Model Simulations

2019-04-22

```r
library(spopmodel)
```

Background

Data

We will use three datasets native to `spopmodel`: `age_dist`; `prob_spawn`; and `number_eggs`'. Below we show a snippet of each. (If you wish, in your R console, run `library(spopmodel)`, and then view each by typing dataset name and pressing <>.)

```r
str(age_dist)
#> 'data.frame':    20 obs. of  2 variables:
#>  $ age : num  0 1 2 3 4 5 6 7 8 9 ...
#>  $ freq: num  2.19e+08 3.24e+03 2.01e+03 7.63e+02 8.93e+02 ...
cat("\n")
str(prob_spawn)
#> 'data.frame':    20 obs. of  3 variables:
#>  $ age : num  0 1 2 3 4 5 6 7 8 9 ...
#>  $ prob: num  0 0 0 0 0 0 0 0 0 0 ...
#>  $ se  : num  0 0 0 0 0 0 0 0 0 0 ...
cat("\n")
str(number_eggs)
#> 'data.frame':    20 obs. of  3 variables:
#>  $ age  : num  0 1 2 3 4 5 6 7 8 9 ...
#>  $ count: num  0 0 0 0 0 0 0 0 0 0 ...
#>  $ se   : num  0 0 0 0 0 0 0 0 0 0 ...
```

Note that all three datasets have an `age` field, and that (you may verify if you wish) age ranges from 0 to 19. We need one more dataset (survival probability), which will discuss below.

Survival Probability

We have available to us (i.e., native to `spopmodel`) dataframe `prob_survival`. However, this dataset only reflects one level of exploitation (μ ; ~0.13), and for this demo we want a range of say 0 to

0.30. To do this, we use `SurvivalProb()`. The resulting variable (`prob_surv`) is a list with 31 elements, one for each level of μ .

`SurvivalProb()` automatically includes ages-0 to -2. These data were culled from literature (<references here?>). We supplied `3:19` to `ages` to maintain consistency with our other datasets herein.

- `sRate` is survival rate (obtained from `ChapmanRobson()` using 2014-2016 CDFW trammel data.)
- `sRateErr`same as `sRate`
- `agesMu` reflect ages susceptible to harvest (obtained from `FitVBGM()` and related slot-limit length to age)

```
# set list of mu from 0 to 0.30
mus <- as.list(seq(from = 0, to = 0.30, by = 0.01))

prob_surv <- SurvivalProb(
  ages = 3:19,
  sRate = 0.94576,
  sRateErr = 0.04281,
  mu = mus,
  agesMu = 10:15
)
```

Simulations

Now that we have our starting data, let's run some simulations.

```
iters <- 1000
```

For simplicity, we've set the number of iterations to 1000. In reality and for improved accuracy, you'd want to run upwards of 5K to 10K or more.

Using `prob_spawn` we'll run simulations for spawning probability. We set the seed arbitrarily to 1234.

```
sims_prob_spawn <- Simulations(
  data = prob_spawn,
  prob = prob,
  std = se,
  iters = iters,
  seed = 1234,
  type = "spawning"
)

str(sims_prob_spawn)
```

```
#>  'simulations' num [1:20, 1:1000] 0 0 0 0 0 0 0 0 0 0 ...
#>  - attr(*, "iterations")= num 1000
```

Next — and like with did with `prob_spawn`, we'll run simulations using `number_eggs`.

```
sims_num_eggs <- Simulations(
  data = number_eggs,
  prob = count,
  std = se,
  iters = iters,
  seed = 1234,
  type = "numeggs"
)
#> Warning: `maxb` set at 3 * data[[prob]]

str(sims_num_eggs)
#>  'simulations' num [1:20, 1:1000] 0 0 0 0 0 0 0 0 0 0 ...
#>  - attr(*, "iterations")= num 1000
```

Simulations for survival probability are a bit more tricky because we must iterate over all items in `prob_surv` (i.e., all levels of mu in `mus`). For this, we use R's `mapply()`. (Note in the `MoreArgs` argument we've had to quote field names. Observe when directly calling `Simulations()` we did not need quotes.) The `recruitment` parameter denotes period (in years) of successful recruitment. We can change accordingly, but for this demonstration we'll keep it at 5. We set `SIMPLIFY = FALSE`, as we want to maintain list datatype.

```
sims_prob_surv <-mapply(
  FUN = Simulations,
  prob_surv,
  MoreArgs = list(
    prob = "Prob",
    std = "Err",
    recruitment = 5,
    iters = iters,
    # makes a difference setting to NULL
    # seed = 1234,
    seed = NULL,
    type = "survival"
  ),
  SIMPLIFY = FALSE
)
```

Next we calculate fecundity simulations using egg count and spawning probability simulations. We assume a 0.5 female:male ratio.

```
sex_ratio <- 0.5

sims_fecund <- sims_num_eggs * sims_prob_spawn * sex_ratio

# str(sims_fecund)
```

Now with all the sims in place, we run poplation projections for each level of μ . First, we create some helpful variables for use with `lapply()`.

