Package ‘popkin’

January 7, 2023

Title Estimate Kinship and FST under Arbitrary Population Structure

Version 1.3.23

Description Provides functions to estimate the kinship matrix of individuals from a large set of biallelic SNPs, and extract inbreeding coefficients and the generalized FST (Wright's fixation index). Method described in Ochoa and Storey (2021) <doi:10.1371/journal.pgen.1009241>.

LazyData true

Depends R (>= 3.5.0)

Imports RColorBrewer, graphics, grDevices, ape

Suggests BEDMatrix, testthat, knitr, rmarkdown, bnpsd

License GPL-3

Encoding UTF-8

RoxygenNote 7.2.3

VignetteBuilder knitr

URL `https://github.com/StoreyLab/popkin/`

BugReports `https://github.com/StoreyLab/popkin/issues`

NeedsCompilation no

Author Alejandro Ochoa [aut, cre] (<`https://orcid.org/0000-0003-4928-3403`>),
John D. Storey [aut] (<`https://orcid.org/0000-0001-5992-402X`>)

Maintainer Alejandro Ochoa <`alejandro.ochoa@duke.edu`>

Repository CRAN

Date/Publication 2023-01-07 00:30:02 UTC

R topics documented:

`popkin-package` *A package for estimating kinship and FST under arbitrary population structure*

Description

The heart of this package is the `popkin()` function, which estimates the kinship matrix of all individual pairs from their genotype matrix. Inbreeding coefficients, the generalized `FST`, and the individual-level pairwise `FST` matrix are extracted from the kinship matrix using `inbr()`, `fst()`, and `pwfst()`, respectively. `fst()` accepts weights for individuals to balance subpopulations obtained with `weights_subpops()`. Kinship matrices can be renormalized (to change the most recent common ancestor population or MRCA) using `rescale_popkin()`. Coancestry matrices can be estimated from allele frequency matrices using `popkin_af()`. Lastly, kinship, coancestry, and pairwise FST matrices can be visualized using `plot_popkin()` (with the help of `inbr_diag()` for kinship matrices only).

Author(s)

Maintainer: Alejandro Ochoa <`alejandro.ochoa@duke.edu`> (ORCID)

Authors:

- John D. Storey <`jstorey@princeton.edu`> (ORCID)

See Also

Useful links:

- `https://github.com/StoreyLab/popkin/`
- Report bugs at `https://github.com/StoreyLab/popkin/issues`

Examples

```
# estimate and visualize kinship and FST from a genotype matrix

# Construct toy data
X <- matrix(c(0,1,2,1,0,1,1,0,2), nrow = 3, byrow = TRUE) # genotype matrix
subpops <- c(1,1,2) # subpopulation assignments for individuals
subpops2 <- 1:3 # alternate labels treat every individual as a different subpop

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
## library(BEDMatrix)
## X <- BEDMatrix(file) # load genotype matrix object

# estimate the kinship matrix from the genotypes "X"!
# all downstream analysis require "kinship", none use "X" after this
kinship <- popkin(X, subpops) # calculate kinship from X and optional subpop labels

# plot the kinship matrix, marking the subpopulations
# note inbr_diag replaces the diagonal of kinship with inbreeding coefficients
plot_popkin( inbr_diag(kinship), labs = subpops )

# extract inbreeding coefficients from kinship
inbreeding <- inbr(kinship)

# estimate FST
weights <- weights_subpops(subpops) # weigh individuals so subpopulations are balanced
Fst <- fst(kinship, weights) # use kinship matrix and weights to calculate fst
Fst <- fst(inbreeding, weights) # estimate more directly from inbreeding vector (same result)

# estimate and visualize the pairwise FST matrix
pairwise_fst <- pwfst(kinship) # estimated matrix
leg_title <- expression(paste('Pairwise ', F[ST])) # fancy legend label
# NOTE no need for inbr_diag() here!
plot_popkin(pairwise_fst, labs = subpops, leg_title = leg_title)

# rescale the kinship matrix using different subpopulations (implicitly changes the MRCA)
kinship2 <- rescale_popkin(kinship, subpops2)

# toy allele frequency data
P <- matrix( runif( 9 ), nrow = 3 )

# estimate coancestry from allele frequency matrix
# (useful from P matrices from admixture models)
coancestry <- popkin_af( P )
```

---

admix_label_cols *Label ancestries based on best match to individual labels*

Description

Returns labels for each ancestry (columns) of an admixture matrix which is the best matching label among the average individual (rows) of each subpopulation. More specifically, each ancestry is associated to the subpopulation label in which its admixture proportion was the highest averaging over all individuals from that subpopulation. If there are two or more ancestries that match to the same label, these are made unique by appending its order of appearance (if the label is "A", then the first column that matches to it is labeled "A1", the next one "A2", etc).

Usage

```
admix_label_cols(Q, labs)
```

Arguments

`Q`	The admixture proportions matrix.
`labs`	Subpopulation labels for individuals (rows of `Q`).

Value

The best label assignments for the ancestries (columns of `Q`), made unique by indexes if there are overlaps.

See Also

`admix_order_cols()` to automatically order ancestries given ordered individuals.

`plot_admix()` for plotting admixture matrices.

Examples

```
# toy admixture matrix with labels for individuals/rows that match well with ancestry/columns
Q <- matrix(
    c(
        0.1, 0.8, 0.1,
        0.1, 0.7, 0.2,
        0.0, 0.4, 0.6,
        0.0, 0.3, 0.7,
        0.9, 0.0, 0.1
    ),
    nrow = 5,
    ncol = 3,
    byrow = TRUE
)
labs <- c('X', 'X', 'Y', 'Y', 'Z')

# to calculate matches and save as column names, do this:
colnames( Q ) <- admix_label_cols( Q, labs )

# expected column names: c('Z', 'X', 'Y')
```

`admix_order_cols` *Reorder admixture matrix columns*

Description

Returns the order of the columns (ancestries) of an admixture matrix so that they are in their average order of appearance in rows (individuals). More specifically, for each ancestry it calculates its mean row (expected row number weighted by this ancestry's proportion distribution among rows), and returns the order in which these mean row values are increasing. In datasets where the rows/individuals are already ordered in a meaningful way (for example, by distance from the species' geographical origin, and generally grouping the most similar individuals together), this function can lead to a more pleasing automated visualization of the admixture proportions.

Usage

```
admix_order_cols(Q)
```

Arguments

`Q`	The admixture proportions matrix.

Value

The desired order of the columns (a vector of indexes).

See Also

`admix_label_cols()` to automatically assign labels to ancestries given labels to individuals.

`plot_admix()` for plotting admixture matrices.

Examples

```
# here is a toy admixture proportions matrix with columns in no meaningful order
Q <- matrix(
    c(
        0.1, 0.8, 0.1,
        0.1, 0.7, 0.2,
        0.0, 0.4, 0.6,
        0.0, 0.3, 0.7,
        0.9, 0.0, 0.1
    ),
    nrow = 5,
    ncol = 3,
    byrow = TRUE
)
# get nicer order
indexes <- admix_order_cols( Q )
# apply reordering to columns
Q <- Q[ , indexes ]
```

```
# notice that now the first columns takes on the highest values initially,
# followed by the second column, and lastly the third column.
```

---

avg_kinship_subpops — *Calculate a kinship matrix between subpopulations by averaging individual data*

Description

This function calculates a kinship matrix between subpopulations, whose values are the average kinship values between all individual pairs where one individual is in the first subpopulation and the other individual is in the second subpopulation. To estimate coancestry instead of kinship, which is recommended to get more interpretable diagonal values, the input kinship matrix should be transformed using `inbr_diag()`.

