Package: vecless (via r-universe)

June 5, 2026

Version 1.1.1

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

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

Depends R (>= 2.13), tensorA

Imports atease, mvbutils

Date 18/3/2015

Title Easy array/matrix ops

Description Algebraic-style array/matrix ops without vectorization, using named indices; a wrapper for the 'tensorA' package

License GPL (>=2)

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

Date/Publication 2025-04-06 00:29:01 UTC

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

RemoteRef HEAD

RemoteSha 8b0c3298f913d1c7d5234186b91b8d5046786d84

Contents

:= *Algebraic treatment of array/matrix ops*

Description

If you love R's vectorization syntax and find it really intuitive, stop reading now. If not, you can use this function to directly code "complicated matrixy multiplication-like things", where the indices are specified by name just as you would in an algebraic equation: the kind of things I want to do all the time in statistical computing. It's easier to use than to describe; one description might be, that it automatically loops/vectorizes "dotty products" over chosen indices, and does "parallel ops" on the remainder. Even the crude approach in version 1.0 of `vecless` is quite quick, *except* that initial parsing can be pretty time-consuming; see `?compile_vecless` for a speed-up.

The basic idea is that you write assignments to (parts of) arrays in the form `LHS := RHS`, rather than using `<-` or `=`. The LHS and RHS refer to arrays via eg `A[i,j,k]` where the `i`, `j`, `k` are just placeholders that have meaning only inside the particular expression being used, to link up the correct elements in different parts of the expression— just as in written algebra. The code automatically loops over all relevant indices, so `A[i] := B[i]+1` will assign to *all* elements of `A`. Just as in algebra, the order of indices matters, but not their names— within any given `LHS := RHS`, you could replace every incidence of `i` by `mooncalf` and the result would be the same.

Basic parallel ops are easy. Any binary arithmetic or logical operator can be used, as well as the unary operators `+-!`:

```
A[i,j] := B[i,j] * C[i] + D[j]
A[i,j] := B[i] * C[j] # outer product
```

So are "generalized" dot-products. If you want a dotty product requiring summation over an index `k` in both tensors on the RHS, then use a fake function `%[k]%` to indicate summation over those indices, like this:

```
A[i,j] := B[i,k] %[k]% C[k,j]
```

which in that particular case is equivalent to matrix multiplication, `A <- B %*% C`, or:

$$A_{ij} := sum_k B_{ik} * C_{kj}$$

But the notation is much more flexible and general than a standard dot-product. You can dot over several indices at once by writing e.g. `%[j,k]%`.

Very often in stats, you want "parallel" operations over some index, here `i` and `l`:

```
A[i,l] := B[i,j,l] %[j]% C[j,l]
```

Auto-promotion works:

```
A[i,l] := B[i,j,l] %[j]% C[j] # C "needs" to be replicated along an extra dim 'l'
```

So do scalar-type ops:

```
A[i,l] := B[i,j] %[j]% (2*C[j,l]+1)
```

Here's a transpose:

```
A[i,j] := B[j,i]
```

This is numerically the same as

```
A[j,i] := B[i,j]
```

but A's indices are named differently; the name-sequence always follows the LHS subscripts. In general, the names of indices are ephemeral; they matter within the lifetime of a single expression, but not afterwards (except for display).

Nesting works, though I haven't checked how thoroughly...

```
A[i] := (B[i,j] %[j]% C[j,k]) %[k]% D[k]
```

"Primitive" functions work, component-wise:

```
A[i,j] := log( B[i,j] + 3)
```

Summation works, via the following creative abuse of notation:

```
u[j] := SUM_ %[i]% v[i,j]
```

Using `SUM_` is very clear, but autosummation can also be done quickly like this:

```
A[i] := B[ i, +j]
A[i] := B[ i, +.]
```

the latter form being convenient if you don't want to bother giving B's 2nd index a "name" when you're only summing over it anyway; `.` is a "name" from R's PoV.

To "drop" a dimension (ie slice at particular index for that dimension), you can use numeric (and possibly character) literals, or wrap a variable/expression in `{}`:

```
D[i]:= A[2,i]
D[i]:= A["cat",i]
D[i]:= A[{var},i]
```

where the braces denote that `var` is not an index. Obviously, `var` has to have length one; otherwise you can use `A[i,j]:=D[i,j=var]` but of course then the dimension won't be dropped. `var` can be numeric, or character if `A` has dimnames.

