Package: kinsimmer2 (via r-universe)

November 8, 2026

Title Fast kin-retaining simulation

Version 1.0.41

Author Mark Bravington, based on ideas from Shane Baylis

Description See title, plus litewt_df objects

Maintainer <markb2@summerinsouth.net>

Depends R (>= 4.2)

Imports utils, mvbutils, atease, offarray

Suggests fastmatch

License GPL-2

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

Date/Publication 2026-11-08 08:29:04 UTC

RemoteUrl https://github.com/markbravington/kinsimmer2

RemoteRef HEAD

RemoteSha 2c3983d754ca96bdc7d511486e5c85742a83a5ec

Contents

litewt_df 2
make_sim_names2 3
mate_multipat_noassoc 4
quadsolve 5
quickin 6
rbernoulli 7
samp_from_catch 8
simkin2 10
subset_samples 13

Index **15**

litewt_df *Lightweight data-frame-like objects*

Description

`data.frame` has all sorts of well-known not-necessarily-welcome baggage, which can slow it down and/or make it cumbersome to use. `tibble` *may* resolve much of that, but it requires some crazy number of extra tidybollocks packages to be installed. We don't all want that. So `litewt_df` offers a super-minimal alternative. It doesn't do many error checks, and is meant for coding tasks rather than user-accessible data manipulation.

Main differences are that `litewt_df` isn't checked much for consistency, after initial setup. Subset reasonableness is mostly not checked. Subsetting with square-brackets doesn't drop rows or columns; it always returns a `litewt_df` (i.e. a `list`). To get a single value or vector of values from a column "mycol", do `x$mycol[...]` or, more horribly, `unlist( x[...,"mycol"])`. Subset-assignment either expects a single scalar assignand (i.e. "value"), or a `litewt_df`; if the latter, its names must match those of the columns to be amended in the assignee (i.e. "x").

There is no `cbind`, since it's so easy to do directly. There is an `rbind` method, which *should* work even when the non-first arguments are `data.frames` as well. However, for simple operations such as adding empty rows, `grow_litewt_df` is better; see **Examples**.

Methods exist for `print`, `dim`, subsetting (`[`), and subset-replacement (`[<-`), single-column assignment/replacement (`$<-` and `[[<-`), and conversion to/from `data.frame`. For the "from" case, you can use either `as.litewt_df( mydf)` or `litewt_df( mydf)`.

If this stands the test of time, I will eventually move it to `mvbutils` or to its own package.

Groan: Presumably someone *could* have made `tibble` really lightweight too, but they decided not to. Whaaaatever.

Usage

```
litewt_df(...)
as.litewt_df( df_or_list)
grow_litewt_df( x, nrows, value=NA)
```

Arguments

`...`	Either one `data.frame` or several vectors, all of the same length or of length 1. Each element should be named in the call, e.g. `litewt_df(x1=1:6,<...>)`, unless it is just the name of a variable, in which case it will be auto-deduced. Only "atomic" objects are allowed (see `is_atomic`). Column-specific S3 classes are left as-is(for now) and may or may not work. The columns must all have unique names.
`df_or_list`	see if you can figure it out for yourself!
`x`	a `litewt_df`
`nrows`	how many rows to append?

value — what value to put in those rows. The same value will be used for all columns (as well as all rows), so be careful not to accidentally change column type; the default of NA is safe.

Value

Object of class litewt_df: just a list with an S3 class.

Examples

```
## I should probably demo differences from 'data.frame'
## Should only occur when you try to do dodgy stuff, I hope.
xx <- litewt_df( x=1:3, y=letters[1:3])
xx[ 2,]
xx[ 3,] <- '0'
xx$z <- 4 # cbind! Note auto-repped
xx$z[ -2]
xx[1,'z'] # not the same as...
xx$z[1]
## Single-arg subsetting a la data.frame (with reluctance on my part...)
xx[2]
xx[[2]] <- letters[ 4:6] # the only way to replace several elements of one column
xx[3,2] <- 'ZZZ'
xx[2:1] # reverse columns
# Subassignment:
xx[,2] <- 'AAA'
try( xx[,2] <- letters[5:7]) # not allowed; only scalar or litewt_df value
xx[[2]] <- letters[ 5:7]
# but better to use names:
xx$y <- letters[ 8:10]
xx <- grow_litewt_df( xx, 2, value=33)
rbind( xx, xx)
# Naming:
threeseq <- 1:3
aa <- litewt_df( threeseq, y=letters[1:3]) # OK
try( aa <- litewt_df( threeseq, 2:4, y=letters[1:3])) # not OK
try( aa <- litewt_df( (threeseq), y=letters[1:3])) # also not OK; gotta be pure name
```

---

make_sim_names2 *Helper for kin-simulation*

Description

ATM this trivial function just returns the definitions of SEXES and CODS (possible Causes-Of-Death) for use in other code, eg as offarray indices.

