Package 'SIMplyBee'

September 20, 2024

Type Package

Title 'AlphaSimR' Extension for Simulating Honeybee Populations and Breeding Programmes

Version 0.4.1

Description An extension of the 'AlphaSimR' package (<https://cran.r-project.org/package=AlphaSimR>) for stochastic simulations of honeybee populations and breeding programmes. 'SIMplyBee' enables simulation of individual bees that form a colony, which includes a queen, fathers (drones the queen mated with), virgin queens, workers, and drones. Multiple colony can be merged into a population of colonies, such as an apiary or a whole country of colonies. Functions enable operations on castes, colony, or colonies, to ease 'R' scripting of whole populations. All 'AlphaSimR' functionality with respect to genomes and genetic and phenotype values is available and further extended for honeybees, including haplo-diploidy, complementary sex determiner locus, colony events (swarming, supersedure, etc.), and colony phenotype values.

URL https://github.com/HighlanderLab/SIMplyBee

Encoding UTF-8

Imports methods, R6, stats, utils, extraDistr (>= 1.9.1), RANN, Rcpp (>= 0.12.7)

Depends R (>= 3.3.0), AlphaSimR (>= 1.5.3)

LinkingTo Rcpp, RcppArmadillo (>= 0.7.500.0.0), BH

RoxygenNote 7.3.2

Suggests rmarkdown, knitr, ggplot2, testthat (>= 3.0.0), Matrix

Config/testthat/edition 3

Config/Needs/website tidyverse/tidytemplate

VignetteBuilder knitr

NeedsCompilation yes

Author Jana Obšteter [aut, cre] (<https://orcid.org/0000-0003-1511-3916>),
Laura Strachan [aut] (<https://orcid.org/0000-0002-2569-0250>),
Jernej Bubnič [aut] (<https://orcid.org/0000-0003-1362-3736>),
Gregor Gorjanc [aut] (<https://orcid.org/0000-0001-8008-2787>)

Maintainer Jana Obšteter `<obsteter.jana@gmail.com>`

Repository CRAN

Date/Publication 2024-09-20 12:30:02 UTC

Contents

`addCastePop` *Add caste individuals to the colony*

Description

Level 2 function that adds (raises) the specified number of a specific caste individuals to a Colony or MultiColony object by producing offspring from a mated queen. If there are already some individuals present in the caste, new and present individuals are combined.

Usage

```
addCastePop(
  x,
  caste = NULL,
  nInd = NULL,
  new = FALSE,
  exact = FALSE,
  year = NULL,
  simParamBee = NULL,
  ...
)

addWorkers(x, nInd = NULL, new = FALSE, exact = FALSE, simParamBee = NULL, ...)

addDrones(x, nInd = NULL, new = FALSE, simParamBee = NULL, ...)

addVirginQueens(
  x,
  nInd = NULL,
  new = FALSE,
  year = NULL,
  simParamBee = NULL,
  ...
)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`caste`	character, "workers", "drones", or "virginQueens"
`nInd`	numeric or function, number of workers to be added, but see `new`; if `NULL` then `SimParamBee$nWorkers` is used. If input is `MultiColony-class`, the input could also be a vector of the same length as the number of colonies. If a single value is provided, the same value will be used for all the colonies.
`new`	logical, should the number of individuals be added to the caste population anew or should we only top-up the existing number of individuals to `nInd`
`exact`	logical, only relevant when adding workers - if the csd locus is turned on and exact is `TRUE`, we add the exact specified number of viable workers (heterozygous at the csd locus)
`year`	numeric, only relevant when adding virgin queens - year of birth for virgin queens
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	additional arguments passed to `nInd` when this argument is a function

Details

This function increases queen's `nWorkers` and `nHomBrood` counters.

Value

`Colony-class` or `MultiColony-class` with workers added

Functions

- `addWorkers()`: Add workers to a colony
- `addDrones()`: Add drones to a colony
- `addVirginQueens()`: Add virgin queens to a colony

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 100)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 5, nDrones = nFathersPoisson)

# Create and cross Colony and MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[4:5], n = 2)
apiary <- cross(apiary, drones = droneGroups[3:4])
```

```
#Here we show an example for workers, but same holds for drones and virgin queens!
# Add workers
addCastePop(colony, caste = "workers", nInd = 20)
# Or use a alias function
addWorkers(colony, nInd = 20)
# Same aliases exist for drones and virgin queens!

# If nInd is NULL, the functions uses the default in SP$nWorkers
# We can change this default
SP$nWorkers <- 15
nWorkers(addWorkers(colony))
# nVirginQueens/nWorkers/nDrones will NOT vary between function calls when a constant is used

# Specify a function that will give a number
nWorkers(addWorkers(colony, nInd = nWorkersPoisson))
nWorkers(addWorkers(colony, nInd = nWorkersPoisson))
# nVirginQueens/nWorkers/nDrones will vary between function calls when a function is used

# Store a function or a value in the SP object
SP$nWorkers <- nWorkersPoisson
(addWorkers(colony))
# nVirginQueens/nWorkers/nDrones will vary between function calls when a function is used

# Queen's counters
getMisc(getQueen(addWorkers(colony)))

# Add individuals to a MultiColony object
apiary <- addWorkers(apiary)
# Add different number of workers to colonies
nWorkers(addWorkers(apiary, nInd = c(50, 100)))
```

`buildUp` *Build up Colony or MultiColony object by adding (raising) workers and drones*

Description

Level 2 function that builds up a Colony or MultiColony object by adding (raising) workers and drones usually in spring or after events such as split or swarming.

Usage

```
buildUp(
  x,
  nWorkers = NULL,
  nDrones = NULL,
  new = TRUE,
  exact = FALSE,
  resetEvents = FALSE,
```

```
  simParamBee = NULL,
  ...
)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`nWorkers`	numeric or function, number of worker to add to the colony, but see `new`; if `NULL` then `SimParamBee$nWorkers` is used. If input is `MultiColony-class`, the input could also be a vector of the same length as the number of colonies. If a single value is provided, the same value will be applied to all the colonies.
`nDrones`	numeric or function, number of drones to add to the colony, but see `new`; if `NULL` then `SimParamBee$nDrones` is used. If input is `MultiColony-class`, the input could also be a vector of the same length as the number of colonies. If a single value is provided, the same value will be applied to all the colonies.
`new`	logical, should the number of workers and drones be added anew or should we only top-up the existing number of workers and drones to `nWorkers` and `nDrones` (see details)
`exact`	logical, if the csd locus is turned on and exact is `TRUE`, create the exact specified number of only viable workers (heterozygous on the csd locus)
`resetEvents`	logical, call `resetEvents` as part of the build up
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	additional arguments passed to `nWorkers` or `nDrones` when these arguments are a function

Details

This function increases queen's `nWorkers`, `nHomBrood`, and `nDrones` counters. It also turns production on.

Argument `new` enables simulation of two common cases. First, if you are modelling year-to-year cycle, you will likely want `new = TRUE`, so that, say, in spring you will replace old (from last year) workers and drones with the new ones. This is the case that we are targeting and hence `new = TRUE` is default. Second, if you are modelling shorter period cycles, you will likely want `new = FALSE` to just top up the current workers and drones - you might also want to look at `replaceWorkers` and `replaceDrones`.

TODO: Discuss on how to model day-to-day variation with `new = FALSE`. We are not sure this is easy to achieve with current implementation just now, but could be expanded. https://github.com/HighlanderLab/SIMplyBee/issue

Value

`Colony-class` or `MultiColony-class` with workers and drones replaced or added

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)
```

```
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
isProductive(colony)
apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
isProductive(apiary)

# Build up
# Using defaults in SP$nWorkers & SP$nDrones
(colony <- buildUp(colony))
isProductive(colony)
# Build-up a MultiColony class
(apiary <- buildUp(apiary))
isProductive(apiary)

# The user can also specify a function that will give a number
colony <- removeWorkers(colony) # Remove workers to start from fresh
colony <- removeDrones(colony) # Remove drones to start from fresh
buildUp(colony, nWorkers = nWorkersPoisson, nDrones = nDronesPoisson)
buildUp(colony, nWorkers = nWorkersPoisson, nDrones = nDronesPoisson)
# nWorkers and nDrones will vary between function calls when a function is used
# You can store these functions or a values in the SP object
SP$nWorkers <- nWorkersPoisson
SP$nDrones <- nDronesPoisson

# Specifying own number
colony <- buildUp(colony, nWorkers = 100)
# Build up a MultiColony class
apiary <- buildUp(apiary, nWorkers = 250)
# Build up with different numbers
apiary <- buildUp(apiary, nWorkers = c(1000, 2000), nDrones = c(100, 150))
nWorkers(apiary)
nDrones(apiary)

# Queen's counters
getMisc(getQueen(buildUp(colony)))
```

`c,NULLOrPop-method` *Combine a NULL and AlphaSimR population*

Description

This combine `c()` method is a hack to combine NULL and an AlphaSimR population object `c(NULL, pop)` (`c(pop, NULL)` works already with AlphaSimR package code).

Usage

```
## S4 method for signature 'NULLOrPop'
c(x, ...)
```

Arguments

`x`	NULL or `Pop-class`
`...`	list of NULL or `Pop-class` objects

`calcBeeGRMIbd` *Calculate Genomic Relatedness Matrix (GRM) for honeybees from Identical By Descent genomic data*

Description

Level 0 function that returns Genomic Relatedness Matrix (GRM) for honeybees from Identical By Descent genomic data (tracked alleles since the founders) - see references on the background theory.

Usage

```
calcBeeGRMIbd(x)
```

Arguments

`x`	`matrix` of haplotypes/genomes with allele indicators for the founders coded as 1, 2, ... Haplotypes/genome are in rows and sites are in columns; no missing values are allowed (this is not checked!). Row names are essential (formated as ind_genome as returned by AlphaSimR IBD functions) to infer the individual and their ploidy (see examples)!

Value

a list with a matrix of gametic relatedness coefficients (genome) and a matrix of individual relatedness coefficients (indiv)

References

Grossman and Eisen (1989) Inbreeding, coancestry, and covariance between relatives for X-chromosomal loci. The Journal of Heredity, doi:10.1093/oxfordjournals.jhered.a110812

Fernando and Grossman (1989) Covariance between relatives for X-chromosomal loci in a population in disequilibrium. Theoretical and Applied Genetics, doi:10.1007/bf00305821

Fernando and Grossman (1990) Genetic evaluation with autosomal and X-chromosomal inheritance. Theoretical and Applied Genetics, doi:10.1007/bf00224018

Van Arendonk, Tier, and Kinghorn (1994) Use of multiple genetic markers in prediction of breeding values. Genetics, doi:10.1093/genetics/137.1.319

Hill and Weir (2011) Variation in actual relationship as a consequence of Mendelian sampling and linkage. Genetics Research, doi:10.1017/s0016672310000480

Examples

```
founderGenomes <- quickHaplo(nInd = 3, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

SP$setTrackRec(TRUE)
SP$setTrackPed(isTrackPed = TRUE)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 1, nDrones = nFathersPoisson)
colony <- createColony(basePop[2])
colony <- cross(x = colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)

haploQ <- getQueenIbdHaplo(colony)
haploW <- getWorkersIbdHaplo(colony)
haploD <- getDronesIbdHaplo(colony)
SP$pedigree

haplo <- rbind(haploQ, haploW, haploD)

GRMs <- calcBeeGRMIbd(x = haplo)
# You can visualise this matrix with the image() functions from the "Matrix" package

# Inspect the diagonal of the relationship matrix between individuals
x <- diag(GRMs$indiv)
hist(x)
summary(x)

 # Inspect the off-diagonal of the relationship matrix between individuals
x <- GRMs$indiv[lower.tri(x = GRMs$indiv, diag = FALSE)]
hist(x)
summary(x)

ids <- getCasteId(colony)
qI <- ids$queen
wI <- sort(ids$workers)
dI <- sort(ids$drones)

qG <- c(t(outer(X = qI, Y = 1:2, FUN = paste, sep = "_")))
wG <- c(t(outer(X = wI, Y = 1:2, FUN = paste, sep = "_")))
dG <- paste(dI, 1, sep = "_")

# Queen vs workers
GRMs$genome[wG, qG]
GRMs$indiv[wI, qI]

# Queen vs drones
GRMs$genome[dG, qG]
GRMs$indiv[dI, qI]

# Workers vs workers
```

```
GRMs$genome[wG, wG]
GRMs$indiv[wI, wI]

# Workers vs drones
GRMs$genome[dG, wG]
GRMs$indiv[dI, wI]
```

---

calcBeeGRMIbs *Calculate Genomic Relatedness Matrix (GRM) for honeybees from Identical By State genomic data*

Description

Level 0 function that returns Genomic Relatedness Matrix (GRM) for honeybees from Identical By State genomic data (bi-allelic SNP represented as allele dosages) following the method for the sex X chromosome (Druet and Legarra, 2020)

Usage

```
calcBeeGRMIbs(x, sex, alleleFreq = NULL)

calcBeeAlleleFreq(x, sex)
```

Arguments

`x`	`matrix` of genotypes represented as allele dosage coded as 0, 1, or 2 in females (queens or workers) and as 0 or 1 in males (fathers or drones); individuals are in rows and sites are in columns; no missing values are allowed (this is not checked - you will get NAs!)
`sex`	character vector denoting sex for individuals with genotypes in `x` - `"F"` for female and `"M"` for male
`alleleFreq`	numeric, vector of allele frequencies for the sites in `x`; if `NULL`, then `calcBeeAlleleFreq` is used

Value

matrix of genomic relatedness coefficients

Functions

- `calcBeeAlleleFreq()`: Calculate allele frequencies from honeybee genotypes

References

Druet and Legarra (2020) Theoretical and empirical comparisons of expected and realized relationships for the X-chromosome. Genetics Selection Evolution, 52:50 doi:10.1186/s12711020005706

Examples

```
founderGenomes <- quickHaplo(nInd = 3, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

SP$setTrackRec(TRUE)
SP$setTrackPed(isTrackPed = TRUE)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 1, nDrones = nFathersPoisson)
colony <- createColony(basePop[2])
colony <- cross(x = colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)

geno <- getSegSiteGeno(colony, collapse = TRUE)
sex <- getCasteSex(x = colony, collapse = TRUE)

GRM <- calcBeeGRMIbs(x = geno, sex = sex)
# You can visualise this matrix with the function image() from the package 'Matrix'

#Look at the diagonal at the relationship matrix
x <- diag(GRM)
hist(x)
summary(x)

#Look at the off-diagonal at the relationship matrix
x <- GRM[lower.tri(x = GRM, diag = FALSE)]
hist(x)
summary(x)

# Compare relationship between castes
ids <- getCasteId(colony)
idQueen <- ids$queen
idWorkers <- ids$workers
idDrones <- ids$drones

# Queen vs others
GRM[idQueen, idWorkers]
GRM[idQueen, idDrones]

# Workers vs worker
GRM[idWorkers, idWorkers]

# Workers vs drones
GRM[idWorkers, idDrones]

# Calculating allele frequencies ourselves (say, to "shift" base population)
aF <- calcBeeAlleleFreq(x = geno, sex = sex)
hist(aF)
GRM2 <- calcBeeGRMIbs(x = geno, sex = sex, alleleFreq = aF)
stopifnot(identical(GRM2, GRM))
```

```
# You can also create relationships with pooled genomes
pooledGenoW <- getPooledGeno(getWorkersSegSiteGeno(colony),
                             type = "mean",
                             sex = getCasteSex(colony, caste="workers"))
queenGeno <- getQueenSegSiteGeno(colony)
# Compute relationship between pooled workers genotype and the queen
calcBeeGRMIbs(x = rbind(queenGeno, pooledGenoW), sex = c("F","F"))
# You can now compare how this compare to relationships between the queen
# individual workers!
```

`calcColonyValue` *Calculate colony value(s)*

Description

Level 0 function that calculate value(s) of a colony.

Usage

```
calcColonyValue(x, FUN = NULL, simParamBee = NULL, ...)

calcColonyPheno(x, FUN = mapCasteToColonyPheno, simParamBee = NULL, ...)

calcColonyGv(x, FUN = mapCasteToColonyGv, simParamBee = NULL, ...)

calcColonyBv(x, FUN = mapCasteToColonyBv, simParamBee = NULL, ...)

calcColonyDd(x, FUN = mapCasteToColonyDd, simParamBee = NULL, ...)

calcColonyAa(x, FUN = mapCasteToColonyAa, simParamBee = NULL, ...)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`FUN`	function, that calculates colony value from values of colony members
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	other arguments of `FUN`

Value

a matrix with one value or a row of values when `x` is `Colony-class` and a row-named matrix when `x` is `MultiColony-class`, where names are colony IDs

Functions

- `calcColonyPheno()`: Calculate colony phenotype value from caste individuals' phenotype values
- `calcColonyGv()`: Calculate colony genetic value from caste individuals' genetic values
- `calcColonyBv()`: Calculate colony breeding value from caste individuals' breeding values
- `calcColonyDd()`: Calculate colony dominance value from caste individuals' dominance values
- `calcColonyAa()`: Calculate colony epistasis value from caste individuals' epistasis value

See Also

`mapCasteToColonyValue` as an example of `FUN`, `selectColonies` for example for to select colonies based on these values, and `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)


# Define two traits that collectively affect colony honey yield:
# 1) queen's effect on colony honey yield, say via pheromone secretion phenotype
# 2) workers' effect on colony honey yield, say via foraging ability phenotype
# The traits will have a negative genetic correlation of -0.5 and heritability
# of 0.25 (on an individual level)
nWorkers <- 10
mean <- c(10, 10 / nWorkers)
varA <- c(1, 1 / nWorkers)
corA <- matrix(data = c(
  1.0, -0.5,
  -0.5, 1.0
), nrow = 2, byrow = TRUE)
varE <- c(3, 3 / nWorkers)
varA / (varA + varE)
SP$addTraitADE(nQtlPerChr = 100,
               mean = mean,
               var = varA, corA = corA,
               meanDD = 0.1, varDD = 0.2, corD = corA,
               relAA = 0.1, corAA = corA)
SP$setVarE(varE = varE)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 200)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create and cross Colony and MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(colony, nWorkers = nWorkers, nDrones = 3)
apiary <- createMultiColony(basePop[3:5], n = 2)
```

```
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(apiary, nWorkers = nWorkers, nDrones = 3)

# Colony value - shorthand version
# (using the default mapCasteToColony*() functions - you can provide yours instead!)
# Phenotype value
calcColonyPheno(colony)
calcColonyPheno(apiary)
# Genetic value
calcColonyGv(colony)
calcColonyGv(apiary)

# Colony value - long version
# (using the default mapCasteToColony*() function - you can provide yours instead!)
calcColonyValue(colony, FUN = mapCasteToColonyPheno)
calcColonyValue(apiary, FUN = mapCasteToColonyPheno)

# Colony value - long version - using a function stored in SimParamBee (SP)
# (using the default mapCasteToColony*() function - you can provide yours instead!)
SP$colonyValueFUN <- mapCasteToColonyPheno
calcColonyValue(colony)
calcColonyValue(apiary)
```

`calcInheritanceCriterion` *Calculate the inheritance criterion*

Description

Level 0 function that calculates the inheritance criterion as the sum of the queen (maternal) and workers (direct) effect from the queen, as defined by Du et al. (2021). This can be seen as the expected value of drones from the queen or half the expected value of virgin queens from the queen.

Usage

```
calcInheritanceCriterion(
  x,
  queenTrait = 1,
  workersTrait = 2,
  use = "gv",
  simParamBee = NULL
)
```

Arguments

`x`	`Pop-class`, `Colony-class` or `MultiColony-class`
`queenTrait`	numeric (column position) or character (column name), trait that represents queen's effect on the colony value; if `NULL` then this effect is 0

`workersTrait`	numeric (column position) or character (column name), trait that represents workers' effect on the colony value; if `NULL` then this effect is 0
`use`	character, the measure to use for the calculation, being either "gv" (genetic value), "ebv" (estimated breeding value), or "pheno" (phenotypic value)
`simParamBee`	`SimParamBee`, global simulation parameters

Value

integer when `x` is `Colony-class` and a named list when `x` is `MultiColony-class`, where names are colony IDs

References

Du, M., et al. (2021) Short-term effects of controlled mating and selection on the genetic variance of honeybee populations. Heredity 126, 733–747. doi:10.1038/s41437021004112

See Also

`calcSelectionCriterion` and `calcPerformanceCriterion` and as well as `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

meanA <- c(10, 10 / SP$nWorkers)
varA <- c(1, 1 / SP$nWorkers)
corA <- matrix(data = c( 1.0, -0.5,
                        -0.5,  1.0), nrow = 2, byrow = TRUE)
SP$addTraitA(nQtlPerChr = 100, mean = meanA, var = varA, corA = corA,
name = c("queenTrait", "workersTrait"))
varE <- c(3, 3 / SP$nWorkers)
corE <- matrix(data = c(1.0, 0.3,
                        0.3, 1.0), nrow = 2, byrow = TRUE)
SP$setVarE(varE = varE, corE = corE)
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])

calcInheritanceCriterion(colony, queenTrait = 1, workersTrait = 2)
calcInheritanceCriterion(apiary, queenTrait = 1, workersTrait = 2)
```

```
apiary[[2]] <- removeQueen(apiary[[2]])
calcInheritanceCriterion(apiary, queenTrait = 1, workersTrait = 2)
```

`calcPerformanceCriterion`

Calculate the performance criterion

Description

Level 0 function that calculates the performance criterion as the sum of the queen (maternal) effect from the queen and the workers (direct) effect from her workers, as defined by Du et al. (2021). This can be seen as the expected value of the colony.

Usage

```
calcPerformanceCriterion(
  x,
  queenTrait = 1,
  workersTrait = 2,
  workersTraitFUN = sum,
  use = "gv",
  simParamBee = NULL
)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`queenTrait`	numeric (column position) or character (column name), trait that represents queen's effect on the colony value; if `NULL` then this effect is 0
`workersTrait`	numeric (column position) or character (column name), trait that represents workers' effect on the colony value; if `NULL` then this effect is 0
`workersTraitFUN`	function, that will be applied to the workers effect values of workers, default is sum (see examples), but note that the correct function will depend on how you will setup simulation!
`use`	character, the measure to use for the calculation, being either "gv" (genetic value),"ebv" (estimated breeding value), or "pheno" (phenotypic value)
`simParamBee`	`SimParamBee`, global simulation parameters

Value

integer when `x` is `Colony-class` and a named list when `x` is `MultiColony-class`, where names are colony IDs

References

Du, M., et al. (2021) Short-term effects of controlled mating and selection on the genetic variance of honeybee populations. Heredity 126, 733–747. doi:10.1038/s41437021004112

See Also

`calcSelectionCriterion` and `calcInheritanceCriterion` and as well as `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

meanA <- c(10, 10 / SP$nWorkers)
varA <- c(1, 1 / SP$nWorkers)
corA <- matrix(data = c( 1.0, -0.5,
                        -0.5,  1.0), nrow = 2, byrow = TRUE)
SP$addTraitA(nQtlPerChr = 100, mean = meanA, var = varA, corA = corA,
name = c("queenTrait", "workersTrait"))
varE <- c(3, 3 / SP$nWorkers)
corE <- matrix(data = c(1.0, 0.3,
                        0.3, 1.0), nrow = 2, byrow = TRUE)
SP$setVarE(varE = varE, corE = corE)
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(colony)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(apiary)

calcPerformanceCriterion(colony, queenTrait = 1, workersTrait = 2, workersTraitFUN = sum)
calcPerformanceCriterion(apiary, queenTrait = 1, workersTrait = 2, workersTraitFUN = sum)

apiary[[2]] <- removeQueen(apiary[[2]])
calcPerformanceCriterion(apiary, queenTrait = 1,
                         workersTrait = 2, workersTraitFUN = sum)
```

`calcQueensPHomBrood` *The expected proportion and a realised number of csd homozygous brood*

Description

Level 0 functions that calculate or report the proportion of csd homozygous brood of a queen or a colony. The csd locus determines viability of fertilised eggs (brood) - homozygous brood is removed by workers. These functions 1) calculate the expected proportion of homozygous brood from the csd allele of the queen and fathers, 2) report the expected proportion of homozygous brood, or 3) report a realised number of homozygous brood due to inheritance process. See `vignette(package = "SIMplyBee")` for more details.

Usage

```
calcQueensPHomBrood(x, simParamBee = NULL)

pHomBrood(x, simParamBee = NULL)

nHomBrood(x, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`simParamBee`	`SimParamBee`, global simulation parameters

Value

numeric, expected csd homozygosity named by colony id when `x` is `MultiColony-class`

Functions

- `pHomBrood()`: Expected percentage of csd homozygous brood of a queen / colony
- `nHomBrood()`: Realised number of csd homozygous brood produced by a queen

See Also

Demo in the introductory vignette `vignette("Honeybee_biology", package="SIMplyBee")`

Examples

```
# This is a bit long example - the key is at the end!
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 120, nDrones = 20)
colony <- addVirginQueens(x = colony, nInd = 1)
```

```
apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 100, nDrones = 10)

# Virgin queen
try(calcQueensPHomBrood(basePop[5]))

# Queens of colony
calcQueensPHomBrood(colony)

# Queens of apiary
calcQueensPHomBrood(apiary)

# Inbreed virgin queen with her brothers to generate csd homozygous brood
colony2 <- createColony(x = getVirginQueens(colony))
colony2 <- cross(x = colony2, drones = pullDrones(x = colony, nInd = nFathersPoisson())[[1]])

# Calculate the expected csd homozygosity
calcQueensPHomBrood(getQueen(colony2))
pHomBrood(colony2)

# Evaluate a realised csd homozygosity
nHomBrood(addWorkers(colony2, nInd = 100))
nHomBrood(addWorkers(colony2, nInd = 100))
# nHomBrood will vary between function calls due to inheritance process
```

`calcSelectionCriterion` *Calculate the selection criterion*

Description

Level 0 function that calculates the selection criterion as the sum of workers (direct) and queen (maternal) effects of workers, as defined by Du et al. (2021). This can be seen as the expected value of virgin queens from the queen (as well as workers, but we would not be selecting workers).

