R cribsheet: CKMR model/design course

Mark Bravington

June 2021

As noted somewhere, you need R version 4.0.x or 4.1.x, and these standard packages:

`Matrix, Rcpp, RcppEigen, TMB`

(The last two are only for my package `deconvodisc` which is only required in a couple of examples.) Plus you need to install some of my packages from

```
install.packages( c( 'mvbutils', 'atease', 'offarray', 'deconvodisc', 'debug'),
 repos='https://markbravington.github.io/Rmvb-repo')
```

Actually you don't *need* `debug`, but I will use it while running scripts and it's very useful generally.

Before my scripts will run, you'll need to *turn off R's byte-compiler*, as described below (thanks to SB for picking this up!)

The course code all sits in a few scripts, and the data in "dot-rda" files (which can only be loaded into R— e.g. they are not spreadsheetable)— the structure is described in section 3 below. The script "csiro-course-2021-master.r" *creates* a bunch of functions that are referred to in the other scripts— you can just `source()` that one. Each other script pertains to one specific dataset/example, e.g. "fit_delfi_A.r"; you're meant to cut'n'paste lines and intersperse your own (especially for diagnostics/output after the actual fitting step), not to source those other scripts holus bolus. They each prepare & fit the data for one example; see below for more.

"IMOAL" <=> "In My Opinion, At Least"

1 Turn off the byte-compiler!

You may not even realize that R has (for the last few years) a built-in byte-compiler which does a lot of very complicated and totally ingenious behind-the-scenes stuff to produce frankly maaaaaaaarginal speedups IMOAL of your code. The BC is switched on by default, and in theory it should "just" work without your ever noticing. In practice, there's a bug in it (I reckon) that interferes with my

`offarray` package, which is ubiquitous in the scripts here. Because I have automatic code in my R startup to turn *off* the byte-compiler, I always forget that it is likely to bite other people...

I don't rate my chances of getting that BC bug fixed anytime soon (and I don't know exactly what causes it— NFI what goes on inside the BC, but my `offarray` code looks legit to me, albeit complicated), and don't fancy the argy bargy, so I just turn the BC off and forget about it. The script that hopefully does the same for you, is in "turn_off_bc.r". Eventually I will add an instruction to do that to all the "fit_<blah>r" scripts, but for now you'll need to run it manually. It needs to be `source()` ed once per R session— the earlier the better, I suspect, and I put it in my own ".Rprofile".

2 Idiosyncrasies

I haven't set out to write obscure R code, but everybody's R experience is different and there's no one best/clearest way. So I've just written stuff that *I* think is reasonably clear, and efficient enough to run in real-time— and, crucially, stuff that *I* can follow. It's commented, succinctly; IMOAL having too many mid-code comments is counterproductive. (One problem is that I didn't yet edit the comments between examples, all of which follow the same pattern but differ in the details. Sorry! Do bear that in mind as you puzzle your way through them...) If you are expecting to grasp CKMR, something which can get a bit complicated, then you should be well able to fight your way through (what *you* see as...) the idiosyncrasies of my R code!

The only bits you *really* need to follow, I reckon, are the log-likelihood functions.

A couple of tips for deciphering function code (tips you probably won't see in Proper Books):

- Don't necessarily start at the top. If you get a bit stuck, just look vaguely for statements in the middle that might do something interesting, and focus on those. Then work backwards to see how they were set up, and forwards to see what comes out at the end.
- If you don't understand a line in detail because it uses "unusual" syntax/functions, then just ignore it (for now, or for ever!). It's likely operationally crucial, but only in a nuts-n-bolts way. There are a surprisingly large number of components in an electric kettle (as you find out if you take one apart and try to reassemble it) but the main thing to understand is that there's a big element that heats stuff up. Eventually you can sort out all the grisly details if you really have to, but don't start by trying to understand every minor operational nuance.
- No-one tries to understand a misbehaving car engine by always starting with where the fuel goes in. You look for the bit where the smoke's coming out, and try to fix that, and work backwards if not. Same goes for code.

That said, there are a few things about the course code that might benefit from a few words. The first few are from my packages `mvbutils` and `offarray`.

- `cq( some, words, here)` is exactly like `c( 'some', 'words', 'here')` but looks less quoty.
- `offarray`: just like a regular R array (or matrix or vector), except that dimensions can start at any number, including negative (regular R arrays must start at 1). Character-indexed dimensions are also allowed, e.g. `SEX=cq(F,M)`. I find `offarray` a lot more natural for "real data" and I make less mistakes. Things basically "just work", but there *are* some differences from base-R behaviour (even if your index happens to start at 1)— see `?offarray` in the eponymous package if you need more info.
  - `dimseq(X)` is useful
  - converting to/from tables is easy
  - Many standard R operations, eg `matplot`, will misbehave on `offarrays`, so just do eg `standardop( unclass(x),...)`.
- `thing <- autoloop( X=<Xvalues>, Y=<Yvalues>, <R expression involving X & Y>)` is like doing this:

```
thing <- offarray(<dimensions derived from X/Yvalues>);
for(X in <Xvalues>) for( Y in <Yvalues>) {
  thing[X,Y] <- <evaluated expression given this X and Y>
}
```

  - but usually much faster and IMOAL actually clearer.
  - Sometimes I use `autoloop` to just construct an empty `offarray`, since the syntax is sometimes easier than the manual dimension-specification of directly calling an `offarray()` constructor. If the expression in the `autoloop` call is 0 etc, that's what's going on.
- Package `atease` lets me write `x@a` instead of `attr(x,"a")` and `x@a<-<something>` works too. This works for any `x`, not "just" S4 objects (which is what `@` is normally reserved for in R).
- `returnList(a,b)` puts `a&b` into a list with names `"a"` and `"b"`. I often use it to simulate the "multi-argument returns" that R used to have.
- I make heavy and unapologetic use of features in base-R, some of which may be unfamiliar (but they are documented, so that's on you). These include:
  `environments` and scoping, `local`, `with`, `evalq`, `mget`, `list2env`, subscript-by-matrix (using `MATSUB=...` syntax for `offarrays`),
- Notation: I am not 100% consistent here, but:
  - "Data" and "pop dyn quantities" often start with capital letters; my general housekeeping variables usually don't. I use underscores a lot, and never use camelCase except as noted next.