Note: In the original kinsimmer, all individual info had to be numeric; also, individual fields needed to have "names" but I didn't want to lug a dimnames around everywhere, so numeric column "names" were useful. But kinsimmer2::simkin2 uses litewt_df which can cope with character variables and which *requires* column names, so there's less need for a separate function.

Usage

```
make_sim_names2()
```

Value

A named list; see **Examples**. You could wrap the call in `extract.named` so that `SEXES` and `CODS` are auto-defined in your ensuing code.

Examples

```
make_sim_names2()
#  $SEXES
#  [1] "FEMALE" "MALE"
#
#  $CODS
#  [1] "NAT" "HUM"
#
```

---

mate_multipat_noassoc *Father-picking functions*

Description

These are examples of father-picking functions. The `singlepat` one assigns the same randomly-chosen (fecundity-weighted) father to all offspring from each clutch; it is fast, and works whether `clutch_size` is 1 or more. The `multipat` one is a bit slower and fairly complex coz it allows multiple paternity per litter ("clutch", i.e. all offspring from one particular female that year). Each male's expected offspring total is proportional to his fecundity. There's no associative mating, nor pair-bonding. The rate of multiple paternity is controlled by `pnewdad`, which should be passed to `simkin2` via `other_vars`.

The first offspring in each clutch gets a random dad, probabilistically but according to fecundity. Conceptually, each further offspring draws a `runif(1)` and if that falls above `pnewdad`, this offspring gets the same dad as the previous offspring; otherwise a new dad is chosen randomly. The new dad *might*, by chance, be the same as the old dad.

The original clutches are split up into sub-clutches, each having the same mother & father. It's possible that the same father-mother combo might generate more than one sub-clutch, but the `mvb_mate2` driver function is totally cool with that.

I have tried to make the `multipat` code really efficient, avoiding loops as much as possible. Father-pickers with associative mating but no pair-bonding could get *very* slow, if each possible father is considered for each possible mother (and offspring...). Those cases could probably be handled adequately by introducing a "group" column, looping over group, and restricting to fathers and mothers within the same group.

Usage

```
mate_singlepat_noassoc( indlivy, clutch_size, imums, year)
mate_multipat_noassoc( indlivy, clutch_size, imums, year)
```

Arguments

indlivy	living animals this year, who might be fathers or mothers
clutch_size	vector of clutch sizes, one per successfully-breeding female
imums	which animals (rows) in indlivy are the successfully-breeding females.
year	obvious

Value

A litewt_df with 3 columns: clutch_size, mum, dad. The latter two should be the row-numbers within indlivy (*not* the ID values).

See Also

mvb_mate2, simkin2

Examples

```
1+1
```

---

quadsolve *Robust quadratic equation solver*

Description

As any fule kno it is easy to solve a quadratic equation, but getting a numerically robust solution can be tricky. This duz it fairly well, and is vectorized so you can solve lots at once. Coefficients must be real, not complex. See The Internet for how it works.

Usage

```
quadsolve( a, b, c)
```

Arguments

a, b, c	real-valued scalars or vectors in a*x^2+bx+c==0. Vectors must be of the same length; scalars will be replicated if necessary.

Value

A matrix with two columns, one for each solution. You *always* get a matrix, even if it's just one row.

Examples

```
# Not compulsory to have an EXAMPLES -- you can put examples into other sections.
# Here's how to make a "don't run" example:
## Not run:
reformat.my.hard.drive()

## End(Not run)
# ... note that CRAN will still try to run "don't run" examples :/
```

`quickin` *Find kin-pairs from IBM simulation*

Description

Start with an IBM `litewt_df` from `simkin2`, which contains full ancestry info. Add a field `$SAMPY`, containing the year the animal was sampled, or NA if it wasn't (which most won't be, normally). Then call `quickin` to extract all the kin-pairs within the *samples*, ie with non-NA `$SAMPY` (although it traces ancestry thru unsampled animals, too). As the name suggests, this is pretty fast: "naive" implementations can take far longer than the IBM itself. By default, everything up to cousins is recorded, but you can reduce that (for speed), and it's easy enough to extend the code to more distant ones.

