Fast loops with offarrays

July 12, 2024

The `offarray` package lets you work with arrays (and matrices and vectors) whose indexing can start wherever you like— not just at 1 as in base-R, nor just at 0 as in misbegotten languages like C. This is great for applications like population dynamics: you can start your years at 2015, you can start your range of ages at 8, etc. Simpler, clearer, less buggity-bug-bug.

Most R-ish things work as you would expect on `offarray` objects. There is documentation! Try `?offarray` (where exceptions are noted) and `utils::example('offarray','offarray')`.

```
library( mvbutils)

##
## Attaching package:  'mvbutils'
## The following object is masked from 'package:graphics':
##
##     clip
## The following objects are masked from 'package:utils':
##
##     ?, help
## The following objects are masked from 'package:base':
##
##     print.default, print.function, rbind, rbind.data.frame

library( offarray)

##
## Attaching package:  'offarray'
## The following objects are masked from 'package:base':
##
##     pmax, pmin

options( digits=4)
bloop <- offarray( 1:6, dimseq=list( 11:12, 21:23))
bloop + 3

##       [,21] [,22] [,23]
## [11,]     4     6     8
## [12,]     5     7     9

bloop > 0

##       [,21] [,22] [,23]
## [11,]  TRUE  TRUE  TRUE
## [12,]  TRUE  TRUE  TRUE

bloop[11,]

##       [,21] [,22] [,23]
## [11,]     1     3     5

bloop[SLICE=11,] # drops the dimension (drop is off by default)
```

```
## [21] [22] [23]
##    1    3    5

## utils::example( 'offarray', 'offarray')...
## ... or even try reading the documentation of '?offarray'...
```

You can also use `offarray` with Kasper Kristensen's stupendous `RTMB` package— so you can get first derivatives in high-dimensional problems really fast, and fit complicated statistical models with remarkable ease. (Of course, the whole point of `RTMB` is that you can use "regular" R code. But it took some work to get `offarray` working with it.) **This is amazing!!!**

0.1 Package `TMBO`

Note that you can also use `offarray` with my package `TMBO`, which deliberately lets you write flexible-offset TMB code (in C++, of course; that's good old "TMB", not "`RTMB`") that is fully `offarray`-compatible on the R side. In marked contrast to "native TMB", TMBO also gives very informative non-crashy error messages when you go Out-Of-Bounds (OOB) or commit other array-access stuff-ups, which are by far the commonest problem. However, I suspect that TMBO — which was a lot of work to write in late 2023, and which I'm kinda proud of— will be obsolete almost immediately, because `RTMB` and `offarray` will just remove the need ever to go near C++ again. And I will shed no tears whatsoever if that is the case.

1 Loops and speed

When you need to to fill in an offarray one-element-at-a-time, for-loops are going to be slow, especially nested ones. A staggering amount of argument-checking gets done behind the scenes in R *every time* a single offarray element is read; and every time one element is changed, R makes a "deep copy"[1] of the entire offarray, which gets very costly for big offarrays. Even if you only intend to run your R code once for `RTMB::MakeADFun`, normal for-loops can *still* be waaaaay too slow. However, you can *completely solve the problem* by using `autoloop()` to do (almost) all the loops for you, with *minimal* cost in comprehension. Cost in simple cases is about 2-3 times slower than base-R, ie negligible penalty. Here's an example:

```
N <- 20 # array has 8000 elements (smallish)
system.time({
  xR <- array( 0, rep( N, 3));
  for( i in 1:N) for( j in 1:N) for( k in 1:N)
    xR[ i,j,k] <- 1/(i+2*j+3*k)
})

##    user  system elapsed
##       0       0       0

system.time({
  xO <- offarray(0, dimseq=list( 1:N, 1:N, 1:N));
  for( i in 1:N) for( j in 1:N) for( k in 1:N)
    xO[ i,j,k] <- 1/(i+2*j+3*k)
})

##    user  system elapsed
##    0.20    0.05    0.39

system.time(
  xA <- autoloop( i=1:N, j=1:N, k=1:N,
    1/(i+2*j+3*k))
)
```

