Changes in secr 4.0

Murray Efford

2019-12-07

Contents

Introduction 1

Multi-threading 2

Faster fitting of ‘proximity’ and ‘count’ data 2

Data compression 2

1. Binary proximity data as binomial count 2

2. Pass only non-zero counts 3

Operation 3

Restrictions 3

Individual mask subsets 3

Miscellaneous 4

Backward compatibility 4

References 4

Appendix 1. Specific changes in secr 4.0 6

Appendix 2. Code for fitting models omitted from secr 4.0 6

House mouse 6

Stoat 7

Ovenbird 7

Brushtail possum 7

Deer mouse 7

DENSITY demo data 8

Ovenbird song 8

Appendix 3. Functions calling multi-threaded C++ in secr 4.1 8

Introduction

Version 4.0 has been substantially re-written. Most code from earlier versions will still work. However, there are major changes to some features, and results may differ. Version 4.1 will soon be released with fixes for bugs in secr 4.0 (covariate models were prone to error).

Table 1. Key changes in secr 4.0 (see also see Appendix 1)

<table>
<thead>
<tr>
<th>Category</th>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Multi-threading of likelihood computation</td>
<td>New</td>
</tr>
<tr>
<td>General</td>
<td>Fast evaluation for proximity and count detectors</td>
<td>New, optional</td>
</tr>
</tbody>
</table>
Multi-threading

Multi-threading is a way to use multiple processor cores to speed computation. secr 4.0 performs multi-threading with `parallelFor` in RcppParallel (Allaire et al. 2018). Multi-threading is used in computationally intensive functions, particularly likelihood evaluation in `secr.fit` (Appendix 3).

Multi-threading was not applied in secr 4.0 for polygon and transect detector types because these used integration code from the R API (Rdqags). Multi-threading is used for these detector types in secr 4.1 which uses the integration interface in RcppNumerical (Qiu et al. 2019).

By default, the code will use one less than the number of available cores; this leaves some capacity for other programs. The number of cores may be controlled directly with the `ncores` argument of `secr.fit`. Parallelisation in `secr.fit` is at the level of entire detection histories: histories are distributed over cores (in general the number of histories is much greater than the number of cores). In a typical application on a quad-core desktop running Windows (8 virtual cores) this can speed up model fitting by a factor of 3–4.

Parallel processing in secr previously relied on the `parallel` package, with mixed success (setting ncores>1 could even slow down computation). The old function `par.secr.fit` no longer speeds up computation when ncores>1, but it is retained.

Faster fitting of ‘proximity’ and ‘count’ data

Data from binary proximity (‘proximity’) and count proximity (‘count’) detectors often take the form of large, sparse 3-D arrays. For one session the array has dimensions $n$ (number of detected individuals), $s$ (number of occasions) and $k$ (number of detectors), and most cells will be zero.

The code in secr 3.2 always iterated over all cells of the array; this does not constrain the models that can be fitted. For many models it is faster to operate on a compressed version of the data, and this is the default in secr 4.0. Fast processing (and the associated compression) can be bypassed in secr 4.0 by setting the details argument `fastproximity = FALSE`. This may be needed to avoid restrictions.

Data compression

Two types of compression are applied automatically by default (`fastproximity = TRUE`):

1. Binary proximity data as binomial count

Binary data for a particular animal and detector over $s$ occasions are summed and modelled as a binomial count of size $s$ on a single occasion. This option has been available with manual preprocessing using code such as

```r
CH1 <- reduce(CH, outputdetector = 'count', by = 'all')
secr.fit(CH1, binomN = 1)
```
The capthist object from the `reduce` step includes detectors with a usage attribute that records the number of occasions pooled for each detector; `binomN = 1` directs `secr.fit` to find the binomial size in the usage attribute.

2. **Pass only non-zero counts**

Even with compression over occasions, the number of detections at most detectors is usually zero. It is efficient to preprocess the data for each animal into a list of those detectors with non-zero counts, and the associated counts. The likelihood code in `secr 4.0` automatically applies this compression under the `fastproximity` option.

**Operation**

Compressed counts may be modelled as Poisson or binomial. A Poisson model is used for the compressed counts if detector type `count` appears anywhere in the input data and a Poisson model is specified via the argument `binomN = 0`.

