Package 'FSmod'

April 10, 2012

Type Package

Title Fraser Sockeye Modelling: Migration simulator for Fraser Sockeye and other similar critters

Version 1.0

Date 2010-07-08

Author Aaron Springford

Maintainer Sean Cox <spcox@sfu.ca>

Depends R (>= 2.10), snow, rlecuyer, VGAM, gdata

Description The FSmod framework simulates directional migration of co-migrating populations (e.g. Fraser Sockeye salmon populations) in the presence of varying environmental conditions. Effects on the movement and survival of the populations as they migrate can be both acute and cumulative thanks to the use of BOTS (BOTS are Objects for Tracking States). The model can be configured to be stochastic or deterministic. It employs discrete timesteps and spatial increments that can be arbitrarily large or small (within computing limits). The framework is driven in large part by a series of configuration files that are read in at run-time and define the characteristics of the system. Due to the computational requirements typical of such large problems, the package can make use of a computing cluster, which greatly speeds computation.

License GPL-2

LazyLoad yes

R topics documented:

`FSmod-package` *Fraser Sockeye Modelling: Migration simulator for Fraser Sockeye and other similar critters*

Description

The FSmod framework simulates directional migration of co-migrating populations (e.g. Fraser Sockeye salmon populations) in the presence of varying environmental conditions. Effects on the movement and survival of the populations as they migrate can be both acute and cumulative thanks to the use of BOTS (BOTS are Objects for Tracking States). The model can be configured to be stochastic or deterministic. It employs discrete timesteps and spatial increments that can be arbitrarily large or small (within computing limits). The framework is driven in large part by a series of configuration files that are read in at run-time and define the characteristics of the system. Due to the computational requirements typical of such large problems, the package can make use of a computing cluster, which greatly speeds computation.

Details

Package:	FSmod
Type:	Package
Version:	1.0
Date:	April 10th, 2012
License:	GPL v2
LazyLoad:	yes

`FSmodRun` is the workhorse function. Most of the inputs to the model are handled via configuration and data files that are read in at runtime. These files are described in the help file for `FSmodRun`, and can be retrieved from the installation directory using `getFSmodFiles`.

Author(s)

Aaron Springford
Contact: Dr. Sean Cox <spcox@sfu.ca>

`.doObs` *In-simulation observation and management.*

Description

These internal functions implement the in-simulation observation and management actions specified by the `obsDesignFile` and `manPlanFile` input files during an `FSmodRun` simulation. The purpose of this help file is to describe the objects that are available for use during a simulation run and to detail how the input files are used within the simulation. These functions are basically useless outside of the `FSmodRun` function and are not intended to be called directly by the user.

Usage

```
.doObs( obsDesign, t )
.doMan( manPlan )
```

Arguments

obsDesign	This is read in from the obsDesignFile by FSmodRun and passed to .doObs at the end of the simulation loop, before .doMan is called.
manPlan	This is read in from the manPlanFile by FSmodRun and passed to .doMan at the end of the simulation loop, after .doObs is called.
t	The current timestep of the FSmodRun simulation.

Details

The observation plan is carried out by making observations at predetermined timesteps. The first column in the obsDesignFile is thus a timestep or a vector of timesteps. The second column contains R code that carries out the observation. When .doObs is called, both columns are parsed and evaluated. If the timestep t matches any of the observation timesteps in the first column of obsDesignFile, then the corresponding row of obsDesignFile (second column) is evaluated, generating the observation.

The management plan, on the other hand, is carried out by a series of conditional statements. That is, when some predetermined event occurs, then a management action is taken. The first column in the manPlanFile is thus a logical statement. The second column contains R code that performs the management action. When .doMan is called, both columns are parsed and evaluated. If the statement in the first column is TRUE, then the corresponding row of manPlanFile (second column) is evaluated, performing the management action.

Of course, these two mechanisms can be used in limitless creative ways to manipulate the course of the simulation as the user sees fit. In other words, you don't need to restrict yourself to their original intended purpose. For example, one management action may be to add an additional survey if uncertainty is too high. This could be done using the manPlanFile even though it is the action is to take an observation. This could also be done by using of the obsDesignFile and including a logical-type statement compounded with the observation call in the second column.