Generally, you would embed a call to `quickin` inside whatever function you are running to generate the samples; see `samp_from_catch` for an example. The call to `quickin` needs to come after you've added the `$SAMPY` field

Usage

```
quickin( inds, max_gen=2, cousins.=FALSE, thiatics.=FALSE)
```

Arguments

`inds`	From `simkin2`; a `litewt_df` (or a dataframe) with one row per individual, including character fields `ID`, `MUM`, `DAD`, and a field `SAMPY`. Only those individuals where `SAMPY` is non-NA will be "genotyped", but you need *all* animals in there so that kinship can be traced back thru unsampled ancestors.
`max_gen`	how many generations to trace back. You need at least 2 for GGPs.
`cousins.`, `thiatics.`	whether to extract those pairs. Especially for cousins, this can take some time (because there might be lots)— so, turn it off if you don't need it!

Details

Algorithm could be described here; it's not obvious.

Value

A list with names like $POP, whose elements are 2-column character matrices containing the IDs of those kin, always within the sampled part of inds. I'm pretty sure the columns are in birth-order, although the analyst wouldn't generally know that for all pairs.

Examples

```
## Not run:
reformat.my.hard.drive()

## End(Not run)
```

---

rbernoulli *Random clutch sizes*

Description

These can be passed as the rclutch_fun argument to simkin2, to generate for each female either a 0/1 clutch size (rbernoulli) or an over-dispersed, Negative Binomial clutch size (rODclutch). The other likely candidate is plain old rpois, for Poisson clutch size.

simkin2 will call its rclutch_fun argument with just two arguments— the number of living females, and the vector of expected clutch size for each such female— so if your function needs extra parameters, you have to specify them in the other_vars argument to simkin2. The environment of your clutch function will be reset so it also knows about other_vars. The classic case is rODclutch, which expects a scalar between 0 and 1, called ODclutch_par: 1 means Poisson, and smaller values mean more and more overdispersion. Like this:

```
simkin2( ...,
  rclutch_fun= rODclutch,
  other_vars= list( ODclutch_par=0.1),
  ...
)
```

Usage

```
rbernoulli( n, prob)
rODclutch( n, expct) # but NB ODclutch_par must be provided separately
```

Arguments

n	Number of living female adults; will be equal to the length() of the other argument.
prob, expct	Expected clutch size, always strictly positive. For rbernoulli, it should be between 0 and 1, though that is not checked by simkin2.

Value

An integer vector of length n.

See Also

rpois, rnbinom

Examples

```
# Not compulsory to have an EXAMPLES -- you can put examples into other sections.
# Here's how to make a "don't run" example:
## Not run:
reformat.my.hard.drive()

## End(Not run)
# ... note that CRAN will still try to run "don't run" examples :/
```

---

samp_from_catch *Sampling, lethal and non-lethal*

Description

After simkin2, you can use samp_from_catch to draw lethal samples (i.e. where COD=="HUM") and find the kin within. For non-lethal samples, or for a mix, you can use samp_from_all. In either case, but certainly for non-lethal samples, you can control additional sampling selectivity via extra_sel_fun; see specimen_sampsel_fun for an age-selective non-lethal example.

Usage

```
samp_from_catch(
  indiv, m_y,
  extra_sel_fun = function(indy, Y) rep(1, nrow(indy)),
  keep_vars= cq( SEX, A),
  extra_keep_vars= character(),
  thiatics. = FALSE,
  cousins. = FALSE
)
samp_from_all(
  indiv,
  m_y_live= 0*m_y_dead,
  m_y_dead= 0*m_y_live,
  live_sel_fun= function( indy, Y) stop( 'Surely not?'),
  dead_sel_fun= function( indy, Y) rep( 1, nrow( indy)),
  keep_vars= cq( SEX, A, COD), # makes it easy to remove some
  extra_keep_vars= character(),
  thiatics.= FALSE,
  cousins.= FALSE
)
specimen_sampsel_fun( indy, Y)
```