Otherwise, a binomial model is used for the compressed counts. The size of the binomial model is specific to each detector. It is set internally by manipulating the `usage` attribute. Most simply (`binomN = NULL`) this is merely the number of occasions or the sum over occasions of the input `usage` if that exists. For specified `binomN > 1` the `usage` is multiplied by `binomN`.

**Restrictions**

`fastproximity = TRUE` is applied by `secr.fit` under these conditions:

- all detectors are of type ‘proximity’, ‘count’ or ‘capped’
- no learned response in model (~b, ~bk etc. prohibited)
- no temporal variation in model (~t, ~T, ~tcov etc. prohibited)
- `groups = NULL`
- no variation over detectors in model [potential, not implemented]

If any condition is not met then processing continues as if `fastproximity = FALSE`.

**Individual mask subsets**

The base algorithm of `secr` treats all points on the habitat mask as potential locations for the activity centre of each individual i.e. integration is over the area of the mask. When the mask is large relative to the home range, most points will contribute almost nothing to the integral, and time is wasted including them in the summation.

A more efficient approach is to consider only mask points in the vicinity of the known detections of an individual. `secr 4.0` allows the user to specify a maximum mask radius (`details$maxdistance`) with respect to the centroid of detections, computed separately for each animal. The centroid is the geometric mean of detection locations for each individual. If `maxdistance` is not specified then summation is over all mask points. The entire mask is always used for the likelihood component due to undetected individuals.

A simple example:
library(secr)
fit0 <- secr.fit(captdata, buffer = 100, trace = FALSE)
fitims <- secr.fit(captdata, buffer = 100, details = list(maxdistance = 100), trace = FALSE)
fits <- secrlist(fit0, fitims)
sapply(fits, '[[', 'proctime')

## fit0.elapsed fitims.elapsed
## 3.43 2.55

collate(fits)[1,,,D']

## estimate SE.estimate lcl ucl
## fit0 5.479818 0.6467416 4.351635 6.900488
## fitims 5.500105 0.6481329 4.369290 6.923585

It is up to the user to select a limit that is large enough not to affect the likelihood. This should be larger than the expected home range radius, as the centroid is only an approximation to the activity centre. The value $5\sigma$ is suggested for a half-normal detection function.

Using individual mask subsets makes fitting much faster, particularly when activity areas are small relative to the overall extent of the mask (examples in Table 2).

Table 2. Relative timings for model fitting in secr.fit with two scenarios for halfnormal $\sigma$ in relation to spacing $s$ between detectors on a $10 \times 10$ square grid. Using individual, local mask achieves $>5$-fold speed improvement with the usual approach to likelihood evaluation (‘Basic’), but only minor improvement when other tools are used to speed up execution (‘Fast’). $D = (0.8/\sigma)^2$; $\lambda_0 = 0.1/\sigma^{0.5}$.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Method</th>
<th>Basic</th>
<th>Fast</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigma = $s$, full mask</td>
<td></td>
<td>303</td>
<td>7</td>
</tr>
<tr>
<td>sigma = 2$s$, full mask</td>
<td></td>
<td>112</td>
<td>6</td>
</tr>
<tr>
<td>sigma = $s$, local mask</td>
<td></td>
<td>51</td>
<td>4</td>
</tr>
<tr>
<td>sigma = 2$s$, local mask</td>
<td></td>
<td>24</td>
<td>3</td>
</tr>
</tbody>
</table>

Miscellaneous

Backward compatibility

Code that worked in previous versions should work in secr 4.0, but this is not guaranteed. Problems are likely with models fitted in earlier versions, as these will not include the new components ‘designD’, ‘designNE’ and ‘learnedresponse’. The solution is to refit the models from scratch in secr 4.0, or continue using the earlier version.

Fewer fitted models are included with the datasets. If you rely on a missing model, fit it yourself with the code in Appendix 2.