By now you may be wondering how to go about writing useful obsDesignFile and manPlanFile files. Some simple observation schemes can be carried out using relAbun.obs, count.obs, or tagFish.obs, but it will be helpful to know what R objects exist within the FSmodRun simulation that can be called upon in the second columns of the obsDesignFile and manPlanFile files. The following list details the relevant objects available:

- *C* Vector of catches by reach for current timestep.
- *cpAllCU* Cutpoints. List of length nCU, each element is a data.frame with number of rows equal to the number of reaches, and the number of columns equal to the number of cutpoints plus the first column containing reachid.
- *cpChanged* Flag to tell when a cutpoint has been changed. Logical vector with length = nCU.
- *CU* The CU object that contains most CU-specific quantities.
  - *reach* List of vectors with reaches traversed by the CU, one vector per path.
  - *T* Timesteps. Vector of integer timesteps.
  - *N* Matrix of current numbers by reach (rows) and path (cols)
  - *bots* Matrix of BOTS for this CU. Each row is one BOTS. First column is reach number, second column is path number, third column is the first state variable (called state1), additional columns for extra states (called state2, state3, and so on).

  - *spawners* Number of spawners that have escaped so far. A numeric vector with length equal to the number of migration paths.
  - *cumulPars* List of cumulative effects parameters. See FSmodRun.
  - *enterProps* Proportion of critters entering via each path (a.k.a. diversion). Vector with length equal to the number of paths.
  - *sc* Scale parameter for the logistic/multinomial movement distribution. See FSmodRun.
  - *CU* The CU number.
  - *CUname* The CU name. Example: L-01-01 for default sockeye.
  - *CUcommon* The common name for the CU. Example: Kamloops.ES.CU for default sockeye.
  - *pNoBOTS* The probability of survival if there are no BOTS in a given reach.
  - *allReaches* A vector of all of the reach numbers that are touched by the CU.
  - *nodes* A `data.frame` with columns `NODES` and `REACHID` containing all of the nodes that the CU migrates past.
  - *cpAll* A `data.frame` with cutpoints for the multinomial movement distribution. Each row is a reach that the CU travels through. The first column contains the reach id numbers; the subsequent columns contain the cutpoints.
  - *cnodes* Numeric vector with the location(s) of the counting nodes.
  - *counts* Matrix of passage past each counting node. One column per `cnode`, one row per timestep.
  - *tnodes* Numeric vector with the location(s) of the tag listening nodes.
- *cuN* List of numbers returning, with length = `nCU`. The names of the list items correspond to the names of the CUs.
- *CUnames* Character vector with the names of the CUs.
- *dens* Numeric matrix with density function values for returns to the map. Names of rows correspond to `CUnames`. Names of columns correspond to timesteps e.g. `T.1, T.2, ...`.
- *envData* Big `data.frame` for environmental data with entries
  - `T:` Timestep
  - `reach:` Reach number
  - Environment variables like `temp`, `flow`
- *harv* Harvest plan. A `list` object with
  - `harvM` Numeric harvest rates with number of rows equal to the number of reaches, number of columns equal to the number of timesteps.
  - `fisheries` Named list with vectors of reaches covered by each fishery.
  - `openings` Named list with timesteps at which each fishery is open.
  - `hrates` Named list with harvest rates, one per timestep the fishery is open.
- *m* Current natural deaths vector, by reach.
- *manPlan* Read-in `data.frame` of management plan.
- *mortM* Matrix of natural mortality due to non-cumulative effects with rows = number of reaches, columns = number of timesteps.
- *move.df* Large `data.frame` with
  - `T:` timestep
  - `reach:` reach number
  - `CU:` CU number
  - `flow:` flow

  - `temp`: temperature
  - `...` other environmental variables
  - `y`: movement value relative to cutpoints. See `FSmodRun` for details.
- *moveRates* `data.frame` with
  - `CU`: The CU name
  - `reach`: The reach number
  - `y`: Movement rate in the *original units* from the input file.
- *nCU* The number of CUs.
- *nReach* The total number of reaches.
- *obs* This object is intended to hold observations and is output at the end of `FSmodRun`.
- *out* Large output matrix. May not contain all of the output information if it is being dumped periodically to an output file! See `FSmodRun` for details.
- *reachDefs* `data.frame` as read in from file.
- *reaches* `list` with vectors of reaches, named for the reach intevals. e.g. `Johnstone Strait: Vancouver` for default sockeye.
- *reachid* Vector of all integer reach IDs.
- *reachNodes* `data.frame` as read in from file.
- *retd* `data.frame` with entry timing. Contains
  - `CU` The CU name
  - `loc` Factor variable with the timing location name. See `FSmodRun`, `retpars` argument.
  - `T.1`
  - `T.2`
  - `...` The distribution function. Should sum to 1.
- *retSplit* `list` of length = `nCU`, containing sublists of numbers entering the map (one vector per path). So, for example if you want the number entering the map for CU = 1, path = 2, and timestep = 123, you want `retSplit[[1]][[2]][123]`.
- *retSplitBOT* Same as `retSplit` but for BOTS.
- *t* The current timestep.
- *y.curr* This timestep's subset of movement effects. This is basically `move.df[move.df$T==t,]`.