Usage

```
calcSelectionCriterion(
  x,
  queenTrait = 1,
  queenTraitFUN = sum,
  workersTrait = 2,
  workersTraitFUN = sum,
  use = "gv",
  simParamBee = NULL
)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`queenTrait`	numeric (column position) or character (column name), trait that represents queen's effect on the colony value; if `NULL` then this contribution is 0
`queenTraitFUN`	function, that will be applied to the queen effect values of workers, default is sum (see examples), but note that the correct function will depend on how you will setup simulation!
`workersTrait`	numeric (column position) or character (column name), trait that represents workers' effect on the colony value; if `NULL` then this contribution is 0
`workersTraitFUN`	function, that will be applied to the workers effect values of workers, default is sum (see examples), but note that the correct function will depend on how you will setup simulation!
`use`	character, the measure to use for the calculation, being either "gv" (genetic value), "ebv" (estimated breeding value), or "pheno" (phenotypic value)
`simParamBee`	`SimParamBee`, global simulation parameters

Value

integer when `x` is `Colony-class` and a named list when `x` is `MultiColony-class`, where names are colony IDs

References

Du, M., et al. (2021) Short-term effects of controlled mating and selection on the genetic variance of honeybee populations. Heredity 126, 733–747. doi:10.1038/s41437021004112

See Also

`calcInheritanceCriterion` and `calcPerformanceCriterion` and as well as

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

meanA <- c(10, 10 / SP$nWorkers)
varA <- c(1, 1 / SP$nWorkers)
corA <- matrix(data = c( 1.0, -0.5,
                        -0.5,  1.0), nrow = 2, byrow = TRUE)
SP$addTraitA(nQtlPerChr = 100, mean = meanA, var = varA, corA = corA,
name = c("queenTrait", "workersTrait"))
varE <- c(3, 3 / SP$nWorkers)
corE <- matrix(data = c(1.0, 0.3,
                        0.3, 1.0), nrow = 2, byrow = TRUE)
SP$setVarE(varE = varE, corE = corE)
basePop <- createVirginQueens(founderGenomes)
```

```
drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(colony)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(apiary)

calcSelectionCriterion(colony,
                       queenTrait = 1, queenTraitFUN = sum,
                       workersTrait = 2, workersTraitFUN = sum)
calcSelectionCriterion(apiary,
                       queenTrait = 1, queenTraitFUN = sum,
                       workersTrait = 2, workersTraitFUN = sum)

apiary[[2]] <- removeQueen(apiary[[2]])
calcSelectionCriterion(apiary, queenTrait = 1,
                       workersTrait = 2, workersTraitFUN = sum)
```

`collapse` *Collapse*

Description

Level 2 function that collapses a Colony or MultiColony object by setting the collapse event slot to `TRUE`. The production status slot is also changed (to `FALSE`).

Usage

```
collapse(x)
```

Arguments

x	`Colony-class` or `MultiColony-class`

Details

You should use this function in an edge-case when you want to indicate that the colony has collapsed, but you still want to collect some values from the colony for a retrospective analysis. It resembles a situation where the colony has collapsed, but dead bees are still in the hive.

Value

`Colony-class` or `MultiColony-class` with the collapse event set to `TRUE`

Examples

```
founderGenomes <- quickHaplo(nInd = 10, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(basePop[1], n = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = 10)

# Create Colony and MultiColony class
colony <- createColony(x = basePop[1])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(x = basePop[2:10], n = 9)
apiary <- cross(apiary, drones = droneGroups[2:10])

# Collapse
hasCollapsed(colony)
colony <- collapse(colony)
hasCollapsed(colony)

hasCollapsed(apiary)
tmp <- pullColonies(apiary, n = 2)
tmp
apiaryLost <- collapse(tmp$pulled)
hasCollapsed(apiaryLost)
apiaryLeft <- tmp$remnant
hasCollapsed(apiaryLeft)
```

`Colony-class` *Honeybee colony*

Description

An object holding honeybee colony

Usage

```
isColony(x)

## S4 method for signature 'Colony'
show(object)

## S4 method for signature 'ColonyOrNULL'
c(x, ...)
```

Arguments

`x`	`Colony-class`
`object`	`Colony-class`
`...`	NULL, `Colony-class`, or `MultiColony-class`

Value

`Colony-class` or `MultiColony-class`

Functions

- `isColony()`: Test if x is a Colony class object
- `show(Colony)`: Show colony object
- `c(ColonyOrNULL)`: Combine multiple colony objects

Slots

`id` integer, unique ID of the colony

`location` numeric, location of the colony (x, y)

`queen` `Pop-class`, the queen of the colony (we use its misc slot for queen's age and drones (fathers) she mated with)

`virginQueens` `Pop-class`, virgin queens of the colony

`drones` `Pop-class`, drones of the colony

`workers` `Pop-class`, workers of the colony

`split` logical, has colony split

`swarm` logical, has colony swarmed

`supersedure` logical, has colony superseded

`collapse` logical, has colony collapsed

`production` logical, is colony productive

`misc` list, available for storing extra information about the colony

See Also

`createColony`

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)
colony1 <- createColony(x = basePop[2])
colony1 <- cross(colony1, drones = droneGroups[[1]])
colony2 <- createColony(x = basePop[3])
colony2 <- cross(colony2, drones = droneGroups[[2]])
colony3 <- createColony(x = basePop[4])
colony3 <- cross(colony3, drones = droneGroups[[3]])

colony1
```

```
show(colony1)
is(colony1)
isColony(colony1)

apiary <- c(colony1, colony2)
is(apiary)
isMultiColony(apiary)

c(apiary, colony3)
c(colony3, apiary)
```

`combine` *Combine two colony objects*

Description

Level 2 function that combines two Colony or MultiColony objects into one or two colonies objects of the same length to one. For example, to combine a weak and a strong colony (or MultiColony). Workers and drones of the weak colony are added to the strong. User has to remove the weak colony (or MultiColony) from the workspace.

Usage

```
combine(strong, weak)
```

Arguments

`strong`	`Colony-class` or `MultiColony-class`
`weak`	`Colony-class` or `MultiColony-class`

Value

a combined `Colony-class` or `MultiColony-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 10, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(basePop[1], n = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = 10)

# Create weak and strong Colony and MultiColony class
colony1 <- createColony(x = basePop[2])
colony1 <- cross(colony1, drones = droneGroups[[1]])
colony2 <- createColony(x = basePop[3])
colony2 <- cross(colony2, drones = droneGroups[[2]])
apiary1 <- createMultiColony(basePop[4:6], n = 3)
```

```
apiary1 <- cross(apiary1, drones = droneGroups[3:5])
apiary2 <- createMultiColony(basePop[7:9], n = 3)
apiary2 <- cross(apiary2, drones = droneGroups[6:8])

# Build-up
colony1 <- buildUp(x = colony1, nWorkers = 100, nDrones = 20)
colony2 <- buildUp(x = colony2, nWorkers = 20, nDrones = 5)
apiary1 <- buildUp(x = apiary1, nWorkers = 100, nDrones = 20)
apiary2 <- buildUp(x = apiary2, nWorkers = 20, nDrones = 5)

# Combine
nWorkers(colony1); nWorkers(colony2)
nDrones(colony1); nDrones(colony2)
colony1 <- combine(strong = colony1, weak = colony2)
nWorkers(colony1); nWorkers(colony2)
nDrones(colony1); nDrones(colony2)
rm(colony2)

nWorkers(apiary1); nWorkers(apiary2)
nDrones(apiary1); nDrones(apiary2)
apiary1 <- combine(strong = apiary1, weak = apiary2)
nWorkers(apiary1); nWorkers(apiary2)
nDrones(apiary1); nDrones(apiary2)
rm(apiary2)
```

`combineBeeGametes` *Create diploid gametes from a mated queen*

Description

Level 1 function that produces diploid offspring from a mated queen. Queen is diploid, while drones are double haploids so we use AlphaSimR diploid functionality to make this cross, but since drones are double haploids we get the desired outcome. This is an utility function, and you most likely want to use the `cross` functions.

Usage

```
combineBeeGametes(queen, drones, nProgeny = 1, simParamBee = NULL)
```

Arguments

`queen`	`Pop-class`, with a single diploid individual
`drones`	`Pop-class`, with one or more diploid (double haploid) individual(s)
`nProgeny`	integer, number of progeny to create per cross
`simParamBee`	`SimParamBee`, global simulation parameters

Value

Pop-class with diploid individuals

Not exporting this function, since its just a helper

---

combineBeeGametesHaploDiploid

Create diploid gametes from a mated queen

Description

Level 1 function that produces diploid offspring from a mated queen. Drones are haploid, while the queen is diploid, so we first generate gametes (with recombination) from her and merge them with drone genomes (=gametes), where we randomly re-sample drones to get the desired number of progeny. This is an utility function, and you most likely want to use the `cross` function.

Usage

```
combineBeeGametesHaploDiploid(queen, drones, nProgeny = 1, simParamBee = NULL)
```

Arguments

`queen`	`Pop-class`, with a single diploid individual
`drones`	`Pop-class`, with one or more haploid individual(s)
`nProgeny`	integer, number of progeny to create per cross
`simParamBee`	`SimParamBee`, global simulation parameters

Details

This would be the right approach to handle haplo-diploid inheritance in bees, but it causes a raft of downstream issues, since AlphaSimR assumes that individuals have the same ploidy. Hence, we don't use this function.

Value

`Pop-class` with diploid individuals

`createCastePop` *Creates caste population individuals from the colony*

Description

Level 1 function that creates the specified number of caste individuals from the colony with a mated queens. If csd locus is active, it takes it into account and any csd homozygotes are removed and counted towards homozygous brood.

Usage

```
createCastePop(
  x,
  caste = NULL,
  nInd = NULL,
  exact = TRUE,
  year = NULL,
  editCsd = TRUE,
  csdAlleles = NULL,
  simParamBee = NULL,
  ...
)

createWorkers(x, nInd = NULL, exact = FALSE, simParamBee = NULL, ...)

createDrones(x, nInd = NULL, simParamBee = NULL, ...)

createVirginQueens(
  x,
  nInd = NULL,
  year = NULL,
  editCsd = TRUE,
  csdAlleles = NULL,
  simParamBee = NULL,
  ...
)
```

Arguments

`x`	`link[AlphaSimR]{MapPop-class}` (only if caste is "virginQueens"), or `Pop` (only if caste is "drones") or `Colony-class` or `MultiColony-class`
`caste`	character, "workers", "drones", or "virginQueens"
`nInd`	numeric or function, number of caste individuals; if `NULL` then `SimParamBee$nWorkers`, `SimParamBee$nDrones` or `SimParamBee$nVirginQueens` is used depending on the caste; only used when `x` is `Colony-class` or `MultiColony-class`, when `x` is `link[AlphaSimR]{MapPop-class}` all individuals in `x` are converted into virgin queens

`exact`	logical, only relevant when creating workers, if the csd locus is active and exact is `TRUE`, create the exactly specified number of viable workers (heterozygous on the csd locus)
`year`	numeric, year of birth for virgin queens
`editCsd`	logical (only active when `x` is `link[AlphaSimR]{MapPop-class}`), whether the csd locus should be edited to ensure heterozygosity at the csd locus (to get viable virgin queens); see `csdAlleles`
`csdAlleles`	`NULL` or list (only active when `x` is `link[AlphaSimR]{MapPop-class}`); If `NULL`, then the function samples a heterozygous csd genotype for each virgin queen from all possible csd alleles. If not `NULL`, the user provides a list of length `nInd` with each node holding a matrix or a data.frame, each having two rows and n columns. Each row must hold one csd haplotype (allele) that will be assigned to a virgin queen. The n columns span the length of the csd locus as specified in `SimParamBee`. The two csd alleles must be different to ensure heterozygosity at the csd locus.
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	additional arguments passed to `nInd` when this argument is a function

Value

when `x` is `MapPop-class` returns `virginQueens` (a `Pop-class`); when `x` is `Colony-class` returns `virginQueens` (a `Pop-class`); when `x` is `MultiColony-class` return is a named list of `virginQueens` (a `Pop-class`); named by colony ID

Functions

- `createWorkers()`: Create workers from a colony
- `createDrones()`: Create drones from a colony
- `createVirginQueens()`: Create virgin queens from a colony

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

SP$setTrackRec(TRUE)
SP$setTrackPed(isTrackPed = TRUE)

# Create virgin queens on a MapPop
basePop <- createCastePop(founderGenomes, caste = "virginQueens")
# Or alias
createVirginQueens(founderGenomes)
# Same aliases exist for all the castes!!!

# Create drones on a Pop
drones <- createDrones(x = basePop[1],  nInd = 200)
# Or create unequal number of drones from multiple virgin queens
drones <- createDrones(basePop[1:2], nInd = c(100, 200))
```

```
droneGroups <- pullDroneGroupsFromDCA(drones, n = 3, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])

# Using default nInd in SP
colony@virginQueens <- createVirginQueens(colony)
colony@workers <- createWorkers(colony)$workers
colony@drones <- createDrones(colony)
# Usually, you would use functions buildUp() or addCastePop()

# These populations hold individual information
# Example on the virgin queens (same holds for all castes!)
virginQueens <- colony@virginQueens
virginQueens@id
virginQueens@sex
virginQueens@mother
virginQueens@father

# Specify own number
SP$nVirginQueens <- 15
SP$nWorkers <- 100
SP$nDrones <- 10
createVirginQueens(colony)
createVirginQueens(apiary)
# Or creating unequal numbers
createVirginQueens(apiary, nInd = c(5, 10))
# nVirginQueens will NOT vary between function calls when a constant is used

# Specify a function that will give a number
createVirginQueens(colony, nInd = nVirginQueensPoisson)
createVirginQueens(apiary, nInd = nVirginQueensPoisson)
# No. of individuals will vary between function calls when a function is used

# Store a function or a value in the SP object
SP$nVirginQueens <- nVirginQueensPoisson
createVirginQueens(colony)
createVirginQueens(colony)
createVirginQueens(apiary)
createVirginQueens(apiary)
# No. of individuals will vary between function calls when a function is used

# csd homozygosity - relevant when creating virgin queens
SP <- SimParamBee$new(founderGenomes, csdChr = 1, nCsdAlleles = 8)

basePop <- createVirginQueens(founderGenomes, editCsd = FALSE)
all(isCsdHeterozygous(basePop))

basePop <- createVirginQueens(founderGenomes, editCsd = TRUE)
all(isCsdHeterozygous(basePop))
```

`createColony` *Create a new Colony*

Description

Level 2 function that creates a new `Colony-class` to initiate simulations.

Usage

```
createColony(x = NULL, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, one queen or virgin queen(s)
`simParamBee`	`SimParamBee`, global simulation parameters

Value

new `Colony-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 15)

# Create an empty Colony class
colony <- createColony()

# Create Colony class with one or multiple virgin queens
colony1 <- createColony(x = basePop[2])
colony1
colony2 <- createColony(x = basePop[3:4])
colony2

# Create a mated Colony
colony1 <- cross(colony1, drones = drones)
colony1
```

`createCrossPlan` *Create a plan for crossing virgin queens*

Description

Level 0 function that creates a plan for crossing virgin queens or virgin colonies by sampling drones or drone producing colonies to mate with a the virgin queens/colonies either at random (`spatial = FALSE`) or according to the distance between colonies (`spatial = TRUE`).

Usage

```
createCrossPlan(
  x,
  drones = NULL,
  droneColonies = NULL,
  nDrones = NULL,
  spatial = FALSE,
  radius,
  simParamBee = NULL,
  ...
)
```

Arguments

`x`	`Pop-class` or `Colony-class` or `MultiColony-class`, the object with the virgin queens that need to be crossed. When `spatial = TRUE`, the argument needs to be a `Colony-class` or `MultiColony-class` with the location set
`drones`	`Pop-class`, a population of drones (resembling a drone congregation area) available for mating. When `spatial = TRUE`, the user can not provide drones, but needs to provide drone producing colonies instead (see argument `droneColonies`)
`droneColonies`	`MultiColony-class`, drone producing colonies available for mating. When `spatial = TRUE`, the object needs to have the location set
`nDrones`	integer or function, number of drones to sample for each crossing. You need to provide this to provide this argument even when sampling drone producing colonies (otherwise, the default value will be used)
`spatial`	logical, whether the drone producing colonies should be sampled according to their distance from the virgin colony (that is, in a radius)
`radius`	numeric, the radius from the virgin colony in which to sample mating partners, only needed when `spatial = TRUE`
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	other arguments for `nDrones`, when `nDrones` is a function

Value

named list with names being virgin queens/colonies IDs with each list element holding the IDs of selected drones or drone producing colonies

Examples

```
founderGenomes <- quickHaplo(nInd = 1000, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

# Create three virgin MultiColony objects with locations
virginColonies1 <- createMultiColony(basePop[1:30])
virginColonies1 <- setLocation(virginColonies1,
                               location = Map(c, runif(30, 0, 2*pi),
                                                 runif(30, 0, 2*pi)))
virginColonies2 <- createMultiColony(basePop[31:60])
virginColonies2 <- setLocation(virginColonies2,
                               location = Map(c, runif(30, 0, 2*pi),
                                                 runif(30, 0, 2*pi)))
virginColonies3 <- createMultiColony(basePop[61:90])
virginColonies3 <- setLocation(virginColonies3,
                               location = Map(c, runif(30, 0, 2*pi),
                                                 runif(30, 0, 2*pi)))

# Create drone colonies
droneColonies <- createMultiColony(basePop[121:200])
droneColonies <- setLocation(droneColonies,
                             location = Map(c, runif(80, 0, 2*pi),
                                               runif(80, 0, 2*pi)))

# Create some drones to mate initial drone colonies with
DCA <- createDrones(basePop[201:300], nInd = 20)
# Cross initial virgin drone colonies to the DCA with a random cross plan
randomCrossPlan <- createCrossPlan(x = droneColonies,
                                   drones = DCA,
                                   nDrones = nFathersPoisson,
                                   spatial = FALSE)
droneColonies <- cross(droneColonies,
                       drones = DCA,
                       nDrones = nFathersPoisson,
                       crossPlan = randomCrossPlan)

# Plot the colonies in space
virginLocations <- as.data.frame(getLocation(c(virginColonies1, virginColonies2, virginColonies3),
                                             collapse= TRUE))
virginLocations$Type <- "Virgin"
droneLocations <- as.data.frame(getLocation(droneColonies, collapse= TRUE))
droneLocations$Type <- "Drone"
locations <- rbind(virginLocations, droneLocations)

plot(x = locations$V1, y = locations$V2,
     col = c("red", "blue")[as.numeric(as.factor(locations$Type))])

# Cross according to a spatial cross plan according to the colonies' locations
crossPlanSpatial <- createCrossPlan(x = virginColonies1,
                                   droneColonies = droneColonies,
```

```
                                   nDrones = nFathersPoisson,
                                   spatial = TRUE,
                                   radius = 1.5)

# Plot the crossing for the first colony in the crossPlan
virginLocations1 <- as.data.frame(getLocation(virginColonies1, collapse= TRUE))
virginLocations1$Type <- "Virgin"
droneLocations <- as.data.frame(getLocation(droneColonies, collapse= TRUE))
droneLocations$Type <- "Drone"
locations1 <- rbind(virginLocations1, droneLocations)

# Blue marks the target virgin colony and blue marks the drone colonies in the chosen radius
plot(x = locations1$V1, y = locations1$V2, pch = c(1, 2)[as.numeric(as.factor(locations1$Type))],
  col = ifelse(rownames(locations1) %in% crossPlanSpatial[[1]],
                             "red",
                            ifelse(rownames(locations1) == names(crossPlanSpatial)[[1]],
                             "blue", "black")))

colonies1 <- cross(x = virginColonies1,
                   crossPlan = crossPlanSpatial,
                   droneColonies = droneColonies,
                   nDrones = nFathersPoisson)
nFathers(colonies1)

# Cross according to a cross plan that is created internally within the cross function
# The cross plan is created at random, regardless the location of the colonies
colonies2 <- cross(x = virginColonies2,
                   droneColonies = droneColonies,
                   nDrones = nFathersPoisson,
                   crossPlan = "create")

# Mate spatially with cross plan created internally by the cross function
colonies3 <- cross(x = virginColonies3,
                   droneColonies = droneColonies,
                   crossPlan = "create",
                   checkCross = "warning",
                   spatial = TRUE,
                   radius = 1)
```

`createDCA` *Create a drone congregation area (DCA)*

Description

Level 1 function that creates a population of drones from a Colony or MultiColony. Such a population is often referred to as a drone congregation area (DCA).

Usage

```
createDCA(x, nInd = NULL, removeFathers = TRUE, simParamBee = NULL)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`nInd`	numeric, number of random drones to pull from each colony, if `NULL` all drones in a colony are pulled
`removeFathers`	logical, removes `drones` that have already mated; set to `FALSE` if you would like to get drones for mating with multiple virgin queens, say via insemination
`simParamBee`	`SimParamBee`, global simulation parameters

Details

In reality, drones leave the colony to mate. They die after that. In this function we only get a copy of drones from `x`, for computational efficiency and ease of use. However, any mating will change the caste of drones to fathers, and they won't be available for future matings (see `cross`). Not unless `removeFathers = FALSE`.

Value

`Pop-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])

colony <- addDrones(colony, nInd = 10)
createDCA(colony)
createDCA(colony, nInd = 10)@id

apiary <- addDrones(apiary)
createDCA(apiary)
createDCA(apiary, nInd = 10)
```

`createMatingStationDCA` *Create a DCA of drones at a mating stations*

Description

Level 1 function that creates a DCA at a classical honeybee mating station of several sister drone producing queens. The functions first creates multiple drone producing queens (DPQs) from one colony; and second, produces drones from the DPQs. All the created drones form a DCA at a mating station.

Usage

```
createMatingStationDCA(
  colony,
  nDPQs = 20,
  nDronePerDPQ = NULL,
  simParamBee = NULL
)
```

Arguments

`colony`	`Colony-class` to produce drone producing queens from
`nDPQs`	integer, the number of drone producing queens
`nDronePerDPQ`	integer, number of drones each DPQ contributed to the DCA
`simParamBee`	`SimParamBee`, global simulation parameters

Value

`Pop-class` with created drones resembling a DCA at a mating station

Examples

```
founderGenomes <- quickHaplo(nInd = 10, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(basePop[1], n = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = 10)

# Create a colony and cross it
colony1 <- createColony(x = basePop[2])
colony1 <- cross(colony1, drones = droneGroups[[1]])

# Create a empty colony
colony2 <- createColony(x = basePop[3])

# Create a mating station from colony1
```

```
matingStation <- createMatingStationDCA(colony1, nDPQs = 20, nDronePerDPQ = 10)

# Cross colony2 on the mating station
fathers <- pullDroneGroupsFromDCA(matingStation, n = 1, nDrones = 15)
colony2 <- cross(colony2, drones = fathers[[1]])
nFathers(colony2)
```

`createMultiColony` *Create MultiColony object*

Description

Level 3 function that creates a set of colonies. Usually to start a simulation.

Usage

```
createMultiColony(x = NULL, n = NULL, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, virgin queens or queens for the colonies (selected at random if there are more than `n` in Pop, while all are used when `n` is `NULL`)
`n`	integer, number of colonies to create (if only `n` is given then `MultiColony-class` is created with `n` `NULL`) individual colony - this is mostly useful for programming)
`simParamBee`	`SimParamBee`, global simulation parameters

Details

When both `x` and `n` are `NULL`, then a `MultiColony-class` with 0 colonies is created.