[1] This deep-copy-on-subassignment seems to be unavoidable feature of R for any non-base-class object, certainly in S3 and AFAIK also in S4. Base-R arrays *can* do in-place modification, but there's no safe general way in your own code for your own classes, not even at the C level. I doubt R-core has much interest in changing this, although the word "might" has been bandied about for years. Packages like `data.table` get round it only by defining new assignment operators, which I didn't want to do for humble `offarray` objects.

```
##    user  system elapsed
##    0.00    0.00    0.01

N <- 100 # try 400 for a real test; then 8e6 elements and 512 MB
system.time(
  xA <- autoloop( i=1:N, j=1:N, k=1:N, 1/(i+2*j+3*k))
)

##    user  system elapsed
##    0.12    0.01    0.16

## Don't even TRY the direct loop over offarrays!!!

system.time({
  xR <- array( 0, rep( N, 3));
  for( i in 1:N) for( j in 1:N) for( k in 1:N)
    xR[ i,j,k] <- 1/(i+2*j+3*k)
})

##    user  system elapsed
##    0.02    0.00    0.05
```

Of course, sometimes you can't entirely avoid for-loops. A classic example is time-stepping thru population-dynamics calculations over the years. That's fine; by all means have a for-loop over years, but just try to do all the work *within* each year via one-or-more `autoloop` calls.

`autoloop` will also do automatic "contractions" for you, i.e. accumulating a sum over an index which does not appear in the result. Matrix-multiplication is the most obvious example. Of course, you would not normally use `autoloop` just for that, but here's how you could. Notice also that `autoloop` can use base-R arrays as well as offarrays :

```
mm <- matrix( 1:6, 2, 3)
xx <- 4:6
mx <- autoloop( i=1:2, SUMOVER=list( j=1:3),
  mm[i,j] * xx[ j]
)
mx

## i
## [1] [2]
##  49  64

mm %*% xx

##      [,1]
## [1,]   49
## [2,]   64
```

Automatic contraction comes into its own with probability calculations in population dynamics, and especially in close-kin mark-recapture, when you might be dealing with a 7-dimensional set of indices that you want to sum down to "just" 5; good luck concocting a base-R expression for that!

You can have several statements within the body of an `autoloop`, wrapped inside {} and separated by semicolons. Some statements might compute ephemeral quantities that are needed only for computing other things, and that don't need to be kept afterwards. Other statements can produce named variables that you do keep at the end. The examples later on show all that.

1.1 How does `autoloop` work?

```
?autoloop
```

If that's too much effort to read: `autoloop` is like base-R `outer`, but generalized and betterized. Which should encourage you to read `?autoloop`...

2 Programming styles with `autoloop`

`autoloop` auto-vectorizes your expression, and then evaluates it over subsets of a few thousand index-combinations at a time. Therefore, what it runs is not *quite* what you wrote, but if you stick to simple rules then you will never see the difference— and if you don't, it probably won't run at all. Basically, every statement within your `autoloop` expression should result in a vector of the same length. Even though array-accesses don't *look* like they will do that, the magic of `autoloop` ensures that they generally do[2].

Mathematical operations within an `autoloop` expression generally work fine (though not matrix-multiply— but that is simple to code manually within the expression). The main "limitation" is that you can't use `if`-statements[3] The R function `ifelse()` can accomplish the same thing, but be aware that `ifelse(cond,ex1,ex2)` always evaluates *both* `ex1` and `ex2` whether `cond` is TRUE or FALSE. That can lead to OOB trouble if you naively try to access arrays, which is usually what I use `autoloop` for. I'd like to put in a general Bayes-theorem example here, but I'd have to think about it, and who's got time for that? So instead, here's a no-thought example from Close-Kin Mark-Recapture. If you are currently or imminently doing the CKMR workshop, maybe defer reading this until Day 2! If not, you might not understand the background at all— never mind :)