Value

None.

Author(s)

Aaron Springford

count.obs *A function for getting fish passage estimates*

Description

This function gets fish passage estimates, applying systematic and random errors to the counts. This can be used in order to simulate data coming from fish counting fences, for example. This function can be used in an observation file (defaulting to "obsDesign.csv") to generate data as FSmodRun progresses.

Usage

```
count.obs(CU, rb.func = NULL, cv = 0, ...)
```

Arguments

CU	This should always be set equal to "CU" to work properly
rb.func	A function to apply bias as a function of true counts (should take true counts as first argument, and return biased counts). Additional arguments can be passed to this function via the ... mechanism.
cv	A coefficient of variation for lognormal observation errors (recycled to the number of counting fences).
...	Additional arguments passed to rb.func

Details

- First, true numbers passing each counting location are extracted from the CU object.
- Next, bias is applied using rb.func.
- Finally, additional lognormal random noise is applied (using rlnorm)

Value

A matrix containing observed counts, with a column for each counting location and a row for each timestep. Fractions of one fish are possible.

Author(s)

Aaron Springford

See Also

FSmodRun

Examples

```
# Unfortunately, FSmod's reliance on configuration and data files precludes
# including examples.  However, once all of the configuration files have been
# correctly copied from the FSmodfiles directory (found in the local
# installation directory, after installing FSmod) to the user's working
# directory, the following code should work:
#
```

```
## Not run:
res <- FSmodRun()
# Get true counts
count.obs( res$CU )
# Get counts with error
count.obs( res$CU, cv = 0.05 )
# Get counts with a negative bias
count.obs( res$CU, rb.func = function(x) 0.95*x, cv = 0.05 )

## End(Not run)

# Example line in obsDesign.csv file
# ---------------------------------
# t, Expression
# 150, obs$mycounts150 <- count.obs( CU )
```

`dev.off.acro` *Close a .pdf device an display the file*

Description

This is a wrapper for `dev.off` that will close the current device if it is a pdf, and if the platform is Windows show it in the default Windows pdf viewer. This is meant to be a companion function to `pdfacro`.

Usage

```
dev.off.acro( filename )
```

Arguments

`filename`	The file name associated with the current pdf device.

Author(s)

Aaron Springford

`FSmodRun` *Migration simulator for Fraser Sockeye and other similar critters*

Description

This is the workhorse function that actually does the simulation. The simulator is designed to be flexible enough to accomodate the complexities of the Fraser Sockeye system as we currently understand them, while leaving open the option to add/remove complexity as one might see fit. Also important is speediness of calculation, which has been carefully considered: movement is handled by C routines, there is minimal overhead computation, and much of the code can be computed on a "localhost" cluster (i.e. to take advantage of modern multi-processor computer architectures). The simulation is controlled in part by arguments to this function, and in part by data from files read in at runtime. The use of files as inputs should make future gui-based controls easier to implement using any number of tools. A key feature of this simulation framework is the ability to handle large

numbers of migrating fish (critters) while keeping track of cumulative exposures along the migration path. The keeping track of exposures is done using BOTS: BOTS are Objects for Tracking States. In essence, fish (critter) locations are handled by location whereas BOTS are tracked individually. Because both BOTS and fish (critters) obey the same rules, the histories of BOTS approximate the histories of fish (critters) at the same location and time in the simulation. Thus, BOTS histories are used when applying cumulative effects to non-BOTS fish (critters).