Value

`MultiColony-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 3, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

# Create 2 empty (NULL) colonies
apiary <- createMultiColony(n = 2)
apiary
apiary[[1]]
apiary[[2]]
```

```
# Create 3 virgin colonies
apiary <- createMultiColony(x = basePop, n = 3) # specify n
apiary <- createMultiColony(x = basePop[1:3]) # take all provided
apiary
apiary[[1]]
apiary[[2]]

# Create mated colonies by crossing
apiary <- createMultiColony(x = basePop[1:2], n = 2)
drones <- createDrones(x = basePop[3], n = 30)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 2, nDrones = 15)
apiary <- cross(apiary, drones = droneGroups)
apiary
apiary[[1]]
apiary[[2]]
```

`cross` *Cross (mate) virgin queen(s) as a population, of a colony, or of all given colonies*

Description

Level 1 function that crosses (mates) a virgin queen to a group of drones. The virgin queen(s) could be within a population (`Pop-class`), in a colony (`Colony-class`), or multi-colony (`MultiColony-class`). This function does not create any progeny, it only stores the mated drones (fathers) so we can later create progeny as needed. When input is a (`Colony-class`) or (`MultiColony-class`), one virgin queens is selected at random, mated, and promoted to the queen of the colony. Other virgin queens are destroyed. Mated drones (fathers) are stored for producing progeny at a later stage. For a better understanding of crossing and the functions have a look at the "Crossing" vignette.

Usage

```
cross(
  x,
  crossPlan = NULL,
  drones = NULL,
  droneColonies = NULL,
  nDrones = NULL,
  spatial = FALSE,
  radius = NULL,
  checkCross = "error",
  simParamBee = NULL,
  ...
)
```

Arguments

`x`	`Pop-class` or `Colony-class` or `MultiColony-class`, one or more virgin queens / colonies to be mated;
`crossPlan`	named list with names being virgin queen or colony IDs with each list element holding the IDs of either selected drones or selected drone producing colonies, OR a string "create" if you want the cross plan to be created internally. The function can create a random (`spatial = FALSE`) or spatial (`spatial = TRUE`) cross plan internally. Also see `createCrossPlan`.
`drones`	a `Pop-class` or a list of `Pop-class`, group(s) of drones that will be mated with virgin queen(s). See `pullDroneGroupsFromDCA`) to create a list of drone "packages". A single `Pop-class` is only allowed when mating a single virgin queen or colony, or when mating according to a cross plan that includes drones' IDs. When creating a spatial cross plan internally, males can not be provided through the `drones` argument as a `Pop-class`, but should be provided through the `droneColonies` argument as or `MultiColony-class`
`droneColonies`	`MultiColony-class` with all available drone producing colonies. Provided when `drones` is not provided (NULL). When providing drone producing colonies, the cross function uses a cross plan, that can either be provided by the user through (`crossPlan`) argument or created internally (when `crossPlan` is "create")
`nDrones`	numeric of function, the number of drones to sample to mate with each virgin queen when using a cross plan
`spatial`	logical, whether the drone producing colonies should be sampled according to their distance from the virgin colony (that is, in a radius)
`radius`	numeric, the radius around the virgin colony in which to sample mating partners, only needed when `spatial = TRUE`
`checkCross`	character, throw a warning (when `checkCross = "warning"`),
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	other arguments for `nDrones`, when `nDrones` is a function

Details

This function changes caste for the mated drones to fathers, and mated virgin queens to queens. See examples. This means that you can not use these individuals in matings any more!

If the supplied drone population is empty (has 0 individuals), which can happen in edge cases or when `nFathersPoisson` is used instead of `nFathersTruncPoisson`, or when performing spatially-aware mating and no drone producing colonies are found in the vicinity, then mating of a virgin queen will fail and she will stay virgin. This can happen for just a few of many virgin queens, which can be annoying to track down, but you can use `isQueen` or `isVirginQueen` to find such virgin queens. You can use `checkCross` to alert you about this situation.

Value

`Pop-class` with mated queen(s). The misc slot of the queens contains additional information about the number of workers, drones, and homozygous brood produced, and the expected percentage of csd homozygous brood.

See Also

Colony-class on how we store the fathers along the queen. For more examples for mating with either externally or internally created cross plan, please see createCrossPlan

For crossing virgin queens according to a cross plan, see createCrossPlan. For crossing virgin queens on a mating stations, see createMatingStationDCA

Examples

```
founderGenomes <- quickHaplo(nInd = 30, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 20, nDrones = nFathersPoisson)

# If input is a Pop class of virgin queen(s)
virginQueen <- basePop[2]
isQueen(virginQueen)
(matedQueen <- cross(
  x = virginQueen,
  drones = droneGroups[1]
))

isQueen(virginQueen)
isQueen(matedQueen)
nFathers(matedQueen)
isDrone(getFathers(matedQueen))
isFather(getFathers(matedQueen))

virginQueens <- basePop[4:5]
matedQueens <- cross(
  x = virginQueens,
  drones = droneGroups[c(3, 4)]
)

isQueen(matedQueens)
nFathers(matedQueens)
getFathers(matedQueens)

# Inbred mated queen (mated with her own sons)
matedQueen2 <- cross(
  x = basePop[1],
  drones = droneGroups[5]
)
# Check the expected csd homozygosity
pHomBrood(matedQueen2)

# If input is a Colony or MultiColony class
# Create Colony and MultiColony class
colony <- createColony(basePop[6])
```

```
isVirginQueen(getVirginQueens(colony))
apiary <- createMultiColony(basePop[7:8])
all(isVirginQueen(mergePops(getVirginQueens(apiary))))

# Cross
colony <- cross(colony, drones = droneGroups[[6]])
isQueenPresent(colony)
apiary <- cross(apiary, drones = droneGroups[c(7, 8)])
all(isQueenPresent(apiary))
nFathers(apiary)

# Try mating with drones that were already used for mating
colony <- createColony(basePop[9])
try((matedColony <- cross(x = colony, drones = droneGroups[[1]])))
# Create new drones and mate the colony with them
drones <- createDrones(x = basePop[1], nInd = 15)
all(isDrone(drones))
any(isFather(drones))
(matedColony <- cross(x = colony, drones = drones))
isQueenPresent(matedColony)

# Mate with drone producing colonies and a given cross plan
droneColonies <- createMultiColony(basePop[10:15])
droneColonies <- cross(droneColonies, drones = droneGroups[10:15])
nFathers(droneColonies)
apiary2 <- createMultiColony(basePop[16:20])
apiary3 <- createMultiColony(basePop[21:25])
apiary4 <- createMultiColony(basePop[26:30])

# Create a random cross plan
randomCrossPlan <- createCrossPlan(x = apiary2,
                                   droneColonies = droneColonies,
                                   nDrones = nFathersPoisson,
                                   spatial = FALSE)
apiary2 <- cross(x = apiary2,
                 droneColonies = droneColonies,
                 crossPlan = randomCrossPlan,
                 nDrones = 15)
nFathers(apiary2)

# Mate colonies according to a cross plan that is created internally within the cross function
apiary3 <- cross(x = apiary3,
                 droneColonies = droneColonies,
                 crossPlan = "create",
                 nDrones = 15)
nFathers(apiary3)

# Mate colonies according to a cross plan that is created internally within the cross function
# For this, all the colonies have to have a set location
droneColonies <- setLocation(droneColonies,
                             location = Map(c, runif(6, 0, 2*pi), runif(6, 0, 2*pi)))
apiary4 <- setLocation(apiary4,
                       location = Map(c, runif(5, 0, 2*pi), runif(5, 0, 2*pi)))
```

```
apiary4 <- cross(x = apiary4,
                 droneColonies = droneColonies,
                 crossPlan = "create",
                 nDrones = nFathersPoisson,
                 spatial = TRUE,
                 checkCross = "warning",
                 radius = 3)
nFathers(apiary4)
```

`downsize` *Reduce number of workers and remove all drones and virgin queens from a Colony or MultiColony object*

Description

Level 2 function that downsizes a Colony or MultiColony object by removing a proportion of workers, all drones and all virgin queens. Usually in the autumn, such an event occurs in preparation for the winter months.

Usage

```
downsize(x, p = NULL, use = "rand", new = FALSE, simParamBee = NULL, ...)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`p`	numeric, proportion of workers to be removed from the colony; if `NULL` then `SimParamBee$downsizeP` is used. If input is `MultiColony-class`, the input could also be a vector of the same length as the number of colonies. If a single value is provided, the same value will be applied to all the colonies
`use`	character, all the options provided by `selectInd`; it guides the selection of workers that will be removed
`new`	logical, should we remove all current workers and add a targeted proportion anew (say, create winter workers)
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	additional arguments passed to `p` when this argument is a function

Value

`Colony-class` or `MultiColony-class` with workers reduced and drones/virgin queens removed

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 100)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 3, nDrones = 12)

# Create and cross Colony and MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(colony)
apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(apiary)

# Downsize
colony <- downsize(x = colony, new = TRUE, use = "rand")
colony
apiary <- downsize(x = apiary, new = TRUE, use = "rand")
apiary[[1]]

# Downsize with different numbers
nWorkers(apiary); nDrones(apiary)
apiary <- downsize(x = apiary, p = c(0.5, 0.1), new = TRUE, use = "rand")
nWorkers(apiary); nDrones(apiary)
```

`downsizePUnif` *Sample the downsize proportion - proportion of removed workers in downsizing*

Description

Sample the downsize proportion - proportion of removed workers in downsizing - used when `p = NULL` (see `SimParamBee$downsizeP`).

This is just an example. You can provide your own functions that satisfy your needs!

Usage

```
downsizePUnif(colony, n = 1, min = 0.8, max = 0.9)
```

Arguments

`colony`	`Colony-class`
`n`	integer, number of samples
`min`	numeric, lower limit for `downsizePUnif`
`max`	numeric, upper limit for `downsizePUnif`

Value

numeric, downsize proportion

See Also

`SimParamBee` field `downsizeP`

Examples

```
downsizePUnif()
downsizePUnif()
p <- downsizePUnif(n = 1000)
hist(p, breaks = seq(from = 0, to = 1, by = 0.01), xlim = c(0, 1))
```

`editCsdLocus` *Edit the csd locus*

Description

Edits the csd locus in an entire population of individuals to ensure heterozygosity. The user can provide a list of csd alleles for each individual or, alternatively, the function samples a heterozygous genotype for each individual from all possible csd alleles. The gv slot is recalculated to reflect the any changes due to editing, but other slots remain the same.

Usage

```
editCsdLocus(pop, alleles = NULL, simParamBee = NULL)
```

Arguments

`pop`	`Pop-class`
`alleles`	`NULL` or list; If `NULL`, then the function samples a heterozygous csd genotype for each virgin queen from all possible csd alleles. If not `NULL`, the user provides a list of length `nInd` with each node holding a matrix or a data.frame, each having two rows and n columns. Each row must hold one csd haplotype (allele) that will be assigned to a virgin queen. The n columns span the length of the csd locus as specified in `SimParamBee`. The two csd alleles must be different to ensure heterozygosity at the csd locus.
`simParamBee`	global simulation parameters.

Value

Returns an object of `Pop-class`

---

getAa *Access epistasis values of individuals in a caste*

Description

Level 0 function that returns epistasis values of individuals in a caste.

Usage

```
getAa(x, caste = NULL, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getQueenAa(x, collapse = FALSE, simParamBee = NULL)

getFathersAa(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getVirginQueensAa(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getWorkersAa(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getDronesAa(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`collapse`	logical, if the return value should be a single matrix with epistatic values of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

vector of epistasis values when `x` is `Colony-class` and list of vectors of epistasis values when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Not exporting this function, since the theory behind it is not fully developed

Functions

- `getQueenAa()`: Access epistasis value of the queen
- `getFathersAa()`: Access epistasis values of fathers
- `getVirginQueensAa()`: Access epistasis values of virgin queens
- `getWorkersAa()`: Access epistasis values of workers
- `getDronesAa()`: Access epistasis values of drones

See Also

dd and `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

---

getBv *Access breeding values of individuals in a caste*

Description

Level 0 function that returns breeding values of individuals in a caste.

Usage

```
getBv(x, caste = NULL, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getQueenBv(x, collapse = FALSE, simParamBee = NULL)

getFathersBv(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getVirginQueensBv(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getWorkersBv(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getDronesBv(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`collapse`	logical, if the return value should be a single matrix with breeding valued of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

vector of breeding values when `x` is `Colony-class` and list of vectors of breeding values when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Not exporting this function, since the theory behind it is not fully developed

Functions

- `getQueenBv()`: Access breeding value of the queen
- `getFathersBv()`: Access breeding values of fathers
- `getVirginQueensBv()`: Access breeding values of virgin queens
- `getWorkersBv()`: Access breeding values of workers
- `getDronesBv()`: Access breeding values of drones

See Also

`bv` and `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

`getCaste` *Report caste of an individual*

Description

Level 0 function that reports caste of an individual

Usage

```
getCaste(x, collapse = FALSE, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`collapse`	logical, if `TRUE`, the function will return a single vector with caste information
`simParamBee`	`SimParamBee`, global simulation parameters

Value

When x is `Pop-class`, character of caste status; if you get `NA` note that this is not supposed to happen. When x is `Colony-class`, list with character vectors (list is named with caste). When x is `MultiColony-class`, list of lists with character vectors (list is named with colony id).

See Also

`getCastePop` and `getCasteId`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 20, nDrones = 5)
colony <- addVirginQueens(colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 10, nDrones = 2)
apiary <- addVirginQueens(apiary, nInd = 4)

getCaste(getQueen(colony))
getCaste(getFathers(colony))
getCaste(getWorkers(colony))
getCaste(getDrones(colony))
getCaste(getVirginQueens(colony))

bees <- c(
  getQueen(colony),
  getFathers(colony, nInd = 2),
  getWorkers(colony, nInd = 2),
  getDrones(colony, nInd = 2),
  getVirginQueens(colony, nInd = 2)
)
getCaste(bees)

getCaste(colony)
# Collapse information into a single vector
getCaste(colony, collapse = TRUE)
getCaste(apiary)

# Create a data.frame with id, colony, and caste information
(tmpC <- getCaste(apiary[[1]]))
(tmpI <- getCasteId(apiary[[1]]))
tmp <- data.frame(caste = unlist(tmpC), id = unlist(tmpI))
head(tmp)
tail(tmp)

(tmpC <- getCaste(apiary))
(tmpI <- getCasteId(apiary))
(tmp <- data.frame(caste = unlist(tmpC), id = unlist(tmpI)))
tmp$colony <- sapply(
  X = strsplit(
```

```
    x = rownames(tmp), split = ".",
    fixed = TRUE
  ),
  FUN = function(z) z[[1]]
)
head(tmp)
tail(tmp)
```

`getCasteId` *Get IDs of individuals of a caste, or ID of all members of colony*

Description

Level 0 function that returns the ID individuals of a caste. To get the individuals, use `getCastePop`. To get individuals' caste, use `getCaste`.

Usage

```
getCasteId(x, caste = "all", collapse = FALSE, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	character, "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`collapse`	logical, if all IDs should be returned as a single vector
`simParamBee`	`SimParamBee`, global simulation parameters

Value

when x is `Pop-class` for `caste != "all"` or list for `caste == "all"` with ID nodes named by caste; when x is `Colony-class` return is a named list of `Pop-class` for `caste != "all"` or named list for `caste == "all"` indluding caste members IDs; when x is `MultiColony-class` return is a named list of `Pop-class` for `caste != "all"` or named list of lists of `Pop-class` for `caste == "all"` indluding caste members IDs

See Also

`getCaste`

`getCastePop` and `getCaste`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
```

```
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 20, nDrones = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 10, nDrones = 2)
apiary <- addVirginQueens(apiary, nInd = 4)

getCasteId(x = drones)
getCasteId(x = colony)
getCasteId(x = apiary, caste = "workers")
getCasteId(x = apiary)
getCasteId(x = apiary, caste = "virginQueens")
# Get all IDs as a single vector
getCasteId(x = colony, caste = "all", collapse = TRUE)
getCasteId(x = apiary, caste = "workers", collapse = TRUE)
getCasteId(x = apiary, caste = "drones", collapse = TRUE)
getCasteId(x = apiary, caste = "all", collapse = TRUE)

# Create a data.frame with id, colony, and caste information
(tmpC <- getCaste(apiary[[1]]))
(tmpI <- getCasteId(apiary[[1]]))
tmp <- data.frame(caste = unlist(tmpC), id = unlist(tmpI))
head(tmp)
tail(tmp)

(tmpC <- getCaste(apiary))
(tmpI <- getCasteId(apiary))
(tmp <- data.frame(caste = unlist(tmpC), id = unlist(tmpI)))
tmp$colony <- sapply(
  X = strsplit(
    x = rownames(tmp), split = ".",
    fixed = TRUE
  ),
  FUN = function(z) z[[1]]
)
head(tmp)
tail(tmp)
```

`getCastePop` *Access individuals of a caste*

Description

Level 1 function that returns individuals of a caste. These individuals stay in the colony (compared to `pullCastePop`).

Usage

```
getCastePop(
  x,
  caste = "all",
  nInd = NULL,
  use = "rand",
  removeFathers = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getQueen(x, collapse = FALSE, simParamBee = NULL)

getFathers(x, nInd = NULL, use = "rand", collapse = FALSE, simParamBee = NULL)

getWorkers(x, nInd = NULL, use = "rand", collapse = FALSE, simParamBee = NULL)

getDrones(
  x,
  nInd = NULL,
  use = "rand",
  removeFathers = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getVirginQueens(
  x,
  nInd = NULL,
  use = "rand",
  collapse = FALSE,
  simParamBee = NULL
)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`, exceptionally `Pop-class` for calling `getFathers` on a queen population
`caste`	character, "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed; if there are less individuals than requested, we return the ones available - this can return `NULL`. If input is `MultiColony-class`, the input could also be a vector of the same length as the number of colonies. If a single value is provided, the same value will be applied to all the colonies.
`use`	character, all options provided by `selectInd` and `"order"` that selects `1:nInd` individuals (meaning it always returns at least one individual, even if `nInd = 0`)

`removeFathers`	logical, removes `drones` that have already mated; set to `FALSE` if you would like to get drones for mating with multiple virgin queens, say via insemination
`collapse`	logical, whether to return a single merged population
`simParamBee`	`SimParamBee`, global simulation parameters

Value

when `x` is `Colony-class` return is `Pop-class` for `caste != "all"` or list for `caste == "all"` with nodes named by caste; when `x` is `MultiColony-class` return is a named list of `Pop-class` for `caste != "all"` or named list of lists of `Pop-class` for `caste == "all"`. You can merge all the populations in the list with `mergePops` function.

Functions

- `getQueen()`: Access the queen
- `getFathers()`: Access fathers (drones the queen mated with)
- `getWorkers()`: Access workers
- `getDrones()`: Access drones
- `getVirginQueens()`: Access virgin queens

See Also

`getQueen`, `getFathers`, `getVirginQueens`, `getWorkers`, and `getDrones`

`getCasteId` and `getCaste`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])

# Build-up and add virgin queens
colony <- buildUp(x = colony)
apiary <- buildUp(x = apiary)
colony <- addVirginQueens(x = colony)
apiary <- addVirginQueens(x = apiary)

# Get the queen of the colony
getCastePop(colony, caste = "queen")
```

```
getQueen(colony)

# Comparison of getCastePop() and getWorkers()
getCastePop(colony, caste = "workers")
getCastePop(colony, caste = "workers")
getCastePop(colony, caste = "workers", nInd = 2)
# Or aliases
getWorkers(colony)
# Same aliases exist for all the castes!

# Input is a MultiColony class - same behaviour as for the Colony!
getCastePop(apiary, caste = "queen")
# Or alias
getQueen(apiary)

# Sample individuals from all the castes
getCastePop(colony, nInd = 5, caste = "all")

# Get different number of workers per colony
getCastePop(apiary, caste = "workers", nInd = c(10, 20))
# Or alias
getWorkers(apiary, nInd = c(10, 20))

# Obtain individuals from MultiColony as a single population
getCastePop(apiary, caste = "queen", collapse = TRUE)
getQueen(apiary, collapse = TRUE)
getWorkers(apiary, nInd = 10, collapse = TRUE)
getDrones(apiary, nInd = 3, collapse = TRUE)
```

`getCasteSex` *Get sex of individuals of a caste, or sex of all members of colony*

Description

Level 0 function that returns the sex individuals of a caste. To get the individuals, use `getCastePop`. To get individuals' caste, use `getCaste`.

Usage

```
getCasteSex(x, caste = "all", collapse = FALSE, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	character, "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`collapse`	logical, if `TRUE`, the function will return a single vector with sex information
`simParamBee`	`SimParamBee`, global simulation parameters

Value

when `x` is `Pop-class` for `caste != "all"` or list for `caste == "all"` with sex nodes named by caste; when `x` is `Colony-class` return is a named list of `Pop-class` for `caste != "all"` or named list for `caste == "all"` indluding caste members sexes; when `x` is `MultiColony-class` return is a named list of `Pop-class` for `caste != "all"` or named list of lists of `Pop-class` for `caste == "all"` indluding caste members sexes

See Also

`getCaste`

`getCastePop` and `getCaste`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 20, nDrones = 5)
colony <- addVirginQueens(colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 10, nDrones = 2)
apiary <- addVirginQueens(apiary, nInd = 4)

getCasteSex(x = drones)
getCasteSex(x = colony)
getCasteSex(x = apiary, caste = "workers")
getCasteSex(x = apiary)
getCasteSex(x = apiary, caste = "virginQueens")
# Collapse information into a single vector
getCasteSex(colony, caste = "all", collapse = TRUE)

# Create a data.frame with sex, colony, and caste information
(tmpC <- getCaste(apiary[[1]]))
(tmpS <- getCasteSex(apiary[[1]]))
(tmpI <- getCasteId(apiary[[1]]))
tmp <- data.frame(caste = unlist(tmpC), sex = unlist(tmpS), id = unlist(tmpI))
head(tmp)
tail(tmp)

(tmpC <- getCaste(apiary))
(tmpS <- getCasteSex(apiary))
```

```
(tmpI <- getCasteId(apiary))
tmp <- data.frame(caste = unlist(tmpC), sex = unlist(tmpS), id = unlist(tmpI))
tmp$colony <- sapply(
  X = strsplit(
    x = rownames(tmp), split = ".",
    fixed = TRUE
  ),
  FUN = function(z) z[[1]]
)
head(tmp)
tail(tmp)
```

`getCsdAlleles` *Get csd alleles*

Description

Level 0 function that returns alleles from the csd locus. See `SimParamBee` for more information about the csd locus.

Usage

```
getCsdAlleles(
  x,
  caste = NULL,
  nInd = NULL,
  allele = "all",
  dronesHaploid = TRUE,
  collapse = FALSE,
  unique = FALSE,
  simParamBee = NULL
)

getQueenCsdAlleles(
  x,
  allele = "all",
  unique = FALSE,
  collapse = FALSE,
  simParamBee = NULL
)

getFathersCsdAlleles(
  x,
  nInd = NULL,
  allele = "all",
  dronesHaploid = TRUE,
  unique = FALSE,
  collapse = FALSE,
```

```
  simParamBee = NULL
)

getVirginQueensCsdAlleles(
  x,
  nInd = NULL,
  allele = "all",
  unique = FALSE,
  collapse = FALSE,
  simParamBee = NULL
)

getWorkersCsdAlleles(
  x,
  nInd = NULL,
  allele = "all",
  unique = FALSE,
  collapse = FALSE,
  simParamBee = NULL
)

getDronesCsdAlleles(
  x,
  nInd = NULL,
  allele = "all",
  dronesHaploid = TRUE,
  unique = FALSE,
  collapse = FALSE,
  simParamBee = NULL
)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, for how many individuals; if `NULL` all individuals are taken; this can be useful as a test of sampling individuals
`allele`	character, either "all" for both alleles or an integer for a single allele, use a value of 1 for female allele and a value of 2 for male allele
`dronesHaploid`	logical, return haploid result for drones?
`collapse`	logical, if `TRUE`, the function will return a set of csd alleles across the entire population, colony, or multicolony (not separately for each caste when `x` is a colony or each caste of each colony when `x` is a multicolony. This is a way to get one single object as an output across castes or colonies. Note this has nothing to do with the colony collapse. It's like `paste(..., collapse = TRUE)`. Default is `FALSE`. See examples about this behaviour.

`unique`	logical, return only the unique set of csd alleles. This argument interacts with `collapse`. Default is `FALSE`. See examples about this behaviour.
`simParamBee`	`SimParamBee`, global simulation parameters

Details

If both collapse and unique are `TRUE`, the function returns a unique set of csd alleles in the entire population, colony, or multicolony

Value

matrix with haplotypes when `x` is `Pop-class`, list of matrices with haplotypes when `x` is `Colony-class` (list nodes named by caste) and list of a list of matrices with haplotypes when `x` is `MultiColony-class`, outer list is named by colony id when `x` is `MultiColony-class`; `NULL` when `x` is `NULL`

Functions

- `getQueenCsdAlleles()`: Access csd alleles of the queen
- `getFathersCsdAlleles()`: Access csd alleles of the fathers
- `getVirginQueensCsdAlleles()`: Access csd alleles of the virgin queens
- `getWorkersCsdAlleles()`: Access csd alleles of the workers
- `getDronesCsdAlleles()`: Access csd alleles of the drones

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes, nCsdAlleles = 5)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)

# Use getCsdAlleles on a Population
getCsdAlleles(getQueen(colony))
getCsdAlleles(getWorkers(colony))

# Use getCsdAlleles on a Colony
getCsdAlleles(colony)
getCsdAlleles(colony, caste = "queen")
getQueenCsdAlleles(colony)
```

```
getCsdAlleles(colony, caste = "workers")
getWorkersCsdAlleles(colony)
# Same aliases exist for all the castes!

getCsdAlleles(colony, unique = TRUE)
getCsdAlleles(colony, collapse = TRUE)
getCsdAlleles(colony, collapse = TRUE, unique = TRUE)

# Use getCsdAlleles on a MultiColony
getCsdAlleles(apiary)
getCsdAlleles(apiary, unique = TRUE)
getCsdAlleles(apiary, collapse = TRUE, unique = TRUE)
getCsdAlleles(apiary, nInd = 2)
```

`getCsdGeno` *Get genotypes from the csd locus*

Description

Level 0 function that returns genotypes from the csd locus. See `SimParamBee` for more information about the csd locus and how we have implemented it.

Usage

```
getCsdGeno(
  x,
  caste = NULL,
  nInd = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getQueenCsdGeno(x, collapse = FALSE, simParamBee = NULL)

getFathersCsdGeno(
  x,
  nInd = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getVirginQueensCsdGeno(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getWorkersCsdGeno(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getDronesCsdGeno(
```

```
    x,
    nInd = NULL,
    dronesHaploid = TRUE,
    collapse = FALSE,
    simParamBee = NULL
  )
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, for how many individuals; if `NULL` all individuals are taken; this can be useful as a test of sampling individuals
`dronesHaploid`	logical, return haploid result for drones?
`collapse`	logical, if the return value should be a single matrix with haplotypes of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Details

The returned genotypes are spanning multiple bi-allelic SNP of a non-recombining csd locus / haplotype. In most cases you will want to use `getCsdAlleles`.