Arguments

indiv	from simkin2 or similar, a litewt_df (or data.frame)
m_y, m_y_dead, m_y_live	Sample sizes in each year— an offarray with proper dimseq. All you can do here is specify the total; you can't stratify directly, but you can control subsampling approximately via the sel_fun argument(s). In samp_from_all, if you only want one type of sampling, then just leave the other one at its default.
extra_sel_fun, live_sel_fun, dead_sel_fun	to sample selectively. See **Details**. In samp_from_all, NULL means don't do that type of sampling.
keep_vars, extra_keep_vars	The covariates named in the union of these two will be retained in the output; useful if e.g. you want to keep a field "LENGTH". You can use either, but the reason for splitting them is that keep_vars is the "default" and you might not want all of the default and yet you might want extra ones.
thiatics., cousins.	whether to look for these higher-order kin too. They'll slow it down, so take note.
indy, Y	the two args to extra_sel_fun. indy will be a litewt_df with info on the samplable animals in year Y.

Details

extra_sel_fun will be called once per year, with two arguments. The second, Y, is the year. The first, indy, will be a litewt_df containing samplable animals that year, which for samp_from_catch is just the catches in year Y. For samp_from_all, the samplable animals are all that were either caught (lethally) in Y or were non-calves that survived year Y: i.e., *excluding* newborns and animals that died naturally that year. Your extra_sel_fun should return a numeric vector of length nrow(indy), containing relative probabilities of sampling each catchee that year. The probs will be normalized afterwards, so they can be unscaled— hence the default. See specimen_sampsel_fun for an example.

Value

List with components Samps, POPs, FSPs, same2KPs, diff2KPs, plus MHSPs and PHSPS. Samps records individual samples, with field ID, Y (of sampling), LETHAL (type of sampling; always TRUE for samp_from_catch) and by default also SEX and A (age at sampling), plus any other specified in extra_keep_vars. If you don't want SEX or A, modify keep_vars accordingly. If you want to add extra fields, eg LQUAN for length-at-age quantile, specify them in extra_keep_vars. Apart from Samps, the other elements are all kin-pairs, i.e. two-column matrices with the ID fields of the two members. same2KPs and diff2KPs combine HSPs and GGPs; "same" and "diff" refer to whether their mtDNA haplotypes are shared (assuming infinite haplotypes...), so these can be distinguished genetically. The true maternal and paternl HSPs (excluding GGPs) are recorded in MHSPs and PHSPs respectively, but you can't expect to know those for sure genetically (unless you are WGSing/etc them); so, if your dataset is to be "realistically difficult", you can delete those post hoc.

See Also

simkin2

Examples

```
# Not compulsory to have an EXAMPLES -- you can put examples into other sections.
# Here's how to make a "don't run" example:
## Not run:
reformat.my.hard.drive()

## End(Not run)
# ... note that CRAN will still try to run "don't run" examples :/
```

---

simkin2 *CKMR simulator v2*

Description

Simulates a single population thru time, keeping ancestry. You supply a few functions that control fecundity, mortality, and perhaps other quirks. Unlike Shane Baylis's fishSim package, this wraps up the loop-over-time into a single function, with a small number of parameters— so simkin2 is convenient to use, but not as flexible as writing your own time-stepper fishSim-style.

Unlike my original simkin, this one lets you add extra animal-level info— for example, current length, or stock— and to have the mating and mortality stuff depend on that. You can do pair-bonding and so on if you are keen.

The only (?) restriction is that simkin2 kills the plus-group entirely every year; so, set a big plus-group age...

simkin2 just generates the population. After you've done so, you can collect samples using eg quickin, or wrappers like prepare_from_sim.

simkin2 requires various functions to specify how mortality and mating work. The various specimen... show examples, mainly so I can document here what arguments they need (i.e. what simkin2 will pass to them); don't expect to use the specimen... functions as-is. Importantly, their environments will be reset inside simkin2 prior to being called, so that they can refer to the variables named in public_vars and secret_vars.

Details : Within each year, breeding happens strictly before dying. New recruits (produced during breeding, and constituting the 0-group for the year) are immune to death— there's no point in generating loads of them only to kill them immediately.

simkin2 does have a notion of "fishing" (specifically, fishing mortality) as well as natural mortality; and as well as the individual-level records, it also generates an array of numbers-caught-by-sex-age-year. This is computed using Baranov's equation: ie the actual numbers dying each year are allocated to "fishing" or "natural" categories in proportion to the two mortality rates. This aggregate-catch information is available later for use in CKMR estimation models.