Usage

```
FSmodRun <- function( cl = NULL, nProcs = 1, plotTF = TRUE
                    , plotTFinteract = TRUE
                    , plotPars = list( plotBOTS = FALSE, whichCU = NULL
                                     , nPerPage = 2, pdfname = NULL
                                     , itype = "barplot" )
                    , buildX = TRUE, yRead = TRUE, loadY = FALSE
                    , mvr.file = "move/moveRates02.txt"
                    , N.file = "CU/RunSize.csv"
                    , CUdefs.file = "CU/CUdefns.csv"
                    , mvdefs.file = "move/moveDefns02.csv"
                    , year = 2002, nT = 360, nBots = 1000, nStates = 1
                    , reachNodes.file = "basemap/ReachNodes.csv"
                    , reachDefs.file = "basemap/ReachDefns.csv"
                    , retdist.file = NULL
                    , retpars = list(
                        func = function( x, p ) dnorm( x, p[1], p[2] )
                      , file = "CU/2002ArrT.csv"
                      , CU.col = 2, par.col = c(5,3)
                      , par.trans = function(x) x*c(1,1/6)
                      , loc = "endpoint" )
                    , adj = list( c(0,0) )
                    , cnodes = c( "Mission", "Qualark" )
                    , tnodes = c( "Hells Gate", "Thompson", "Timber's House" )
                    , envData.dirs = c( "environment/DFO_TempFiles"
                                      , "environment/WSC_DisFiles" )
                    , envData.files = c( "environment/tempDefns.csv"
                                       , "environment/flowDefns.csv" )
                    , envData.varnames = c( "temp", "flow" )
                    , fishLocs = "fisheries/FishDefns.csv"
                    , fishOpens = "fisheries/FishOpens.csv"
                    , manPlanFile = "man/mP.csv"
                    , bneck = NULL
                    , cpars = list( list( normMu = 600, normSD = 30 ) )
                    , enterProps = list( c( 0.3, 0.7 ) )
                    , sc = 0.5, cpVec = 1:8, cpAll = NULL, cpVals = -1:8
                    , pNoBOTS = 0, natM = 0
                    , stoch.nat = TRUE, stoch.fish = TRUE
                    , obsDesignFile = "obs/obsDesign.csv"
                    , out.file = "outdf.txt"
                    , speedRatio = 20, maxrows = 1e6, YAB = FALSE
                    , seed = 999
                    )
```

Arguments

cl	A snow cluster. This is optional.
nProcs	If cl is NULL, FSmodRun can still make use of a local multi-processor. By setting nProcs > 1, a SNOW cluster is created via a call to makeSOCKcluster, i.e. makeSOCKcluster(rep.int("localhost", nProcs)). nProcs is ignored if a SNOW cluster is supplied via cl.
plotTF	Whether or not to produce a plot showing the migration, catch, and natural mortality resulting from the simulation. If TRUE, a call to plotFSmodOut is made at the end of the simulation.
plotTFinteract	If TRUE, a call to plotFSmodOut.interactive is made at the end of the simulation.
plotPars	A named list of parameters to pass to plotFSmodOut in the event that plotTF is TRUE. If plotTFinteract is TRUE, then the itype element is passed to the type argument of plotTFinteract.
buildX	When using a linear model formulation for movement rates, this controls whether or not X should be built from scratch, or loaded from the file "movedf.rsv". Also, controls whether environmental data should be built from scratch or loaded from the file "envdata.rsv". Currently, we don't use a linear model for movement rates.
yRead	If TRUE, suppresses calculating X and builds movement rates directly from values supplied in mvdefs.file. Useful if not interested in using the linear model movement formulation. NOTE: We have abandoned development of the linear model movement formulation at least for the time being.
loadY	If we are building movement rates directly from mvdefs.file, this process can be very time consuming. If loadY is TRUE, we will skip the building and read in the movement rate objects directly from the file "movedfY.rsv".
mvr.file	File with movement rate data.
N.file	File with numbers returning to the first location(s) on the map.
CUdefs.file	File with CU definitions.
mvdefs.file	File with movement rate definitions (for mapping movement rates to locations).
year	The year of the simulation (used when referencing some input data.frames).
nT	Length of season (In the sockeye example, 12 hr timestep, beginning June 1st)
nBots	The number of BOTS that each CU gets (recycled)
nStates	The number of state variables that are tracked by BOTS
reachNodes.file	File with reach (location) node definitions
reachDefs.file	File with reach (location) interval definitions
retdist.file	File with run timing distribution. The first column must contain the CU code, the second column the named location (from reachNodes.file) at which the timing distribution is measured, and the 3rd through (2+nT) columns the timing distribution (which need not be normalized). If one does not want to specify a file, then retdist.file should be set to NULL. In this case, the run timings are approximated using the retpars argument and the movement rates along the map. When this occurs, a file called "CU/retdist-'seed'.csv", where 'seed' is the seed argument to this function is created containing the retdist.file corresponding to the retpars specification, and this file is then used as input to the simulation.

`retpars` A list that controls the run timing distribution in case it is not specified with `retdist.file`. The elements of the list are:

- `func` is a distribution function that takes `x` (the timestep) and `p` (a vector of parameters).
- `file` is a file that contains the parameters to pass to `func`.
- `CU.col` is the column in `file` that contains the CU code.
- `par.col` are the column(s) in `file` that contain the parameters to be passed to `func`.
- `par.trans` is a function that takes the vector of raw parameters specified by `par.col` and transforms them before they are passed to `func`.
- `loc` are the named locations (from `reachNodes.file`) at which the timing distribution is specified. This argument is recycled. Setting `loc = "endpoint"` specifies the end of each CU's migration path (i.e. spawning escapements).