Value

matrix with genotypes when `x` is `Pop-class`, list of matrices with genotypes when `x` is `Colony-class` (list nodes named by caste) and list of a list of matrices with genotypes when `x` is `MultiColony-class`, outer list is named by colony id when `x` is `MultiColony-class`; `NULL` when `x` is `NULL`

Functions

- `getQueenCsdGeno()`: Access csd genotypes of the queen
- `getFathersCsdGeno()`: Access csd genotypes of the fathers
- `getVirginQueensCsdGeno()`: Access csd genotypes of the virgin queens
- `getWorkersCsdGeno()`: Access csd genotypes of the virgin queens
- `getDronesCsdGeno()`: Access csd genotypes of the virgin queens

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
```

```
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 4)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

# Use getCsdGeno on a Population
getCsdGeno(getQueen(colony))
getCsdGeno(getWorkers(colony))

# Using dronesHaploid = TRUE returns drones as haploids instead of double haploids
getCsdGeno(getDrones(colony), nInd = 3, dronesHaploid = TRUE)
# Using dronesHaploid = FALSE returns drones as double haploids
getCsdGeno(getDrones(colony), nInd = 3, dronesHaploid = FALSE)

# Use getCsdGeno on a Colony
getCsdGeno(colony)
getCsdGeno(colony, caste = "queen")
getQueenCsdGeno(colony)
getCsdGeno(colony, caste = "workers")
getWorkersCsdGeno(colony)
# Same aliases exist for all the castes!

# Use getCsdGeno on a MultiColony - same behaviour as for the Colony!
getCsdGeno(apiary)
getCsdGeno(apiary, nInd = 2)
```

`getDd` *Access dominance values of individuals in a caste*

Description

Level 0 function that returns dominance values of individuals in a caste.

Usage

```
getDd(x, caste = NULL, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getQueenDd(x, collapse = FALSE, simParamBee = NULL)

getFathersDd(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getVirginQueensDd(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)
```

```
getWorkersDd(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getDronesDd(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`collapse`	logical, if the return value should be a single matrix with dominance values of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

vector of dominance values when `x` is `Colony-class` and list of vectors of dominance values when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Not exporting this function, since the theory behind it is not fully developed

Functions

- `getQueenDd()`: Access dominance value of the queen
- `getFathersDd()`: Access dominance values of fathers
- `getVirginQueensDd()`: Access dominance values of virgin queens
- `getWorkersDd()`: Access dominance values of workers
- `getDronesDd()`: Access dominance values of drones

See Also

`dd` and `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

`getEvents` *Report which colony events have occurred*

Description

Level 0 function that returns a matrix of logicals reporting the status of the colony events. The events are: split, swarm, supersedure, collapse, and production. These events impact colony status, strength, and could also impact downstream phenotypes.

Usage

```
getEvents(x)
```

Arguments

x	Colony-class or MultiColony-class

Value

matrix of logicals, named by colony id when x is MultiColony-class

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(apiary, nInd = 4)

getEvents(colony)
getEvents(apiary)

tmp <- swarm(colony)
getEvents(tmp$swarm)
getEvents(tmp$remnant)

apiary <- supersede(apiary)
getEvents(apiary)
```

`getGv` *Access genetic values of individuals in a caste*

Description

Level 0 function that returns genetic values of individuals in a caste.

Usage

```
getGv(x, caste = NULL, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getQueenGv(x, collapse = FALSE, simParamBee = NULL)

getFathersGv(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getVirginQueensGv(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getWorkersGv(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getDronesGv(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`collapse`	logical, if the return value should be a single matrix with genetic values of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

vector of phenotype values when `x` is `Colony-class` and list of vectors of genetic values when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Functions

- `getQueenGv()`: Access genetic value of the queen
- `getFathersGv()`: Access genetic values of fathers
- `getVirginQueensGv()`: Access genetic values of virgin queens
- `getWorkersGv()`: Access genetic values of workers
- `getDronesGv()`: Access genetic values of drones

See Also

`gv` and `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

SP$addTraitA(nQtlPerChr = 10, var = 1)
SP$addSnpChip(5)
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

# Input is a population
getGv(x = getQueen(colony))
queens <- getQueen(apiary, collapse = TRUE)
getGv(queens)

# Input is a colony
getGv(colony, caste = "queen")
getQueenGv(colony)

getGv(colony, caste = "workers")
getWorkersGv(colony)
# Same aliases exist for all the castes!

# Get genetic values for all individuals
getGv(colony, caste = "all")
# Get all genetic values in a single matrix
getGv(colony, caste = "all", collapse = TRUE)

# Input is a MultiColony - same behaviour as for the Colony!
getGv(apiary, caste = "queen")
getQueenGv(apiary)

# Get the genetic values of all individuals either by colony or in a single matrix
getGv(apiary, caste = "all")
getGv(apiary, caste = "all", collapse = TRUE)
```

`getIbdHaplo` *Access IBD haplotypes of individuals in a caste*

Description

Level 0 function that returns IBD (identity by descent) haplotypes of individuals in a caste.

Usage

```
getIbdHaplo(
  x,
  caste = NULL,
  nInd = NULL,
  chr = NULL,
  snpChip = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getQueenIbdHaplo(
  x,
  chr = NULL,
  snpChip = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getFathersIbdHaplo(
  x,
  nInd = NULL,
  chr = NULL,
  snpChip = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getVirginQueensIbdHaplo(
  x,
  nInd = NULL,
  chr = NULL,
  snpChip = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getWorkersIbdHaplo(
  x,
  nInd = NULL,
  chr = NULL,
  snpChip = NULL,
```

```
  collapse = FALSE,
  simParamBee = NULL
)

getDronesIbdHaplo(
  x,
  nInd = NULL,
  chr = NULL,
  snpChip = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`chr`	numeric, chromosomes to retrieve, if `NULL`, all chromosome are retrieved
`snpChip`	integer, indicating which SNP array loci are to be retrieved, if `NULL`, all sites are retrieved
`dronesHaploid`	logical, return haploid result for drones?
`collapse`	logical, if the return value should be a single matrix with haplotypes of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

matrix with haplotypes when `x` is `Colony-class` and list of matrices with haplotypes when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Functions

- `getQueenIbdHaplo()`: Access IBD haplotype data of the queen
- `getFathersIbdHaplo()`: Access IBD haplotype data of fathers
- `getVirginQueensIbdHaplo()`: Access IBD haplotype data of virgin queens
- `getWorkersIbdHaplo()`: Access IBD haplotype data of workers
- `getDronesIbdHaplo()`: Access IBD haplotype data of drones

See Also

`getIbdHaplo` and `pullIbdHaplo`

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

SP$setTrackRec(TRUE)
SP$setTrackPed(isTrackPed = TRUE)
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 200)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

# Input is a population
getIbdHaplo(x = getQueen(colony))
queens <- getQueen(apiary, collapse = TRUE)
getIbdHaplo(queens)

# Input is a colony
getIbdHaplo(x = colony, caste = "queen")
getQueenIbdHaplo(colony)

getIbdHaplo(colony, caste = "workers", nInd = 3)
getWorkersIbdHaplo(colony)
# Same aliases exist for all castes!

# Get haplotypes for all individuals
getIbdHaplo(colony, caste = "all")
# Get all haplotypes in a single matrix
getIbdHaplo(colony, caste = "all", collapse = TRUE)

# Input is a MultiColony
getIbdHaplo(x = apiary, caste = "queen")
getQueenIbdHaplo(apiary)
# Or collapse all the haplotypes into a single matrix
getQueenIbdHaplo(apiary, collapse = TRUE)

# Get the haplotypes of all individuals either by colony or in a single matrix
getIbdHaplo(apiary, caste = "all")
getIbdHaplo(apiary, caste = "all", collapse = TRUE)
```

`getId` *Get the colony ID*

Description

Level 0 function that returns the colony ID. This is by definition the ID of the queen.

Usage

```
getId(x)
```

Arguments

x	`Pop-class`, `Colony-class`, or `MultiColony-class`

Value

character, `NA` when queen not present

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])

getId(getQueen(colony)) # Pop class
getId(colony) # Colony Class
getId(apiary) # MultiColony Class

colony2 <- removeQueen(colony)
getId(colony2)
```

`getLocation` *Get the colony location*

Description

Level 0 function that returns the colony location as (x, y) coordinates.

Usage

```
getLocation(x, collapse = FALSE)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`collapse`	logical, if the return value should be a single matrix with locations of all the colonies; only applicable when input is a `MultiColony-class` object

Value

numeric with two values when `x` is `Colony-class` and a list of numeric with two values when `x` is `MultiColony-class` (list named after colonies); `c(NA, NA)` when location not set

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])

getLocation(colony)
getLocation(apiary[[1]])
getLocation(apiary)
getLocation(apiary, collapse = TRUE)

loc <- c(123, 456)
colony <- setLocation(colony, location = loc)
getLocation(colony)

loc1 <- c(512, 722)
```

```
colony1 <- setLocation(apiary[[1]], location = loc1)
getLocation(colony1)

loc2 <- c(189, 357)
colony2 <- setLocation(apiary[[2]], location = loc2)
getLocation(colony2)

getLocation(c(colony1, colony2))

# Assuming one location (as in bringing colonies to an apiary at a location!)
apiary <- setLocation(apiary, location = loc1)
getLocation(apiary)

# Assuming different locations (so tmp is not an apiary in one location!)
tmp <- setLocation(c(colony1, colony2), location = list(loc1, loc2))
getLocation(tmp)
```

`getMisc` *Get miscellaneous information in a population*

Description

Get miscellaneous information in a population

Usage

```
getMisc(x, node = NULL)
```

Arguments

`x`	`Pop-class`
`node`	character, name of the node to get from the `x@misc` slot; if `NULL` the whole `x@misc` slot is returned

Value

The `x@misc` slot or its nodes `x@misc[[*]][[node]]`

getPheno *Access phenotype values of individuals in a caste*

Description

Level 0 function that returns phenotype values of individuals in a caste.

Usage

```
getPheno(x, caste = NULL, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getQueenPheno(x, collapse = FALSE, simParamBee = NULL)

getFathersPheno(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getVirginQueensPheno(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getWorkersPheno(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)

getDronesPheno(x, nInd = NULL, collapse = FALSE, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`collapse`	logical, if the return value should be a single matrix with phenotypes of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

vector of genetic values when `x` is `Colony-class` and list of vectors of genetic values when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Functions

- `getQueenPheno()`: Access phenotype value of the queen
- `getFathersPheno()`: Access phenotype values of fathers
- `getVirginQueensPheno()`: Access phenotype values of virgin queens
- `getWorkersPheno()`: Access phenotype values of workers
- `getDronesPheno()`: Access phenotype values of drones

See Also

pheno and `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

SP$addTraitA(nQtlPerChr = 10, var = 1)
SP$setVarE(varE = 1)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

# Input is a population
getPheno(x = getQueen(colony))
queens <- getQueen(apiary, collapse = TRUE)
getPheno(queens)

# Input is a colony
getPheno(colony, caste = "queen")
getQueenPheno(colony)

getPheno(colony, caste = "fathers")
getPheno(colony, caste = "fathers", nInd = 2)
getPheno(colony, caste = "fathers", nInd = 2) # random sample!
getFathersPheno(colony)
getFathersPheno(colony, nInd = 2)

getPheno(colony, caste = "workers")
getWorkersPheno(colony)
# Same aliases exist for all the castes!!!

# Get phenotypes for all individuals
getPheno(colony, caste = "all")
# Get all phenotypes in a single matrix
getPheno(colony, caste = "all", collapse = TRUE)

# Input is a MultiColony - same behaviour as for the Colony!
```

```
getPheno(apiary, caste = "queen")
getQueenPheno(apiary)

# Get the phenotypes of all individuals either by colony or in a single matrix
getPheno(apiary, caste = "all")
getPheno(apiary, caste = "all", collapse = TRUE)
```

`getPooledGeno` *Get a pooled genotype from true genotypes*

Description

Level 0 function that returns a pooled genotype from true genotypes to mimic genotyping of a pool of colony members.

Usage

```
getPooledGeno(x, type = NULL, sex = NULL)
```

Arguments

`x`	matrix, true genotypes with individuals in rows and sites in columns
`type`	character, "mean" for average genotype or "count" for the counts of reference and alternative alleles
`sex`	character, vector of "F" and "M" to denote the sex of individuals in `x`

Value

a numeric vector with average allele dosage when `type = "mean"` and a two-row matrix with the counts of reference (1st row) and alternative (2nd row) alleles

Examples

```
founderGenomes <- quickHaplo(nInd = 3, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)


basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)
apiary <- createMultiColony(basePop[2:3], n = 2)
apiary <- cross(x = apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

genoQ <- getQueenSegSiteGeno(apiary[[1]])
genoF <- getFathersSegSiteGeno(apiary[[1]])
genoW <- getWorkersSegSiteGeno(apiary[[1]])
genoD <- getDronesSegSiteGeno(apiary[[1]])
```

```
genoV <- getVirginQueensSegSiteGeno(apiary[[1]])

# Pool of drones
sexD <- getCasteSex(apiary[[1]], caste = "drones")
getPooledGeno(x = genoD, type = "count", sex = sexD)[, 1:10]
(poolD <- getPooledGeno(x = genoD, type = "mean", sex = sexD))[, 1:10]
# ... compare to queen's genotype
genoQ[, 1:10]
plot(
  y = poolD, x = genoQ, ylim = c(0, 2), xlim = c(0, 2),
  ylab = "Average allele dosage in drones",
  xlab = "Allele dosage in the queen"
)

# As an exercise you could repeat the above with different numbers of drones!

# Pool of workers
getPooledGeno(x = genoW, type = "count")[, 1:10]
(poolW <- getPooledGeno(x = genoW, type = "mean"))[, 1:10]
# ... compare to fathers' and queen's avearage genotype
sexF <- getCasteSex(apiary[[1]], caste = "fathers")
sexQ <- rep(x = "F", times = nrow(genoF))
sexFQ <- c(sexF, sexQ)
genoFQ <- rbind(genoF, genoQ[rep(x = 1, times = nrow(genoF)), ])
(poolFQ <- getPooledGeno(x = genoFQ, type = "mean", sex = sexFQ))[, 1:10]
plot(
  y = poolW, x = poolFQ, ylim = c(0, 2), xlim = c(0, 2),
  ylab = "Average allele dosage in workers",
  xlab = "Average allele dosage in the queen and fathers"
)

# As an exercise you could repeat the above with different numbers of workers!
```

`getQtlGeno` *Access QTL genotypes of individuals in a caste*

Description

Level 0 function that returns QTL genotypes of individuals in a caste.

Usage

```
getQtlGeno(
  x,
  caste = NULL,
  nInd = NULL,
  trait = 1,
  chr = NULL,
  dronesHaploid = TRUE,
```

```
  collapse = FALSE,
  simParamBee = NULL
)

getQueenQtlGeno(x, trait = 1, chr = NULL, collapse = FALSE, simParamBee = NULL)

getFathersQtlGeno(
  x,
  nInd = NULL,
  trait = 1,
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getVirginQueensQtlGeno(
  x,
  nInd = NULL,
  trait = 1,
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getWorkersQtlGeno(
  x,
  nInd = NULL,
  trait = 1,
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getDronesQtlGeno(
  x,
  nInd = NULL,
  trait = 1,
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is

	a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`trait`	numeric (trait position) or character (trait name), indicates which trait's QTL genotypes to retrieve
`chr`	numeric, chromosomes to retrieve, if `NULL`, all chromosome are retrieved
`dronesHaploid`	logical, return haploid result for drones?
`collapse`	logical, if the return value should be a single matrix with genotypes of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

matrix with genotypes when `x` is `Colony-class` and list of matrices with genotypes when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Functions

- `getQueenQtlGeno()`: Access QTL genotype data of the queen
- `getFathersQtlGeno()`: Access QTL genotype data of fathers
- `getVirginQueensQtlGeno()`: Access QTL genotype data of virgin queens
- `getWorkersQtlGeno()`: Access QTL genotype data of workers
- `getDronesQtlGeno()`: Access QTL genotype data of drones

See Also

`getQtlGeno` and `pullQtlGeno` as well as `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

SP$addTraitA(nQtlPerChr = 10)
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 200)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
```

```
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

# Input is a population
getQtlGeno(x = getQueen(colony))
queens <- getQueen(apiary, collapse = TRUE)
getQtlGeno(queens)

# Input is a colony
getQtlGeno(colony, caste = "queen")
getQueenQtlGeno(colony)

getQtlGeno(colony, caste = "workers", nInd = 3)
getWorkersQtlGeno(colony)
# Same aliases exist for all the castes!

# Get genotypes for all individuals
getQtlGeno(colony, caste = "all")
# Get all haplotypes in a single matrix
getQtlGeno(colony, caste = "all", collapse = TRUE)

# Input is a MultiColony - same behaviour as for the Colony!
getQtlGeno(apiary, caste = "queen")
getQueenQtlGeno(apiary)

# Get the genotypes of all individuals either by colony or in a single matrix
getQtlGeno(apiary, caste = "all")
getQtlGeno(apiary, caste = "all", collapse = TRUE)
```

`getQtlHaplo` *Access QTL haplotypes of individuals in a caste*

Description

Level 0 function that returns QTL haplotypes of individuals in a caste.

Usage

```
getQtlHaplo(
  x,
  caste = NULL,
  nInd = NULL,
  trait = 1,
  haplo = "all",
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
```

```
)

getQueenQtlHaplo(
  x,
  trait = 1,
  haplo = "all",
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getFathersQtlHaplo(
  x,
  nInd = NULL,
  trait = 1,
  haplo = "all",
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getVirginQueensQtlHaplo(
  x,
  nInd = NULL,
  trait = 1,
  haplo = "all",
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getWorkersQtlHaplo(
  x,
  nInd = NULL,
  trait = 1,
  haplo = "all",
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getDronesQtlHaplo(
  x,
  nInd = NULL,
  trait = 1,
  haplo = "all",
  chr = NULL,
```

```
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`trait`	numeric (trait position) or character (trait name), indicates which trait's QTL haplotypes to retrieve
`haplo`	character, either "all" for all haplotypes or an integer for a single set of haplotypes, use a value of 1 for female haplotypes and a value of 2 for male haplotypes
`chr`	numeric, chromosomes to retrieve, if `NULL`, all chromosome are retrieved
`dronesHaploid`	logical, return haploid result for drones?
`collapse`	logical, if the return value should be a single matrix with haplotypes of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

matrix with haplotypes when `x` is `Colony-class` and list of matrices with haplotypes when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Functions

- `getQueenQtlHaplo()`: Access QTL haplotype data of the queen
- `getFathersQtlHaplo()`: Access QTL haplotype data of fathers
- `getVirginQueensQtlHaplo()`: Access QTL haplotype data of virgin queens
- `getWorkersQtlHaplo()`: Access QTL haplotype of workers
- `getDronesQtlHaplo()`: Access QTL haplotype data of drones

See Also

`getQtlHaplo` and `pullQtlHaplo` as well as `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

SP$addTraitA(nQtlPerChr = 10)
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 200)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

# Input is a population
getQtlHaplo(x = getQueen(colony))
queens <- getQueen(apiary, collapse = TRUE)
getQtlHaplo(queens)

# Input is a Colony
getQtlHaplo(colony, caste = "queen")
getQueenQtlHaplo(colony)

getQtlHaplo(colony, caste = "workers", nInd = 3)
getWorkersQtlHaplo(colony)
# Same aliases exist for all the castes!

# Get haplotypes for all individuals
getQtlHaplo(colony, caste = "all")
# Get all haplotypes in a single matrix
getQtlHaplo(colony, caste = "all", collapse = TRUE)

# Input is a MultiColony - same behaviour as for the Colony
getQtlHaplo(apiary, caste = "queen")
getQueenQtlHaplo(apiary)

# Get the haplotypes of all individuals either by colony or in a single matrix
getQtlHaplo(apiary, caste = "all")
getQtlHaplo(apiary, caste = "all", collapse = TRUE)
```

`getQueenAge` *Get (calculate) the queen's age*

Description

Level 0 function that returns the queen's age.

Usage

```
getQueenAge(x, currentYear, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`currentYear`	integer, current year
`simParamBee`	`SimParamBee`, global simulation parameters

Value

numeric, the age of the queen(s); named when theres is more than one queen; `NA` if queen not present

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])

queen <- getQueen(colony)
queen <- setQueensYearOfBirth(queen, year = 2020)
getQueenAge(queen, currentYear = 2022)

colony <- setQueensYearOfBirth(colony, year = 2021)
getQueenAge(colony, currentYear = 2022)

apiary <- setQueensYearOfBirth(apiary, year = 2018)
getQueenAge(apiary, currentYear = 2022)
```

`getQueenYearOfBirth` *Access the queen's year of birth*

Description

Level 0 function that returns the queen's year of birth.

Usage

```
getQueenYearOfBirth(x, simParamBee = NULL)
```

Arguments

`x`	`Pop-class` (one or more than one queen), `Colony-class` (one colony), or `MultiColony-class` (more colonies)
`simParamBee`	`SimParamBee`, global simulation parameters

Value

numeric, the year of birth of the queen(s); named when theres is more than one queen; `NA` if queen not present

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])

queen <- getQueen(colony)
queen <- setQueensYearOfBirth(queen, year = 2022)
getQueenYearOfBirth(queen)

getQueenYearOfBirth(getQueen(colony))
colony <- setQueensYearOfBirth(colony, year = 2030)
getQueenYearOfBirth(colony)

apiary <- setQueensYearOfBirth(apiary, year = 2022)
getQueenYearOfBirth(apiary)
```

`getSegSiteGeno` *Access genotypes for all segregating sites of individuals in a caste*

Description

Level 0 function that returns genotypes for all segregating sites of individuals in a caste.

Usage

```
getSegSiteGeno(
  x,
  caste = NULL,
  nInd = NULL,
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getQueenSegSiteGeno(x, chr = NULL, collapse = FALSE, simParamBee = NULL)

getFathersSegSiteGeno(
  x,
  nInd = NULL,
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getVirginQueensSegSiteGeno(
  x,
  nInd = NULL,
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getWorkersSegSiteGeno(
  x,
  nInd = NULL,
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getDronesSegSiteGeno(
```

```
  x,
  nInd = NULL,
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`chr`	numeric, chromosomes to retrieve, if `NULL`, all chromosome are retrieved
`dronesHaploid`	logical, return haploid result for drones?
`collapse`	logical, if the return value should be a single matrix with genotypes of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

matrix with genotypes when `x` is `Colony-class` and list of matrices with genotypes when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Functions

- `getQueenSegSiteGeno()`: Access genotype data for all segregating sites of the queen
- `getFathersSegSiteGeno()`: Access genotype data for all segregating sites of fathers
- `getVirginQueensSegSiteGeno()`: Access genotype data for all segregating sites of virgin queens
- `getWorkersSegSiteGeno()`: Access genotype data for all segregating sites of workers
- `getDronesSegSiteGeno()`: Access genotype data for all segregating sites of drones

See Also

`getSegSiteGeno` and `pullSegSiteGeno`

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
```

```
drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

# Input is a population
getSegSiteGeno(x = getQueen(colony))
queens <- getQueen(apiary, collapse = TRUE)
getSegSiteGeno(queens)

# Input is a colony
getSegSiteGeno(colony, caste = "queen")
getQueenSegSiteGeno(colony)

getSegSiteGeno(colony, caste = "workers", nInd = 3)
getWorkersSegSiteGeno(colony)
# same aliases exist for all the castes!

# Get genotypes for all individuals
getSegSiteGeno(colony, caste = "all")
# Get all genotypes in a single matrix
getSegSiteGeno(colony, caste = "all", collapse = TRUE)

# Input is a MultiColony - same behaviour as for the Colony
getSegSiteGeno(apiary, caste = "queen")
getQueenSegSiteGeno(apiary)

# Get the genotypes of all individuals either by colony or in a single matrix
getSegSiteGeno(apiary, caste = "all")
getSegSiteGeno(apiary, caste = "all", collapse = TRUE)
```

`getSegSiteHaplo` *Access haplotypes for all segregating sites of individuals in a caste*

Description

Level 0 function that returns haplotypes for all segregating sites of individuals in a caste.

Usage

```
getSegSiteHaplo(
```

```
  x,
  caste = NULL,
  nInd = NULL,
  haplo = "all",
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getQueenSegSiteHaplo(
  x,
  haplo = "all",
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getFathersSegSiteHaplo(
  x,
  nInd = NULL,
  haplo = "all",
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getVirginQueensSegSiteHaplo(
  x,
  nInd = NULL,
  haplo = "all",
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getWorkersSegSiteHaplo(
  x,
  nInd = NULL,
  haplo = "all",
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getDronesSegSiteHaplo(
  x,
```

```
  nInd = NULL,
  haplo = "all",
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`haplo`	character, either "all" for all haplotypes or an integer for a single set of haplotypes, use a value of 1 for female haplotypes and a value of 2 for male haplotypes
`chr`	numeric, chromosomes to retrieve, if `NULL`, all chromosome are retrieved
`dronesHaploid`	logical, return haploid result for drones?
`collapse`	logical, if the return value should be a single matrix with haplotypes of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

matrix with haplotypes when `x` is `Colony-class` and list of matrices with haplotypes when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Functions

- `getQueenSegSiteHaplo()`: Access haplotype data for all segregating sites of the queen
- `getFathersSegSiteHaplo()`: Access haplotype data for all segregating sites of fathers
- `getVirginQueensSegSiteHaplo()`: Access haplotype data for all segregating sites of virgin queens
- `getWorkersSegSiteHaplo()`: Access haplotype data for all segregating sites of workers
- `getDronesSegSiteHaplo()`: Access haplotype data for all segregating sites of drones

See Also

`getSegSiteHaplo` and `pullSegSiteHaplo`

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

# Input is a population
getSegSiteHaplo(x = getQueen(colony))
queens <- getQueen(apiary, collapse = TRUE)
getSegSiteHaplo(queens)

# Input is a colony
getSegSiteHaplo(colony, caste = "queen")
getQueenSegSiteHaplo(colony)

getSegSiteHaplo(colony, caste = "workers", nInd = 3)
getWorkersSegSiteHaplo(colony)
#Same aliases exist for all the castes!