  - for (off)arrays, I usually embed the "names" of the subscriptors/dimensions at the end, eg `n_comp_POP_BYA` will be 3D with dimensions `B`(irth), `Y`(anked year— of sampling), `A`(ge).
  - Probabilities start with `Pr_`, and are (supposed to be...) of the form `Pr_<things>_<givens>`, so that `Pr_AtruB_CDE` is the probability of "Atru" and "B" (both) given values for "C", "D", and "E"; it will be a 5D array.
  - I am not 100% consistent, and sometimes favour aesthetics or brevity or laziness over convention. Just so you know.

- I haven't used any tidyverse or data.table stuff. FWIW Eric Anderson did implement a tidyverse version of (one of) the `boring_data_prep...` and `generic_lglk...` functions, so it can be done that way too. But I'm not going to.

3 Structure of code and data

You need to `source()` the "master script file" to create all the functions that will be used in the various examples.

The core of each example is a log-likelihood function called something like `generic_lglk_<blah>`. That's the one piec of code you really need to understand for the example (and to able to write yourself for you own data).

The actual fitting of CKMR model(s) to example <blah> is done by the *script* in file "fit_<blah>.r". The steps are:

1. `load()` the data for that example into the R session ;
2. summarize the data into a form suitable for CKMR models (eg adding up the number of kin-pairs and comparisons by "type", ie individual covariate values and nature-of-kinship)— by calling `boring_data_prep_<blah>(...)`. It's got that name for a good reason; you can look at the code if you really want, but I suggest not— concentrate on CKMR! The summaries get stored in an environment, called e.g. `env` — so that e.g. `env$n_MOP` has the MOP-counts.
3. Make a copy of the generic lglk that will be specific for this example, called eg `lglk_<blah>`.
   (a) The reason you have to make a copy is that `generic_lglk_<blah>` mentions the data by name (eg `n_MOP`) but can't be used as-is, because it hasn't been told what those data actually are for this *specific* example. And you might well be able to re-use the same generic in different examples, so I don't want to tie it to a single dataset from the get-go.

4. "Tell" your copy `lglk_<blah>` about the actual <blah> data, by resetting its environment to `env`. (This is fundamental R stuff, but lots of people don't know about it— google "R lexical scoping".)
   (a) Whenever `lglk_<blah>` gets called, it will now update various intermediate quantities such as the fitted numbers-at-age, which are also stored in `env`. That's how you get at estimated quantities-of-interest after actually fitting the model.
   (b) Note that `env` and `environment(lglk_<blah>)` are *identical*— they refer to exactly the same bit of memory and they're not copies of each other. Changing something in `env` will also change it in `environment(lglk_<blah>)`, and conversely.
5. pick starting values and fit the model using `nlminb()`. As I said, you've now got access to eg estimated numbers-at-age-and-year matrix inside `env`.
6. Tiddle around with the output in various ways: plots, diagnostics, covariances;... All based on stuff in `env`.

3.1 Data

The data for example "<blah>" lives in the file "samp_<blah>_raw.rda", which you need to `load()` (as per the script) to create an object `samp_<blah>` in your R session. It's a list with components `Samps` and the various kin-pairs, e.g `POPs`, `MHSPs`, `PHSPs`. Some non-sample-related "background" data for the example (e.g. catches-at-age, age-at-maturity) is stored in the attribute `public` which, if you have loaded my package `atease`, you can get at via `samp_<blah>@public`. There might also be "secret true stuff" that you're *not* allowed to look at while fitting, in `samp_<blah>@secret`.

`$Samps` will include fields `$Me` (a name), `$Sex`, `$Y`, plus any sample-specific age, length, or other data. It also has two logicals `$poss_par` and `$poss_off` . The former says whether the sample was used as a potential parent in POP-checks. `$poss_off` says whether it was used as potential offspring in POP-checks and HSP-checks. The reason they are not both usually `TRUE` for each sample, is that I may deliberately only have "done" (or kept the result of) certain types of comparisons, because they're the ones I actually expect to use in the modelling. In the boring-data-prep and lglk steps, comparisons are only totted up across samples that were listed as "suitable" via their `$poss` fields.

In CKMR you don't *have* to use every type of comparison— part of the art is deciding which ones aren't worth the trouble of trying to use. You just can't cheat by doing a comparison and then deciding whether to retain it based on its outcome!).

3.2 How I obtained the data

Naturally, via an elaborate process of stakeholder engagement, consultation, and respectful outreach— plus, of course, br;ibery, flattery, and downright extortion as required (i.e. often).

Or, more prosaically:

I did an IBM simulation of each population with my R package `kinsimmer`, using `simkin` which keeps track of ancestry; sampled from its dead ones and picked the kin-pairs in the sample, using `prep_from_sim2`; and saved the result into a file "samp_<blah>_raw.rda". Package `kinsimmer` is very fast (considering it's all in R, anyway) and reasonably flexible; it's based on Shane Baylis' package `fishSim` but tweaked for my own fell purposes. *If* you're doing your own simulations, then you might prefer to use Shane's package, which is probably better-documented and more general (it doesn't have to be fish!) but slower; *however*— as I'll explain during the course— I generally *don't* recommend simulations for routine CKMR anyway!