Fairly general subsetting works, at least up to a point... but may not have been fully tested. You can use it on the RHS (more tested), and even on the LHS (less tested) if you are assigning into an existing variable. Except for (combinations of) pure slices and subranges, these forms are computationally more expensive— but bloody useful.

```
D[i]:= A[i=2:3] # use a name if you're specifying a subrange
var <- 2:3
D[i]:= A[i=var] # NB 'var' is NOT in braces...
# ... because it's not being dropped and its new "dimension" will have a name.
D := A[ 2] # you don't need 'vecless' for this! Note...
# ... lack of LHS brackets given scalar result; see also *Limitations* below.
C[j]:= A[ j=1:2, 4-j]
D[i]:= A[i,i] # ie diag(A)
u[j]:= mm[ j, ll[ j]] # Lookup table!
A[i,i] := mm[ i, k] %[k]% nn[ k] # diagonal subassignment--- seems to work. A must pre-exist
yy <- matrix( 1:9, 3, 3); yy[j,j+1] := symm[ j=2:3, {jj}] # weird stuff--- also seems to work
yy <- 1:5; yy[ 4-j] := A[ j=2:3] # error, at least for now; too weird
```

Note that you do have to assign the result to something, via `:=`. Without the `:=`, the RHS is meaningless; you can't just make up fake functions willy-sodding-nilly and expect them to work, because they don't exist, do they?

Synonyms: `%:=%` is a near-synonym, *except* its RHS *must* be wrapped in parentheses (because the precedence order is not what you'd like). **NYI** : `%::=%` is the globalized version **NYI** ; `%::=%` is to `%:=` as `<<-` is to `<-`.

Limitations: Aside from bugs, that is...

The requirements for subsetting on the LHS are currently not very sensible. Part of the issue is whether the LHS variable already exists and therefore has known size.

- Any "free" indices on the RHS (ie not summed/dotted-over etc) must appear in the LHS.
- If there are no free indices on the RHS (so that RHS will return a scalar), the LHS must not be subscripted. This is illogical; should be able to assign into a fixed position in a vector/matrix
- For some reason, you *can* assign a lower-D RHS result into a fixed slice of a higher-D existing LHS (using `{}` or numeric or possibly even character literals), except if the RHS is scalar.

Things to be added in version 2:

- proper comprehensive matching of indices on LHS & RHS
- non-1 offsets
- "masked" subscripts, eg `A[i,j=j<i]`. Plans are ready, though I can't remember the exact syntax...
- adding `PROD_` as well as `SUM_`, and doing that more efficiently
- proper (??) handling of character indices
- vast speed-ups using native C code (done) and smarter parsing (not)
- `%xor%` is missing from list of binary operators

Usage

```
# eg A[i,j] := B[i,k] %[k]% C[k,j]
# A[i,j] %::=% (B[i] * C[j]) # NB parentheses with %%-forms
# Now, I have to put the next lines in to avoid offending the CRANiacs, but
# ... it's not how you're meant to use ':=', so...
# ... don't !
`:=`(LHS, RHS)
LHS %:=% RHS
LHS %::=% RHS
```

Arguments

LHS	What to assign the result to, with (or rarely without) notional subscripts/indices:eg A or A[i,j]. Subscripts must correspond to the index-names of the RHS result, but can be in a different order, and some tricks are allowed for subassigment if A already exists. The subscripts only have meaning within the lifetime of the call to :=.
RHS	algebraic-type expression, using fake functions such as %[k]% for dot-product summation over an index k. With the %...% versions, **be sure** to wrap the RHS in parentheses, otherwise R's precedence rules will screw you over royally—and it will all be your fault. And yes, I *have* made that mistake myself.

Details

By some happy quirk, R's parser understands ":=". My version of := examines its arguments and uses them to construct a call to mul.tensor etc in package **tensorA**. That package is extremely powerful, but I still find it hard to follow or use efficiently/elegantly; the documentation assumes a working knowledge of general relativity, but I just count fish.

I should extend this to the following (some of which might work already). These will all be in the successor package **vecless2** or indicial.

If you're writing a package, the use of fake functions in your code will no doubt generate notes from R CMD CHECK about "binding not found". They are harmless. You can either (i) point this out to the CRANia ("go on— make my day"), (ii) make manual globalVariable declarations in your NAMESPACE file, or (iii) use mvbutils to maintain the package, and include the suppress.nannys.missingvar.checks mechanism (NYI..?).

Value

A tensor (or scalar, if that's how the cookie crumbles) is formed, and stored in the top-level variable on the LHS (placed in the calling environment, as you'd expect). These largely behave just like normal matrices and arrays, but you can force that via unclass. 1D tensors may need to be coerced via c() or unclass() to behave as you'd expect afterwards. The stuff on the RHS does **not** have to be tensors.