`final_age` gets survival probability of the oldest fish (i.e., age-19) for each level of μ . The model assumes last age does not die. (We hardcode `20` and `2:3` because we know we have 20 age groups (0-19) and columns 2 & 3 are probability (“Prob”) and error (“Err”). Ideally, it would be best to generate these numbers programmatically.) `mu_levels` creates a vector of μ levels given the names of `sim_prob_surv`. We’ll use this as our “looping” variable in `lapply()`.

```
final_age <- lapply(prob_surv, FUN = "[", 20, 2:3)

# should be TRUE
# identical(
#   names(sims_prob_surv),
#   names(final_age)
# )

mu_levels <- setNames(
  object = names(sims_prob_surv),
  nm = names(sims_prob_surv)
)
```

Population projections

Admitingly, there are cleaner ways to perform these next steps. In the future, we may create some functions or methods to handle such processes but for now this will suffice.

For each value in `mu_levels` we need to get population projections. We essentially do this using `popbio::pop.projection()` (`PopProjections()` is basically a convenient wrapper.) `popbio::pop.projection()` requires a projection matrix (Leslie Matrix in our case), an age vector (this is our `freq` field in `age_dist`), and iterations (or `period` as implemented by `PopProjections()`).

Creation of the Leslie Matrix is handled within `PopProjections()`. So we just supply the proper arguments. Recall `sims_prob_surv` and `final_age` are lists each with `n = length(mus)` elments. Thus, the use of `[[x]]` within our `lapply()` loop.

```
pop_proj <- lapply(mu_levels, FUN = function(x) {

  PopProjections(
    fSims = sims_fecund,
```

```
      sSims = sims_prob_surv[[x]],
      mn = final_age[[x]][["Prob"]],
      sdev = final_age[[x]][["Err"]],
      ageFreq = age_dist[["freq"]],
      period = 20
    )

})
```

Lambda

…and now what we paid top dollar for: **Lambda** (λ), the population growth rate. `pop_proj` is a massive list within a list. Within each μ level exists an even larger list (size is 5 * iters, or in our case 5000). Five (5) is the number of values returned by `popbio::pop.projection()` (see help file for names of return values). We need to extract `pop.changes` (e.g., pop_proj[["mu_0"]]["pop.changes",], which gets all 1000 `pop.changes` for μ level 0).

`Lambda()` returns a dataframe. Currently `MuLevel` is set to TBD. So below we use a cheap way of replacing TBD with the appropriate value. In the future, we'll improve `Lambda()` to handle this. Numeric `MuLevel` is important for plotting, the next and final step.

```
lambda_mu <- lapply(mu_levels, FUN = function(x) {

  mu <- as.numeric(sub(pattern = "mu_", replacement = "", x = x))

  out <- Lambda(popChanges = pop_proj[[x]]["pop.changes", ])

  out[["MuLevel"]] <- mu

  out

})

lambda_mu <- do.call(what = rbind, args = lambda_mu)
rownames(lambda_mu) <- NULL

lambda_mu
#>    MuLevel NumSims MeanLambda MedLambda  LBLambda UBLambda
#> 1     0.00    1000  1.0182567 1.0092605 0.9513938 1.183160
#> 2     0.01    1000  1.0148649 1.0058614 0.9503390 1.162420
#> 3     0.02    1000  1.0145084 1.0043394 0.9513826 1.165231
#> 4     0.03    1000  1.0115063 1.0027001 0.9458925 1.164427
#> 5     0.04    1000  1.0069457 0.9999962 0.9409417 1.159131
#> 6     0.05    1000  1.0067535 0.9971523 0.9400273 1.170781
#> 7     0.06    1000  1.0021715 0.9947435 0.9402286 1.152717
#> 8     0.07    1000  0.9990580 0.9917455 0.9366875 1.147496
#> 9     0.08    1000  0.9976594 0.9900108 0.9293552 1.154299
```

```
#> 10    0.09    1000  0.9941981 0.9868757 0.9265558 1.152466
#> 11    0.10    1000  0.9919172 0.9853065 0.9239571 1.140664
#> 12    0.11    1000  0.9909891 0.9839054 0.9285465 1.135044
#> 13    0.12    1000  0.9851585 0.9802888 0.9261591 1.108791
#> 14    0.13    1000  0.9848172 0.9788385 0.9140775 1.134598
#> 15    0.14    1000  0.9807339 0.9759609 0.9150239 1.112881
#> 16    0.15    1000  0.9773120 0.9746546 0.9109660 1.107820
#> 17    0.16    1000  0.9755107 0.9709586 0.9114579 1.106613
#> 18    0.17    1000  0.9734219 0.9687661 0.9047649 1.113315
#> 19    0.18    1000  0.9695270 0.9667187 0.9006204 1.109995
#> 20    0.19    1000  0.9691114 0.9638878 0.9039897 1.086659
#> 21    0.20    1000  0.9667529 0.9648807 0.8932784 1.098186
#> 22    0.21    1000  0.9621530 0.9592392 0.8964956 1.081295
#> 23    0.22    1000  0.9608333 0.9587785 0.8919415 1.072102
#> 24    0.23    1000  0.9578869 0.9563753 0.8879434 1.074871
#> 25    0.24    1000  0.9591810 0.9564630 0.8904286 1.068210
#> 26    0.25    1000  0.9550264 0.9535679 0.8801565 1.075110
#> 27    0.26    1000  0.9530801 0.9512719 0.8840972 1.065095
#> 28    0.27    1000  0.9509013 0.9502048 0.8717639 1.060333
#> 29    0.28    1000  0.9475277 0.9477816 0.8763844 1.057907
#> 30    0.29    1000  0.9449264 0.9448142 0.8723496 1.047645
#> 31    0.30    1000  0.9433686 0.9439012 0.8649053 1.053774
```