# Get haplotypes for all individuals
getSegSiteHaplo(colony, caste = "all")
# Get all haplotypes in a single matrix
getSegSiteHaplo(colony, caste = "all", collapse = TRUE)

#Input is a MultiColony - same behaviour as for the Colony!
getSegSiteHaplo(apiary, caste = "queen")
getQueenSegSiteHaplo(apiary)

# Get the haplotypes of all individuals either by colony or in a single matrix
getSegSiteHaplo(apiary, caste = "all")
getSegSiteHaplo(apiary, caste = "all", collapse = TRUE)
```

`getSnpGeno` *Access SNP array genotypes of individuals in a caste*

Description

Level 0 function that returns SNP array genotypes of individuals in a caste.

Usage

```
getSnpGeno(
  x,
  caste = NULL,
  nInd = NULL,
  snpChip = 1,
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getQueenSnpGeno(
  x,
  snpChip = 1,
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getFathersSnpGeno(
  x,
  nInd = NULL,
  snpChip = 1,
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getVirginQueensSnpGeno(
  x,
  nInd = NULL,
  snpChip = 1,
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getWorkersSnpGeno(
  x,
  nInd = NULL,
  snpChip = 1,
  chr = NULL,
```

```
  collapse = FALSE,
  simParamBee = NULL
)

getDronesSnpGeno(
  x,
  nInd = NULL,
  snpChip = 1,
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`snpChip`	numeric, indicates which SNP array genotypes to retrieve
`chr`	numeric, chromosomes to retrieve, if `NULL`, all chromosome are retrieved
`dronesHaploid`	logical, return haploid result for drones?
`collapse`	logical, if the return value should be a single matrix with genotypes of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

matrix with genotypes when `x` is `Colony-class` and list of matrices with genotypes when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Functions

- `getQueenSnpGeno()`: Access SNP array genotype data of the queen
- `getFathersSnpGeno()`: Access SNP array genotype data of fathers
- `getVirginQueensSnpGeno()`: Access SNP array genotype data of virgin queens
- `getWorkersSnpGeno()`: Access SNP array genotype data of workers
- `getDronesSnpGeno()`: Access SNP array genotype data of drones

See Also

`getSnpGeno` and `pullSnpGeno`

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

SP$addSnpChip(nSnpPerChr = 5)
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

# Input is a population
getSnpGeno(x = getQueen(colony))
queens <- getQueen(apiary, collapse = TRUE)
getSnpGeno(queens)

# Input is a colony
getSnpGeno(colony, caste = "queen")
getQueenSnpGeno(colony)

getSnpGeno(colony, caste = "workers", nInd = 3)
getWorkersSnpGeno(colony)
# Same aliases exist for all the castes!

# Get genotypes for all individuals
getSnpGeno(colony, caste = "all")
# Get all haplotypes in a single matrix
getSnpGeno(colony, caste = "all", collapse = TRUE)

# Input is a MultiColony - same behaviour as for the Colony!
getSnpGeno(apiary, caste = "queen")
getQueenSnpGeno(apiary)

# Get the haplotypes of all individuals either by colony or in a single matrix
getSnpGeno(apiary, caste = "all")
getSnpGeno(apiary, caste = "all", collapse = TRUE)
```

`getSnpHaplo` *Access SNP array haplotypes of individuals in a caste*

Description

Level 0 function that returns SNP array haplotypes of individuals in a caste.

Usage

```
getSnpHaplo(
  x,
  caste = NULL,
  nInd = NULL,
  snpChip = 1,
  haplo = "all",
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getQueenSnpHaplo(
  x,
  snpChip = 1,
  haplo = "all",
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getFathersSnpHaplo(
  x,
  nInd = NULL,
  snpChip = 1,
  haplo = "all",
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

getVirginQueensSnpHaplo(
  x,
  nInd = NULL,
  snpChip = 1,
  haplo = "all",
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getWorkersSnpHaplo(
```

```
  x,
  nInd = NULL,
  snpChip = 1,
  haplo = "all",
  chr = NULL,
  collapse = FALSE,
  simParamBee = NULL
)

getDronesSnpHaplo(
  x,
  nInd = NULL,
  snpChip = 1,
  haplo = "all",
  chr = NULL,
  dronesHaploid = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`caste`	NULL or character, NULL when `x` is a `Pop-class`, and character when `x` is a `Colony-class` or `MultiColony-class` with the possible values of "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`nInd`	numeric, number of individuals to access, if `NULL` all individuals are accessed, otherwise a random sample
`snpChip`	numeric, indicates which SNP array haplotypes to retrieve
`haplo`	character, either "all" for all haplotypes or an integer for a single set of haplotypes, use a value of 1 for female haplotypes and a value of 2 for male haplotypes
`chr`	numeric, chromosomes to retrieve, if `NULL`, all chromosome are retrieved
`dronesHaploid`	logical, return haploid result for drones?
`collapse`	logical, if the return value should be a single matrix with haplotypes of all the individuals
`simParamBee`	`SimParamBee`, global simulation parameters

Value

matrix with haplotypes when `x` is `Colony-class` and list of matrices with haplotypes when `x` is `MultiColony-class`, named by colony id when `x` is `MultiColony-class`

Functions

- `getQueenSnpHaplo()`: Access SNP array haplotype data of the queen
- `getFathersSnpHaplo()`: Access SNP array haplotype data of fathers

- `getVirginQueensSnpHaplo()`: Access SNP array haplotype data of virgin queens
- `getWorkersSnpHaplo()`: Access SNP array haplotype of workers
- `getDronesSnpHaplo()`: Access SNP array haplotype data of drones

See Also

`getSnpHaplo` and `pullSnpHaplo`

Examples

```
founderGenomes <- quickHaplo(nInd = 4, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

SP$addSnpChip(nSnpPerChr = 5)
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)

# Input is a population
getSnpHaplo(x = getQueen(colony))
queens <- getQueen(apiary, collapse = TRUE)
getSnpHaplo(queens)

# Input is a colony
getSnpHaplo(colony, caste = "queen")
getQueenSnpHaplo(colony)

getSnpHaplo(colony, caste = "workers", nInd = 3)
getWorkersSnpHaplo(colony)
# Same aliases exist for all the castes!

# Get haplotypes for all individuals
getSnpHaplo(colony, caste = "all")
# Get all haplotypes in a single matrix
getSnpHaplo(colony, caste = "all", collapse = TRUE)

# Input is a MultiColony - same behaviour as for the Colony!
getSnpHaplo(apiary, caste = "queen")
getQueenSnpHaplo(apiary)
```

```
# Get the haplotypes of all individuals either by colony or in a single matrix
getSnpHaplo(apiary, caste = "all")
getSnpHaplo(apiary, caste = "all", collapse = TRUE)
```

`hasCollapsed` *Test if colony has collapsed*

Description

Level 0 function that returns colony collapse status.

Usage

```
hasCollapsed(x)
```

Arguments

x	`Colony-class` or `MultiColony-class`

Value

logical, named by colony id when `x` is `MultiColony-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(colony, nInd = 5)

hasCollapsed(colony)
colony <- collapse(colony)
hasCollapsed(colony)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)

hasCollapsed(apiary)
apiary <- collapse(apiary)
hasCollapsed(apiary)
```

`hasSplit` *Test if colony has split*

Description

Level 0 function that returns colony split status. This will obviously impact colony strength.

Usage

```
hasSplit(x)
```

Arguments

x	`Colony-class` or `MultiColony-class`

Value

logical, named by colony id when x is `MultiColony-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)

hasSplit(colony)
tmp <- split(colony)
hasSplit(tmp$split)
hasSplit(tmp$remnant)

hasSplit(apiary)
tmp2 <- split(apiary)
hasSplit(tmp2$split)
hasSplit(tmp2$remnant)
```

`hasSuperseded` *Test if colony has superseded*

Description

Level 0 function that returns colony supersedure status.

Usage

```
hasSuperseded(x)
```

Arguments

x	`Colony-class` or `MultiColony-class`

Value

logical, named by colony id when `x` is `MultiColony-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)

hasSuperseded(colony)
colony <- supersede(colony)
hasSuperseded(colony)

hasSuperseded(apiary)
apiary <- supersede(apiary)
hasSuperseded(apiary)
```

`hasSwarmed` *Test if colony has swarmed*

Description

Level 0 function that returns colony swarmed status. This will obviously have major impact on the colony and its downstream events.

Usage

```
hasSwarmed(x)
```

Arguments

x	`Colony-class` or `MultiColony-class`

Value

logical, named by colony id when `x` is `MultiColony-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(colony, nInd = 5)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)

hasSwarmed(colony)
tmp <- swarm(colony)
hasSwarmed(tmp$swarm)
hasSwarmed(tmp$remnant)

hasSwarmed(apiary)
tmp2 <- swarm(apiary)
hasSwarmed(tmp2$swarm)
hasSwarmed(tmp2$remnant)
```

---

isCaste *Is individual a member of a specific caste*

Description

Level 0 function that tests if individuals are members of a specific caste

Usage

```
isCaste(x, caste, simParamBee = NULL)

isQueen(x, simParamBee = NULL)

isFather(x, simParamBee = NULL)

isWorker(x, simParamBee = NULL)

isDrone(x, simParamBee = NULL)

isVirginQueens(x, simParamBee = NULL)

isVirginQueen(x, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`
`caste`	character, one of "queen", "fathers", "workers", "drones", or "virginQueens"; only single value is used
`simParamBee`	`SimParamBee`, global simulation parameters

Value

logical

Functions

- `isQueen()`: Is individual a queen
- `isFather()`: Is individual a father
- `isWorker()`: Is individual a worker
- `isDrone()`: Is individual a drone
- `isVirginQueens()`: Is individual a virgin queen
- `isVirginQueen()`: Is individual a virgin queen

See Also

`isQueen`, `isFather`, `isVirginQueen`, `isWorker`, and `isDrone`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 120, nDrones = 20)
colony <- addVirginQueens(x = colony, nInd = 4)

isCaste(getQueen(colony), caste = "queen")
isCaste(getFathers(colony, nInd = 2), caste = "fathers")
isCaste(getWorkers(colony, nInd = 2), caste = "workers") # random sample!
isCaste(getDrones(colony, nInd = 2), caste = "drones")
isCaste(getVirginQueens(colony, nInd = 2), caste = "virginQueens")

bees <- c(
  getQueen(colony),
  getFathers(colony, nInd = 2),
  getWorkers(colony, nInd = 2),
  getDrones(colony, nInd = 2),
  getVirginQueens(colony, nInd = 2)
)
isCaste(bees, caste = "queen")
isCaste(bees, caste = "fathers")
isCaste(bees, caste = "workers")
isCaste(bees, caste = "drones")
isCaste(bees, caste = "virginQueens")

isQueen(getQueen(colony))
isQueen(getFathers(colony, nInd = 2))

isFather(getQueen(colony))
isFather(getFathers(colony, nInd = 2))

isWorker(getQueen(colony))
isWorker(getFathers(colony, nInd = 2))
isWorker(getWorkers(colony, nInd = 2))

isDrone(getQueen(colony))
isDrone(getFathers(colony, nInd = 2))
isDrone(getDrones(colony, nInd = 2))

isVirginQueen(getQueen(colony))
isVirginQueen(getFathers(colony, nInd = 2))
isVirginQueen(getVirginQueens(colony, nInd = 2))
```

`isCsdActive` *Is csd locus activated*

Description

Level 0 function that checks if the csd locus has been activated. See `SimParamBee` for more information about the csd locus.

Usage

```
isCsdActive(simParamBee = NULL)
```

Arguments

`simParamBee`	`SimParamBee`, global simulation parameters

Value

logical

Examples

```
founderGenomes <- quickHaplo(nInd = 3, nChr = 3, segSites = 100)
SP <- SimParamBee$new(founderGenomes, csdChr = NULL)

isCsdActive()

SP <- SimParamBee$new(founderGenomes)

isCsdActive()
```

`isCsdHeterozygous` *Test if individuals are heterozygous at the csd locus*

Description

Level 0 function that returns if individuals of a population are heterozygous at the csd locus. See `SimParamBee` for more information about the csd locus.

Usage

```
isCsdHeterozygous(pop, simParamBee = NULL)
```

Arguments

`pop`	`Pop-class`
`simParamBee`	`SimParamBee`, global simulation parameters

Details

We could expand `isCsdHeterozygous` to work also with `Colony-class` and `MultiColony-class` if needed

Value

logical

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 4)

# Use isCsdHeterozygous on a Population
isCsdHeterozygous(getQueen(colony))
isCsdHeterozygous(getWorkers(colony))
```

`isDronesPresent` *Are drones present*

Description

Level 0 function that returns drones presence status (are they present or not).

Usage

```
isDronesPresent(x, simParamBee = NULL)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`simParamBee`	`SimParamBee`, global simulation parameters

Value

logical, named by colony id when `x` is `MultiColony-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 120, nDrones = 20)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 100, nDrones = 10)

isDronesPresent(colony)
isDronesPresent(removeDrones(colony))
isDronesPresent(apiary)
isDronesPresent(removeDrones(apiary))
```

`isEmpty` *Check whether a population, colony or a multicolony object has no individuals within*

Description

Check whether a population, colony or a multicolony object has no individuals within.

Usage

```
isEmpty(x)
```

Arguments

`x`	`Pop-class` or `Colony-class` or `MultiColony-class`

Value

boolean when `x` is `Pop-class` or `Colony-class`, and named vector of boolean when `x` is `MultiColony-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
```

```
isEmpty(new(Class = "Pop"))
isEmpty(basePop[0])
isEmpty(basePop)

emptyColony <- createColony()
nonEmptyColony <- createColony(basePop[1])
isEmpty(emptyColony)
isEmpty(nonEmptyColony)

emptyApiary <- createMultiColony(n = 3)
emptyApiary1 <- c(createColony(), createColony())
emptyApiary2 <- createMultiColony()
nonEmptyApiary <- createMultiColony(basePop[2:5], n = 4)

isEmpty(emptyApiary)
isEmpty(emptyApiary1)
isEmpty(nonEmptyApiary)
isNULLColonies(emptyApiary)
isNULLColonies(emptyApiary1)
isNULLColonies(nonEmptyApiary)

nEmptyColonies(emptyApiary)
nEmptyColonies(emptyApiary1)
nEmptyColonies(nonEmptyApiary)
nNULLColonies(emptyApiary)
nNULLColonies(emptyApiary1)
nNULLColonies(nonEmptyApiary)
```

`isFathersPresent` *Are fathers present (=queen mated)*

Description

Level 0 function that returns fathers presence status (are they present or not, which means the queen is mated).

Usage

```
isFathersPresent(x, simParamBee = NULL)

areFathersPresent(x, simParamBee = NULL)
```

Arguments

x	`Colony-class` or `MultiColony-class`
simParamBee	`SimParamBee`, global simulation parameters

Value

logical, named by colony id when `x` is `MultiColony-class`

Functions

- `areFathersPresent()`: Are fathers present

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
isFathersPresent(colony)
apiary <- createMultiColony(basePop[3:4], n = 2)
isFathersPresent(apiary)

colony <- cross(colony, drones = droneGroups[[1]])
isFathersPresent(removeDrones(colony))

apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
isFathersPresent(removeDrones(apiary))
```

`isGenoHeterozygous` *Test if a multilocus genotype is heterozygous*

Description

Level 0 function that returns heterozygote status for a multilocus genotype.

Usage

```
isGenoHeterozygous(x)
```

Arguments

`x`	integer or matrix, output from `getCsdGeno`

Value

logical # Not exporting this function, since its just a helper

---

isNULLColonies *Check which of the colonies in a multicolony are NULL*

Description

Check which of the colonies in a multicolony are NULL

Usage

```
isNULLColonies(multicolony)
```

Arguments

`multicolony`	`MultiColony-class`

Value

Named vector of boolean

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

emptyApiary <- createMultiColony(n = 3)
emptyApiary1 <- c(createColony(), createColony())
nonEmptyApiary <- createMultiColony(basePop[2:5], n = 4)

isEmpty(emptyApiary)
isEmpty(emptyApiary1)
isEmpty(nonEmptyApiary)
isNULLColonies(emptyApiary)
isNULLColonies(emptyApiary1)
isNULLColonies(nonEmptyApiary)

nEmptyColonies(emptyApiary)
nEmptyColonies(emptyApiary1)
nEmptyColonies(nonEmptyApiary)
nNULLColonies(emptyApiary)
nNULLColonies(emptyApiary1)
nNULLColonies(nonEmptyApiary)
```

isProductive *Test if colony is currently productive*

Description

Level 0 function that returns colony production status. This can be used to decided if colony production can be simulated.

Usage

```
isProductive(x)
```

Arguments

x	Colony-class or MultiColony-class

Value

logical, named by colony id when x is MultiColony-class

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])

isProductive(colony)
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
isProductive(colony)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])

isProductive(apiary)
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
isProductive(apiary)
```

`isQueenPresent` *Is the queen present*

Description

Level 0 function that returns queen's presence status (is she present/alive or not).

Usage

```
isQueenPresent(x, simParamBee = NULL)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`simParamBee`	`SimParamBee`, global simulation parameters

Value

logical, named by colony id when `x` is `MultiColony-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 120, nDrones = 20)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 100, nDrones = 10)

isQueenPresent(colony)
isQueenPresent(apiary)

colony <- removeQueen(colony)
isQueenPresent(colony)
```

---

isSimParamBee *Test if x is a SimParamBee class object*

Description

Test if x is a `SimParamBee` class object

Usage

```
isSimParamBee(x)
```

Arguments

`x`	`SimParamBee`

Value

logical

Examples

```
founderGenomes <- quickHaplo(nInd = 2, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

isSimParamBee(SP)
```

---

isVirginQueensPresent *Are virgin queen(s) present*

Description

Level 0 function that returns virgin queen(s) presence status.

Usage

```
isVirginQueensPresent(x, simParamBee = NULL)

isVirginQueenPresent(x, simParamBee = NULL)

areVirginQueensPresent(x, simParamBee = NULL)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`simParamBee`	`SimParamBee`, global simulation parameters

Value

logical, named by colony id when `x` is `MultiColony-class`

Functions

- `isVirginQueenPresent()`: Are virgin queen(s) present
- `areVirginQueensPresent()`: Are virgin queen(s) present

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- addVirginQueens(x = colony, nInd = 4)
isVirginQueenPresent(colony)
isVirginQueenPresent(pullVirginQueens(colony)$remnant)
isVirginQueenPresent(removeQueen(colony))

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 100, nDrones = 10)
isVirginQueenPresent(apiary)

tmp <- swarm(x = apiary)
isVirginQueenPresent(tmp$swarm)
isVirginQueenPresent(tmp$remnant)
```

`isWorkersPresent` *Are workers present*

Description

Level 0 function that returns workers presence status (are they present or not).

Usage

```
isWorkersPresent(x, simParamBee = NULL)

areWorkersPresent(x, simParamBee = NULL)

areDronesPresent(x, simParamBee = NULL)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`simParamBee`	`SimParamBee`, global simulation parameters

Value

logical, named by colony id when `x` is `MultiColony-class`

Functions

- `areWorkersPresent()`: Are workers present
- `areDronesPresent()`: Are drones present

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 120, nDrones = 20)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 100, nDrones = 10)

isWorkersPresent(colony)
isWorkersPresent(removeWorkers(colony))
isWorkersPresent(apiary)
isWorkersPresent(removeWorkers(apiary))
```

`mapCasteToColonyValue` *Map caste member (individual) values to a colony value*

Description

Maps caste member (individual) values to a colony value - for phenotype, genetic, breeding, dominance, and epistasis values. This function can be used as `FUN` argument in `calcColonyValue` function(s). It can also be saved in `SimParamBee$colonyValueFUN` as a default function called by `calcColonyValue` function(s).

This is just an example - quite a flexible one! You can provide your own "caste functions" that satisfy your needs within this mapping function (see `queenFUN`, `workersFUN`, and `dronesFUN` below) or provide a complete replacement of this mapping function! For example, this mapping function does not cater for indirect (social) genetic effects where colony individuals value impacts value of other colony individuals. Note though that you can achieve this impact also via multiple correlated traits, such as a queen and a workers trait.

Usage

```
mapCasteToColonyValue(
  colony,
  value = "pheno",
  queenTrait = 1,
  queenFUN = function(x) x,
  workersTrait = 2,
  workersFUN = colSums,
  dronesTrait = NULL,
  dronesFUN = NULL,
  traitName = NULL,
  combineFUN = function(q, w, d) q + w,
  checkProduction = TRUE,
  notProductiveValue = 0,
  simParamBee = NULL
)

mapCasteToColonyPheno(colony, simParamBee = NULL, ...)

mapCasteToColonyGv(colony, simParamBee = NULL, ...)

mapCasteToColonyBv(colony, simParamBee = NULL, ...)

mapCasteToColonyDd(colony, simParamBee = NULL, ...)

mapCasteToColonyAa(colony, simParamBee = NULL, ...)
```

Arguments

`colony`	`Colony-class`
`value`	character, one of `pheno` or `gv`
`queenTrait`	numeric (column position) or character (column name), trait(s) that represents queen's contribution to colony value(s); if `NULL` then this contribution is 0; you can pass more than one trait here, but make sure that `combineFUN` works with these trait dimensions
`queenFUN`	function, function that will be applied to queen's value
`workersTrait`	numeric (column position) or character (column name), trait(s) that represents workers' contribution to colony value(s); if `NULL` then this contribution is 0; you can pass more than one trait here, but make sure that `combineFUN` works with these trait dimensions
`workersFUN`	function, function that will be applied to workers values

`dronesTrait`	numeric (column position) or character (column name), trait(s) that represents drones' contribution to colony value(s); if `NULL` then this contribution is 0; you can pass more than one trait here, but make sure that `combineFUN` works with these trait dimensions
`dronesFUN`	function, function that will be applied to drone values
`traitName`	the name of the colony trait(s), say, honeyYield; you can pass more than one trait name here, but make sure to match them with `combineFUN` trait dimensions
`combineFUN`	function that will combine the queen, worker, and drone contributions - this function should be defined as `function(q, w, d)` where `q` represents queen's, `q` represents workers', and `d` represents drones' contribution.
`checkProduction`	logical, does the value depend on the production status of colony; if yes and production is `FALSE`, the return is `notProductiveValue` - this will often make sense for colony phenotype value only; you can pass more than one logical value here (one per trait coming out of `combineFUN`)
`notProductiveValue`	numeric, returned value when colony is not productive; you can pass more than one logical value here (one per trait coming out of `combineFUN`)
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	other arguments of `mapCasteToColonyValue` (for its aliases)

Details

This is a utility/mapping function meant to be called by `calcColonyValue`. It only works on a single colony - use `calcColonyValue` to get Colony or MultiColony values.

Value

numeric matrix with one value or a row of values

Functions

- `mapCasteToColonyPheno()`: Map caste member (individual) phenotype values to a colony phenotype value
- `mapCasteToColonyGv()`: Map caste member (individual) genetic values to a colony genetic value
- `mapCasteToColonyBv()`: Map caste member (individual) breeding values to a colony breeding value
- `mapCasteToColonyDd()`: Map caste member (individual) dominance values to a colony dominance value
- `mapCasteToColonyAa()`: Map caste member (individual) epistasis values to a colony epistasis value

See Also

`SimParamBee` field `colonyValueFUN` and functions `calcColonyValue`, `calcColonyPheno`, `calcColonyGv`, `getEvents`, `pheno`, and `gv`, as well as `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)


# Define two traits that collectively affect colony honey yield:
# 1) queen's effect on colony honey yield, say via pheromone secretion phenotype
# 2) workers' effect on colony honey yield, say via foraging ability phenotype
# The traits will have a negative genetic correlation of -0.5 and heritability
# of 0.25 (on an individual level)
nWorkers <- 10
mean <- c(10, 10 / nWorkers)
varA <- c(1, 1 / nWorkers)
corA <- matrix(data = c(
  1.0, -0.5,
  -0.5, 1.0
), nrow = 2, byrow = TRUE)
varE <- c(3, 3 / nWorkers)
varA / (varA + varE)
SP$addTraitADE(nQtlPerChr = 100,
               mean = mean,
               var = varA, corA = corA,
               meanDD = 0.1, varDD = 0.2, corD = corA,
               relAA = 0.1, corAA = corA)
SP$setVarE(varE = varE)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 10)
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = drones)
colony <- buildUp(colony, nWorkers = nWorkers, nDrones = 3)

# Colony value
mapCasteToColonyPheno(colony)
mapCasteToColonyGv(colony)

# To understand where the above values come from, study the contents of
# mapCasteToColonyValue() and the values below:

# Phenotype values
getQueenPheno(colony)
getWorkersPheno(colony)

# Genetic values
getQueenGv(colony)
getWorkersGv(colony)
```

`mapLoci` *Finds loci on a genetic map and return a list of positions*

Description

Finds loci on a genetic map and return a list of positions. This function is adopted from AlphaSimR (Gaynor et al., 2021)

Usage

```
mapLoci(markers, genMap)
```

Arguments

`markers`	character, vector of marker positions as "chr_position"
`genMap`	list, genetic map

Value

A list with number of loci per chromosome and genetic positions of the markers

---

`MultiColony-class` *Honeybee multicolony object*

Description

An object holding a collection of honeybee colonies. It behaves like a list.