Examples

```
# These may be WRONG! and are not comprehensive. See *Description* for up-to-date
B <- matrix( 1:6, 3, 2)
C <- matrix( 5:12, 2, 4)
A[i,j] := B[i,k] %[k]% C[k,j]
A[j,i] := B[i,k] %[k]% C[k,j] # A will be "transposed"
V <- 1:2
A[i,j] := V[i] * V[j] + 3 # outer product, and nested ops; A's dims are now [i,j]
# Slicing/dropping
A[i] := C[i,2]
jj <- 2
A[i]:= C[i,{jj}] # avoid treating j as index
Ascalar := V[{jj}] # no subscripts needed/allowed on LHS here
A[i,j] := B[j,i] + 9
A[i,{jj}] := B[{jj},i]
```

```
A[i,jj] := B[ jj, i] # different!
A[ j]:= C[j,j+1]
# Some errors: partly user stupids, partly limitations of current vecless
try( A[j]:= C[j,{j}]) # wrong answer really; your fault
```

`compile_vecless` *Compile function that uses vecless*

Description

Using `vecless` operator `:=` can be a bit slow in loops, because the parsing takes a while. Compiling means the parsing need only be done once, which can speed things up quite a bit (eg 5X).

In version 1.0 of `vecless`, this is done rather sloppily— the compiled version will only work if exactly the same sequence of `:=` calls is made every time. It will fail if, for example, `while`-loops execute a different number of times, or `if`-statements branch differently. Also, the operations themselves are still fairly slow compared to base R. Version 2 will fix everything— but version 1 is still pretty good.

Usage

```
compile_vecless( acall, tidy=FALSE) # pointless without assigning the result...
# Always do eg cfunc <- compile_vecless( ofunc( 5), tidy=...)
```

Arguments

`acall`	a bona fide call to your function, which must be runnable— since it will actually be run (the result is discarded). You have to define the function first, then compile it; you can't do it all-in-one by putting the definition into the call to `compile_vecless` (because I am too lazy to write the necessary fiddly code).
`tidy`	if TRUE, the compiled code will look almost identical to the original, which helps debugging— though you should debug **before** you compile. The downside is that the closure-environment of the compiled function will be changed; it will be a child of the original closure-environment, containing a redefinition of `:=`. Normally that won't matter at all— it would only be an issue if your function is weird enough to want to *change* things in its closure-environment (which is different to the temporary environment where the statements of the function actually happen); most functions just access their closure-environment (and its parents) implicitly, to look for symbols. However, if you are doing something really weird and don't want to touch the closure-environment, set `tidy=FALSE` (the default) for some ugly but safer code.

Details

Lots of environment magic is used. During compilation, the `:=` operator is replaced so that it records, in sequence, the post-parsing pre-evaluation version of each call to `:=`, in a special environment which also contains a counter. In "replay", the counter is first set to zero; `:=` is overloaded/replaced (depending on `tidy`) so that it updates the counter and then evaluates the next recorded expression.

A better method (insofar as not requiring exactly the same execution-path on each invocation of the function, and thus robust to loops and ifs) would be to replace or augment each := call with a version that has all the parsing-info for that call attached. It's entirely possible, but requires "manual traversal of the parse-tree" in your function during compilation, which is a lot of work. For version 2.

Value

A modified version of your function.

See Also

:=

Examples

```
vctest <- function( tt) {
  x2 <- 1:2
  for( i in 1:1000) {
    K[j,l] := tt[ j] + x2[ l]
  }
return( K)
}
vctest2 <- compile_vecless( vctest( 2))
system.time( print( vctest( 7:9)))
system.time( print( vctest2( 7:9)))
vctest2 # ugly
vctest3 <- compile_vecless( vctest( 2), tidy=TRUE)
vctest3 # slightly prettier
# Native R equiv, without using 'outer'; actually faster than 'outer'!
vctest4 <- function( tt) {
  x2 <- 1:2
  K <- matrix( 0, length( tt), length( x2))
  for( i in 1:1000) {
    K[] <- tt[ row( K)] + x2[ col( K)]
  }
return( K)
}
system.time( print( vctest4( 7:9)))
# R equiv, using 'outer' (which does not generalize, and is incomprehensible to mortal humans)
vctest5 <- function( tt) {
  x2 <- 1:2
  K <- matrix( 0, length( tt), length( x2))
  for( i in 1:1000) {
    K <- outer( tt, x2, '+')
  }
return( K)
}
system.time( print( vctest5( 7:9)))
```