Usage

```
avg_kinship_subpops(kinship, subpops, subpop_order = unique(subpops))
```

Arguments

`kinship`	A symmetric n-by-n kinship matrix.
`subpops`	The length-n vector of subpopulation assignments for each individual.
`subpop_order`	The optional order of subpopulations in the output matrix. `subpop_order` must contain every unique subpopulation in `subpops`. Any additional subpopulations in `subpop_order` (missing in `subpops`) are ignored. By default, subpopulations are in the order of first appearance in `subpops`.

Value

The symmetric K-by-K kinship matrix between subpopulations, where K is the number of unique subpopulations in `subpops`, ordered as in `subpop_order`.

Examples

```
# a toy kinship matrix with 5 individuals belonging to 2 subpopulations
kinship <- matrix(
    c(
        0.7, 0.4, 0.4, 0.1, 0.0,
        0.4, 0.7, 0.4, 0.2, 0.1,
        0.4, 0.4, 0.7, 0.2, 0.0,
        0.1, 0.2, 0.2, 0.6, 0.1,
        0.0, 0.1, 0.0, 0.1, 0.6
    ),
    nrow = 5,
    ncol = 5
)
subpops <- c(1, 1, 1, 2, 2)

# calculate mean kinship between (and within) subpopulations
# a 2x2 matrix
avg_kinship_subpops( kinship, subpops )

# calculate coancestry estimate instead (difference is diagonal)
avg_kinship_subpops( inbr_diag( kinship ), subpops )
```

---

fst *Calculate FST from a population-level kinship matrix or vector of inbreeding coefficients*

Description

This function simply returns the weighted mean inbreeding coefficient. If weights are `NULL` (default), the regular mean is calculated. If a kinship matrix is provided, then the inbreeding coefficients are extracted from its diagonal using `inbr()` (requires the diagonal to contains self-kinship values as `popkin()` returns, and not inbreeding coefficients as `inbr_diag()` returns). If there is local inbreeding and it can be estimated (from known pedigrees, for example), it can be subtracted from the total inbreeding coefficients, resulting in a vector of structural inbreeding that correctly averages into FST.

Usage

```
fst(x, weights = NULL, x_local = NULL)
```

Arguments

`x`	The vector of inbreeding coefficients, or the kinship matrix if `x` is a matrix.
`weights`	Weights for individuals (optional, defaults to uniform weights)
`x_local`	An optional vector of inbreeding coefficients, or a local kinship matrix if `x_local` is a matrix.

Details

The returned weighted mean inbreeding coefficient equals the generalized FST if all individuals are "locally outbred" (i.e. if the self-relatedness of every individual stems entirely from the population structure rather than due partly to having unusually closely related parents, such as first or second cousins). Note most individuals in population-scale human data are locally outbred. If there are locally-inbred individuals, but their local inbreeding cannot be estimated, then the returned value will overestimate FST. Good estimates of local inbreeding can be passed (parameter `x_local`), in which case the code will subtract their effect and FST will be more accurate.

Value

FST

Examples

```
# Get FST from a genotype matrix

# Construct toy data
X <- matrix(c(0,1,2,1,0,1,1,0,2), nrow = 3, byrow = TRUE) # genotype matrix
subpops <- c(1,1,2) # subpopulation assignments for individuals

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
## library(BEDMatrix)
## X <- BEDMatrix(file) # load genotype matrix object

# estimate the kinship matrix "kinship" from the genotypes "X"!
kinship <- popkin(X, subpops) # calculate kinship from X and optional subpop labels
weights <- weights_subpops(subpops) # can weigh individuals so subpopulations are balanced
Fst <- fst(kinship, weights) # use kinship matrix and weights to calculate fst

Fst <- fst(kinship) # no (or NULL) weights implies uniform weights

inbr <- inbr(kinship) # if you extracted inbr for some other analysis...
Fst <- fst(inbr, weights) # ...use this inbreeding vector as input too!
```

`hgdp_subset` *HGDP subset*

Description

Subset of the HGDP dataset.

Usage

```
hgdp_subset
```

Format

a matrix of 0's, 1's and 2's.

Value

genotype matrix

Source

Stanford HGDP http://www.hagsc.org/hgdp/files.html

inbr — *Extract inbreeding coefficients from a kinship matrix*

Description

The kinship matrix contains transformed inbreeding coefficients along the diagonal. This function extracts the vector of inbreeding values from the input kinship matrix, by transforming the diagonal using the formula `2 * x - 1`.

Usage

```
inbr(kinship)
```

Arguments

`kinship`	The n-by-n kinship matrix.

Value

The length-n vector of inbreeding coefficient for each individual.

See Also

`inbr_diag()` to replace kinship diagonal with inbreeding values (better for plots)

Examples

```
#########
# illustrate the main transformation on a 2x2 kinship matrix:
# same inbreeding values for both individuals
inbr <- 0.2
# corresponding self kinship (diagonal values) for both individuals
kinship_self <- (1 + inbr)/2
# actual kinship matrix (zero kinship between individuals)
kinship <- matrix(c(kinship_self, 0, 0, kinship_self), nrow=2)
# expected output of inbr (extracts inbreeding coefficients)
inbr_exp <- c(inbr, inbr)
# actual output from this function
inbr_obs <- inbr(kinship)
# verify that they match (up to machine precision)
stopifnot( all( abs(inbr_obs - inbr_exp) < .Machine$double.eps ) )

#########
# Construct toy data
X <- matrix(c(0,1,2,1,0,1,1,0,2), nrow=3, byrow=TRUE) # genotype matrix
subpops <- c(1,1,2) # subpopulation assignments for individuals

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
```

```
## library(BEDMatrix)
## X <- BEDMatrix(file) # load genotype matrix object

# estimate the kinship matrix from the genotypes "X"!
kinship <- popkin(X, subpops) # calculate kinship from X and optional subpop labels

# extract inbreeding coefficients from Kinship
inbr <- inbr(kinship)
```

`inbr_diag` *Replace kinship diagonal with inbreeding coefficients*

Description

The usual kinship matrix contains self-kinship values along their diagonal given by `diag(kinship) = ( 1 + inbr ) / 2`, where `inbr` is the vector of inbreeding coefficient. This function returns a modified kinship matrix with diagonal values replaced with `inbr` (off-diagonal values stay the same). The resulting matrix is better for visualization, but is often not appropriate for modeling (e.g. in mixed-effects models for association or heritability estimation).

Usage

```
inbr_diag(kinship)
```

Arguments

`kinship`	A kinship matrix with self-kinship values along the diagonal. Can pass multiple kinship matrices contained in a list. If `NULL`, it is returned as-is.

Value

The modified kinship matrix, with inbreeding coefficients along the diagonal, preserving column and row names. If the input was a list of kinship matrices, the output is the corresponding list of transformed matrices. `NULL` inputs are preserved without causing errors.

See Also

The inverse function is given by `bnpsd::coanc_to_kinship()`.

`inbr()` to extract the vector of inbreeding values from a kinship matrix.