Usage

```
simkin2(
  N0, M_sa, #fecbar_sa
  F_fun,        # function( indlivy, y, ...): Per capita fishing mort prob
 fec_fun,    # function( parsex, indlivy, y): per cap baseline fecundity, before DDep
  densidep_fun, # function( indlivy, eqm_fec_scalor): multiplier on previous
  rclutch_fun,  # function( n, expct_clutch): n random integer clutch
    # sizes around individual expected values
  father_fun,   # function( indlivy, clutch_size, imothers, year):
    # modified subclutch sizes, each with 1 mum and 1 dad
    # eg mate_singlepat_noassoc
  YEARS,
  prob_FEMALE0,
  prelim_fun= identity,
  update_extras_fun= identity,
  secret_vars= list(),
  public_vars= list(),
  archive_int= 10, # every N years
  show_report= 0,
  RAW_only=FALSE
)
specimen_densidep_fun( indlivy, eqm_fec_scalor, y)
specimen_fec_fun( parsex, indlivy, y)
specimen_Fmort_fun(
  indlivy, y,
  a= y - indlivy$BIRTHY,
  l= indlivy$LENGTH
)
```

Arguments

N0	total numbers in first year (year 1). Age starts at 1 (so 0-groups can exist, but are not tracked in the simulation.)
M_sa	Natural mortality by sex and age. Also determines AMAX. If you supply a single vector, it's assumed to apply to both sexes. If for both sexes, it needs to be an offarray with first dimension having names "F" and "M". Constant over time. NB there's no mortality on 0-groups.
fec_fun	Should return individual *expected* clutch sizes per year, before DensDep
F_fun	Should return individual probs of death-by-fishing that year.
densidep_fun	Should return a scalar that multiplies all fecundities that year to produce expected clutch size.
rclutch_fun	Function to generate an integer random clutch size (i.e. number of maternal offspring), given its expected value. Only 2 params will be passed in, but its environment will be reset to know about other_vars so you can also give it extra info, e.g. overdispersion. Basic options are rbernoulli for 0/1, rpoisson for guess what, rODclutch.
father_fun	See mate_singlepat_noassoc for more.

YEARS — range of years to sim over. Will be contigualzed.

prob_FEMALE0 — among newborns

prelim_fun — Optional function to be called just before the year-loop starts, meant for adding columns to the census. For non-individual-specific info, use other_vars.

update_extras_fun — Optional function to be called every year before breeding and mortality, eg to update length info. Meant for user-defined columns.

public_vars, secret_vars — named lists, or environments, containing things that the user-specified functions need to know about. Will be appended to @public and @secret attributes of returnee. See DETAILS.

archive_int — How often to move dead ones into the archive, to keep the main census uncluttered.

show_report — controls annual print summaries. Default 0 means nothing; 1 or more means deaths-and-births; 2 or more means age compo; 3 or more for detailed mortality and breeding info.

RAW_only — ?return original dataset only, without fancy names or nice summaries?

y, indlivy — (densidep_fun, F_fun) y is one year; litewt_df is that year's living animals

parsex — (fec_fun) Which parental sex this fecundity is for

eqm_fec_scalor — (densidep_fun) "Eqm popn size" used in densidep_fun; basically if the "population" is bigger than that, then densidep_fun should return something <1 (but never less than 0), or something >1 if the population is larger.

a, l — (F_fun) These are examples of extra params you might put into your F_fun, if you want to be able to calculate fishing mort for specific covariate combinations. Anything in here should be equal-length vectors. simkin2 won't use them (it only passes the indlivy and y arguments to F_fun, but post hoc you might want to yourself.

Value

simkin2 normally returns a litewt_df (qv) with columns described next, and two attributes secret and public described afterwards. The non-numeric elements are mode character, not factor.

- ID — a unique string for each animal, starting with "X" and followed by a number (possibly negative; see DAD & MUM).
- SEX — "FEMALE" or "MALE"
- DAD, MUM — IDs of these two kin. For founders (individuals alive at the very start), the parents will have arbitary negative names such as "X-1". No founders will be related, regardless of reality.
- BIRTHY, DEATHY — too obvious for words. DEATHY is NA if the animal was still alive at the end.