References


Appendix 1. Specific changes in secr 4.0

<table>
<thead>
<tr>
<th>Function</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>secr.fit etc.</td>
<td>Argument <code>ncores</code> now refers to the number of cores used for multi-threading by <code>RcppParallel</code>, not the number of clusters used by <code>parallel</code>; default NULL uses 1 less than available cores</td>
</tr>
<tr>
<td>secr.fit</td>
<td><code>autoini = 'all'</code> for start values from multi-session data (experimental)</td>
</tr>
<tr>
<td>secr.fit</td>
<td>New components designD, designNE and learnedresponse in output <code>secr</code> object</td>
</tr>
<tr>
<td>secr.fit</td>
<td><code>LLonly = TRUE</code> uses automatic parameter values when start not provided</td>
</tr>
<tr>
<td>secr.fit</td>
<td><code>LLonly = TRUE</code> output has new attributes ‘npar’, ‘preptime’ and ‘LLtime’</td>
</tr>
<tr>
<td>sim.capthist</td>
<td>Detection function for polygons and transects must be of hazard type (14:19) as required by <code>secr.fit</code></td>
</tr>
<tr>
<td>make.systematic</td>
<td>New argument <code>order</code> controls sequence of clusters in generated traps object</td>
</tr>
<tr>
<td>par.secr.fit</td>
<td>No longer automatically sets <code>ncores=1</code> in each call to <code>secr.fit</code></td>
</tr>
<tr>
<td>RSE</td>
<td>New function to extract precision of ‘real’ parameter estimates</td>
</tr>
<tr>
<td>summary.capthist</td>
<td>Component ‘versions.time’ includes elapsed time</td>
</tr>
<tr>
<td>summary.secr</td>
<td>Acquire <code>ncores</code> argument</td>
</tr>
<tr>
<td>autoini</td>
<td><code>region.N</code>, <code>suggest.buffer</code></td>
</tr>
<tr>
<td>fx.secr</td>
<td>Argument <code>normal</code> dropped</td>
</tr>
<tr>
<td>fxi.contour</td>
<td>Argument <code>ncores</code> dropped</td>
</tr>
<tr>
<td>mask.check</td>
<td><code>pmixProfileLL</code>, <code>simulate.secr</code></td>
</tr>
<tr>
<td>read.SPACECAP</td>
<td>Removed</td>
</tr>
<tr>
<td>write.SPACECAP</td>
<td></td>
</tr>
</tbody>
</table>

The new components in ‘secr’ objects returned by `secr.fit` are described on the help page for the function.

Appendix 2. Code for fitting models omitted from secr 4.0

Earlier versions of `secr` included pre-fitted models to demonstrate various features. Most of these have been dropped in `secr 4.0` to save space and to reduce maintenance costs. The omitted models may be regenerated with the code in this appendix.

House mouse

```r
morning <- subset(housemouse, occ = c(1,3,5,7,9))
afternoon <- subset(housemouse, occ = c(2,4,6,8,10))
morning.0 <- secr.fit(morning, buffer = 20, trace = FALSE)
morning.h2 <- secr.fit(morning, buffer = 20, model=list(g0~h2), trace = FALSE)
morning.0h2 <- secr.fit(morning, buffer = 20, model=list(sigma~h2), trace = FALSE)
morning.h2h2 <- secr.fit(morning, buffer = 20, model=list(g0~h2, sigma~h2), trace = FALSE)
morning.t <- secr.fit(morning, buffer = 20, model=g0~t, trace = FALSE)
morning.b <- secr.fit(morning, buffer = 20, model=g0~b, trace = FALSE)
housemouse.0 <- secr.fit(housemouse, buffer = 20, trace = FALSE)
```
housemouse.ampm <- secr.fit(housemouse, model = g0-tcov, timecov = 
  c(0, 1, 0, 1, 0, 1, 0, 1, 0, 1), buffer = 20, trace = FALSE)
housemouse.ampmh2h2 <- secr.fit(housemouse, model=list(g0-tcov+h2, sigma-tcov+h2),
  timecov = c(0, 1, 0, 1, 0, 1, 0, 1, 0, 1), buffer = 20, trace = FALSE)

Stoat

stoat.model.HN <- secr.fit(stoatCH, buffer = 1000, detectfn = 0, trace = FALSE)
stoat.model.HZ <- secr.fit(stoatCH, buffer = 1000, detectfn = 1, trace = FALSE,
  biasLimit = NA)
stoat.model.EX <- secr.fit(stoatCH, buffer = 1000, detectfn = 2, trace = FALSE)