For a range of "adult" sampling years Y and ages-at-sampling A, and "juvenile" birth-years B, we consider a comparison between a potential parent "adult" and potential offspring "juvenile". The adult is lethally caught in year Ya at age Aa, and the juvenile in Yj at Aj. What is the demographic probability that the two will be a Parent-Offspring Pair, aka "POP"? In simple cases, that probability is "just" the adult's expected fecundity in the birth-year Bj (which will depend on the adult's age at Bj, plus whether the adult would actually be alive then), divided by the total fecundity across *all* adults in year B, which I've "pre-calculated" for simplicity. Let's make all this sex-specific while we're at it, because you always *should* in CKMR. And we'll use `autoloop`. Here's a naive example, showing off multi-statement code while we're again at it:

```
  SEXES <- c( "Female", "Male")
  adSAMPYEARS <- 1995:2000     # when our "adults" were sampled
  adAGES <- 6:9                # ... of adults (ie potential parents)
  juSAMPYEARS <- 1990:1999     # when our potential offspring were sampled
  juAGES <- 1:3                # ... and their range of ages at sampling
  juBYEARS <- (1990-3):(1999-1)  # implied range of juvenile birth-years

  # Any old values for fecundity-at-age...
  fec_SA <- autoloop( S=SEXES, A=adAGES, log( A))

  # Any old values for total fecundity over the years... say from 1e6 typical age-7 adults
  totfec_SB <- autoloop( S=SEXES, B=juBYEARS, 1e6 * fec_SA[ S, 7])

  try( Pr_POP_SYAYA <- autoloop(
      Sa=SEXES, Ya=adSAMPYEARS, Aa=adAGES,
      Yj=juSAMPYEARS, Aj=juAGES, {  # curly required coz multiple statements
    Bj <- Yj - Aj;                  # birth of Our juvenile
    Aa_at_Bj <- Aa - (Ya - Bj);     # how old was Our adult then?
    Pr <- ifelse( Ya >= Bj,         # was adult still alive then?
      fec_SA[ Sa, Aa_at_Bj] / totfec_SB[ Sa, Bj],  # yes: competing against other adults
      0)                            # no: dead already
    return( Pr)
   }))

## Error in autoloop(Sa = SEXES, Ya = adSAMPYEARS, Aa = adAGES, Yj = juSAMPYEARS,  :
##   @Sa="Female", Ya=1995L, Aa=9L, Yj=1992L, Aj=1L@  index woe for fec_SA[Sa, Aa_at_Bj]
```

[2]Provided you stick to the rules. Are the rules documented? Not really, though there are some in `?autoloop`. Just stick to them anyway, is my advice.

[3]Nor `for/while/repeat/break/next`, but `if` is more significant here.

Well, that didn't work, but at least the error message gives you a chance to figure out why. The problem is with accessing `fec_SA[Sa,Aa_at_Bj]` and clearly `Sa` won't be the cause, so it must be `Aa_at_Bj` . The error message doesn't show you `Aa_at_Bj` directly[4], because it's calculated only within the expression, but we can see where it came from. With `Ya=1995,Aa=9,Yj=1992,Aj=1` we would have `Bj=1992-1=1991` and `Ya-Bj=1995-1991=4` and `Aa_at_Bj=9-4=5`. But the range of adult ages, `adAGES`, which indexes `fec_SA`, is only 6:9. So the real problem is that our "adult" wasn't adult back then; s/he was too young. (The calculation could even have led to a negative age for the adult, if the juvenile was born before s/he was!) While we're at it, there is also the possibility of exceeding the maximum adult age— it won't be the case in the samples, because sampling kills the adults, but if we compare an adult to a juvenile born *after* the adult capture, then er *projected* age could be OOB. Let's fix that quickly:

```
  try( Pr_POP_SYAYA <- autoloop(
      Sa=SEXES, Ya=adSAMPYEARS, Aa=adAGES,
      Yj=juSAMPYEARS, Aj=juAGES, {
    Bj <- Yj - Aj;                  # birth of Our juvenile
    Aa_at_Bj <- Aa - (Ya - Bj);     # how old was Our adult then?
    Pr <- ifelse(
        (Ya >= Bj) &                   # too dead; note single ampersand coz vector
        (Aa_at_Bj >= min( adAGES)) &   # too young
        (Aa_at_Bj <= max( adAGES)),    # too old in future (and will actually be already dead!)
      fec_SA[ Sa, Aa_at_Bj] / totfec_SB[ Sa, Bj],  # yes: competing
      0)                               # no: not around
    return( Pr)
   }))

## Error in autoloop(Sa = SEXES, Ya = adSAMPYEARS, Aa = adAGES, Yj = juSAMPYEARS,  :
##   @Sa="Female", Ya=1995L, Aa=9L, Yj=1992L, Aj=1L@  index woe for fec_SA[Sa, Aa_at_Bj]
```