Usage

```
isMultiColony(x)

## S4 method for signature 'MultiColony'
show(object)

## S4 method for signature 'MultiColony'
c(x, ...)

## S4 method for signature 'MultiColonyOrNULL'
c(x, ...)

## S4 method for signature 'MultiColony,integerOrNumericOrLogical'
x[i, j, drop]

## S4 method for signature 'MultiColony,character'
x[i, j, drop]

## S4 method for signature 'MultiColony,integerOrNumericOrLogical'
x[[i]]

## S4 method for signature 'MultiColony,character'
```

```
x[[i]]

## S4 replacement method for signature
## 'MultiColony,integerOrNumericOrLogicalOrCharacter,ANY,MultiColony'
x[i, j] <- value

## S4 replacement method for signature
## 'MultiColony,integerOrNumericOrLogicalOrCharacter,ANY,Colony'
x[[i, j]] <- value
```

Arguments

`x`	`MultiColony-class`
`object`	`MultiColony-class`
`...`	`NULL`, `Colony-class`, or `MultiColony-class`
`i`	integer, numeric, logical, or character, index or ID to select a colony (see examples)
`j`	not used
`drop`	not used
`value`	`Colony-class` or `MultiColony-class` to assign into `x` based on colony index or name `i`

Value

`MultiColony-class` or `Colony-class`

Functions

- `isMultiColony()`: Test if x is a MultiColony class object
- `show(MultiColony)`: Show MultiColony object
- `c(MultiColony)`: Combine multiple Colony and MultiColony objects
- `c(MultiColonyOrNULL)`: Combine multiple Colony and MultiColony objects
- `x[i`: Extract a colony (one or more!) with an integer/numeric/logical index (position) (return `MultiColony-class`)
- `x[i`: Extract a colony (one or more!) with a character ID (name) (return `MultiColony-class`)
- `x[[i`: Extract a colony (just one!) with an integer/numeric/logical index (position) (return `Colony-class`)
- `x[[i`: Extract a colony (just one!) with a character ID (name) (return `Colony-class`)
- `` `[`(x = MultiColony, i = integerOrNumericOrLogicalOrCharacter, j = ANY) <- value ``: Assign colonies into MultiColony
- `` `[[`(x = MultiColony, i = integerOrNumericOrLogicalOrCharacter, j = ANY) <- value ``: Assign Colony into MultiColony

Slots

`colonies` list, a collection of `Colony-class` objects

See Also

createMultiColony

Examples

```
founderGenomes <- quickHaplo(nInd = 10, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)
apiary <- createMultiColony(basePop[1:6], n = 6)
apiary <- cross(apiary, drones = droneGroups[1:6])
apiary
show(apiary)
is(apiary)
isMultiColony(apiary)

getId(apiary)
apiary[1]
getId(apiary[1])
getId(apiary["2"])
getId(apiary[2])
getId(apiary[-1])
getId(apiary[5])

getId(apiary)
getId(apiary[c(1, 3)])
getId(apiary[c("2", "4")])
getId(apiary[c(TRUE, FALSE, TRUE, FALSE)])
getId(apiary[c(TRUE, FALSE)]) # beware of recycling!
getId(apiary[c(5, 6)])
getId(apiary[c("6", "7")])

apiary[[1]]
apiary[["2"]]
apiary[[3]]
apiary[["4"]]
try(apiary[[6]])
apiary[["7"]]

getId(c(apiary[c(1, 3)], apiary[2]))
getId(c(apiary[2], apiary[c(1, 3)]))

getId(c(apiary[2], apiary[0]))
getId(c(apiary[0], apiary[2]))

getId(c(apiary[2], NULL))
getId(c(NULL, apiary[2]))

apiary1 <- apiary[1:2]
```

```
apiary2 <- apiary[3:4]
getId(apiary1)
getId(apiary2)
apiary1[[1]] <- apiary2[[1]]
getId(apiary1)
try(apiary2[[1]] <- apiary2[[2]])

apiary1 <- apiary[1:2]
apiary2 <- apiary[3:5]
getId(apiary1)
getId(apiary2)
apiary2[1:2] <- apiary1
getId(apiary2)
try(apiary2[1] <- apiary1)
try(apiary2[1:3] <- apiary1)
try(apiary2[1:2] <- apiary1[[1]])

apiary2 <- apiary[3:5]
getId(apiary2)
try(apiary2[c("4", "5")] <- apiary1)
try(apiary2[c("4", "5")] <- apiary1)
```

`nCaste` *Level 0 function that returns the number of individuals of a caste in a colony*

Description

Returns the number of individuals of a caste in a colony

Usage

```
nCaste(x, caste = "all", simParamBee = NULL)

nQueens(x, simParamBee = NULL)

nFathers(x, simParamBee = NULL)

nWorkers(x, simParamBee = NULL)

nDrones(x, simParamBee = NULL)

nVirginQueens(x, simParamBee = NULL)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`caste`	character, "queen", "fathers", "workers", "drones", "virginQueens", or "all"
`simParamBee`	`SimParamBee`, global simulation parameters

Value

when x is `Colony-class` return is integer for `caste != "all"` or list for `caste == "all"` with nodes named by caste; when `x` is `MultiColony-class` return is named integer for `caste != "all"` or named list of lists for `caste == "all"`

Functions

- `nQueens()`: Number of queens in a colony
- `nFathers()`: Number of fathers in a colony
- `nWorkers()`: Number of workers in a colony
- `nDrones()`: Number of drones in a colony
- `nVirginQueens()`: Number of virgin queens in a colony

See Also

`nQueens`, `nFathers`, `nVirginQueens`, `nWorkers`, and `nDrones`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 100, nDrones = 10)
colony <- addVirginQueens(x = colony, nInd = 3)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 100, nDrones = 10)
apiary <- addVirginQueens(x = apiary, nInd = 3)

# Check caste members
nCaste(colony, caste = "queen")
nCaste(colony, caste = "fathers")
nCaste(colony, caste = "virginQueens")
nCaste(colony, caste = "workers")
nCaste(colony, caste = "drones")
nCaste(colony, caste = "all")

nCaste(apiary, caste = "queen")
nCaste(apiary, caste = "fathers")
nCaste(apiary, caste = "virginQueens")
nCaste(apiary, caste = "workers")
```

```
nCaste(apiary, caste = "drones")
nCaste(apiary, caste = "all")

# Check number of queens
nQueens(colony)
nQueens(apiary)
apiary <- removeQueen(apiary)
nQueens(apiary)

# Check number of fathers
nFathers(colony)
nFathers(apiary)

# Check number of workers
nWorkers(colony)
nWorkers(apiary)

# Check number of drones
nDrones(colony)
nDrones(apiary)

# Check number of virgin queens
nVirginQueens(colony)
nVirginQueens(apiary)
```

`nColonies` *Number of colonies in a MultiColony object*

Description

Level 0 function that returns the number of colonies in a MultiColony object.

Usage

```
nColonies(multicolony)

nNULLColonies(multicolony)

nEmptyColonies(multicolony)
```

Arguments

`multicolony`	`MultiColony-class`

Value

integer

Functions

- `nNULLColonies()`: Number of `NULL` colonies in a MultiColony object
- `nEmptyColonies()`: Number of empty colonies in a MultiColony object

See Also

`nNULLColonies` and `nEmptyColonies`

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
emptyApiary <- createMultiColony(n = 3)
emptyApiary1 <- c(createColony(), createColony())
nonEmptyApiary <- createMultiColony(basePop[2:3], n = 2)

nColonies(nonEmptyApiary)
nColonies(emptyApiary)

isEmpty(emptyApiary)
isEmpty(emptyApiary1)
isEmpty(nonEmptyApiary)
isNULLColonies(emptyApiary)
isNULLColonies(emptyApiary1)
isNULLColonies(nonEmptyApiary)

nEmptyColonies(emptyApiary)
nEmptyColonies(emptyApiary1)
nEmptyColonies(nonEmptyApiary)
nNULLColonies(emptyApiary)
nNULLColonies(emptyApiary1)
nNULLColonies(nonEmptyApiary)
```

`nCsdAlleles` *Report the number of distinct csd alleles*

Description

Level 0 function that returns the number of distinct csd alleles in input. See `SimParamBee` for more information about the csd locus.

Usage

```
nCsdAlleles(x, collapse = FALSE, simParamBee = NULL)
```

Arguments

`x`	`Pop-class`, `Colony-class`, or `MultiColony-class`
`collapse`	logical, if `TRUE`, the function will return the number of distinct csd alleles in either the entire population, colony, or multicolony. Note this has nothing to do with the colony collapse. It's like `paste(..., collapse = TRUE)`. Default is `FALSE`. See examples about this behaviour.Default is `FALSE`.
`simParamBee`	`SimParamBee`, global simulation parameters

Details

Queen has 2 distinct csd alleles, since she has to be heterozygous to be viable. The same holds for individual virgin queens and workers, but note that looking at csd genotypes of virgin queens or workers we are looking at a sample of 1 csd allele from the queen and 1 csd allele from their fathers, noting that homozygous genotypes are excluded. Therefore, `nCsdAlleles()` from virgin queens and workers is a noisy realisation of `nCsdAlleles()` from queens and fathers. For this reason, we also report `nCsdAlleles()` from queens and fathers combined (see the `queenAndFathers` list node) when `x` is `Colony-class`. This last measure is then the expected number of csd alleles in a colony as opposed to realised number of csd alleles in a sample of virgin queens and workers. Similarly as for virgin queens and workers, `nCsdAlleles()` from drones gives a noisy realisation of `nCsdAlleles()` from queens. The amount of noise will depend on the number of individuals, so in most cases with reasonable number of individuals there should be minimal amount of noise.

Value

integer representing the number of distinct csd alleles when `x` is `Pop-class` (or), list of integer when `x` is `Colony-class` (list nodes named by caste) and list of a list of integer when `x` is `MultiColony-class`, outer list is named by colony id when `x` is `MultiColony-class`; the integer rep

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 6, nDrones = 3)
colony <- addVirginQueens(x = colony, nInd = 4)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 6, nDrones = 3)
apiary <- addVirginQueens(x = apiary, nInd = 5)
```

```
nCsdAlleles(getQueen(colony))
nCsdAlleles(getWorkers(colony))

nCsdAlleles(colony)
nCsdAlleles(colony, collapse = TRUE)

nCsdAlleles(apiary)
nCsdAlleles(apiary, collapse = TRUE)
```

`nDronesPoisson` *Sample a number of drones*

Description

Sample a number of drones - used when `nDrones = NULL` (see `SimParamBee$nDrones`).

This is just an example. You can provide your own functions that satisfy your needs!

Usage

```
nDronesPoisson(x, n = 1, average = 100)

nDronesTruncPoisson(x, n = 1, average = 100, lowerLimit = 0)

nDronesColonyPhenotype(
  x,
  queenTrait = 1,
  workersTrait = NULL,
  checkProduction = FALSE,
  lowerLimit = 0,
  simParamBee = NULL,
  ...
)
```

Arguments

`x`	`Pop-class` or `Colony-class`
`n`	integer, number of samples
`average`	numeric, average number of drones
`lowerLimit`	numeric, returned numbers will be above this value
`queenTrait`	numeric (column position) or character (column name), trait that represents queen's effect on the colony phenotype (defined in `SimParamBee` - see examples); if `0` then this effect is 0
`workersTrait`	numeric (column position) or character (column name), trait that represents workers's effect on the colony phenotype (defined in `SimParamBee` - see examples); if `0` then this effect is 0

`checkProduction`	logical, does the phenotype depend on the production status of colony; if yes and production is not `TRUE`, the result is above `lowerLimit`
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	other arguments of `mapCasteToColonyPheno`

Details

`nDronesPoisson` samples from a Poisson distribution with a given average, which can return a value 0.

`nDronesTruncPoisson` samples from a zero truncated Poisson distribution.

`nDronesColonyPhenotype` returns a number (above `lowerLimit`) as a function of colony phenotype, say queen's fecundity. Colony phenotype is provided by `mapCasteToColonyPheno`. You need to set up traits influencing the colony phenotype and their parameters (mean and variances) via `SimParamBee` (see examples).

When `x` is `Pop-class`, only `workersTrait` is not used, that is, only `queenTrait` is used.

Value

numeric, number of drones

Functions

- `nDronesTruncPoisson()`: Sample a non-zero number of drones
- `nDronesColonyPhenotype()`: Sample a non-zero number of drones based on colony phenotype, say queen's fecundity

See Also

`SimParamBee` field `nDrones` and `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
nDronesPoisson()
nDronesPoisson()
n <- nDronesPoisson(n = 1000)
hist(n, breaks = seq(from = min(n), to = max(n)), xlim = c(0, 200))
table(n)

nDronesTruncPoisson()
nDronesTruncPoisson()
n <- nDronesTruncPoisson(n = 1000)
hist(n, breaks = seq(from = min(n), to = max(n)), xlim = c(0, 200))
table(n)

# Example for nDronesColonyPhenotype()
founderGenomes <- quickHaplo(nInd = 3, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)
```

```
average <- 100
h2 <- 0.1
SP$addTraitA(nQtlPerChr = 100, mean = average, var = average * h2)
SP$setVarE(varE = average * (1 - h2))
basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 50)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 2, nDrones = 15)
colony1 <- createColony(x = basePop[2])
colony2 <- createColony(x = basePop[3])
colony1 <- cross(colony1, drones = droneGroups[[1]])
colony2 <- cross(colony2, drones = droneGroups[[2]])
colony1@queen@pheno
colony2@queen@pheno
createDrones(colony1, nInd = nDronesColonyPhenotype)
createDrones(colony2, nInd = nDronesColonyPhenotype)
```

`nFathersPoisson` *Sample a number of fathers*

Description

Sample a number of fathers - use when `nFathers` = `NULL` (see `SimParamBee$nFathers`).

This is just an example. You can provide your own functions that satisfy your needs!

Usage

```
nFathersPoisson(n = 1, average = 15)

nFathersTruncPoisson(n = 1, average = 15, lowerLimit = 0)
```

Arguments

`n`	integer, number of samples
`average`	numeric, average number of fathers
`lowerLimit`	numeric, returned numbers will be above this value

Details

`nFathersPoisson` samples from a Poisson distribution, which can return a value 0 (that would mean a failed queen mating).

`nFathersTruncPoisson` samples from a truncated Poisson distribution (truncated at zero) to avoid failed matings.

Value

numeric, number of fathers

Functions

- `nFathersTruncPoisson()`: Sample a non-zero number of fathers

See Also

`SimParamBee` field `nFathers`

Examples

```
nFathersPoisson()
nFathersPoisson()
n <- nFathersPoisson(n = 1000)
hist(n, breaks = seq(from = min(n), to = max(n)), xlim = c(0, 40))
table(n)

nFathersTruncPoisson()
nFathersTruncPoisson()
n <- nFathersTruncPoisson(n = 1000)
hist(n, breaks = seq(from = min(n), to = max(n)), xlim = c(0, 40))
table(n)
```

`nVirginQueensPoisson` *Sample a number of virgin queens*

Description

Sample a number of virgin queens - used when `nFathers = NULL` (see `SimParamBee$nVirginQueens`).

This is just an example. You can provide your own functions that satisfy your needs!

Usage

```
nVirginQueensPoisson(colony, n = 1, average = 10)

nVirginQueensTruncPoisson(colony, n = 1, average = 10, lowerLimit = 0)

nVirginQueensColonyPhenotype(
  colony,
  queenTrait = 1,
  workersTrait = 2,
  checkProduction = FALSE,
  lowerLimit = 0,
  simParamBee = NULL,
  ...
)
```

Arguments

`colony`	`Colony-class`
`n`	integer, number of samples
`average`	numeric, average number of virgin queens
`lowerLimit`	numeric, returned numbers will be above this value
`queenTrait`	numeric (column position) or character (column name), trait that represents queen's effect on the colony phenotype (defined in `SimParamBee` - see examples); if `NULL` then this effect is 0
`workersTrait`	numeric (column position) or character (column name), trait that represents workers's effect on the colony phenotype (defined in `SimParamBee` - see examples); if `NULL` then this effect is 0
`checkProduction`	logical, does the phenotype depend on the production status of colony; if yes and production is not `TRUE`, the result is above `lowerLimit`
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	other arguments of `mapCasteToColonyPheno`

Details

`nVirginQueensPoisson` samples from a Poisson distribution, which can return a value 0 (that would mean a colony will fail to raise a single virgin queen after the queen swarms or dies).

`nVirginQueensTruncPoisson` samples from a truncated Poisson distribution (truncated at zero) to avoid failure.

`nVirginQueensColonyPhenotype` returns a number (above `lowerLimit`) as a function of colony phenotype, say swarming tendency. Colony phenotype is provided by `mapCasteToColonyPheno`. You need to set up traits influencing the colony phenotype and their parameters (mean and variances) via `SimParamBee` (see examples).

Value

numeric, number of virgin queens

Functions

- `nVirginQueensTruncPoisson()`: Sample a non-zero number of virgin queens
- `nVirginQueensColonyPhenotype()`: Sample a non-zero number of virgin queens based on colony's phenotype, say, swarming tendency

See Also

`SimParamBee` field `nVirginQueens` and `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
nVirginQueensPoisson()
nVirginQueensPoisson()
n <- nVirginQueensPoisson(n = 1000)
hist(n, breaks = seq(from = min(n), to = max(n)), xlim = c(0, 30))
table(n)

nVirginQueensTruncPoisson()
nVirginQueensTruncPoisson()
n <- nVirginQueensTruncPoisson(n = 1000)
hist(n, breaks = seq(from = min(n), to = max(n)), xlim = c(0, 30))
table(n)

# Example for nVirginQueensColonyPhenotype()
founderGenomes <- quickHaplo(nInd = 3, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

# Setting trait scale such that mean is 10 split into queen and workers effects
meanP <- c(5, 5 / SP$nWorkers)
# setup variances such that the total phenotype variance will match the mean
varA <- c(3 / 2, 3 / 2 / SP$nWorkers)
corA <- matrix(data = c(
  1.0, -0.5,
  -0.5, 1.0
), nrow = 2, byrow = TRUE)
varE <- c(7 / 2, 7 / 2 / SP$nWorkers)
varA / (varA + varE)
varP <- varA + varE
varP[1] + varP[2] * SP$nWorkers
SP$addTraitA(nQtlPerChr = 100, mean = meanP, var = varA, corA = corA)
SP$setVarE(varE = varE)
basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 50)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 2, nDrones = 15)
colony1 <- createColony(x = basePop[2])
colony2 <- createColony(x = basePop[3])
colony1 <- cross(colony1, drones = droneGroups[[1]])
colony2 <- cross(colony2, drones = droneGroups[[2]])
colony1 <- buildUp(colony1)
colony2 <- buildUp(colony2)
nVirginQueensColonyPhenotype(colony1)
nVirginQueensColonyPhenotype(colony2)
```

`nWorkersPoisson` *Sample a number of workers*

Description

Sample a number of workers - used when `nInd = NULL` (see `SimParamBee$nWorkers`).

This is just an example. You can provide your own functions that satisfy your needs!

Usage

```
nWorkersPoisson(colony, n = 1, average = 100)

nWorkersTruncPoisson(colony, n = 1, average = 100, lowerLimit = 0)

nWorkersColonyPhenotype(
  colony,
  queenTrait = 1,
  workersTrait = NULL,
  checkProduction = FALSE,
  lowerLimit = 0,
  simParamBee = NULL,
  ...
)
```

Arguments

`colony`	`Colony-class`
`n`	integer, number of samples
`average`	numeric, average number of workers
`lowerLimit`	numeric, returned numbers will be above this value
`queenTrait`	numeric (column position) or character (column name), trait that represents queen's effect on the colony phenotype (defined in `SimParamBee` - see examples); if `0` then this effect is 0
`workersTrait`	numeric (column position) or character (column name), trait that represents workers's effect on the colony phenotype (defined in `SimParamBee` - see examples); if `0` then this effect is 0
`checkProduction`	logical, does the phenotype depend on the production status of colony; if yes and production is not `TRUE`, the result is above `lowerLimit`
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	other arguments of `mapCasteToColonyPheno`

Details

`nWorkersPoisson` samples from a Poisson distribution with a given average, which can return a value 0. `nDronesTruncPoisson` samples from a zero truncated Poisson distribution.

`nWorkersColonyPhenotype` returns a number (above `lowerLimit`) as a function of colony phenotype, say queen's fecundity. Colony phenotype is provided by `mapCasteToColonyPheno`. You need to set up traits influencing the colony phenotype and their parameters (mean and variances) via `SimParamBee` (see examples).

Value

numeric, number of workers

Functions

- `nWorkersTruncPoisson()`: Sample a non-zero number of workers
- `nWorkersColonyPhenotype()`: Sample a non-zero number of workers based on colony phenotype, say queen's fecundity

See Also

`SimParamBee` field `nWorkers` and `vignette(topic = "QuantitativeGenetics", package = "SIMplyBee")`

Examples

```
nWorkersPoisson()
nWorkersPoisson()
n <- nWorkersPoisson(n = 1000)
hist(n, breaks = seq(from = min(n), to = max(n)), xlim = c(0, 200))
table(n)

nWorkersTruncPoisson()
nWorkersTruncPoisson()
n <- nWorkersTruncPoisson(n = 1000)
hist(n, breaks = seq(from = min(n), to = max(n)), xlim = c(0, 200))
table(n)

# Example for nWorkersColonyPhenotype()
founderGenomes <- quickHaplo(nInd = 3, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

average <- 100
h2 <- 0.1
SP$addTraitA(nQtlPerChr = 100, mean = average, var = average * h2)
SP$setVarE(varE = average * (1 - h2))
basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 50)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 2, nDrones = 15)
colony1 <- createColony(x = basePop[2])
colony2 <- createColony(x = basePop[3])
colony1 <- cross(colony1, drones = droneGroups[[1]])
colony2 <- cross(colony2, drones = droneGroups[[2]])
colony1@queen@pheno
colony2@queen@pheno
createWorkers(colony1, nInd = nWorkersColonyPhenotype)
createWorkers(colony2, nInd = nWorkersColonyPhenotype)
```

`pullCastePop` *Pull individuals from a caste in a colony*

Description

Level 1 function that pulls individuals from a caste in a colony. These individuals are removed from the colony (compared to `getCaste`).

Usage

```
pullCastePop(
  x,
  caste,
  nInd = NULL,
  use = "rand",
  removeFathers = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

pullQueen(x, collapse = FALSE, simParamBee = NULL)

pullWorkers(x, nInd = NULL, use = "rand", collapse = FALSE, simParamBee = NULL)

pullDrones(
  x,
  nInd = NULL,
  use = "rand",
  removeFathers = TRUE,
  collapse = FALSE,
  simParamBee = NULL
)

pullVirginQueens(
  x,
  nInd = NULL,
  use = "rand",
  collapse = FALSE,
  simParamBee = NULL
)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`caste`	character, "queen", "workers", "drones", or "virginQueens"
`nInd`	numeric, number of individuals to pull, if `NULL` all individuals are pulled. If input is `MultiColony-class`, the input could also be a vector of the same length as the number of colonies. If a single value is provided, the same value will be applied to all the colonies.
`use`	character, all options provided by `selectInd`
`removeFathers`	logical, removes `drones` that have already mated; set to `FALSE` if you would like to get drones for mating with multiple virgin queens, say via insemination
`collapse`	logical, whether to return a single merged population for the pulled individuals (does not affect the remnant colonies)
`simParamBee`	`SimParamBee`, global simulation parameters

Value

list of `Pop-class` and `Colony-class` when `x` is `Colony-class` and list of (a list of `Pop-class` named by colony id) and `MultiColony-class` when `x` is `MultiColony-class`

Functions

- `pullQueen()`: Pull queen from a colony
- `pullWorkers()`: Pull workers from a colony
- `pullDrones()`: Pull drones from a colony
- `pullVirginQueens()`: Pull virgin queens from a colony

See Also

`pullQueen`, `pullVirginQueens`, `pullWorkers`, and `pullDrones`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(x = colony, nWorkers = 100, nDrones = 10, exact = TRUE)
colony <- addVirginQueens(x = colony, nInd = 3)

apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])
apiary <- buildUp(x = apiary, nWorkers = 100, nDrones = 10, exact = TRUE)
apiary <- addVirginQueens(x = apiary, nInd = 3)

# pullCastePop on Colony class
# We can't pull the queen and leave the colony queenless
pullCastePop(colony, caste = "virginQueens")
pullCastePop(colony, caste = "virginQueens", nInd = 2)
# Or use aliases
pullVirginQueens(colony)
pullVirginQueens(colony, nInd = 2)
# Same aliases exist for all the castes!!!

# pullCastePop on MultiColony class - same behaviour as for the Colony!
pullCastePop(apiary, caste = "workers")
# Or pull out unequal number of workers from colonies
pullCastePop(apiary, caste = "workers", nInd = c(10, 20))
pullWorkers(apiary)
```

```
nWorkers(apiary)
nWorkers(pullWorkers(apiary)$remnant)


# Merge all the pulled populations into a single population
pullCastePop(apiary, caste = "queen", collapse = TRUE)
pullCastePop(apiary, caste = "virginQueens", collapse = TRUE)
```

`pullColonies` *Pull out some colonies from the MultiColony object*

Description

Level 3 function that pulls out some colonies from the MultiColony based on colony ID or random selection.