`adj` A vector list with adjustments to be made to the travel time when building entry timing distributions, for example to account for bottlenecks when the entry timing is being estimated from movement rates. Positive values are for increasing the travel time (i.e. making entry times earlier).

`cnodes` A character vector with the counting nodes (must be in `reachNodes.file`)

`tnodes` A character vector with the tag receiver nodes (must be in `reachNodes.file`)

`envData.dirs` The path(s) to the environmental data files

`envData.files` The environmental data files

`envData.varnames`
The variable names corresponding to the environmental data

`fishLocs` A file defining the locations of the fisheries

`fishOpens` A file defining the times when the fisheries are open, and their harvest rates

`manPlanFile` A file describing the in-season management plan. The first column contains a conditional statement to be evaluated at each timestep. The second column contains an expression to be evaluated if the conditional statement is `TRUE`. The expressions are evaluated in the order that they appear in the `manPlanFile`. Note that the conditional statement can be set to `TRUE` if an expression is meant to be evaluated at each timestep. See `Management` for more information.

`bneck` An optional data.frame with the following columns:

- `CU` is the CU for which the bottleneck applies. If `NA`, then the bottleneck is applied to all CU's.
- `t` is the timestep for which the bottleneck applies. If `NA`, then the bottleneck is applied to all timesteps.
- `loc` is the named locations at which the bottleneck is applied. These names must be in the `reachNodes.file` and the `reachDefs.file`.
- `pen` is the bottleneck penalty. The larger the value, the higher the penalty. See Details.
- `offset` is the bottleneck offset. This the the number of reaches away from the bottleneck that will be affected by the bottleneck. For example, setting the offset to 1 means that a critter that is 1 reach from the bottleneck will not be affected - only fish that are further than 1 reach away will be affected.

`cpars` Vector list of cumulative effects parameters, recycled if necessary

`enterProps` Vector list of proportion of individuals entering the map by path (a.k.a. diversion rates for Fraser sockeye). Each element of the the vector list is itself a vector of length equal to the number of starting branches on the map. These subvectors will be normalized if they don't sum to one. In addition, the vector list will be recycled if necessary to be the same length as CU.

`sc` Vector of jump "scale" parameters, recycled if necessary

`cpVec` A vector of cutpoints to use in movement calculations (this same vector will be used for all reaches/locations subject to potential bottleneck adjustments). These cutpoints are overridden by a non-NULL `cpAll` argument. See Details.

`cpAll` If NULL, cutpoints are obtained from cpVec and any bottleneck penalties. This argument allows the user to define different cutpoints by reach/location. If non-NULL, `cpAll` needs to be a data.frame whose first column is named "reach" (and contains all of the reaches/locations on the map), and whose subsequent columns are named "cp.2", "cp.3", and so on (and contain the cutpoints by reach/location).

`cpVals` A vector of the movements associated with each cutpoint interval. See Details.

`pNoBOTS` Vector of probabilities of death for fish in reaches without BOTS (recycled). This controls what happens when natural mortality is applied to fish that are far away from the nearest BOTS, meaning that cumulative exposures are not available with which to calculate the probability of death.

`natM` Underlying, constant natural mortality rate.

`stoch.nat` If TRUE, natural mortality is stochastic (binomial draw)

`stoch.fish` If TRUE, catches are stochastic (binomial draw)

`obsDesignFile` A file containing an observation design protocol. This is a .csv file whose first column contains the timestep at which the observation is to take place, and a second column with code that will implement the observation. See `relAbun.obs`, `count.obs`, `tagFish.obs` for examples. Also see `Observation` for more details.

`out.file` The simulation builds a matrix as it progresses with columns:

- *t* for timestep
- *CU* for conservation unit (population)
- *reachid* for reach ID number (location)
- *N* for numbers present
- *m* for numbers dying of natural causes
- *C* for numbers being caught in a fishery

This matrix might be very large and uncomfortable to handle. When this matrix has more rows than `maxrows`, the matrix will be dumped in `maxrows` increments to the file `out.file`.

`speedRatio` The ratio of the movement rate units (in the input files) to the units in the simulation model (i.e. reaches/timestep). For example, if the movement rates in the input files are in km/day, and the movement rates in the simulation are in 10km/12hr, then speedRatio should be 20.

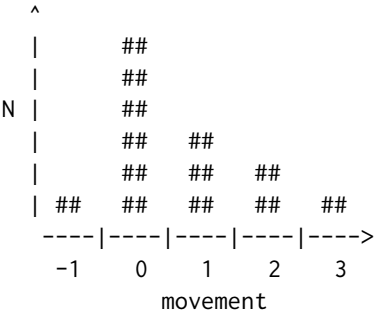
`maxrows` The maximum number of rows that the simulation output matrix is allowed to have.