Examples

```
#########
# illustrate the main transformation on a 2x2 kinship matrix:
# same inbreeding values for both individuals
inbr <- 0.2
# corresponding self kinship (diagonal values) for both individuals
kinship_self <- (1 + inbr)/2
```

```
# kinship between the two individuals
kinship_between <- 0.1
# actual kinship matrix
kinship <- matrix(c(kinship_self, kinship_between, kinship_between, kinship_self), nrow=2)
# expected output of inbr_diag (replaces self kinship with inbreeding)
kinship_inbr_diag_exp <- matrix(c(inbr, kinship_between, kinship_between, inbr), nrow=2)
# actual output from this function
kinship_inbr_diag_obs <- inbr_diag(kinship)
# verify that they match (up to machine precision)
stopifnot( all( abs(kinship_inbr_diag_obs - kinship_inbr_diag_exp) < .Machine$double.eps ) )

# for a list of matrices, returns list of transformed matrices:
inbr_diag( list(kinship, kinship) )

# a list with NULL values also works
inbr_diag( list(kinship, NULL, kinship) )

#########
# Construct toy data (to more closely resemble real data analysis)
X <- matrix(c(0,1,2,1,0,1,1,0,2), nrow=3, byrow=TRUE) # genotype matrix
subpops <- c(1,1,2) # subpopulation assignments for individuals

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
## library(BEDMatrix)
## X <- BEDMatrix(file) # load genotype matrix object

# estimate the kinship matrix from the genotypes "X"!
kinship <- popkin(X, subpops) # calculate kinship from X and optional subpop labels

# lastly, replace diagonal of kinship matrix with inbreeding coefficients
kinship_inbr_diag <- inbr_diag(kinship)
```

`mean_kinship` *Calculate the weighted mean kinship*

Description

This function computes a particular weighted mean kinship that arises in the context of kinship and FST estimators and in the definition of the effective sample size. This function allows for weights to be zero or even negative, but they are internally normalized to sum to one.

Usage

```
mean_kinship(kinship, weights = NULL)
```

Arguments

`kinship`	The kinship matrix
`weights`	Weights for individuals (optional). If `NULL` (default), uniform weights are used.

Value

The weighted mean kinship matrix, equivalent to `drop( weights %*% kinship %*% weights )` after normalizing weights to sum to one.

Examples

```
# construct a dummy kinship matrix
kinship <- matrix(c(0.5, 0, 0, 0.6), nrow=2)

# this is the ordinary mean
mean_kinship(kinship)

# weighted mean with twice as much weight on the second individual
# (weights are internally normalized to sum to one)
weights <- c(1, 2)
mean_kinship(kinship, weights)
```

`n_eff` *Calculates the effective sample size of the data*

Description

The effective sample size `n_eff` is the equivalent number of independent haplotypes that gives the same variance as that observed under the given population. The variance in question is for the weighted sample mean ancestral allele frequency estimator. It follows that `n_eff` equals the inverse of the weighted mean kinship. If `max = TRUE`, a calculation is performed that implicitly uses optimal weights which maximize `n_eff`, which equals the sum of the elements of the inverse kinship matrix. However, if `nonneg = TRUE` and if the above solution has negative weights (common), optimal non-negative weights are found instead (there are three algorithms available, see `algo`). If `max = FALSE`, then the input weights are used in this calculation, and if weights are `NULL`, uniform weights are used.

Usage

```
n_eff(
  kinship,
  max = TRUE,
  weights = NULL,
  nonneg = TRUE,
  algo = c("gradient", "newton", "heuristic"),
  tol = 1e-10
)
```

Arguments

`kinship`	An `n`-by-`n` kinship matrix.
`max`	If `TRUE`, returns the maximum `n_eff` value among those computed using all possible vectors of weights that sum to one (and which are additionally non-negative if `nonneg = TRUE`). If `FALSE`, `n_eff` is computed using the specific weight vector provided.
`weights`	Weights for individuals (optional). If `NULL`, uniform weights are used. This parameter is ignored if `max = TRUE`.
`nonneg`	If `TRUE` (default) and `max = TRUE`, non-negative weights that maximize `n_eff` are found. See `algo`. This has no effect if `max = FALSE`.
`algo`	Algorithm for finding optimal non-negative weights (applicable only if `nonneg = TRUE` and `max = TRUE` and the weights found by matrix inversion are non-negative). May be abbreviated. If "gradient" (default), an optimized gradient descent algorithm is used (fastest; recommended). If "newton", the exact multivariate newton's Method is used (slowest since `(n+1)`-by-`(n+1)` Hessian matrix needs to be inverted at every iteration; use if possible to confirm that "gradient" gives the best answer). If "heuristic", if the optimal solution by the inverse matrix method contains negative weights, the most negative weight in an iteration is forced to be zero in all subsequent iterations and the rest of the weights are solved for using the inverse matrix method, repeating until all resulting weights are non-negative (also slow, since inversion of large matrices is required; least likely to find optimal solution).
`tol`	Tolerance parameter for "gradient" and "newton" algorithms. The algorithms converge when the norm of the step vector is smaller than this tolerance value.

Details

The maximum `n_eff` possible is `2 * n`, where `n` is the number of individuals; this value is attained only when all haplotypes are independent (a completely unstructured population in Hardy-Weinberg equilibrium). The minimum `n_eff` possible is 1, which is attained in an extremely structured population with FST of 1, where every individual has exactly the same haplotype at every locus (no heterozygotes). Moreover, for `K` extremely-differentiated subpopulations (FST = 1 per subpopulation) `n_eff = K`. In this way, `n_eff` is smaller than the ideal value of `2 * n` depending on the amount of kinship (covariance) in the data.

Occasionally, depending on the quality of the input kinship matrix, the estimated `n_eff` may be outside the theoretical `[1, 2*n]` range, in which case the return value is set to the closest boundary value. The quality of the results depends on the success of matrix inversion (which for numerical reasons may incorrectly contain negative eigenvalues, for example) or of the gradient optimization.

Value

A list containing `n_eff` and `weights` (optimal weights if `max = TRUE`, input weights otherwise).

Examples

```
# Get n_eff from a genotype matrix
# Construct toy data
X <- matrix(c(0,1,2,1,0,1,1,0,2), nrow=3, byrow=TRUE) # genotype matrix
subpops <- c(1,1,2) # subpopulation assignments for individuals

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
## library(BEDMatrix)
## X <- BEDMatrix(file) # load genotype matrix object

# estimate the kinship matrix "kinship" from the genotypes "X"!
kinship <- popkin(X, subpops) # calculate kinship from X and optional subpop labels
weights <- weights_subpops(subpops) # can weigh individuals so subpopulations are balanced

# use kinship matrix to calculate n_eff
# default mode returns maximum n_eff possible across all non-negative weights that sum to one
# also returns the weights that were optimal
obj <- n_eff(kinship)
n_eff_max <- obj$n_eff
w_max <- obj$weights

# version that uses weights provided
obj <- n_eff(kinship, max = FALSE, weights = weights)
n_eff_w <- obj$n_eff
w <- obj$weights # returns input weights renormalized for good measure

# no (or NULL) weights implies uniform weights
obj <- n_eff(kinship, max = FALSE)
n_eff_u <- obj$n_eff
w <- obj$weights # uniform weights
```

plot_admix *Make a structure/admixture plot*

Description

This function facilitates structure plots with options that resemble those of plot_popkin() in name and results. The biggest difference is this function plots single panels (technically 2 panels including the legend, unless it is omitted), whereas plot_popkin() can plot multiple kinship matrices with a shared legend.

