B
#    [1,] 1 1
#    [2,] 0 0
#    [3,] 0 0
## And test the trimming facility
offarray( flubbo2, dimseq=list( X=2:3, # X=1 will be discarded
    Y=c( 'A', 'B')), warn_trim=TRUE)
# ... warning
slindoff( test, 'X')
#        Y
#  X      A b
#    [3,] 3 3
#    [4,] 4 4
#    [5,] 5 5
slindoff( test, 2)
## NB no offset operational since dimnames()[[2]] is non-empty
#        Y
#  X      A b
#    [3,] 1 2
#    [4,] 1 2
#    [5,] 1 2
whichoff( test>4) # unlike which( test>4, arr.ind=T)
## NB obvious approach *doesn't* work:
whichoff( test>4) # loses offsets, cos 'test>4' just returns arrayi
```

`partial_expand_dot_grid` *Like expand.grid but a few rows at a time*

Description

expand.grid can produce a overgigantic object, so you can instead call partial_expand_dot_grid repeatedly in a loop to generate the "next" lot of rows.

Usage

```
partial_expand_dot_grid( dots, n = prod(lengths(dots)), already = 0, force_data.frame = FALSE, switch_v
```

Arguments

dots	list of expandees. Unlike expand.grid, you have to put them explicitly into a list, rather than using ... facility.
n	how many rows to produce this time. Defaults to all remaining rows. If you overflow the "natural" size of the grid, it will cycle back to the start, and will always give you n rows.
already	how many rows in previous runs (default 0, for the first time).
force_data.frame	whether to convert the result to a data.frame, or leave it as a list. The latter might be fine if you just want an environment to evaluate expressions in, or to use for MATSUB= subsetting in offarray. Note that conversion will enforce stringsAsFactors=FALSE because I hate the alternative.
switch_var	if non-0, then an attribute will be added showing the rows where the switch_var-th expandee changed value.

Examples

```
expand.grid( A=1:3, B=4:5)
partial_expand_dot_grid( list( A=1:3, B=4:5), force=TRUE) # the same, thanks to defaults
partial_expand_dot_grid( list( A=1:3, B=4:5), n=6, already=0, force=TRUE) # explicit defaults
partial_expand_dot_grid( list( A=1:3, B=4:5), n=3, already=2)
```

---

pmax *Parallel min and max*

Description

Just like base-R pmax and pmin , except that my ones work properly with offarray :/ so that pmax(offar,0) will return an offarray of the same dimension as offar. Note that pmax(0,offar) will return a vector, as per base::pmax (attributes of the return-value are based on the first argument).

Details: This is a pretty ugly fix, to necessitate a special-case like this. A theoretically-better option might be to make pmax/pmin into S3 generics in offarray, but since they are already in some sense S4-aware, I didn't want to accidentally mess that up; also, functions with dotdot-dot args are awkward with S3 anyway, viz cbind. It's possible that this could all be handled "gracefully" by making offarray S4 and doing some wizardry, but I hate S4 and don't want to

get embroiled in it just to fix bugs in other packages that might still manifest anyway. A simple option is `base::pmax( unclass( offar), 0)` but of course that does not return an `offarray`, which I think `pmax` should.

Usage

```
pmax(..., na.rm = FALSE)
pmin(..., na.rm = FALSE)
```

Arguments

`..., na.rm`	as per `base::pmax` (qv), a logical indicating whether missing values should be removed.

Value

If the *first* argument is an `offarray`, and the return-value from uncompiled base `pmax/pmin` is of the same length (ie the first element wasn't auto-replicated), then the result is an `offarray` with the same attributes. If not, the result is whatever base-R gives (without compilation).

Examples