The public attribute comprises population-level things you'd expect to know, or easily deduce, in a real CKMR setting: always C_sya (which might be all-zero). You can add other things, e.g. Amat, len_fun for growth curve info. The secret attribute comprises real summaries from the

simulation: currently N_sya, M_sa, rclutch_fun and densidep_fun. These would not be known for real data! But they can be used to check estimator performance (or to sneakily help you get close-by starting values). If RAW_only==TRUE, you don't get the @public or @secret summaries, just the litewt_df. Might save a *tiny* bit of time...

Examples

```
## Leftvoers from kinsimmer v1; needs updating
## Not run:
# Needs prior work to set up F, M, etc...
fisho <-   indiv <- simkin2(
    N0=100000,
    F_sya=F_ya,
    M_sa=M,
    fec_sa=fec_sa,
    prob_FEMALE0=0.5,
    clutch_expr=quote({
     # Allow overdispersion in "litter size". Here Var propto Mean; could be user-supplied function
        # Big "fec_gamma_shape" means less ODis
        n_clutch <- length( imothers)
        e_clutch_size <- rgamma( n_clutch, scale=fec_sa[ FEMALE, ind_AGE[ imothers]],
           shape=50) # CV = 15%
     # It's pretty certain that n_sprogs exceeds number of potential mothers, so always use the built-in rmultinom.
     # ... rather than my sparse-checking alternative in handy2--- at least for portability
        # RCT is pre-calculated via prelim_expr
        stats::rmultinom( 1, prelim$RCT, e_clutch_size)
      }),
    prelim_expr=quote({
        list( RCT= sum( indiv[ !is.na( indiv[,ID]), BIRTHY]==0))
      })
  )
fisho@public$wt_a <- wt_a # other stuff is alread there

## End(Not run)
```

---

subset_samples *Filter samples and kin-pairs*

Description

Given a *full* set of samples and kin-pairs, produced by a kin-sampler such as samp_from_catch (qv), apply a filter expression to it to get rid of some. EG you might only want to retain *female* samples. The code is simple, but the issue is that it has to be applied to all the kin-pairs as well as to the samples.

Usage

```
subset_samples( kinsamps, expr, varenv= parent.frame())
```

Arguments

kinsamps	a list with components $Samps and also various kin-pairs, such as POPs, containing 2-col char matrices of IDs (which will match into $Samps$ID). Presumably produced by a kin-sampler such as samp_from_catch.
expr	Filter expression that works "within" kinsamps$Samps, eg SEX=="FEMALE"
varenv	The filter expression might want to refer to variables that live in the calling environment, e.g. an age-of-maturity Amat. The default should work for that, but to be ultra-careful, you can either set varenv yourself manually, or set it to NULL to avoid any such lookup.

Value

Like kinsamps but presumably with fewer rows and fewer kin-pairs. Attributes of kinsamps will be retained, e.g. @public, @secret.

See Also

doc2Rd, flatdoc

Examples

```
## Not run:
reformat.my.hard.drive()

## End(Not run)
```

Index

∗ **misc**
litewt_df, 2
make_sim_names2, 3
mate_multipat_noassoc, 4
quadsolve, 5
quickin, 6
rbernoulli, 7
samp_from_catch, 8
simkin2, 10
subset_samples, 13

as.data.frame.litewt_df *(*litewt_df*)*, 2
as.litewt_df *(*litewt_df*)*, 2

extract.named, *4*

grow_litewt_df *(*litewt_df*)*, 2

litewt_df, 2, *3*, *5, 6*, *9*, *12, 13*

make_sim_names2, 3
mate_multipat_noassoc, 4
mate_singlepat_noassoc
*(*mate_multipat_noassoc*)*, 4

quadsolve, 5
quickin, 6, *10*

rbernoulli, 7
rODclutch, *11*
rODclutch *(*rbernoulli*)*, 7

samp_from_all *(*samp_from_catch*)*, 8
samp_from_catch, *6*, 8, *13*
simkin2, *4–8*, 10, *10*
specimen_densidep_fun *(*simkin2*)*, 10
specimen_fec_fun *(*simkin2*)*, 10
specimen_Fmort_fun *(*simkin2*)*, 10
specimen_sampsel_fun *(*samp_from_catch*)*,
8
subset_samples, 13