Usage

```
plot_admix(
  Q,
  col = RColorBrewer::brewer.pal(max(ncol(Q), 3), "Paired"),
  mar_pad = 0.2,
  panel_letters = NA,
  panel_letters_cex = 1.5,
  panel_letters_adj = -0.1,
  axis_lab_cex = 1,
  xlab = "Individuals",
  xlab_line = 1,
  xlab_cex = axis_lab_cex,
  ylab = "Ancestry",
  ylab_line = 2,
  ylab_side = 2,
  ylab_cex = axis_lab_cex,
  leg_title = "Ancestries",
  leg_title_cex = axis_lab_cex,
  leg_title_line = 2,
  leg_cex = 1,
  leg_mar = leg_title_line + 1,
  leg_width = 0.2,
  leg_las = 0,
  leg_omit = FALSE,
  layout_add = !leg_omit,
  names = FALSE,
  names_cex = 1,
  names_line = NA,
  names_las = 2,
  labs = NULL,
  labs_cex = 1,
  labs_las = 0,
  labs_line = 0,
  labs_sep = TRUE,
  labs_lwd = 1,
  labs_col = "black",
  labs_ticks = FALSE,
  labs_text = TRUE,
  labs_even = FALSE,
  ...
)
```

Arguments

`Q`	The admixture proportions matrix, with `n` individuals along rows and `K` ancestries along columns. Rows should sum to 1, but this is not enforced. There must be at least 2 ancestries. The ancestry labels used by the legend must be the column names, which are unlabeled if the column names are missing.
`col`	A vector of at least `K` colors for the ancestries (extra colors are ignored). By default uses the "Paired" palette of `RColorBrewer`, which has at most 12 colors, so please provide colors if `K > 12`. Since the minimum number of colors for "Paired" is 3, when `K = 2` we ask for 3 colors, then remove the middle color internally.
`mar_pad`	Margin padding used for legend panel only (margins for first/main panel are not altered by this function).

panel_letters	Panel letter to include in first/main panel (default NA is no letter). Despite name (matches plot_popkin()), must be scalar.
panel_letters_cex	Scaling factor of panel letter (default 1.5).
panel_letters_adj	X-axis adjustment for panel letter (default -0.1). Negative values place the letter into the left margin area. Might need adjustment depending on the size of the left margin. AXIS LABEL OPTIONS
axis_lab_cex	Scaling factor for x-axis, y-axis, and legend title labels (which can also be set individually, see below).
xlab	X-axis label (default "Individuals"). Set to NA to omit.
xlab_line	The value of line for xlab passed to graphics::mtext().
xlab_cex	Scaling factor for x-axis label.
ylab	Y-axis label (default "Ancestry"). Set to NA to omit.
ylab_line	The value of line for ylab passed to graphics::mtext().
ylab_side	The value of side for ylab passed to graphics::mtext() (2 is y-axis, 1 is x-axis, can also place on top (3) or right (4)).
ylab_cex	Scaling factor for y-axis label. LEGEND (COLOR KEY) OPTIONS
leg_title	The name of the categorical ancestry variable (default "Ancestries").
leg_title_cex	Scaling factor for legend title label.
leg_title_line	The value of line for leg_title passed to graphics::mtext().
leg_cex	Scaling factor for ancestry labels.
leg_mar	Margin values for the kinship legend panel only. A length-4 vector (in c(bottom, left, top, right) format that graphics::par() 'mar' expects) specifies the full margins, to which mar_pad is added. Otherwise, the margins used in the last panel are preserved with the exception that the left margin is set to mar_pad, and if leg_mar is length-1 (default), it is added to mar_pad to specify the right margin. By default the right margin is large enough to accommodate leg_title for the given value of leg_title_line.
leg_width	The width of the legend panel, relative to the width of the main panel. This value is passed to graphics::layout() (ignored if layout_add = FALSE).
leg_las	The ancestry label orientations (in format that graphics::mtext() expects).
leg_omit	If TRUE, no legend (second panel) is produced (default FALSE is to include legend).
layout_add	If TRUE (default) then graphics::layout() is called internally to create two panels: the main panel and the color key legend. The original layout is reset when plotting is complete and if layout_add = TRUE. If a non-standard layout or additional panels (beyond those provided by this function) are desired, set to FALSE and call graphics::layout() yourself beforehand. INDIVIDUAL LABEL OPTIONS

`names`	If `TRUE`, the row (individual) names are plotted in the structure barplot.
`names_cex`	Scaling factor for the individual names.
`names_line`	Line where individual names are placed.
`names_las`	Orientation of labels relative to axis. Default (2) makes labels perpendicular to axis.
	SUBPOPULATION LABEL OPTIONS
`labs`	Subpopulation labels for individuals in the admixture matrix. Use a matrix of labels to show groupings at more than one level (for a hierarchy or otherwise).
`labs_cex`	A vector of label scaling factors for each level of labs.
`labs_las`	A vector of label orientations (in format that `graphics::mtext()` expects) for each level of labs.
`labs_line`	A vector of lines where labels are placed (in format that `graphics::mtext()` expects) for each level of labs.
`labs_sep`	A vector of logicals that specify whether lines separating the subpopulations are drawn for each level of labs.
`labs_lwd`	A vector of line widths for the lines that divide subpopulations (if `labs_sep = TRUE`) for each level of labs.
`labs_col`	A vector of colors for the lines that divide subpopulations (if `labs_sep = TRUE`) for each level of labs.
`labs_ticks`	A vector of logicals that specify whether ticks separating the subpopulations are drawn for each level of labs.
`labs_text`	A vector of logicals that specify whether the subpopulation labels are shown for each level of labs. Useful for including separating lines or ticks without text.
`labs_even`	A vector of logicals that specify whether the subpopulations labels are drawn with equal spacing for each level of labs. When `TRUE`, lines mapping the equally-spaced labels to the unequally-spaced subsections of the heatmap are also drawn.
`...`	Additional options passed to `graphics::barplot()`.

See Also

`admix_order_cols()` to automatically order ancestries given ordered individuals.

`admix_label_cols()` to automatically assign labels to ancestries given labels to individuals.

Examples

```
# create random proportions for two ancestries
Q <- runif( 10 )
Q <- cbind( Q, 1 - Q )
# add ancestry names
colnames( Q ) <- c('A1', 'A2')

# plot this data!
plot_admix( Q )

# See vignette for more elaborate examples!
```

plot_phylo *Plot a* phylo *tree object*

Description

This is a wrapper around `ape::plot.phylo()` that makes several adjustments so plots agree more with accompanying kinship matrices. In particular, tree is reversed on the y-axis to match matrix orientation, y-axis spacing is more padded for small trees, and an x-axis scale is always added.

Usage

```
plot_phylo(
  tree,
  xlab = "Coancestry",
  xmax = NULL,
  leg_n = 5,
  edge_width = 1,
  ...
)
```

Arguments

`tree`	A phylo object to plot.
`xlab`	The x-axis label (default "Coancestry").
`xmax`	X-axis maximum limit.
`leg_n`	The desired number of ticks in the x-axis (input to `pretty()`, see that for more details).
`edge_width`	The width of the tree edges (passed to `ape::plot.phylo()` as `edge.width`).
`...`	Additional parameters passed to `ape::plot.phylo()`. However, these parameters cannot be passed: `x.lim` (controlled via `xmax`), `y.lim` (a better default for small trees is passed and cannot be changed) and `font` (takes the value of `par('font')` instead of ape's default of 3 (italic)).

See Also

`plot_popkin()` can create multipanel figures including kinship matrices and trees (calling the present function in the process).

Examples

```
# create a small random tree
library(ape)
tree <- rtree( 3 )

# plot it!
plot_phylo( tree )
```

`plot_popkin` *Visualize one or more kinship matrices and other related objects*

Description

This function plots one or more kinship matrices, trees (class `phylo` objects, see `ape` package), and arbitrary functions, and a shared legend for the kinship color key. Many options allow for fine control of individual or subpopulation labeling.