```
dumbo <- offarray( 1:4, dim=c( 2, 2), first=c( 3, 5))
try( base::pmax( 0, dumbo))
#Error in (function (sub, this_offset, ii, e)  :
#  in offarray, logical subsets must match length of object's dimension
try( base::pmax( dumbo, 0))
#Error in `[<-.offarray`(`*tmp*`, change, value = numeric(0)) :
#  Wrong number of indices
# 1D case
dumbo <- offarray( 1:2, dim=2, first=3)
try( base::pmax( 0, dumbo))
# [1] 1 2
# ... sorta works (!), but loses offarrayness
pmax(dumbo, 0)
# [3] [4]
#   1   2
# yahooo!
```

---

reclasso *Prepare offarray code for RTMB etc*

Description

Suppose you have `offarray`-using code that computes a log-likelihood or suchlike, based on parameters. And you've debugged it, well done! Now you would like to also run under package **RMTB**, say (or some other hypothetical package where the "parameters" might not just be "numbers"). To make things work nicely, you just wrap the body of your code in a call to `reclasso`, like this:

```
myfun <- function( parzzz, <...>) reclasso(by=parzzz, {
  <old code of myfun>
})
```

This will have *no effect* if you just run `myfun` in R normally, so you don't need to keep modifying your code when you do/don't want to use it with `RTMB` etc.

But if `parzzz` has a specific class— eg if called from `RTMB::MakeADFun`, then `parzzz` will be class "advector"— then a special form of `reclasso` gets called, to pre-tweak your code so it works nicely with that class. (For `RTMB` package, it has to modify addition, multiplication, etc, plus `REPORTO`, and to tweak environments.) This behaviour depends on having loaded an appropriate "helper" library, which for `RTMB` is the **offartmb** package.

If you are annoyed by having to wrap your code like that, you can abbreviate it to this:

```
myfun <- function( parzzz, <...>) {<old code>} |> reclasso( by=parzzz)
```

Details: The main reason for `reclasso` is that R's multiple-dispatch rules for addition etc simply don't work right, unless you go to great lengths with S4. Which sounds like a lot of work for *me*, and I'm not sure it's even guaranteed to work fully without hacking, since package **RTMB** is of necessity quite hacky itself. Instead, I have implemented my own versions of "+" etc which *do* dispatch correctly, taking account of and preserving "offarray" and/or "advector" etc classes. So the trick is to replace calls to base-R "+" etc, with calls to my more culturally-attuned versions.

[Note, primarily to my future self] There are tedious internal nuances with dispatching when the parameters are a `list` (which `RTMB` allows, basically unlisting them for you) and which method gets picked up. The bottom line is that it works, but e.g. quite why I had to define a `reclasso.numeric` instead of `reclasso.default` is a mystery to me. But ultimately I don't really care, and you shouldn't either.

Oh, and the reasons that I don't just provide a replacement for `MakeADFun` that auto-does this, are:

- because this notion might eventually be used not with the **RTMB** package but with something else, so I didn't want to tie it in (no idea what, this is just a note)
- your function might call other functions you wrote, all of which need their own `reclasso`.

So it's better for it to be *your* responsibility to add `reclasso` calls to *your* functions as needed. Calling `reclasso` costs almost nothing run-time-wise, BTW.

Usage

```
## Never use it from the command line!
## Only use it as a wrapper for function code,
## EG if your original function looks like this, and
## 'p' might be modified by some callers:
#   myorigfun <- function( p, otherpars) {<body>}
## then either of
#  myfun <- function( p, otherpars) reclasso( by=p, {<body of myfun})
#  myfun <- function( p, otherpars) {<body>} |> reclasso(by=p)
reclasso( expr, by, ...)
```

Arguments

expr	Code of your function
by	What variable to look at, to decide whether expr needs to be pre-modified before evaluation. Expected use-case is for by to be a vector of parameters.
...	Possibly required by some methods

Examples

```
## Not run:
reformat.my.hard.drive()