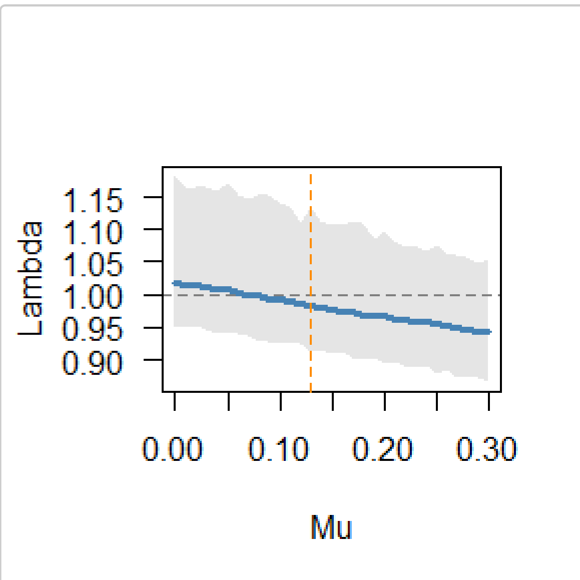
```
# x & y values for drawing polygon as lower & upper bounds
poly_list <- list(
  x = c(
    lambda_mu[["MuLevel"]][1],
    lambda_mu[["MuLevel"]],
    rev(lambda_mu[["MuLevel"]][-1])
  ),
  y = c(
    lambda_mu[["LBLambda"]][1],
    lambda_mu[["UBLambda"]],
    rev(lambda_mu[["LBLambda"]][-1])
  )
)

# create the plot with appropriate limits
plot(
  x = range(lambda_mu[["MuLevel"]]),
  y = range(lambda_mu[, c("LBLambda", "UBLambda")]),
  type = "n",
  panel.last = abline(h = 1, col = "grey50", lty = 2, lwd = 0.25),
  panel.first = polygon(poly_list, col = "grey90", border = NA),
  las = 1,
  xlab = "Mu",
```

```
  ylab = "Lambda"
)

# add data (mean lambda over mu)
lines(
  x = lambda_mu[["MuLevel"]],
  y = lambda_mu[["MeanLambda"]],
  col = "steelblue",
  lty = 1,
  lwd = 3
)

# optional: add current mu
# adding point might be better but would need to workout accurate y-val
# points(x = 0.13, y = 0.98, col = "darkorange", pch = 19)
abline(v = 0.13, col = "darkorange", lty = 2, lwd = 0.5)
```




```
# optional: lower & upper bounds as lines
# lines(
#   x = c(lambda_mu[["MuLevel"]]),
#   y = c(lambda_mu[["LBLambda"]]),
#   col = "blue"
# )
# lines(
#   x = c(lambda_mu[["MuLevel"]]),
#   y = c(lambda_mu[["UBLambda"]]),
#   col = "red"
# )
```