Usage

```
plot_popkin(
  kinship,
  titles = NULL,
  col = NULL,
  col_n = 100,
  mar = NULL,
  mar_pad = 0.2,
  oma = 1.5,
  diag_line = FALSE,
  panel_letters = toupper(letters),
  panel_letters_cex = 1.5,
  panel_letters_adj = -0.1,
  ylab = "Individuals",
  ylab_adj = NA,
  ylab_line = 0,
  ylab_side = 2,
  ylab_per_panel = FALSE,
  layout_add = TRUE,
  layout_rows = 1,
  leg_per_panel = FALSE,
  leg_title = "Kinship",
  leg_cex = 1,
  leg_n = 5,
  leg_mar = 3,
  leg_width = 0.3,
  leg_column = NA,
  names = FALSE,
  names_cex = 1,
  names_line = NA,
  names_las = 2,
  labs = NULL,
  labs_cex = 1,
  labs_las = 0,
  labs_line = 0,
  labs_sep = TRUE,
  labs_lwd = 1,
  labs_col = "black",
  labs_ticks = FALSE,
  labs_text = TRUE,
  labs_even = FALSE,
  null_panel_data = FALSE,
  weights = NULL,
  raster = is.null(weights),
  sym = FALSE,
  ...
)
```

Arguments

`kinship`	A numeric kinship matrix, a `phylo` or `function` object, or a list of any such objects (at least one kinship matrix is expected). This list may contain `NULL` elements (makes blank panels with titles; good for placeholders or non-existent data) `phylo` objects are plotted with `plot_phylo()`, which is a wrapper around `ape::plot.phylo()` that makes some adjustments so resulting plots are more consistent with accompanying kinship matrices. `function` elements are executed without arguments, and are expected to produce single arbitrary plot panels.
`titles`	Titles to add to each matrix panel (default is no titles). Applied to kinship and phylo panels only.
`col`	Colors for kinship heatmap (default is a red-white-blue palette symmetric about zero constructed using `RColorBrewer`).
`col_n`	The number of colors to use in the heatmap (applies if `col = NULL`).
`mar`	Margins shared by all panels (if a vector) or for each panel (if a list of such vectors). If the vector has length 1, `mar` corresponds to the shared lower and left margins, while the top and right margins are set to zero. If this length is 2, `mar[1]` is the same as above, while `mar[2]` is the top margin. If this length is 4, then `mar` is a fully-specified margin vector in the standard format `c(bottom, left, top, right)` that `graphics::par()` 'mar' expects. Vectors of invalid lengths produce a warning. Note the padding `mar_pad` below is added to every margin if set. If `NULL`, the original margin values are used without change, and are reset for every panel that has a `NULL` value. The original margins are also reset after plotting is complete. Applied to panels of all types (kinship, phylo, and function).
`mar_pad`	Margin padding added to all panels (`mar` above and `leg_mar` below). Default 0.2. Must be a scalar or a vector of length 4 to match `graphics::par()` 'mar'. Applied to panels of all types (kinship, phylo, and function).
`oma`	Outer margin vector. If length 1, the value of `oma` is applied to the left outer margin only (so `ylab` below displays correctly) and zero outer margins elsewhere. If length 4, all outer margins are expected in standard format `graphics::par()` 'mar' expects (see `mar` above). `mar_pad` above is never added to outer margins. If `NULL`, no outer margins are set (previous settings are preserved). Vectors of invalid lengths produce a warning. Note: if `layout_add = FALSE`, this function still (re)sets the outer margins if `oma` is not `NULL`, which can be convenient if

	`plot_popkin` generates the first few panels, but otherwise a partial multipanel figure will be reset unless `oma = NULL` is also set!
`diag_line`	If `TRUE` adds a line along the kinship diagonal (default no line). May also be a vector of logicals to set per panel (lengths must agree). Has no effect on non-kinship panels.
`panel_letters`	Vector of strings for labeling panels (default A-Z). No labels are added if `NULL`, or when there is only one panel except if its set to a single letter in that case (this behavior is useful if goal is to have multiple external panels but popkin only creates one of these panels). Applied to panels of all types (kinship, phylo, and function).
`panel_letters_cex`	Scaling factor of panel letters (default 1.5).
`panel_letters_adj`	X-axis adjustment for panel letters (default -0.1). Negative values place the letter into the left margin area. Might need adjustment depending on the size of the left margin.
`ylab`	The y-axis label (default "Individuals"). If `length(ylab) == 1`, the label is placed in the outer margin (shared across panels); otherwise `length(ylab)` must equal the number of panels and each label is placed in the inner margin of the respective panel. Applied to panels of all types (kinship, phylo, and function).
`ylab_adj`	The value of `adj` for `ylab` passed to `graphics::mtext()`. If `length(ylab) == 1`, only the first value is used, otherwise `length(ylab_adj)` must equal the number of panels.
`ylab_line`	The value of `line` for `ylab` passed to `graphics::mtext()`. If `length(ylab) == 1`, only the first value is used, otherwise `length(ylab_line)` must equal the number of panels.
`ylab_side`	The value of `side` for `ylab` passed to `graphics::mtext()` (2 is y-axis, 1 is x-axis, can also place on top (3) or right (4)). If `length(ylab) == 1`, only the first value is used, otherwise `length(ylab_side)` must equal the number of panels.
`ylab_per_panel`	Forces y-axis labels to appear for each panel, in the inner margins. Most useful to cover the case where there is a single panel but no outer margins (`oma = NULL`). LAYOUT OPTIONS
`layout_add`	If `TRUE` (default) then `graphics::layout()` is called internally with appropriate values for the required number of panels for each matrix, the desired number of rows (see `layout_rows` below) plus the color key legend. The original layout is reset when plotting is complete if `layout_add = TRUE`. If a non-standard layout or additional panels (beyond those provided by `plot_popkin`) are desired, set to `FALSE` and call `graphics::layout()` yourself beforehand; in this case you may want to set `oma = NULL` (above) as well!
`layout_rows`	Number of rows in layout, used only if `layout_add = TRUE`. LEGEND (COLOR KEY) OPTIONS
`leg_per_panel`	If `TRUE`, every kinship matrix get its own legend/color key (best for matrices with very different scales), and each phylo tree has its own x-axis range. If `FALSE` (default), a single legend/color key is shared by all kinship matrix panels, and also every tree has the same x-axis range (different from the kinship range).