## End(Not run)
```

---

subset.offarray *Subsetting offarrays*

Description

These are subset (extraction) and replacement functions for the class offarray. You would normally call x[...] directly, rather than subset(x,...), but the latter is a convenient place for documentation!

Although I have tried to write them as efficiently as possible, there is unavoidable overhead compared to base-R let alone raw C. In particular, there is AFAIK *no* safe way to avoid a deep-copy during any subset-replacement, except for "classless" objects. So, every time you try to change one single element of an offarray with 1e8 elements, you will trigger a "deep copy" of the whole thing. **Therefore** you should avoid as far as possibly writing for-loops to do replacements. **Instead**, use autoloop; it's *fast* and memory-efficient.

Special features of subset extraction: There are a few tricks on subset-extraction. Normally you use anonymous indices, eg x[i,,3:7], but you can optionally *name* each index with one of the values "SLICE" or "NOOFF", eg:

```
x[ 7, 5, c(8,7,6)] # 3D offarray, with warning
x[ SLICE=7, SLICE=5, NOOFF=c(8,6)] # 1D offarray, no warning
x[ 7, 5, c( 8, 6), drop=T, nooff=T] # same
x[ 7, 5, c( 8, 6), drop=c(T,T,F), nooff=c(F,F,T)] # more of same
```

Naming particular indices is usually more convenient than using the vector-forms of the arguments drop and nooff.

Usage

```
## S3 method for class 'offarray'
subset(x, ..., VECSUB, MATSUB, drop=FALSE, nooff=FALSE) # S3 method for offarray
## S3 method for class 'offarray'
x[..., VECSUB, MATSUB, drop = FALSE, nooff = FALSE] # S3 method for offarray
## S3 replacement method for class 'offarray'
x[..., VECSUB, MATSUB] <- value  # S3 method for offarray
```

Arguments

x	thing to subset or subset-assign
...	subset indices. For extraction only, each of these can have special names.
VECSUB	standard R "vector subset" (even for multidimensional offarray), either logical with the same length as x, or numeric. Offsets are ignored, so the first element of x is always 1 here. Negative indices are allowed, but not 0, NA, or beyond the length of the object.
MATSUB	standard R "matrix subset of array", respecting offsets, except that MATSUB does not have to be a numeric matrix, but can also be a data.frame or list with character elements for dimensions with non-null dimnames.
drop	whether to drop dimension(s) that end up with length-1 after extraction. Either a logical vector of length dim(x), or a logical scalar which will be replicated to that length. drop is overridden by naming indices. drop elements are over-ridden to TRUE by any indices that are named "SLICE".
nooff	whether to abandon the offset for dimension(s). nooff should either be a logical vector of length dim(x), or a logical scalar which will be replicated to that length. You would only want to set nooff if you are deliberately extracting a non-consecutive subset from soem dimension(s). R will do the extraction regardless, but will warn if a non-consecutive subset is used without the corresponding element of nooff being set to TRUE. nooff elements are over-ridden to TRUE by any indices that are named NOOFF.
value	the value...

Details

Both subset and subset-replacement methods use similar trickery, to avoid unnecessary deep-copies. The typical operation of an S3 method is to call NextMethod after modifying the arguments, but this can't be done directly for any ... arguments. And re-calling the generic leads to deep-copies. So I modify the arguments, set a special flag, then use Recall; the code checks for the special flag first, and if so calls NextMethod. That seems to work and to be byte-compiler-robust, fortunately.

Previous versions of offarray did not work properly when the byte-compiler was in use, because of truly obscure features in the latter (p73 of the manual, and the manual is not exactly easy to find...).

Value

Subset-extraction will return an offarray unless the result is length-1 and drop or nooff has been set/over-ridden, in which case a scalar will result. Dimensions of length-1 are dropped iff SLICE or drop is set for those dimensions. When non-consecutive elements are extracted from a dimension, the offset of the result will be set to NA in that dimension, so that *that dimension* behaves just like base-R in further subsetting.

See Also

offarray, autoloop

Examples

```
## See ?offarray
```

Index