`YAB` Yet Another Boxcar: When set to TRUE, movement rate variation is set to zero, and corrections are made to attempt to make average movement rates match over the length of the migration path (i.e. to avoid bias caused by discretization in space and time). NOT YET IMPLEMENTED.

seed — Random number seed. When using nProcs > 1, this seed in incremented by 1 and used to seed the cluster's random number generator as well.

Details

The movement of fish is handled by reach/location. For each reach, it is a something like a multinomial. For example, suppose there are some number of fish in a particular reach/location. The distribution of those fish at the next timestep, relative to their current location, might be:



This distribution is defined by an average movement response y, which might be calculated from a linear model of effects b, like so:

$$y = Xb$$

Then, the probability that a fish ends up at any one of the series of possible future locations m is (in our example):

$$\begin{aligned} p(m \geq 0) &= logit^{-1}(y) \\ p(m \geq 1) &= logit^{-1}(y - c_2) \\ p(m \geq 2) &= logit^{-1}(y - c_3) \\ p(m \geq 3) &= logit^{-1}(y - c_4) \end{aligned}$$

where the cutpoint parameters are

$$0 = c_1 < c_2 < c_3 < c_4$$

.

The procedure is:

1. Determine the value of the latent variable z, where $z_i = y_i + \epsilon_i$, and $\epsilon_i \sim \text{logistic}(\texttt{sc})$, sc being an argument to this function.
2. Then

$$\begin{array}{rrll} m_i = & \text{-1} & \text{if} & z_i < 0 \\ & 0 & \text{if} & 0 < z_i < c_2 \\ & 1 & \text{if} & c_2 < z_i < c_3 \\ & 2 & \text{if} & c_3 < z_i < c_4 \\ & 3 & \text{if} & z_i > c_4 \end{array}$$

An identical procedure is used to move each BOTS and tagged fish individually.

Bottlenecks can be inserted on the migration map to limit the ability of fish (critters) to bypass an obstacle such as a fish ladder too quickly. The cutpoints are adjusted upwards so that it is less likely that a fish (critter) will be able to pass a bottleneck when approaching from a larger distance.

The bottleneck mechanism works by increasing the value of the cutpoints in the reaches (locations) approaching the bottleneck location. The cutpoints are increased so that movement to the reach (location) at the bottleneck is unaffected, but movement past the bottleneck from any reach before it is equally unlikely (given that movement past the bottleneck is possible from a given reach). The probability of movement past the bottleneck is further penalized by adding `bneckPen` to all of the affected cutpoints.

As an example, consider what happens if we are performing a simulation with the default parameter values, and there is a bottleneck at reach (location) 61. The relevant portion of `cpAll` is

reach	cp.2	cp.3	cp.4	cp.5	cp.6	cp.7	cp.8	cp.9
53	1	2	3	4	5	6	7	8
54	1	2	3	4	5	6	7	8
55	1	2	3	4	5	6	7	8
56	1	2	3	4	5	6	7	8
57	1	2	3	4	5	6	7	8
58	1	2	3	4	5	6	7	8
59	1	2	3	4	5	6	7	8
60	1	2	3	4	5	6	7	8
61	1	2	3	4	5	6	7	8

After applying the bottleneck with `bneckPen` = 1, `cpAll` is

reach	cp.2	cp.3	cp.4	cp.5	cp.6	cp.7	cp.8	cp.9
53	1	2	3	4	5	6	7	8
54	1	2	3	4	5	6	7	9
55	1	2	3	4	5	6	9	10
56	1	2	3	4	5	9	10	11
57	1	2	3	4	9	10	11	12
58	1	2	3	9	10	11	12	13
59	1	2	9	10	11	12	13	14
60	1	9	10	11	12	13	14	15
61	1	2	3	4	5	6	7	8

Recall that the default movement vector is -1:8. Using this movement vector in combination with the new cutpoint matrix, movement to reach 61 is unaffected, but it is very unlikely that a fish downstream of reach 61 will move *past* reach 61. This is the intent of the bottleneck mechanism. The higher the `bneckPen`, the less likely movement past reach 61 will be. For example, with `bneckPen` = 4, `cpAll` becomes

reach	cp.2	cp.3	cp.4	cp.5	cp.6	cp.7	cp.8	cp.9
53	1	2	3	4	5	6	7	8
54	1	2	3	4	5	6	7	12
55	1	2	3	4	5	6	12	13
56	1	2	3	4	5	12	13	14
57	1	2	3	4	12	13	14	15
58	1	2	3	12	13	14	15	16
59	1	2	12	13	14	15	16	17
60	1	12	13	14	15	16	17	18
61	1	2	3	4	5	6	7	8

Of course, a user can specify more complex schemes by manually specifying all of the cutpoints using the cpAll argument.