`leg_title`	The name of the variable that the kinship heatmap colors measure (default "Kinship"), or a vector of such values if they vary per panel.
`leg_cex`	Scaling factor for `leg_title` (default 1), or a vector of such values if they vary per panel.
`leg_n`	The desired number of ticks in the kinship legend y-axis, and phylo x-axis (input to `pretty()`, see that for more details), or a vector of such values if they vary per panel.
`leg_mar`	Margin values for the kinship legend panel only, or a list of such values if they vary per panel. A length-4 vector (in `c( bottom, left, top, right )` format that `graphics::par()` 'mar' expects) specifies the full margins, to which `mar_pad` is added. Otherwise, the margins used in the last panel are preserved with the exception that the left margin is set to `mar_pad`, and if `leg_mar` is length-1, it is added to `mar_pad` to specify the right margin.
`leg_width`	The width of the legend panel, relative to the width of a single main panel. This value is passed to `graphics::layout()` (ignored if `layout_add = FALSE`).
`leg_column`	The column number in which to place the kinship legend (default `NA` is for last column). Ignored if `leg_per_panel = TRUE`.
	INDIVIDUAL LABEL OPTIONS
`names`	If `TRUE`, the column and row names are plotted in the kinship heatmap, or a vector of such values if they vary per panel. (`names` has no effect on `phylo` panels, whose tip labels are always plotted, or other panel types.)
`names_cex`	Scaling factor for the column and row names of a kinship matrix, or the tip labels of a `phylo` object, or a vector of such values if they vary per panel.
`names_line`	Line where kinship column and row names are placed, or a vector of such values if they vary per panel. Has no effect on non-kinship panels.
`names_las`	Orientation of labels relative to axis. Default (2) makes labels perpendicular to axis. Has no effect on non-kinship panels.
	SUBPOPULATION LABEL OPTIONS
`labs`	Subpopulation labels for individuals in kinship matrices. Use a matrix of labels (individuals along rows, levels along columns) to show groupings at more than one level (for a hierarchy or otherwise). If input is a vector or a matrix, the same subpopulation labels are shown for every kinship matrix; the input must be a list of such vectors or matrices if the labels vary per panel. Has no effect on non-kinship panels.
`labs_cex`	A vector of label scaling factors for each level of labs, or a list of such vectors if labels vary per panel.
`labs_las`	A vector of label orientations (in format that `graphics::mtext()` expects) for each level of labs, or a list of such vectors if labels vary per panel.
`labs_line`	A vector of lines where labels are placed (in format that `graphics::mtext()` expects) for each level of labs, or a list of such vectors if labels vary per panel.
`labs_sep`	A vector of logicals that specify whether lines separating the subpopulations are drawn for each level of labs, or a list of such vectors if labels vary per panel.
`labs_lwd`	A vector of line widths for the lines that divide subpopulations (if `labs_sep = TRUE`) for each level of labs, or a list of such vectors if labels vary per panel.

`labs_col`	A vector of colors for the lines that divide subpopulations (if `labs_sep = TRUE`) for each level of labs, or a list of such vectors if labels vary per panel.
`labs_ticks`	A vector of logicals that specify whether ticks separating the subpopulations are drawn for each level of labs, or a list of such vectors if labels vary per panel.
`labs_text`	A vector of logicals that specify whether the subpopulation labels are shown for each level of labs, or a list of such vectors if labels vary per panel. Useful for including separating lines or ticks without text.
`labs_even`	A vector of logicals that specify whether the subpopulations labels are drawn with equal spacing for each level of labs, or a list of such vectors if labels vary per panel. When `TRUE`, lines mapping the equally-spaced labels to the unequally-spaced subsections of the heatmap are also drawn.
`null_panel_data`	If `FALSE` (default), panels with `NULL` kinship matrices must not have titles or other parameters set, and no panel letters are used in these cases. If `TRUE`, panels with `NULL` kinship matrices must have titles and other parameters set. In the latter case, these `NULL` panels also get panel letters. The difference is important when checking that lengths of non-singleton parameters agree.
`weights`	A vector with weights for every individual, or a list of such vectors if they vary per panel. The width of every individual in the kinship matrix becomes proportional to their weight. Individuals with zero or negative weights are omitted. Has no effect on non-kinship panels.
`raster`	A logical equivalent to `useRaster` option in the `image` function used internally, or a vector of such logicals if the choice varies per panel. If `weights` are non-`NULL` in a given panel, `raster = FALSE` is forced (this is necessary to plot images where columns and rows have variable width). If `weights` are `NULL`, the default is `raster = TRUE`, but in this case the user may override (for example, so panels are visually coherent when some use weights while others do not, as there are small differences in rendering implementation for each value of `raster`). Note that a multipanel figure with a list of `weights` sets `raster = FALSE` to all panels by default, even if the weights were only applied to a subset of panels. Has no effect on non-kinship panels.
`sym`	If `FALSE` (default), plots non-symmetric (but square) kinship matrices without issues. If `TRUE`, stops if any input kinship matrices (excluding `phylo` or `function` objects) are not symmetric. AXIS LABEL OPTIONS
`...`	Additional options passed to `graphics::image()`. These are shared across kinship panels. Have no effect on non-kinship panels.

Details

`plot_popkin` plots the input kinship matrices as-is. For best results, a standard kinship matrix (such as the output of `popkin()`) should have its diagonal rescaled to contain inbreeding coefficients using `inbr_diag()` before `plot_popkin` is used.

This function permits the labeling of individuals (from row and column names when `names = TRUE`) and of subpopulations (passed through `labs`). The difference is that the labels passed through `labs` are assumed to be shared by many individuals, and lines (or other optional visual aids) are added to demarcate these subgroups.

Examples

```
# Construct toy data
X <- matrix(c(0,1,2,1,0,1,1,0,2), nrow = 3, byrow = TRUE) # genotype matrix
subpops <- c(1,1,2) # subpopulation assignments for individuals

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
## library(BEDMatrix)
## X <- BEDMatrix(file) # load genotype matrix object

# estimate the kinship matrix from the genotypes "X"!
kinship <- popkin(X, subpops) # calculate kinship from X and optional subpop labels

# simple plot of the kinship matrix, marking the subpopulations only
# note inbr_diag replaces the diagonal of kinship with inbreeding coefficients
# (see vignette for more elaborate examples)
plot_popkin( inbr_diag(kinship), labs = subpops )
```

---

popkin — *Estimate kinship from a genotype matrix and subpopulation assignments*

Description

Given the biallelic genotypes of `n` individuals, this function returns the `n`-by-`n` kinship matrix such that the kinship estimate between the most distant subpopulations is zero on average (this sets the ancestral population to the most recent common ancestor population).

Usage

```
popkin(
  X,
  subpops = NULL,
  n = NA,
  loci_on_cols = FALSE,
  mean_of_ratios = FALSE,
  mem_factor = 0.7,
  mem_lim = NA,
  want_M = FALSE,
  m_chunk_max = 1000
)
```

Arguments

`X`	Genotype matrix, `BEDMatrix` object, or a function `X(m)` that returns the genotypes of all individuals at `m` successive locus blocks each time it is called, and `NULL` when no loci are left. If a regular matrix, `X` must have values only in `c(0,`

	`1`, `2`, `NA`), encoded to count the number of reference alleles at the locus, or `NA` for missing data.
`subpops`	The length-`n` vector of subpopulation assignments for each individual. If `NULL`, every individual is effectively treated as a different population.
`n`	Number of individuals (required only when `X` is a function, ignored otherwise). If `n` is missing but `subpops` is not, `n` is taken to be the length of `subpops`.
`loci_on_cols`	If `TRUE`, `X` has loci on columns and individuals on rows; if `FALSE` (default), loci are on rows and individuals on columns. Has no effect if `X` is a function. If `X` is a `BEDMatrix` object, `loci_on_cols` is ignored (set automatically to `TRUE` internally).
`mean_of_ratios`	Chose how to weigh loci. If `FALSE` (default) loci have equal weights (in terms of variance, rare variants contribute less than common variants; also called the "ratio-of-means" version, this has known asymptotic behavior). If `TRUE`, rare variant loci are upweighed (in terms of variance, contributions are approximately the same across variant frequencies; also called the "mean-of-ratios" version, its asymptotic behavior is less well understood but performs better for association testing).
`mem_factor`	Proportion of available memory to use loading and processing data. Ignored if `mem_lim` is not `NA`.
`mem_lim`	Memory limit in GB, used to break up data into chunks for very large datasets. Note memory usage is somewhat underestimated and is not controlled strictly. Default in Linux is `mem_factor` times the free system memory, otherwise it is 1GB (Windows, OSX and other systems).
`want_M`	If `TRUE`, includes the matrix `M` of non-missing pair counts in the return value, which are sample sizes that can be useful in modeling the variance of estimates. Default `FALSE` is to return the relatedness matrix only.
`m_chunk_max`	Sets the maximum number of loci to process at the time. Actual number of loci loaded may be lower if memory is limiting.