Usage

```
pullColonies(
  multicolony,
  ID = NULL,
  n = NULL,
  p = NULL,
  by = NULL,
  pullTop = TRUE,
  simParamBee = NULL
)
```

Arguments

`multicolony`	`MultiColony-class`
`ID`	character or numeric, ID of a colony (one or more) to be pulled out
`n`	numeric, number of colonies to select
`p`	numeric, percentage of colonies pulled out (takes precedence over `n`)
`by`	matrix, matrix of values to select by with names being colony IDs (can be obtained with `calcColonyValue`. If NULL, the colonies are pulled at random. This parameter is used in combination with `n` or `p` to determine the number of pulled colonies, and `pullTop` to determine whether to pull the best or the worst colonies.
`pullTop`	logical, pull highest (lowest) values if `TRUE` (`FALSE`)
`simParamBee`	`SimParamBee`, global simulation parameters

Value

list with two `MultiColony-class`, the `pulled` and the `remnant`

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

mean <- c(10, 10 / SP$nWorkers)
varA <- c(1, 1 / SP$nWorkers)
corA <- matrix(data = c(
  1.0, -0.5,
  -0.5, 1.0
), nrow = 2, byrow = TRUE)
varE <- c(3, 3 / SP$nWorkers)
varA / (varA + varE)
SP$addTraitADE(nQtlPerChr = 100,
               mean = mean,
               var = varA, corA = corA,
               meanDD = 0.1, varDD = 0.2, corD = corA,
               relAA = 0.1, corAA = corA)
SP$setVarE(varE = varE)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1:4], nInd = 100)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = 10)
apiary <- createMultiColony(basePop[2:5], n = 4)
apiary <- cross(apiary, drones = droneGroups[1:4])
apiary <- buildUp(apiary)
getId(apiary)

tmp <- pullColonies(apiary, ID = c(1, 2))
getId(tmp$pulled)
getId(tmp$remnant)

tmp <- pullColonies(apiary, ID = c("3", "4"))
getId(tmp$pulled)
getId(tmp$remnant)

tmp <- pullColonies(apiary, n = 2)
getId(tmp$pulled)
getId(tmp$remnant)

tmp <- pullColonies(apiary, p = 0.75)
getId(tmp$pulled)
getId(tmp$remnant)

# How to pull out colonies based on colony values?
colonyGv <- calcColonyGv(apiary)
pullColonies(apiary, n = 1, by = colonyGv)
```

`pullDroneGroupsFromDCA` *Pulls drone groups from a Drone Congregation Area (DCA)*

Description

Level 1 function that pulls drone groups from a Drone Congregation Area (DCA) to use them later in mating. Within the function drones are pulled (removed) from the DCA to reflect the fact that drones die after mating, so they can't be present in the DCA anymore. Be careful what you do with the DCA object outside function to avoid drone "copies".

Usage

```
pullDroneGroupsFromDCA(DCA, n, nDrones = NULL, simParamBee = NULL, ...)
```

Arguments

`DCA`	`Pop-class`, population of drones
`n`	integer, number of drone groups to be created
`nDrones`	numeric of function, number of drones that a virgin queen mates with; if `NULL` then `SimParamBee$nFathers` is used
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	additional arguments passed to `nDrones` when this argument is a function

Value

list of `Pop-class`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- addDrones(colony, nInd = 100)

# Create colony DCA
DCA <- createDCA(colony)
pullDroneGroupsFromDCA(DCA, n = 4, nDrones = 5)
pullDroneGroupsFromDCA(DCA, n = 5, nDrones = nFathersPoisson)
```

`pullInd` *Pull individuals from a population*

Description

Level 1 function that pulls individuals from a population and update the population (these individuals don't stay in a population).

Usage

```
pullInd(pop, nInd = NULL, use = "rand", simParamBee = NULL)
```

Arguments

`pop`	`Pop-class`
`nInd`	numeric, number of individuals to pull, if `NULL` pull all individuals
`use`	character, all options provided by `selectInd`
`simParamBee`	`SimParamBee`, global simulation parameters

Value

list with a node `pulled` holding `Pop-class` of pulled individuals and a node `remnant)` holding `Pop-class` of remaining individuals

Examples

```
founderGenomes <- quickHaplo(nInd = 3, nChr = 1, segSites = 100)
SP <- SimParam$new(founderGenomes)

basePop <- newPop(founderGenomes)

pullInd(basePop, nInd = 2)
pullInd(basePop, nInd = 3)
pullInd(basePop)
```

`rcircle` *Sample random points within a circle*

Description

Level 0 function that samples random points (x, y) within a circle via rejection sampling.

Usage

```
rcircle(n = 1, radius = 1, uniform = TRUE, normScale = 1/3)
```

Arguments

`n`	integer, number of samples points
`radius`	numeric, radius of the sampled circle
`uniform`	logical, should sampling be uniform or according to a bi-variate spherical (uncorrelated) Gaussian distribution (see examples)
`normScale`	numeric, if `uniform = FALSE`, a factor to scale radius to standard deviation of the Gaussian density in x and in y (see examples)

Value

matrix with two columns for the x and y coordinates of the points.

References

nubDotDev (2021) The BEST Way to Find a Random Point in a Circle https://youtu.be/4y_nmpv-9lI

Wolfram MathWorld (2023) Disk Point Picking https://mathworld.wolfram.com/DiskPointPicking.html

Examples

```
x <- rcircle(n = 500)
lim <- range(x)
plot(x, xlim = lim, ylim = lim, main = "Uniform")

x <- rcircle(n = 500, uniform = FALSE)
lim <- range(x)
plot(x, xlim = lim, ylim = lim, main = "Gaussian")
```

`reduceDroneGeno` *Reduce drones' genotype to a single haplotype*

Description

Level 0 function that reduces drone's genotype to a single haplotype, because we internally simulate them as diploid (doubled haploid). This is an internal utility function that you likely don't need to use.

Usage

```
reduceDroneGeno(geno, pop)
```

Arguments

`geno`	`matrix-class`
`pop`	`Pop-class`

Value

matrix with genotype as one haplotype per drone instead of two - the order of individuals and the number of rows stays the same!

Examples

```
founderGenomes <- quickHaplo(nInd = 3, nChr = 1, segSites = 5)
SP <- SimParamBee$new(founderGenomes, csdChr = NULL)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 2)

(tmp <- getSegSiteGeno(drones))
reduceDroneGeno(geno = tmp, pop = drones)

(tmp <- getSegSiteGeno(c(basePop, drones)))
reduceDroneGeno(geno = tmp, pop = c(basePop, drones))
```

`reduceDroneHaplo` *Reduce drone's double haplotypes to a single haplotype*

Description

Level 0 function that returns one haplotype of drones, because we internally simulate them as diploid (doubled haploid). This is an internal utility function that you likely don't need to use.

Usage

```
reduceDroneHaplo(haplo, pop)
```

Arguments

`haplo`	`matrix-class`
`pop`	`Pop-class`

Details

While this function is meant to work on male (drone) haplotypes, we handle cases where the `haplo` matrix contains male and female haplotypes, which is why you need to provide pop. We only reduce haplotypes for males though.

Value

matrix with one haplotype per drone instead of two - the order of individuals stays the same, but there will be less rows!

Examples

```
founderGenomes <- quickHaplo(nInd = 3, nChr = 1, segSites = 5)
SP <- SimParamBee$new(founderGenomes, csdChr = NULL)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 2)

(tmp <- getSegSiteHaplo(drones, dronesHaploid = FALSE))
reduceDroneHaplo(haplo = tmp, pop = drones)

(tmp <- getSegSiteHaplo(c(basePop, drones), dronesHaploid = FALSE))
reduceDroneHaplo(haplo = tmp, pop = c(basePop, drones))
```

`removeCastePop` *Remove a proportion of caste individuals from a colony*

Description

Level 2 function that removes a proportion of virgin queens of a Colony or MultiColony object

Usage

```
removeCastePop(
  x,
  caste = NULL,
  p = 1,
  use = "rand",
  addVirginQueens = FALSE,
  nVirginQueens = NULL,
  year = NULL,
  simParamBee = NULL
)

removeQueen(
  x,
  addVirginQueens = FALSE,
  nVirginQueens = NULL,
  year = NULL,
  simParamBee = NULL
)

removeWorkers(x, p = 1, use = "rand", simParamBee = NULL)

removeDrones(x, p = 1, use = "rand", simParamBee = NULL)

removeVirginQueens(x, p = 1, use = "rand", simParamBee = NULL)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`caste`	character, "queen", "workers", "drones", or "virginQueens"
`p`	numeric, proportion to be removed; if input is `MultiColony-class`, the input could also be a vector of the same length as the number of colonies. If a single value is provided, the same value will be applied to all the colonies
`use`	character, all the options provided by `selectInd` - guides selection of virgins queens that will stay when p < 1
`addVirginQueens`	logical, whether virgin queens should be added; only used when removing the queen from the colony
`nVirginQueens`	integer, the number of virgin queens to be created in the colony; only used when removing the queen from the colony. If `0`, no virgin queens are added; If `NULL`, the value from `simParamBee$nVirginQueens` is used
`year`	numeric, only relevant when adding virgin queens - year of birth for virgin queens
`simParamBee`	`SimParamBee`, global simulation parameters

Value

`Colony-class` or `MultiColony-class` without virgin queens

Functions

- `removeQueen()`: Remove queen from a colony
- `removeWorkers()`: Remove workers from a colony
- `removeDrones()`: Remove workers from a colony
- `removeVirginQueens()`: Remove virgin queens from a colony

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 100)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 5, nDrones = nFathersPoisson)

# Create and cross Colony and MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
colony <- buildUp(colony)
apiary <- createMultiColony(basePop[4:5], n = 2)
apiary <- cross(apiary, drones = droneGroups[3:4])
apiary <- buildUp(apiary)
```

```
# Remove workers
nWorkers(colony)
colony <- removeCastePop(colony, caste = "workers", p = 0.3)
# or alias:
colony <- removeWorkers(colony, p = 0.3)
# Same aliases exist for all the castes!!

nWorkers(apiary)
apiary <- removeCastePop(apiary, caste = "workers", p = 0.3)
nWorkers(apiary)

# Remove different proportions
apiary <- buildUp(apiary)
nWorkers(apiary)
nWorkers(removeWorkers(apiary, p = c(0.1, 0.5)))
```

`removeColonies` *Remove some colonies from the MultiColony object*

Description

Level 3 function that removes some colonies from the MultiColony object based on their ID.

Usage

```
removeColonies(
  multicolony,
  ID = NULL,
  n = NULL,
  p = NULL,
  by = NULL,
  removeTop = FALSE,
  simParamBee = NULL
)
```

Arguments

`multicolony`	`MultiColony-class`
`ID`	character or numeric, ID of a colony (one or more) to be removed
`n`	numeric, number of colonies to remove
`p`	numeric, percentage of colonies removed (takes precedence over `n`)
`by`	matrix, matrix of values to select by with names being colony IDs (can be obtained with `calcColonyValue`. If NULL, the colonies are removed at random. This parameter is used in combination with `n` or `p` to determine the number of removed colonies, and `removeTop` to determine whether to remove the best or the worst colonies.
`removeTop`	logical, remove highest (lowest) values if `TRUE` (`FALSE`)
`simParamBee`	`SimParamBee`, global simulation parameters

Value

MultiColony-class with some colonies removed

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

mean <- c(10, 10 / SP$nWorkers)
varA <- c(1, 1 / SP$nWorkers)
corA <- matrix(data = c(
  1.0, -0.5,
  -0.5, 1.0
), nrow = 2, byrow = TRUE)
varE <- c(3, 3 / SP$nWorkers)
varA / (varA + varE)
SP$addTraitADE(nQtlPerChr = 100,
               mean = mean,
               var = varA, corA = corA,
               meanDD = 0.1, varDD = 0.2, corD = corA,
               relAA = 0.1, corAA = corA)
SP$setVarE(varE = varE)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1:4], nInd = 100)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = 10)
apiary <- createMultiColony(basePop[2:5], n = 4)
apiary <- cross(apiary, drones = droneGroups[1:4])
apiary <- buildUp(apiary)
getId(apiary)

getId(removeColonies(apiary, ID = 1))
getId(removeColonies(apiary, ID = c("3", "4")))

nColonies(apiary)
apiary <- removeColonies(apiary, ID = "2")
nColonies(apiary)

# How to remove colonies based on colony values?
# Obtain colony phenotype
colonyPheno <- calcColonyPheno(apiary)
# Remove the worst colony
removeColonies(apiary, n = 1, by = colonyPheno)
```

`replaceCastePop` *Replace a proportion of caste individuals with new ones*

Description

Level 2 function that replaces a proportion of caste individuals with new individuals from a Colony or MultiColony object. Useful after events like season change, swarming, supersedure, etc. due to the short life span honeybees.

Usage

```
replaceCastePop(
  x,
  caste = NULL,
  p = 1,
  use = "rand",
  exact = TRUE,
  year = NULL,
  simParamBee = NULL
)

replaceWorkers(x, p = 1, use = "rand", exact = TRUE, simParamBee = NULL)

replaceDrones(x, p = 1, use = "rand", simParamBee = NULL)

replaceVirginQueens(x, p = 1, use = "rand", simParamBee = NULL)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`caste`	character, "workers", "drones", or "virginQueens"
`p`	numeric, proportion of caste individuals to be replaced with new ones; if input is `MultiColony-class`, the input could also be a vector of the same length as the number of colonies. If a single value is provided, the same value will be applied to all the colonies
`use`	character, all the options provided by `selectInd` - guides selection of caste individuals that stay when `p < 1`
`exact`	logical, only relevant when adding workers - if the csd locus is turned on and exact is `TRUE`, we replace the exact specified number of viable workers (heterozygous at the csd locus). You probably want this set to TRUE since you want to replace with the same number of workers.
`year`	numeric, only relevant when replacing virgin queens, year of birth for virgin queens
`simParamBee`	`SimParamBee`, global simulation parameters

Value

`Colony-class` or or `MultiColony-class` with replaced virgin queens

Functions

- `replaceWorkers()`: Replaces some workers in a colony
- `replaceDrones()`: Replaces some drones in a colony
- `replaceVirginQueens()`: Replaces some virgin queens in a colony

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 100)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 5, nDrones = nFathersPoisson)

# Create and cross Colony and MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[4:5], n = 2)
apiary <- cross(apiary, drones = droneGroups[3:4])

# Add individuals
colony <- buildUp(colony, nWorkers = 5, nDrones = 2)
apiary <- buildUp(apiary, nWorkers = 5, nDrones = 2)

# Replace workers in a colony
getCasteId(colony, caste = "workers")
colony <- replaceCastePop(colony, caste = "workers", p = 0.5)
# You can also use an alias
replaceWorkers(colony, p = 0.5)
# Same aliases exist for all the castes!!!
getCasteId(colony, caste = "workers")

getCasteId(apiary, caste="workers")
apiary <- replaceWorkers(apiary, p = 0.5)
getCasteId(apiary, caste="workers")
```

`reQueen` *Re-queen*

Description

Level 2 function that re-queens a Colony or MultiColony object by adding a mated or a virgin queen, removing the previous queen, and changing the colony id to the new mated queen.

Usage

```
reQueen(x, queen, removeVirginQueens = TRUE, simParamBee = NULL)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`queen`	`Pop-class` with one individual that will be the queen of the colony; if she is not mated, she will be added as a virgin queen that will have to be mated later; test will be run if the individual `isVirginQueen` or `isQueen`
`removeVirginQueens`	logical, remove existing virgin queens, default is `TRUE` since bee-keepers tend to remove any virgin queen cells to ensure the provided queen prevails (see details)
`simParamBee`	`SimParamBee`, global simulation parameters

Details

If the provided queen is mated, then she is saved in the queen slot of the colony. If she is not mated, then she is saved in the virgin queen slot (replacing any existing virgin queens) and once she is mated will be promoted to the queen of the colony.

Value

`Colony-class` or `MultiColony-class` with new queen(s) (see details)

Examples

```
founderGenomes <- quickHaplo(nInd = 12, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 200)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 7, nDrones = nFathersPoisson)

# Create and cross Colony and MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[2:3])

# Check queen and virgin queens IDs
getCasteId(colony, caste = "queen")
getCasteId(colony, caste = "virginQueens")
getCasteId(apiary, caste = "queen")
getCasteId(apiary, caste = "virginQueens")

# Requeen with virgin queens
virginQueens <- basePop[5:8]
# Requeen a Colony class
colony <- reQueen(colony, queen = virginQueens[1])
# Check queen and virgin queens IDs
getCasteId(colony, caste = "queen")
getCasteId(colony, caste = "virginQueens")
```

```
#' # Requeen with mated queens
matedQueens <- cross(x = basePop[9:12], drones = droneGroups[4:7])
colony <- reQueen(colony, queen = matedQueens[1])
# Check queen and virgin queens IDs
getCasteId(colony, caste = "queen")
getCasteId(colony, caste = "virginQueens")

# Requeen a MultiColony class
apiary <- reQueen(apiary, queen = virginQueens[2:3])
# Check queen and virgin queens IDs
getCasteId(apiary, caste = "queen")
getCasteId(apiary, caste = "virginQueens")
```

`resetEvents` *Reset colony events*

Description

Level 2 function that resets the slots swarm, split, supersedure, collapsed, and production to FALSE in a Colony or MultiColony object. Useful at the end of a yearly cycle to reset the events, allowing the user to track new events in a new year.

Usage

```
resetEvents(x, collapse = NULL)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`collapse`	logical, reset the collapse event (only sensible in setting up a new colony, which the default of `NULL` caters for; otherwise, a collapsed colony should be left collapsed forever, unless you force resetting this event with `collapse = TRUE`)

Value

`Colony-class` or `MultiColony-class` with events reset

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 100)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 5, nDrones = nFathersPoisson)

# Create and cross Colony and MultiColony class
```

```
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[4:5], n = 2)
apiary <- cross(apiary, drones = droneGroups[3:4])

# Build-up - this sets Productive to TRUE
(colony <- buildUp(colony, nWorkers = 100))
isProductive(colony)
resetEvents(colony)

apiary <- buildUp(apiary, nWorkers = 100)
isProductive(apiary)
resetEvents(apiary)

# Split - this sets Split to TRUE
tmp <- split(colony)
(split <- tmp$split)
hasSplit(split)
resetEvents(split)
(remnant <- tmp$remnant)
hasSplit(remnant)
resetEvents(remnant)

# Swarm - this sets Swarm to TRUE
tmp <- swarm(colony)
(swarm <- tmp$swarm)
hasSwarmed(swarm)
resetEvents(swarm)
(remnant <- tmp$remnant)
hasSwarmed(remnant)
resetEvents(remnant)

# Supersede - this sets Supersede to TRUE
(tmp <- supersede(colony))
hasSuperseded(tmp)
resetEvents(tmp)

# Collapse - this sets Collapse to TRUE
(tmp <- collapse(colony))
hasCollapsed(tmp)
resetEvents(tmp)
resetEvents(tmp, collapse = TRUE)

# Same behaviour for MultiColony (example for the split)
tmp <- split(apiary)
(splits <- tmp$split)
hasSplit(splits[[1]])
resetEvents(splits)[[1]]
(remnants <- tmp$remnant)
hasSplit(remnants[[1]])
resetEvents(remnants)[[1]]
```

`selectColonies` *Select colonies from MultiColony object*

Description

Level 3 function that selects colonies from MultiColony object based on colony ID or random selection. Whilst user can provide all three arguments ID, p and n, there is a priority list: ID takes first priority. If no ID is provided, p takes precedence over n.

Usage

```
selectColonies(
  multicolony,
  ID = NULL,
  n = NULL,
  p = NULL,
  by = NULL,
  selectTop = TRUE,
  simParamBee = NULL
)
```

Arguments

`multicolony`	`MultiColony-class`
`ID`	character or numeric, ID of a colony (one or more) to be selected
`n`	numeric, number of colonies to select
`p`	numeric, percentage of colonies selected (takes precedence over `n`)
`by`	matrix, matrix of values to select by with names being colony IDs (can be obtained with `calcColonyValue`. If NULL, the colonies are selected at random. This parameter is used in combination with `n` or `p` to determine the number of selected colonies, and `selectTop` to determine whether to select the best or the worst colonies.
`selectTop`	logical, selects highest (lowest) values if `TRUE` (`FALSE`)
`simParamBee`	`SimParamBee`, global simulation parameters

Value

`MultiColony-class` with selected colonies

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

mean <- c(10, 10 / SP$nWorkers)
varA <- c(1, 1 / SP$nWorkers)
```

```
corA <- matrix(data = c(
  1.0, -0.5,
  -0.5, 1.0
), nrow = 2, byrow = TRUE)
varE <- c(3, 3 / SP$nWorkers)
varA / (varA + varE)
SP$addTraitADE(nQtlPerChr = 100,
               mean = mean,
               var = varA, corA = corA,
               meanDD = 0.1, varDD = 0.2, corD = corA,
               relAA = 0.1, corAA = corA)
SP$setVarE(varE = varE)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1:4], nInd = 100)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = 10)
apiary <- createMultiColony(basePop[2:5], n = 4)
apiary <- cross(apiary, drones = droneGroups[1:4])
apiary <- buildUp(apiary)
getId(apiary)

getId(selectColonies(apiary, ID = 1))
getId(selectColonies(apiary, ID = c("3", "4")))
# ... alternative
getId(apiary[1])
getId(apiary[["4"]])

# Select a random number of colonies
selectColonies(apiary, n = 3)
# Select a percentage of colonies
selectColonies(apiary, p = 0.2)

# Since selection is random, you would get a different set of colonies with
# each function call
getId(selectColonies(apiary, p = 0.5))
getId(selectColonies(apiary, p = 0.5))

# How to select colonies based on colony values?
# Obtain colony phenotype
colonyPheno <- calcColonyPheno(apiary)
# Select the best colony
selectColonies(apiary, n = 1, by = colonyPheno)

# Select the worst 2 colonies
selectColonies(apiary, n = 2, by = colonyPheno, selectTop = FALSE)

# Select best colony based on queen's genetic value for trait 1
queenGv <- calcColonyGv(apiary, FUN = mapCasteToColonyGv, workersTrait = NULL)
selectColonies(apiary, n = 1, by = queenGv)
```

---

setLocation *Set colony location*

Description

Level 2 function that to set a Colony or MultiColony object location to (x, y) coordinates.

Usage

```
setLocation(x, location = c(0, 0))
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`location`	numeric, list, or data.frame, x and y coordinates of colony locations as `c(x1, y1)` (the same location set to all colonies), `list(c(x1, y1), c(x2, y2))`, or `data.frame(x = c(x1, x2), y = c(y1, y2))`

Value

`Colony-class` or `MultiColony-class` with set location

Examples

```
founderGenomes <- quickHaplo(nInd = 5, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(basePop[1], n = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 4, nDrones = 10)

# Create Colony and MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[3:5])
apiary <- cross(apiary, drones = droneGroups[2:4])

getLocation(colony)
getLocation(apiary)

loc <- c(1, 1)
colony <- setLocation(colony, location = loc)
getLocation(colony)

# Assuming one location (as in bringing colonies to one place!)
apiary <- setLocation(apiary, location = loc)
getLocation(apiary)

# Assuming different locations
```

```
locList <- list(c(0, 0), c(1, 1), c(2, 2))
apiary <- setLocation(apiary, location = locList)
getLocation(apiary)

locDF <- data.frame(x = c(0, 1, 2), y = c(0, 1, 2))
apiary <- setLocation(apiary, location = locDF)
getLocation(apiary)
```

`setMisc` *Set miscellaneous information in a population*

Description

Set miscellaneous information in a population

Usage

```
setMisc(x, node = NULL, value = NULL)
```

Arguments

`x`	`Pop-class`
`node`	character, name of the node to set within the `x@misc` slot
`value`	value to be saved into `x@misc[[*]][[node]]`; length of `value` should be equal to `nInd(x)`; if its length is 1, then it is repeated using `rep` (see examples)

Details

A `NULL` in `value` is ignored

Value

`Pop-class`

`setQueensYearOfBirth` *Set the queen's year of birth*

Description

Level 1 function that sets the queen's year of birth.