recordar *Quasi-automatic vectorization*

Description

Suppose you have a cooked up a complicated numerical function `fhard` that works for a "scalar" case (say, one locus at a time), using lots of intricate subset-via-lookups. And say it's vectorizable in principle, but the task is beyond you. (The real crunch comes if you want to use matrix-subsetting of matrix, or vector-subsetting of matrix; you cannot "just" vectorize those cases by adding dimensions at the start.) Then, after a few slight tweaks, you can apply the `recordar` tools to generate a Quasi-Auto-Vectorized version, that will run **quickly** ie using proper R vectorization.

The sequence of ops is:

- make some tweaks to `fhard`, specifically:

– add a `record` arg to `fhard`, default FALSE

– say which arrays/matrices/vectors/scalars in `fhard` need to have their ops recorded (ie could have extra dims at the start), using `set_recording`

– postfix key numerical and definitional statements of `fhard` with `? 0` or sometimes `? 1`

- run `template <- fhard( <example>, record=TRUE)`
- run `vfhard <- make_playback( fhard, template)`
- then you are good to go with `vfhard( <realvectorargs>)`

Any arguments of `fhard` that themselves need recording, must all be pre-vectorized in the same way during playback, I think... but it's OK to have other args that aren't.

One trick lies in the redefinition of `?` to record an operation, rather than summon up help! This is the only way to get the tweak to work, because of operator precedence rules; if you hate it, you can instead wrap your statements in `recordar( <blah>)` but it makes the code harder to read.

Usage

```
# Easy tweaking via eg x[ cbind( a1, a2)] <- y[ lu] ? 0
recordar(assig, expand_dim = FALSE) # It's easier to tweak with '?' but you can instead wrap your stateme
set_recording( vars, record=TRUE) # put eg 'set_recording( c( "ar1", "vec2"))' in the first line in your
make_playback( fhard, template, record_arg_name='record')
```

Arguments

`assig`	(recordar) Statement to be recorded
`expand_dim`	(recordar) ?In the vectorized version, should the prefixdims be explicitly prepended? Thanks to recording magic, this is is normally not needed (and wrong) if the RHS of the assignment includes recorded variables; expansion is needed only if the "scalar" version creates a fixed number of dimensions, eg `myrecar <- matrix( 1:6, 3, 2)`. Hence the usual tweak is to add `? 0` but occasionally you need `? 1` or `? TRUE` (whatever you find clearer).

vars	(set_recording) Names of objects (numeric arrays, matrices, and/or vectors) that need to be tracked
record	(set_recording) If FALSE, nothing is recorded, and your function executes as normal. So you would set this to be the record argument of fhard, if there is one, or leave blank if you always want to record.
fhard	(make_playback) Your fhard, post tweaks.
template	(make_playback) Result of one call to tweaked fhard in its recording mode
record_arg_name	(make_playback) Your original function probably should have an argument named *something* like record, which will be removed from the QAV version. If your argument isn't exactly named record, you can set record_arg_name instead. If you don't have any such argument (ie you always record), then set this to "".

Details

There are some things you can't do... so, don't.

Value

set_recording	returns a function which is a tuned version of recordar, ready for your specific tasks. See **Examples**.
recordar	result of the expression (though you are not meant to use this directly, AFAICS).
make_playback	a function, the QAV version of fhard.

Examples

```
# This one REALLY DOES need an example!
```

---

xtensor *Private tools for vecless*

Description

All these routines are exported for technical reasons only in vecless v1— you should not call them yourself. Basically, vecless reorganizes all := calls into a combination of calls to these tools, then executes the call. I think they need to be exported to be visible.

xtensor adapts the tensorA::tensor class to make it workable for vecless v1.0. The others I'm not even going to document. xtensor itself has various methods which I don't seem to be documenting.

There's now an as.data.frame.xtensor method that works as well as can be expected. If dim>2, the result won't work right (printing triggers an error, just like data.frame on a 3D array), but that's an existing R issue. The as.array.xtensor method exists to get round some problems with names of names and names of dimnames.

Usage

```
#xtensor(X, inds='I' %&% seq_along( dim( X), ...)
```

Arguments

X	(xtensor)
inds	(xtensor)
...	(xtensor)

Value

Immense. NB this section isn't compulsory.

Examples

```
# Not compulsory to have an EXAMPLES -- you can put examples into other sections.
```

Index