Value

A list with:

out.df	A matrix with columns t (timestep), CU (conservation unit/population), reachid (reach ID number/location), N (numbers present), m (numbers dying of natural causes), and C (numbers caught in a fishery)
CU	The CU object at the end of the simulation
cuN	A named list with the numbers returning
harv	A list with harvM (the harvest rate matrix), fisheries (a list object mapping fisheries to reaches/locations), openings (a list object with the timesteps when fisheries were open), and hrates (a list object with harvest rates)
moveRates	A data.frame with CU, reach, and y (the mean movement rate response - see Details)
obs	A list object that can be used to return observations via the obsDesignFile mechanism

Author(s)

Aaron Springford

See Also

For observation functions (and examples of how to use them), see count.obs, relAbun.obs, and tagFish.obs.

Examples

```
# Unfortunately, FSmod's reliance on configuration and data files precludes
# including examples.  However, once all of the configuration files have been
# correctly copied from the FSmodfiles directory (found in the local
# installation directory, after installing FSmod) to the user's working
# directory, the following code should work:

## Not run:
# Run using three snow processes
sim <- FSmodRun( nProcs = 3 )

## End(Not run)
```

`getFSmodFiles` *Retrieve configuration and data files from the FSmod installation*

Description

Because FSmodRun depends on a number of configuration and data files that need to be editable by the user, these files are included in a subdirectory of the FSmod installation called "FSmodFiles".

This helper function retrieves the contents of the FSmodFiles directory and copies them to the current working directory (by default).

Usage

```
getFSmodFiles(wd = getwd(), libPath = .libPaths(), overwrite = TRUE)
```

Arguments

`wd`	The working directory where the files should be copied. By default, this is the current working directory.
`libPath`	Vector of paths to package installation directories. By default, those supplied by `.libPaths`.
`overwrite`	Whether or not to overwrite files of the same name in the current working directory. Defaults to TRUE.

Value

From `file.copy`, a logical vector indicating which operation succeeded for each of the files attempted. Using a missing value for a file or path name will always be regarded as a failure.

Note

It is a good idea to call this function first, to get things rolling. FSmod is best learned using these files as templates.

Author(s)

Aaron Springford

`Next` *Extract numbers from a CU object*

Description

A function that extracts numbers by reach and CU

Usage

```
Next(x, r)
```

Arguments

x	The CU object
r	The reach ID number

Value

If there are multiple migration paths (i.e. multiple ways to get to the reach "r"), then a vector of the numbers in each path occurring at reach "r". If there is only one migration path, then a scalar.

Note

This really isn't very useful unless you stop the simulation partway or something. Mostly, it is used within other functions in the FSmod package.

Author(s)

Aaron Springford

---

pdfacro *pdf device wrapper that kills Adobe Acrobat if necessary*

Description

This is a wrapper for pdf that will attempt to kill a file with the same name as file that is open in Adobe Acrobat before opening the pdf device, if the system platform is Windows. This hopes to circumvent errors associated with Adobe's file locking behaviour.

Usage

```
pdfacro( file = ifelse(onefile, "Rplots.pdf", "Rplot%03d.pdf"), ... )
```

Arguments

file	The file name to be passed to pdf.
...	Further arguments passed to pdf.

Author(s)

Aaron Springford

plotFSmodOut *Plot FSmodRun simulation results*

Description

Plot FSmodRun simulation results, including numbers, catch, and natural mortality by reach (location) and timestep.

Usage

```
plotFSmodOut( out.df = NULL, fn = NULL, CU, cuN, plotBOTS = FALSE
             , whichCU = NULL, nPerPage = 4, pdfname = NULL )
```

Arguments

out.df	The out.df object returned by FSmodRun. If this is not available (e.g. if nrow(out.df) > maxrows) then a file output by FSmodRun can be specified by providing the file name as fn.
fn	File name to read in out.df output from FSmodRun.
CU	CU object returned by FSmodRun.
cuN	cuN object returned by FSmodRun.
plotBOTS	Whether or not to plot individual BOTS trajectories, if available (these would have been stored in the CU object). Currently, not implemented.
whichCU	A logical vector with the same length as CU specifying which CUs to plot. Alternatively, an integer vector specifying the index of the CUs to plot.
nPerPage	The number of CUs to plot per page.
pdfname	An optional filename for .pdf output. If specified, plotFSmodOut will call pdfacro and dev.off.acro when producing .pdf output.

Note

This function is called at the end of an FSmodRun simulation run when plotTF = TRUE.