Ovenbird

ovenmask <- make.mask(traps(ovenCH), type = 'pdot', buffer = 400,
  spacing = 15, detectpar = list(g0 = 0.03, sigma = 90), nocc = 10)
ovenbird.model.1 <- secr.fit(ovenCH, mask = ovenmask, trace = FALSE)
ovenbird.model.1b <- secr.fit(ovenCH, mask = ovenmask, model = list(g0 ~ b), trace = FALSE)
ovenbird.model.1T <- secr.fit(ovenCH, mask = ovenmask, model = list(g0 ~ T), trace = FALSE)
ovenbird.model.h2 <- secr.fit(ovenCH, mask = ovenmask, model = list(g0~h2), trace = FALSE)
ovenbird.model.D <- secr.fit(ovenCH, mask = ovenmask, model = list(D ~ Session),
  trace = FALSE)

Brushtail possum

possum.model.0 <- secr.fit(possumCH, mask = possummask, trace = FALSE)
possum.model.b <- secr.fit(possumCH, mask = possummask, model = g0-b,
  trace = FALSE)
possum.model.Ds <- secr.fit(possumCH, mask = possummask, model =
  list(D ~ d.to.shore), link = list(D = 'identity'), trace = FALSE,
  method = 'Nelder-Mead')

Deer mouse

ESG.0 <- secr.fit(deermouse.ESG, trace = FALSE)
ESG.b <- secr.fit(deermouse.ESG, model=g0-b, trace = FALSE)
ESG.t <- secr.fit(deermouse.ESG, model=g0-t, trace = FALSE)
ESG.h2 <- secr.fit(deermouse.ESG, model=g0-h2, trace = FALSE)
WSG.0 <- secr.fit(deermouse.WSG, model=g0-1, trace = FALSE)
WSG.b <- secr.fit(deermouse.WSG, model=g0-b, trace = FALSE)
WSG.t <- secr.fit(deermouse.WSG, model=g0-t, trace = FALSE)
WSG.h2 <- secr.fit(deermouse.WSG, model=g0-h2, trace = FALSE)
DENSITY demo data

```r
secrdemo.0 <- secr.fit(captdata, trace = FALSE)
secrdemo.CL <- secr.fit(captdata, CL = TRUE, trace = FALSE)
secrdemo.b <- secr.fit(captdata, model = list(g0 = ~b), trace = FALSE)
```

Ovenbird song

```r
signalCH.525 <- subset(signalCH, cutval = 52.6)
omask <- make.mask(traps(signalCH), buffer=200)
ostart <- c(log(20), 80, log(0.1), log(2))
oven.song.model.1 <- secr.fit(signalCH.525, mask = omask, start = ostart, detectfn = 11, trace = FALSE)
oven.song.model.2 <- secr.fit(signalCH.525, mask = omask, start = ostart, trace = FALSE)
```

Appendix 3. Functions calling multi-threaded C++ in secr 4.1

These internal C++ functions use multi-threading. Set ‘ncores = 1’ or the `secr.fit` details argument ‘grain = 0’ to suppress multi-threading. ‘Level’ refers to the units that are distributed over parallel threads. `makegkPointcpp` precomputes probability and hazard values for each combination of parameter values, detector, and mask point.

<table>
<thead>
<tr>
<th>Function</th>
<th>Level</th>
<th>Called by user-facing functions...</th>
</tr>
</thead>
<tbody>
<tr>
<td>makegkPointcpp</td>
<td>cell of habitat mask</td>
<td>autoini, esa, expected.n, fxi, region.N, secr.fit, sim.secr</td>
</tr>
<tr>
<td>makegkPolygoncpp</td>
<td>cell of habitat mask</td>
<td>esa, expected.n, fxi, region.N, secr.fit</td>
</tr>
<tr>
<td>fasthistoriescpp</td>
<td>detection history</td>
<td>secr.fit</td>
</tr>
<tr>
<td>polygonhistoriescpp</td>
<td>detection history</td>
<td>secr.fit</td>
</tr>
<tr>
<td>sightingchatcpp</td>
<td>detection history</td>
<td>secr.fit</td>
</tr>
<tr>
<td>signalhistoriescpp</td>
<td>detection history</td>
<td>secr.fit</td>
</tr>
<tr>
<td>simplehistoriescpp</td>
<td>detection history</td>
<td>secr.fit</td>
</tr>
<tr>
<td>simplehistoriesfxicpp</td>
<td>detection history</td>
<td>fxi</td>
</tr>
</tbody>
</table>