Details

The subpopulation assignments are only used to estimate the baseline kinship (the zero value). If the user wants to re-estimate the kinship matrix using different subpopulation labels, it suffices to rescale it using `rescale_popkin()` (as opposed to starting from the genotypes again, which gives the same answer but more slowly).

Value

If `want_M = FALSE`, returns the estimated `n`-by-`n` kinship matrix only. If `X` has names for the individuals, they will be copied to the rows and columns of this kinship matrix. If `want_M = TRUE`, a named list is returned, containing:

- `kinship`: the estimated `n`-by-`n` kinship matrix
- `M`: the `n`-by-`n` matrix of non-missing pair counts (see `want_M` option).

See Also

`popkin_af()` for coancestry estimation from allele frequency matrices.

Examples

```
# Construct toy data
X <- matrix(
    c(0, 1, 2,
      1, 0, 1,
      1, 0, 2),
    nrow = 3,
    byrow = TRUE
) # genotype matrix
subpops <- c(1,1,2) # subpopulation assignments for individuals

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
## library(BEDMatrix)
## X <- BEDMatrix(file) # load genotype matrix object

kinship <- popkin(X, subpops) # calculate kinship from genotypes and subpopulation labels
```

---

popkin_A   *Compute popkin's `A` and `M` matrices from genotypes*

Description

This function returns lower-level, intermediate calculations for the main `popkin` function. These are not intended for most users, but rather for researchers studying the estimator.

Usage

```
popkin_A(
  X,
  n = NA,
  loci_on_cols = FALSE,
  mean_of_ratios = FALSE,
  mem_factor = 0.7,
  mem_lim = NA,
  m_chunk_max = 1000
)
```

Arguments

`X`	Genotype matrix, `BEDMatrix` object, or a function `X(m)` that returns the genotypes of all individuals at `m` successive locus blocks each time it is called, and `NULL` when no loci are left. If a regular matrix, `X` must have values only in `c(0, 1, 2, NA)`, encoded to count the number of reference alleles at the locus, or `NA` for missing data.
`n`	Number of individuals (required only when `X` is a function, ignored otherwise). If `n` is missing but `subpops` is not, `n` is taken to be the length of `subpops`.

`loci_on_cols`	If `TRUE`, `X` has loci on columns and individuals on rows; if `FALSE` (default), loci are on rows and individuals on columns. Has no effect if `X` is a function. If `X` is a `BEDMatrix` object, `loci_on_cols` is ignored (set automatically to `TRUE` internally).
`mean_of_ratios`	Chose how to weigh loci. If `FALSE` (default) loci have equal weights (in terms of variance, rare variants contribute less than common variants; also called the "ratio-of-means" version, this has known asymptotic behavior). If `TRUE`, rare variant loci are upweighed (in terms of variance, contributions are approximately the same across variant frequencies; also called the "mean-of-ratios" version, its asymptotic behavior is less well understood but performs better for association testing).
`mem_factor`	Proportion of available memory to use loading and processing data. Ignored if `mem_lim` is not `NA`.
`mem_lim`	Memory limit in GB, used to break up data into chunks for very large datasets. Note memory usage is somewhat underestimated and is not controlled strictly. Default in Linux is `mem_factor` times the free system memory, otherwise it is 1GB (Windows, OSX and other systems).
`m_chunk_max`	Sets the maximum number of loci to process at the time. Actual number of loci loaded may be lower if memory is limiting.

Value

A named list containing:

- `A`: n-by-n matrix, for individuals `j` and `k`, of average `w_i * ( ( x_ij - 1 ) * ( x_ik - 1 ) - 1)` values across all loci `i` in `X`; if `mean_of_ratios = FALSE`, `w_i = 1`, otherwise `w_i = 1 / (p_est_i * (1 - p_est_i) )` where `p_est_i` is the reference allele frequency.
- `M`: n-by-n matrix of sample sizes (number of loci with non-missing individual `j` and `k` pairs, used to normalize `A`)

See Also

The main `popkin()` function (a wrapper of this `popkin_A` function and `popkin_A_min_subpops()` to estimate the minimum `A` value).

Examples

```
# Construct toy data
X <- matrix(c(0,1,2,1,0,1,1,0,2), nrow = 3, byrow = TRUE) # genotype matrix

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
# library(BEDMatrix)
# X <- BEDMatrix(file) # load genotype matrix object

obj <- popkin_A(X) # calculate A and M from genotypes
A <- obj$A
M <- obj$M
```

popkin_af	*Estimate coancestry from an allele frequency matrix and subpopulation assignments*

Description

Given the individual-specific allele frequencies of n individuals, this function returns the n-by-n coancestry matrix. This function is the analog of popkin() for allele frequencies rather than genotypes, and as a consequence estimates coancestry instead of kinship. These coancestry estimates are unbiased if the true allele frequencies are provided, but may be less accurate when the allele frequencies themselves are estimated. This function is intended for cases where allele frequencies, but not individual genotypes, are available; otherwise it is best to use the individual genotypes and popkin(). An application of interest is the allele frequency matrices from admixture models, in which case the columns correspond to subpopulations rather than individuals, and subpops = NULL is an acceptable choice.

Usage

```
popkin_af(
  P,
  subpops = NULL,
  loci_on_cols = FALSE,
  mem_factor = 0.7,
  mem_lim = NA,
  want_M = FALSE,
  m_chunk_max = 1000
)
```

Arguments

P	m-by-n matrix of individual-specific allele frequencies, which should have values between [0, 1] (range is not strictly required) or NA for missing data.
subpops	The length-n vector of subpopulation assignments for each individual. If NULL, every individual is effectively treated as a different population.
loci_on_cols	If TRUE, P has loci on columns and individuals on rows; if FALSE (default), loci are on rows and individuals on columns.
mem_factor	Proportion of available memory to use loading and processing data. Ignored if mem_lim is not NA.
mem_lim	Memory limit in GB, used to break up data into chunks for very large datasets. Note memory usage is somewhat underestimated and is not controlled strictly. Default in Linux is mem_factor times the free system memory, otherwise it is 1GB (Windows, OSX and other systems).
want_M	If TRUE, includes the matrix M of non-missing pair counts in the return value, which are sample sizes that can be useful in modeling the variance of estimates. Default FALSE is to return the relatedness matrix only.
m_chunk_max	Sets the maximum number of loci to process at the time. Actual number of loci loaded may be lower if memory is limiting.

Value

If `want_M = FALSE`, returns the estimated `n`-by-`n` coancestry matrix only. If `P` has names for the individuals, they will be copied to the rows and columns of this coancestry matrix. If `want_M = TRUE`, a named list is returned, containing:

- `coancestry`: the estimated `n`-by-`n` coancestry matrix
- `M`: the `n`-by-`n` matrix of non-missing pair counts (see `want_M` option).

See Also

`popkin()` for kinship estimation from genotype matrices.

Examples

```
# a matrix of random uniform allele frequencies
# (unstructured, unlike real data)
P <- matrix( runif( 9 ), nrow = 3 )

coancestry <- popkin_af( P )
```

---

popkin_A_min_subpops — *Estimate the minimum expected value of a matrix* `A` *using subpopulations*

Description

This function averages the values of a square matrix `A` between every subpopulation pair and returns the minimum of these averages. If no subpopulation partition is provided, the function returns the minimum value of `A` excluding the diagonal, to agree when the code treats each individual as a subpopulation. The return value can be used to adjust an `A` matrix to yield the kinship matrix.