Author(s)

Aaron Springford

`plotFSmodOut.interactive` *Plot FSmodRun simulation results interactively*

Description

Plot FSmodRun simulation results interactively by examining time and location profiles.

Usage

```
plotFSmodOut.interactive( out.df = NULL, fn = NULL, CU, cuN
                          , whichCU = NULL, intNodes = "Mission" )
```

Arguments

`out.df`	The out.df object returned by `FSmodRun`. If this is not available (e.g. if `nrow( out.df ) > maxrows`) then a file output by `FSmodRun` can be specified by providing the file name as `fn`.
`fn`	File name to read in out.df output from `FSmodRun`.
`CU`	CU object returned by `FSmodRun`.
`cuN`	cuN object returned by `FSmodRun`.
`whichCU`	A logical vector with the same length as `CU` specifying which CU to plot. Alternatively, an integer specifying the index of the CU to plot. If NULL, the user is presented with a list of options.
`intNodes`	A vector of node names defining areas for which harvest rates and en-route mortality rates are to be calculated. Defaults to "Mission".

Note

This function is called at the end of an `FSmodRun` simulation run when plotTFinteract = TRUE.

Author(s)

Aaron Springford

`relAbun.obs` *Obtain a relative abundance estimate.*

Description

This function gets a relative abundance estimate at a particular location (from a test fishery or counting wheel, for example). It also estimates population proportions at that particular location. This function can be used in an observation file (defaulting to "obsDesign.csv") to generate data as `FSmodRun` progresses.

Usage

```
relAbun.obs(CU, reach, q, cvq, skill = diag(1, nrow = length(CU)
                                            , ncol = length(CU)))
```

Arguments

CU	The CU object containing true abundances
reach	The reach (location)
q	The catchability (proportion sampled) by CU (recycled to length(CU))
cvq	The catchability error by CU (recycled to length(CU)). q's are drawn from a beta() distribution
skill	A correlation matrix specifying stock identification rates (that is, the correlation between realized identifications and "true" CU. This obviously needs to be an nxn matrix where n = length(CU). Defaults to no errors in identification.

Details

- First, q is drawn from a beta distribution, whose parameters are calculated from cvq and q as follows:

$$\sigma_q = cv_q q$$

$$\alpha = q\left(\frac{q(1-q)}{\sigma_q^2} - 1\right)$$

$$\beta = (1-q)\left(\frac{q(1-q)}{\sigma_q^2} - 1\right)$$

- Next, numbers of fish are sampled ("caught") from each CU using a binomial (rbinom) draw.
- Finally, observed proportions are drawn from a multinomial distribution (rmultinom) using the true proportions caught in the previous step.

Value

A list with components

N	Total number of fish at reach (location)
props	Proportions belonging to each CU

Author(s)

Aaron Springford

Examples

```
# Example line in obsDesign.csv file:
# -----------------------------------
# t, Expression
# 150, obs$relAbun150 <- relAbun.obs( CU, reach = 100, q = 1, cvq = 0 )
```

`tagFish.obs` *Tag fish during the course of an FSmodRun simulation*

Description

This function applies tags to fish, effectively removing them from a CU's population pool and placing them in a $tags object. They are then kept track of individually for the rest of the simulation, and are counted at tagging nodes (see FSmodRun). This function is designed to be used in an observation file (defaulting to "obsDesign.csv") to tag fish as FSmodRun progresses.

Usage

```
tagFish.obs(CU, tags, reach, stoch = TRUE)
```

Arguments

CU	The CU object.
tags	The number of tags to be deployed. It is assumed that all tags are deployed, subject to the constraint that there are enough fish at the location (reach) to receive them. If stoch is FALSE, then it could happen that there aren't *exactly* the specified number of tags deployed, due to rounding.
reach	The location at which tagging takes place.
stoch	Whether or not the fish caught for tagging are caught stochastically. If FALSE, fish are caught and tagged exactly in proportion to abundance. If TRUE, the actual number of tags deployed for each CU is drawn using rmultinom.

Details

Tags are kept track of using the same mechanisms as BOTS, except of course that tags are not repopulated if killed. A simplifying assumption is made that the absolute number of tags on a CU is much less than the CU abundance, so that tagged fish are not counted in catch calculations (i.e. they are treated the same way that BOTS are for catch calculations).

Value

Returns the CU object, including a $tags object (which is appended to if it already existed).

Author(s)

Aaron Springford

See Also

FSmodRun

Examples

```
# Example line in obsDesign.csv file
# ----------------------------------
# t, Expression
# 150, CU <- tagFish.obs( CU, tags = 500, reach = 19, stoch = TRUE )
```

Index