Usage

```
setQueensYearOfBirth(x, year, simParamBee = NULL)
```

Arguments

x	Pop-class (one or more than one queen), Colony-class (one colony), or MultiColony-class (more colonies)
year	integer, the year of the birth of the queen
simParamBee	SimParamBee, global simulation parameters

Value

Pop-class, Colony-class, or MultiColony-class with queens having the year of birth set

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = nFathersPoisson)

# Create a Colony and a MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(x = colony, drones = droneGroups[[1]])
apiary <- createMultiColony(basePop[3:4], n = 2)
apiary <- cross(apiary, drones = droneGroups[c(2, 3)])

# Example on Colony class
getQueenYearOfBirth(colony)
getQueenYearOfBirth(apiary)

queen1 <- getQueen(colony)
queen1 <- setQueensYearOfBirth(queen1, year = 2022)
getQueenYearOfBirth(queen1)

colony <- setQueensYearOfBirth(colony, year = 2022)
getQueenYearOfBirth(colony)

apiary <- setQueensYearOfBirth(apiary, year = 2022)
getQueenYearOfBirth(apiary)
```

`SimParamBee` *Honeybee simulation parameters*

Description

Container for global honeybee simulation parameters. Saving this object as `SP` will allow it to be accessed by SIMplyBee functions without repeatedly (and annoyingly!) typing out `someFun(argument, simParamBee = SP)`. `SimParamBee` inherits from AlphaSimR `SimParam`, so all `SimParam` slots and functions are available in addition to `SimParamBee`-specific slots and functions. Some `SimParam` functions could have upgraded behaviour as documented in line with honeybee biology.

Details

This documentation shows details specific to `SimParamBee`. We suggest you also read all the options provided by the AlphaSimR `SimParam`. Below we show minimal usage cases for each `SimParamBee` function.

See also `vignette(package = "SIMplyBee")` for descriptions of how SIMplyBee implements the specific honeybee biology.

Super class

`AlphaSimR::SimParam` -> `SimParamBee`

Public fields

`nWorkers` numeric or function, a number of workers generated in a colony - used in `createWorkers`, `addWorkers`, `buildUp`.

The default value is 100, that is, queen generates 100 workers - this is for a down-scaled simulation (for efficiency) assuming that this represents ~60,000 workers in a full/strong colony (Seeley, 2019). This value is set in `SimParamBee$new()` to have a number to work with.

You can change this setting to your needs!

When `nWorkers` is a function, it should work with internals of other functions. Therefore, the function MUST be defined like `function(colony, arg = default) someCode` , that is, the first argument MUST be `colony` and any following arguments MUST have a default value. For flexibility you can add ... argument to pass on any other argument. See `nWorkersPoisson`, `nWorkersTruncPoisson`, or `nWorkersColonyPhenotype` for examples.

You can provide your own functions that satisfy your needs!

`nDrones` numeric or function, a number of drones generated in a colony - used in `createDrones`, `addDrones`, `buildUp`.

The default value is 100, that is, queen generates 100 drones - this is for a down-scaled simulation (for efficiency) assuming that this represents ~1,000 drones in a full/strong colony (Seeley, 2019). This value is set in `SimParamBee$new()` to have a number to work with.

You can change this setting to your needs!

When `nDrones` is a function, it should work with internals of other functions. Therefore, the function MUST be defined like `function(x, arg = default) someCode` , that is, the first argument MUST be `x` and any following arguments MUST have a default value. For flexibility you can add ... argument to pass on any other argument. See `nDronesPoisson`, `nDronesTruncPoisson`, or `nDronesColonyPhenotype` for examples.

You can provide your own functions that satisfy your needs!

`nVirginQueens` numeric or function, a number of virgin queens generated when a queen dies or other situations - used in `createVirginQueens` and `addVirginQueens`.

The default value is 10, that is, when the queen dies, workers generate 10 new virgin queens (Seeley, 2019). This value is set in `SimParamBee$new()` to have a number to work with.

You can change this setting to your needs!

When `nVirginQueens` is a function, it should work with internals of other functions. Therefore, the function MUST be defined like `function(colony, arg = default) someCode` , that is, the first argument MUST be `colony` and any following arguments MUST have a default value. For flexibility you can add ... argument to pass on any other argument. See `nVirginQueensPoisson`, `nVirginQueensTruncPoisson`, or `nVirginQueensColonyPhenotype` for examples.

You can provide your own functions that satisfy your needs!

`nFathers` numeric or function, a number of drones a queen mates with - used in `pullDroneGroupsFromDCA`, `cross`.

The default value is 15, that is, a virgin queen mates on average with 15 drones (Seeley, 2019). This value is set in `SimParamBee$new()` to have a number to work with.

You can change this setting to your needs!

When `nFathers` is a function, it should work with internals of other functions. Therefore, the function MUST be defined like `function(arg = default) someCode` , that is, any arguments MUST have a default value. We did not use the `colony` argument here, because `nFathers` likely does not depend on the colony. Let us know if we are wrong! For flexibility you can add ... argument to pass on any other argument. See `nFathersPoisson` or `nFathersTruncPoisson` for examples.

You can provide your own functions that satisfy your needs!

`swarmP` numeric or a function, the swarm proportion - the proportion of workers that leave with the old queen when the colony swarms - used in `swarm`.

The default value is 0.50, that is, about a half of workers leave colony in a swarm (Seeley, 2019). This value is set in `SimParamBee$new()` to have a proportion to work with.

You can change this setting to your needs!

When `swarmP` is a function, it should work with internals of other functions. Therefore, the function MUST be defined like `function(colony, arg = default) someCode` , that is, the first argument MUST be `colony` and any following arguments MUST have a default value. For flexibility you can add ... argument to pass on any other argument. See `swarmPUnif` for examples.

You can provide your own functions that satisfy your needs!

`swarmRadius` numeric, radius within which to sample a location of of the swarm - used in `swarm` - see its `radius` argument.

The default value is `0`, that is, swarm gets the same location as the original colony.

You can change this setting to your needs!

`splitP` numeric or a function, the split proportion - the proportion of workers removed in a managed split - used in `split`.

The default value is 0.30, that is, about a third of workers is put into a split colony from a strong colony (Seeley, 2019). This value is set in `SimParamBee$new()` to have a proportion to work with.

You can change this setting to your needs!

When `splitP` is a function, it should work with internals of other functions. Therefore, the function MUST be defined like `function(colony, arg = default) someCode` , that is, the first argument MUST be `colony` and any following arguments MUST have a default value. For flexibility you can add ... argument to pass on any other argument. See `splitPUnif` or `splitPColonyStrength` for examples.

You can provide your own functions that satisfy your needs!

`downsizeP` numeric or a function, the downsize proportion - the proportion of workers removed from the colony when downsizing, usually in autumn - used in `downsize`.

The default value is 0.85, that is, a majority of workers die before autumn or all die but some winter workers are created (Seeley, 2019). This value is set in `SimParamBee$new()` to have a proportion to work with.

You can change this setting to your needs!

When `downsizeP` is a function, it should work with internals of other functions. Therefore, the function MUST be defined like `function(colony, arg = default) someCode` , that is, the first argument MUST be `colony` and any following arguments MUST have a default value. For flexibility you can add ... argument to pass on any other argument. See `downsizePUnif` for example.

You can provide your own functions that satisfy your needs!

`colonyValueFUN` function, to calculate colony values - used in `calcColonyValue` - see also `calcColonyPheno` and `calcColonyGv`.

This function should work with internals of others functions - therefore the function MUST be defined like `function(colony, arg = default) someCode` , that is, the first argument MUST be `colony` and any following arguments MUST have a default value. For flexibility you can add ... argument to pass on any other argument. See `mapCasteToColonyValue` for an example.

You can provide your own functions that satisfy your needs!

Active bindings

`caste` character, caste information for every individual ever created

`lastColonyId` integer, ID of the last Colony object created with `createColony`

`csdChr` integer, chromosome of the csd locus

`csdPos` numeric, starting position of the csd locus on the `csdChr` chromosome (relative at the moment, but could be in base pairs in the future)

`nCsdAlleles` integer, number of possible csd alleles

`nCsdSites` integer, number of segregating sites representing the csd locus

`csdPosStart` integer, starting position of the csd locus

`csdPosStop` integer, ending position of the csd locus

`version` list, versions of AlphaSimR and SIMplyBee packages used to generate this object

Methods

Public methods:

- `SimParamBee$new()`
- `SimParamBee$addToCaste()`
- `SimParamBee$changeCaste()`
- `SimParamBee$updateLastColonyId()`
- `SimParamBee$clone()`

Method `new()`**:** Starts the process of building a new simulation by creating a new SimParamBee object and assigning a founder population of genomes to the this object.

Usage:

```
SimParamBee$new(
  founderPop,
  nWorkers = 100,
  nDrones = 100,
  nVirginQueens = 10,
  nFathers = 15,
  swarmP = 0.5,
  swarmRadius = 0,
  splitP = 0.3,
  downsizeP = 0.85,
  csdChr = 3,
  csdPos = 0.865,
  nCsdAlleles = 128,
  colonyValueFUN = NULL
)
```

Arguments:

`founderPop` `MapPop-class`, founder population of genomes

`nWorkers` see `SimParamBee` field `nWorkers`

`nDrones` see `SimParamBee` field `nDrones`

`nVirginQueens` see `SimParamBee` field `nVirginQueens`

`nFathers` see `SimParamBee` field `nFathers`

`swarmP` see `SimParamBee` field `swarmP`

`swarmRadius` see `SimParamBee` field `swarmRadius`

`splitP` see `SimParamBee` field `splitP`

`downsizeP` see `SimParamBee` field `downsizeP`

`csdChr` integer, chromosome that will carry the csd locus, by default 3, but if there are less chromosomes (for a simplified simulation), the locus is put on the last available chromosome (1 or 2); if `NULL` then csd locus is ignored in the simulation

`csdPos` numeric, starting position of the csd locus on the `csdChr` chromosome (relative at the moment, but could be in base pairs in future)

`nCsdAlleles` integer, number of possible csd alleles (this determines how many segregating sites will be needed to represent the csd locus from the underlying bi-allelic SNP; the minimum number of bi-allelic SNP needed is `log2(nCsdAlleles)`); if set to `0` then `csdChr=NULL` is triggered. By default we set `nCsdAlleles` to 128, which is at the upper end of the reported number of csd alleles (Lechner et al., 2014; Zareba et al., 2017; Bovo et al., 2021).

`colonyValueFUN` see `SimParamBee` field `colonyValueFUN`

Examples:

```
founderGenomes <- quickHaplo(nInd = 10, nChr = 3, segSites = 10)
SP <- SimParamBee$new(founderGenomes, nCsdAlleles = 2)
\dontshow{SP$nThreads = 1L}

# We need enough segregating sites
try(SP <- SimParamBee$new(founderGenomes, nCsdAlleles = 100))
\dontshow{SP$nThreads = 1L}
founderGenomes <- quickHaplo(nInd = 10, nChr = 3, segSites = 100)
```

```
SP <- SimParamBee$new(founderGenomes, nCsdAlleles = 100)
\dontshow{SP$nThreads = 1L}

# We can save the csd locus on chromosome 1 or 2, too, for quick simulations
founderGenomes <- quickHaplo(nInd = 10, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes, nCsdAlleles = 100)
\dontshow{SP$nThreads = 1L}
```

Method `addToCaste()`**:** Store caste information (for internal use only!)

Usage:

```
SimParamBee$addToCaste(id, caste)
```

Arguments:

`id` character, individuals whose caste will be stored

`caste` character, single "Q" for queens, "W" for workers, "D" for drones, "V" for virgin queens, and "F" for fathers

Examples:

```
founderGenomes <- quickHaplo(nInd = 2, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)
\dontshow{SP$nThreads = 1L}
SP$setTrackPed(isTrackPed = TRUE)
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 10)
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = drones)
colony <- addWorkers(colony, nInd = 5)
colony <- addDrones(colony, nInd = 5)
colony <- addVirginQueens(colony, nInd = 2)

SP$pedigree
SP$caste
```

Method `changeCaste()`**:** Change caste information (for internal use only!)

Usage:

```
SimParamBee$changeCaste(id, caste)
```

Arguments:

`id` character, individuals whose caste will be changed

`caste` character, single "Q" for queens, "W" for workers, "D" for drones, "V" for virgin queens, and "F" for fathers

Examples:

```
founderGenomes <- quickHaplo(nInd = 2, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)
\dontshow{SP$nThreads = 1L}
SP$setTrackPed(isTrackPed = TRUE)
basePop <- createVirginQueens(founderGenomes)
```

```
SP$pedigree
SP$caste

drones <- createDrones(x = basePop[1], nInd = 10)
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = drones)
SP$pedigree
SP$caste
```

Method `updateLastColonyId()`**:** A function to update the colony last ID everytime we create a Colony-class with createColony. For internal use only.

Usage:

```
SimParamBee$updateLastColonyId()
```

Arguments:

`lastColonyId` integer, last colony ID assigned

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SimParamBee$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Bovo et al. (2021) Application of Next Generation Semiconductor-Based Sequencing for the Identification of Apis mellifera Complementary Sex Determiner (csd) Alleles from Honey DNA. Insects, 12(10), 868. doi:10.3390/insects12100868

Lechner et al. (2014) Nucleotide variability at its limit? Insights into the number and evolutionary dynamics of the sex-determining specificities of the honey bee Apis mellifera Molecular Biology and Evolution, 31, 272-287. doi:10.1093/molbev/mst207

Seeley (2019) The Lives of Bees: The Untold Story of the Honey Bee in the Wild. Princeton: Princeton University Press. doi:10.1515/9780691189383

Zareba et al. (2017) Uneven distribution of complementary sex determiner (csd) alleles in Apis mellifera population. Scientific Reports, 7, 2317. doi:10.1038/s41598017026299

Examples

```
## ------------------------------------------------
## Method `SimParamBee$new`
## ------------------------------------------------

founderGenomes <- quickHaplo(nInd = 10, nChr = 3, segSites = 10)
SP <- SimParamBee$new(founderGenomes, nCsdAlleles = 2)


# We need enough segregating sites
try(SP <- SimParamBee$new(founderGenomes, nCsdAlleles = 100))
```

```
founderGenomes <- quickHaplo(nInd = 10, nChr = 3, segSites = 100)
SP <- SimParamBee$new(founderGenomes, nCsdAlleles = 100)


# We can save the csd locus on chromosome 1 or 2, too, for quick simulations
founderGenomes <- quickHaplo(nInd = 10, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes, nCsdAlleles = 100)


## ------------------------------------------------
## Method `SimParamBee$addToCaste`
## ------------------------------------------------

founderGenomes <- quickHaplo(nInd = 2, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

SP$setTrackPed(isTrackPed = TRUE)
basePop <- createVirginQueens(founderGenomes)

drones <- createDrones(x = basePop[1], nInd = 10)
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = drones)
colony <- addWorkers(colony, nInd = 5)
colony <- addDrones(colony, nInd = 5)
colony <- addVirginQueens(colony, nInd = 2)

SP$pedigree
SP$caste

## ------------------------------------------------
## Method `SimParamBee$changeCaste`
## ------------------------------------------------

founderGenomes <- quickHaplo(nInd = 2, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

SP$setTrackPed(isTrackPed = TRUE)
basePop <- createVirginQueens(founderGenomes)
SP$pedigree
SP$caste

drones <- createDrones(x = basePop[1], nInd = 10)
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = drones)
SP$pedigree
SP$caste
```

`simulateHoneyBeeGenomes`

Simulate the Honey bee genome

Description

Level 0 function that returns simulated honeybee genomes

Usage

```
simulateHoneyBeeGenomes(
  nMelN = 0L,
  nMelS = 0L,
  nCar = 0L,
  nLig = 0L,
  Ne = 170000L,
  ploidy = 2L,
  nChr = 16L,
  nSegSites = 100L,
  nBp = 225200000/16,
  genLen = 3.199121,
  mutRate = 3.4e-09,
  recRate = 2.3e-07,
  nThreads = NULL
)
```

Arguments

`nMelN`	integer, number of Apis mellifera mellifera North individuals to simulate
`nMelS`	integer, number of Apis mellifera mellifera South individuals to simulate
`nCar`	integer, number of Apis mellifera carnica individuals to simulate
`nLig`	integer, number of Apis mellifera ligustica individuals to simulate
`Ne`	integer, effective size of the simulated population. Currently set to 170,000, according to Wallberg et al., 2014. Would discourage you to change it since it is linked to the parameters of the demographic model we use for the simulation. However, there might be some edge cases when using a different Ne is necessary, but proceed with caution.
`ploidy`	integer, the ploidy of the individuals
`nChr`	integer, number of chromosomes to simulate
`nSegSites`	integer, number of segregating sites to keep per chromosome
`nBp`	integer, base pair length of chromosome
`genLen`	numeric, genetic length of chromosome in Morgans
`mutRate`	numeric, per base pair mutation rate
`recRate`	numeric, per base pair recombination rate
`nThreads`	integer, if OpenMP is available, this will allow for simulating chromosomes in parallel. If `NULL`, the number of threads is automatically detected

Value

`MapPop-class`

References

Wallberg, A., Bunikis, I., Pettersson, O.V. et al. A hybrid de novo genome assembly of the honeybee, Apis mellifera, with chromosome-length scaffolds. 2019, BMC Genomics 20:275. doi:10.1186/s1286401956420

Beye M, Gattermeier I, Hasselmann M, et al. Exceptionally high levels of recombination across the honey bee genome. 2006, Genome Res 16(11):1339-1344. doi:10.1101/gr.5680406

Wallberg, A., Han, F., Wellhagen, G. et al. A worldwide survey of genome sequence variation provides insight into the evolutionary history of the honeybee Apis mellifera. 2014, Nat Genet 46:1081–1088. doi:10.1038/ng.3077

Yang S, Wang L, Huang J, Zhang X, Yuan Y, Chen JQ, Hurst LD, Tian D. Parent-progeny sequencing indicates higher mutation rates in heterozygotes. 2015, Nature 523(7561):463-7. doi:10.1038/nature14649.

See Also

Due to the computational time and resources required to run this function, we do not include an example here, but we demonstrate its use in the Honeybee biology vignette.

`split` *Split colony in two MultiColony*

Description

Level 2 function that splits a Colony or MultiColony object into two new colonies to prevent swarming (in managed situation). The remnant colony retains the queen and a proportion of the workers and all drones. The split colony gets the other part of the workers, which raise virgin queens, of which only one prevails. Location of the split is the same as for the remnant.

Usage

```
split(x, p = NULL, year = NULL, simParamBee = NULL, ...)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`p`	numeric, proportion of workers that will go to the split colony; if `NULL` then `SimParamBee$splitP` is used. If input is `MultiColony-class`, the input could also be a vector of the same length as the number of colonies. If a single value is provided, the same value will be applied to all the colonies
`year`	numeric, year of birth for virgin queens
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	additional arguments passed to `p` when this argument is a function

Value

list with two `Colony-class` or `MultiColony-class`, the `split` and the `remnant` (see the description what each colony holds!); both outputs have the split even slot set do `TRUE`

Examples

```
founderGenomes <- quickHaplo(nInd = 10, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(basePop[1], n = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = 10)

# Create Colony and MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
(colony <- buildUp(colony, nWorkers = 100))
apiary <- createMultiColony(basePop[3:8], n = 6)
apiary <- cross(apiary, drones = droneGroups[2:7])
apiary <- buildUp(apiary, nWorkers = 100)

# Split a colony
tmp <- split(colony)
tmp$split
tmp$remnant

# Split all colonies in the apiary with p = 0.5 (50% of workers in each split)
tmp <- split(apiary, p = 0.5)
tmp$split[[1]]
tmp$remnant[[1]]
# Split with different proportions
nWorkers(apiary)
tmp <- split(apiary, p = c(0.1, 0.2, 0.3, 0.4, 0.5, 0.6))
nWorkers(tmp$split)
nWorkers(tmp$remnant)

# Split only specific colonies in the apiary
tmp <- pullColonies(apiary, ID = c(4, 5))
# Split only the pulled colonies
(split(tmp$pulled, p = 0.5))
```

`splitPUnif` *Sample the split proportion - proportion of removed workers in a managed split*

Description

Sample the split proportion - proportion of removed workers in a managed split - used when `p = NULL` - (see `SimParamBee$splitP`).

This is just an example. You can provide your own functions that satisfy your needs!

Usage

```
splitPUnif(colony, n = 1, min = 0.2, max = 0.4)

splitPColonyStrength(colony, n = 1, nWorkersFull = 100, scale = 1)
```

Arguments

`colony`	`Colony-class`
`n`	integer, number of samples
`min`	numeric, lower limit for `splitPUnif`
`max`	numeric, upper limit for `splitPUnif`
`nWorkersFull`	numeric, average number of workers in a full/strong colony for `splitPColonyStrength` (actual number can go beyond this value)
`scale`	numeric, scaling of numbers in `splitPColonyStrength` to avoid to narrow range when colonies have a large number of bees (in that case change `nWorkersFull` too!)

Details

`splitPUnif` samples from a uniform distribution between values 0.2 and 0.4 irrespective of colony strength.

`splitPColonyStrength` samples from a beta distribution with mean `a / (a + b)`, where `a = nWorkers + nWorkersFull` and `b = nWorkers`. This beta sampling mimics larger splits for strong colonies and smaller splits for weak colonies - see examples. This is just an example - adapt to your needs!

The `nWorkersFull` default value used in this function is geared towards a situation where we simulate ~100 workers per colony (down-scaled simulation for efficiency). If you simulate more workers, you should change the default accordingly.

Value

numeric, split proportion

Functions

- `splitPColonyStrength()`: Sample the split proportion - the proportion of removed workers in a managed split based on the colony strength

See Also

`SimParamBee` field `splitP`

Examples

```
splitPUnif()
splitPUnif()
p <- splitPUnif(n = 1000)
hist(p, breaks = seq(from = 0, to = 1, by = 0.01), xlim = c(0, 1))
```

```
# Example for splitPColonyStrength()
founderGenomes <- quickHaplo(nInd = 2, nChr = 1, segSites = 100)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(x = basePop[1], nInd = 15)
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = drones)
colony <- addWorkers(colony, nInd = 10)
nWorkers(colony) # weak colony
splitPColonyStrength(colony)
splitPColonyStrength(colony)
colony <- addWorkers(colony, nInd = 100)
nWorkers(colony) # strong colony
splitPColonyStrength(colony)
splitPColonyStrength(colony)

# Logic behind splitPColonyStrength()
nWorkersFull <- 100
nWorkers <- 0:200
splitP <- 1 - rbeta(
  n = length(nWorkers),
  shape1 = nWorkers + nWorkersFull,
  shape2 = nWorkers
)
plot(splitP ~ nWorkers, ylim = c(0, 1))
abline(v = nWorkersFull)
pKeep <- 1 - splitP
plot(pKeep ~ nWorkers, ylim = c(0, 1))
abline(v = nWorkersFull)
```

`supersede` *Supersede*

Description

Level 2 function that supersedes a Colony or MultiColony object - an event where the queen dies. The workers and drones stay unchanged, but workers raise virgin queens, of which only one prevails.

Usage

```
supersede(x, year = NULL, nVirginQueens = NULL, simParamBee = NULL, ...)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`year`	numeric, year of birth for virgin queens

`nVirginQueens`	integer, the number of virgin queens to be created in the colony; of these one is randomly selected as the new virgin queen of the remnant colony. If `NULL`, the value from `simParamBee$nVirginQueens` is used
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	additional arguments passed to `nVirginQueens` when this argument is a function

Value

`Colony-class` or `MultiColony-class` with the supersede event set to `TRUE`

Examples

```
founderGenomes <- quickHaplo(nInd = 10, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(basePop[1], n = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = 10)

# Create Colony and MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
(colony <- buildUp(colony, nWorkers = 100))
apiary <- createMultiColony(basePop[3:8], n = 6)
apiary <- cross(apiary, drones = droneGroups[2:7])
apiary <- buildUp(apiary, nWorkers = 100)

# Supersede a colony
isQueenPresent(colony)
isVirginQueenPresent(colony)
colony <- supersede(colony)
isQueenPresent(colony)
isVirginQueenPresent(colony)

# Supersede all colonies in the apiary
isQueenPresent(colony)
isVirginQueenPresent(colony)
apiary1 <- supersede(apiary)
isQueenPresent(colony)
isVirginQueenPresent(colony)

# Sample colonies from the apiary that will supersede (sample with probability of 0.2)
tmp <- pullColonies(apiary, p = 0.2)
# Swarm only the pulled colonies
(supersede(tmp$pulled))
```

---

swarm *Swarm*

Description

Level 2 function that swarms a Colony or MultiColony object - an event where the queen leaves with a proportion of workers to create a new colony (the swarm). The remnant colony retains the other proportion of workers and all drones, and the workers raise virgin queens, of which only one prevails. Location of the swarm is the same as for the remnant or sampled as deviation from the remnant.

Usage

```
swarm(
  x,
  p = NULL,
  year = NULL,
  nVirginQueens = NULL,
  sampleLocation = TRUE,
  radius = NULL,
  simParamBee = NULL,
  ...
)
```

Arguments

`x`	`Colony-class` or `MultiColony-class`
`p`	numeric, proportion of workers that will leave with the swarm colony; if `NULL` then `SimParamBee$swarmP` is used. If input is `MultiColony-class`, the input could also be a vector of the same length as the number of colonies. If a single value is provided, the same value will be applied to all the colonies
`year`	numeric, year of birth for virgin queens
`nVirginQueens`	integer, the number of virgin queens to be created in the colony; of these one is randomly selected as the new virgin queen of the remnant colony. If `NULL`, the value from `simParamBee$nVirginQueens` is used
`sampleLocation`	logical, sample location of the swarm by taking the current colony location and adding deviates to each coordinate using `rcircle`
`radius`	numeric, radius of a circle within which swarm will go; if `NULL` then `SimParamBee$swarmRadius` is used (which uses `0`, so by default swarm does not fly far away)
`simParamBee`	`SimParamBee`, global simulation parameters
`...`	additional arguments passed to `p` or `nVirginQueens` when these arguments are functions

Value

list with two `Colony-class` or `MultiColony-class`, the `swarm` and the `remnant` (see the description what each colony holds!); both outputs have the swarm event set to `TRUE`

Examples

```
founderGenomes <- quickHaplo(nInd = 8, nChr = 1, segSites = 50)
SP <- SimParamBee$new(founderGenomes)

basePop <- createVirginQueens(founderGenomes)
drones <- createDrones(basePop[1], n = 1000)
droneGroups <- pullDroneGroupsFromDCA(drones, n = 10, nDrones = 10)

# Create Colony and MultiColony class
colony <- createColony(x = basePop[2])
colony <- cross(colony, drones = droneGroups[[1]])
(colony <- buildUp(colony, nWorkers = 100))
apiary <- createMultiColony(basePop[3:8], n = 6)
apiary <- cross(apiary, drones = droneGroups[2:7])
apiary <- buildUp(apiary, nWorkers = 100)

# Swarm a colony
tmp <- swarm(colony)
tmp$swarm
tmp$remnant

# Swarm all colonies in the apiary with p = 0.6 (60% of workers leave)
tmp <- swarm(apiary, p = 0.6)
nWorkers(tmp$swarm)
nWorkers(tmp$remnant)
# Swarm with different proportions
nWorkers(apiary)
tmp <- swarm(apiary, p = c(0.4, 0.6, 0.5, 0.5, 0.34, 0.56))
nWorkers(tmp$swarm)
nWorkers(tmp$remnant)

# Sample colonies from the apiary that will swarm (sample with probability of 0.2)
tmp <- pullColonies(apiary, p = 0.2)
# Swarm only the pulled colonies
(swarm(tmp$pulled, p = 0.6))
```

`swarmPUnif` *Sample the swarm proportion - the proportion of workers that swarm*

Description

Sample the swarm proportion - the proportion of workers that swarm - used when `p = NULL` (see `SimParamBee$swarmP`).

This is just an example. You can provide your own functions that satisfy your needs!

Usage

```
swarmPUnif(colony, n = 1, min = 0.4, max = 0.6)
```

Arguments

`colony`	`Colony-class`
`n`	integer, number of samples
`min`	numeric, lower limit for `swarmPUnif`
`max`	numeric, upper limit for `swarmPUnif`

Details

`swarmPUnif` samples from a uniform distribution between values 0.4 and 0.6 irrespective of colony strength.

The `nWorkersFull` default value used in this function is geared towards a situation where we simulate ~100 workers per colony (down-scaled simulation for efficiency). If you simulate more workers, you should change the default accordingly.

Value

numeric, swarm proportion

See Also

`SimParamBee` field `swarmP`

Examples

```
swarmPUnif()
swarmPUnif()
p <- swarmPUnif(n = 1000)
hist(p, breaks = seq(from = 0, to = 1, by = 0.01), xlim = c(0, 1))
```

Index