Same goddamn problem. The logic is fine now, but the problem is that `ifelse` still evaluates both the "yes" and "no" cases every time. So what we have to do, is to fix up the array access to *never* OOB, by tweaking the index; the `ifelse(...,0)` will give the right answer overall in such cases anyway, and we won't offend the OOB gods. There are many ways to code that, but the easiest is perhaps to pipe thru the `mvbutils::clamp` function, which restricts its input to a range:

```
  # No try() needed this time!
  Pr_POP_SYAYA <- autoloop(
      Sa=SEXES, Ya=adSAMPYEARS, Aa=adAGES,
      Yj=juSAMPYEARS, Aj=juAGES, {
    Bj <- Yj - Aj;                  # birth of Our juvenile
    Aa_at_Bj <- Aa - (Ya - Bj);     # how old was Our adult then?
    Pr <- ifelse(
        (Ya >= Bj) &                  # too dead
        (Aa_at_Bj >= min( adAGES)) &  # too young
        (Aa_at_Bj <= max( adAGES)),   # too old
      fec_SA[ Sa, Aa_at_Bj |> clamp( adAGES)] /
        totfec_SB[ Sa, Bj],           # yes, competing
      0)                              # not around
    return( Pr)
   })
```

Over time, I have come to feel that `ifelse` offers little by way of clarity and speed in contexts like this. Instead, I take advantage of "multiplication by 0" and R's automatic type-casting of logicals to numerics, like so:

```
  Pr_POP_SYAYA <- autoloop(
      Sa=SEXES, Ya=adSAMPYEARS, Aa=adAGES,
```

[4]While producing this example, I have started to think that the error message should include *all* available variables (including ones just calculated), not just the indices, because that might clarify things a lot! Next version...

```
    Yj=juSAMPYEARS, Aj=juAGES, {
   Bj <- Yj - Aj;                  # birth of Our juvenile
   Aa_at_Bj <- Aa - (Ya - Bj);     # how old was Our adult then?
   Pr <-
     (Ya >= Bj) *                   # if FALSE, entire expression will give 0
     (Aa_at_Bj >= min( adAGES)) *   # ditto
     (Aa_at_Bj <= max( adAGES)) *   # ditto
     fec_SA[ Sa, Aa_at_Bj |> clamp( adAGES)] /
       totfec_SB[ Sa, Bj]

   return( Pr)
  })
```

Lovely! And while I'm still at it, I'll illustrate another `autoloop` feature. First, you can return multiple arguments— they will each be an `offarray` with the same dimensions, and they'll will be put together into a named list. Second, you can use `mvbutils::extract.named` to immediately turn those list elements into actual variables, in the place where you do the call. The demonstration will compute the expected number of kin-pairs by covariate category, based on the relevant numbers of samples:

```
  # Make up some sample sizes...
  nsamp_ad_SYA <- autoloop( S=SEXES, Y=adSAMPYEARS, A=adAGES,
      10 * sqrt( A) * log( Y))
  nsamp_ju_YA <- autoloop( Y=juSAMPYEARS, A=juAGES,
      5 * sqrt( A+Y) * log( Y-A))

  extract.named( autoloop(
      Sa=SEXES, Ya=adSAMPYEARS, Aa=adAGES,
      Yj=juSAMPYEARS, Aj=juAGES, {
    Bj <- Yj - Aj;                  # birth of Our juvenile
    Aa_at_Bj <- Aa - (Ya - Bj);     # how old was Our adult then?
    Pr_POP_SYAYA <-
        (Ya >= Bj) *                   # if FALSE, entire expression will give 0
        (Aa_at_Bj >= min( adAGES)) *   # ditto
        (Aa_at_Bj <= max( adAGES)) *   # ditto
        fec_SA[ Sa, Aa_at_Bj |> clamp( adAGES)] /
          totfec_SB[ Sa, Bj]  ;
    E_POP_SYAYA <-
        Pr_POP_SYAYA *
        nsamp_ad_SYA[ Sa, Ya, Aa] *
        nsamp_ju_YA[ Yj, Aj]  ;
    returnList( Pr_POP_SYAYA, E_POP_SYAYA)
   }))

ls( patt='_POP_') # both are there...

## [1] "E_POP_SYAYA"  "Pr_POP_SYAYA"

sum( E_POP_SYAYA)

## [1] 86.35

E_POP_SYAYA[,1999:2000,8,1998:1999,1]

## , , Aa = 8, Yj = 1998, Aj = 1
##
##         Ya
## Sa       [,1999] [,2000]
```

```
##   Female  0.3363       0
##   Male    0.3363       0
##
## , , Aa = 8, Yj = 1999, Aj = 1
##
##         Ya
## Sa       [,1999] [,2000]
##   Female  0.3653  0.3364
##   Male    0.3653  0.3364

# Aggregate over adult sex, post hoc.
# Works coz autoloop produces things with named dimensions.
sumover( E_POP_SYAYA[,1999:2000,8,1998:1999,1],    'Sa')

## , , Yj = 1998, Aj = 1
##
##          Aa
## Ya          [,8]
##   [1999,] 0.6725
##   [2000,] 0.0000
##
## , , Yj = 1999, Aj = 1
##
##          Aa
## Ya          [,8]
##   [1999,] 0.7306
##   [2000,] 0.6728
```

3 With package RTMB

This isn't the right place to document the RTMB link! There should be a separate demo, probably in package offartmb. Anyway, the general point is: you can write a function using offarray and autoloop, and with minimal care you can evaluate it either directly in R, or via RTMB::MakeADFun, which creates a C-level version of your function *and* its derivative. That means it will run really fast, especially in the context of fitting to data. Plus, RTMB can automatically do Laplace approximation to integrate out random effects, etc etc. WOW!

What you need to do:

- Load the package offartmb, after[5] loading offarray and RTMB.
- Wrap the body of your "objective function" in a call to reclasso, as shown below. This will have *no effect* if you are running the function directly in R, but will also make it work nicely with RTMB::MakeADFun. It won't work if you don't.
- If you want to store variables calculated inside your objective function, so they are available afterwards, name them in a call to REPORTO() somewhere (like the C macro REPORT() in good old TMB).
- Avoid comparison-operations in your code, eg "==", ">". RTMB hates them. You might be able to get away with them if they only apply to "data", but in that case you are better off doing them beforehand and storing the results as additional data objects.
- Good practice is to put the data needed by your function into a separate environment, and attach that environment to your function. The things from REPORTO() will be stored there too. This is a general principle well worth learning about in R— not specific to RTMB, but perhaps particularly relevant there. The CKMR course examples all do it.

[5] I have added a hack that tries to make the loading-order irrelevant. "Tries".

An example, but not one to run:

```
if( FALSE){ # don't wanna run this!
  library( offartmb)

  myobjfun <- function( pars) reclasso( by=pars, { # magic line
      Ey <- X %% pars
      resid <- y-Ey
      REPORT0( Ey, resid) # two at once; could separate
    return( sum( sqr( y-Ey)))
    })

  X <- cbind( 1, 1:5) # regression
  y <- c( 5, 3, 4, 1, 2)
  pp0 <- 1:2

  # Base-R
  myobjfun( pp0)
  numvbderiv( myobjfun, pp0) # numerical deriv (rough)

  # RTMB
  ooo <- RTMB::MakeADFun( myobjfun, pp0)
  ooo$fn( pp0)
  ooo$gr( pp0)
  # ... and they all lived happily ever after
}
```

As yet, I *haven't* tested/implemented the following:

- several named parameter subsets (it's all just one vector at present, to be unpacked inside your function—coz that's what `optim/nlminb` expects of a base-R function);
- random effects (should just work, but...)
- there's definitely no simulation or OSA capability yet.