Usage

```
popkin_A_min_subpops(A, subpops = NULL)
```

Arguments

`A`	A symmetric `n`-by-`n` matrix with values between every individual pair, including self comparisons.
`subpops`	A length-`n` vector of subpopulation assignments for each individual. If missing, every individual is treated as a different subpopulation.

Value

The minimum of the average between-subpopulation `A` values, which estimates the minimum expected value of `A`

See Also

popkin_A() to generate the `A` matrix usually inputted into this function (`popkin_A_min_subpops`). popkin() is the wrapper function around both of these.

avg_kinship_subpops() for the full matrix of mean kinship values between subpopulations.

Examples

```
# Construct toy data
X <- matrix(c(0,1,2,1,0,1,1,0,2), nrow=3, byrow=TRUE) # genotype matrix
subpops <- c(1,1,2) # subpopulation assignments for individuals

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
## library(BEDMatrix)
## X <- BEDMatrix(file) # load genotype matrix object

# calculate A from genotypes
A <- popkin_A(X)$A

# the recommended form using appropriate subpopulation labels
A_min_est <- popkin_A_min_subpops( A, subpops )

# this recovers the popkin estimate
kinship <- 1 - A / A_min_est
stopifnot( kinship == popkin( X, subpops ) )

# a simple default for exploratory analysis, equals min( A ) for correctly-calculated A
A_min_est <- popkin_A_min_subpops( A )
stopifnot( A_min_est == min( A ) )
```

pwfst — *Estimate the individual-level pairwise FST matrix*

Description

This function construct the individual-level pairwise FST matrix implied by the input kinship matrix. If the input is the true kinship matrix, the return value corresponds to the true pairwise FST matrix. On the other hand, if the input is the estimated kinship returned by popkin(), the same code results in the return value being the pairwise FST estimates described in our paper. In all cases the diagonal of the pairwise FST matrix is zero by definition.

Usage

```
pwfst(kinship)
```

Arguments

`kinship`	The n-by-n kinship matrix

Value

The n-by-n pairwise FST matrix

Examples

```
# Construct toy data
X <- matrix(c(0,1,2,1,0,1,1,0,2), nrow=3, byrow=TRUE) # genotype matrix
subpops <- c(1,1,2) # subpopulation assignments for individuals

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
## library(BEDMatrix)
## X <- BEDMatrix(file) # load genotype matrix object

# estimate the kinship matrix from the genotypes "X"!
kinship <- popkin(X, subpops) # calculate kinship from X and optional subpop labels

# lastly, compute pairwise FST matrix from the kinship matrix
pwF <- pwfst(kinship)
```

`rescale_popkin` *Rescale kinship matrix to set a given kinship value to zero.*

Description

If you already have a population kinship matrix, and you desire to estimate the kinship matrix in a subset of the individuals, you could do it the slow way (reestimating starting from the genotypes of the subset of individuals) or you can do it the fast way: first subset the kinship matrix to only contain the individuals of interest, then use this function to rescale this kinship matrix so that the minimum kinship is zero. This rescaling is required when subsetting results in a more recent Most Recent Common Ancestor (MRCA) population compared to the original dataset (for example, if the original data had individuals from across the world but the subset only contains individuals from a single continent).

Usage

```
rescale_popkin(kinship, subpops = NULL, min_kinship = NA)
```

Arguments

`kinship`	An n-by-n kinship matrix.
`subpops`	The length-n vector of subpopulation assignments for each individual.
`min_kinship`	A scalar kinship value to define the new zero kinship.

Details

This function rescales the input `kinship` matrix so that the value `min_kinship` in the original kinship matrix becomes zero, using the formula `kinship_rescaled = ( kinship - min_kinship ) / ( 1 - min_kinship )`. This is equivalent to changing the ancestral population of the data. If subpopulation labels `subpops` are provided (recommended), they are used to estimate `min_kinship` using the function `popkin_A_min_subpops()`, which is the recommended way to set the MRCA population correctly. If both `subpops` and `min_kinship` are provided, only `min_kinship` is used. If both `subpops` and `min_kinship` are omitted, the function sets `min_kinship = min( kinship )`.

Value

The rescaled `n`-by-`n` kinship matrix, with the desired level of relatedness set to zero.

Examples

```
# Construct toy data
X <- matrix(c(0,1,2,1,0,1,1,0,2), nrow=3, byrow=TRUE) # genotype matrix
subpops <- c(1,1,2) # subpopulation assignments for individuals
subpops2 <- 1:3 # alternate labels treat every individual as a different subpop

# NOTE: for BED-formatted input, use BEDMatrix!
# "file" is path to BED file (excluding .bed extension)
## library(BEDMatrix)
## X <- BEDMatrix(file) # load genotype matrix object

# suppose we first estimate kinship without subpopulations, which will be more biased
kinship <- popkin(X) # calculate kinship from genotypes, WITHOUT subpops
# then we visualize this matrix, figure out a reasonable subpopulation partition

# now we can adjust the kinship matrix!
kinship2 <- rescale_popkin(kinship, subpops)
# prev is faster but otherwise equivalent to re-estimating kinship from scratch with subpops:
# kinship2 <- popkin(X, subpops)

# can also manually set the level of relatedness min_kinship we want to be zero:
min_kinship <- min(kinship) # a naive choice for example
kinship2 <- rescale_popkin(kinship, min_kinship = min_kinship)

# lastly, omiting both subpops and min_kinship sets the minimum value in kinship to zero
kinship3 <- rescale_popkin(kinship2)
# equivalent to both of:
# kinship3 <- popkin(X)
# kinship3 <- rescale_popkin(kinship2, min_kinship = min(kinship))
```

`validate_kinship` *Validate a kinship matrix*

Description

Tests that the input is a valid kinship matrix (a numeric, square, and optionally symmetric R matrix). Intended for matrices to plot and for other uses, including biased estimates, so there is flexibility as to what constitutes a valid kinship matrix. Throws errors if the input is not as above. Can instead return `TRUE`/`FALSE` if `logical = TRUE`.

Usage

```
validate_kinship(kinship, sym = TRUE, name = "kinship", logical = FALSE)
```

Arguments

`kinship`	The kinship matrix to validate.
`sym`	If `TRUE` (default), the matrix is required to be symmetric. Otherwise this particular test is skipped.
`name`	Default "kinship". Change to desired variable name for more informative error messages (i.e. "A" when used to validate the `A` matrix inside `popkin_A_min_subpops`).
`logical`	If `FALSE` (default), function stops with an error message if the input is not a kinship matrix. If `TRUE`, function instead returns `TRUE` if the input passed all tests (appears to be a valid kinship matrix) or `FALSE` otherwise.

Details

True kinship matrices have values strictly between 0 and 1, and diagonal values strictly between 0.5 and 1. However, estimated matrices may contain values slightly out of range. For greater flexibility, this function does not check for out-of-range values.

Value

If `logical = FALSE` (default), nothing. If `logical = TRUE`, returns `TRUE` if the input is a valid kinship matrix, `FALSE` otherwise.

Examples

```
# this is a valid (positive) example
kinship <- matrix(c(0.5, 0, 0, 0.6), nrow=2)
# this will run without errors or warnings
validate_kinship(kinship)

# negative examples

# dies if input is missing
try( validate_kinship() )

# and if input is not a matrix
try( validate_kinship( 1:5 ) )

# and for non-numeric matrices
char_mat <- matrix(c('a', 'b', 'c', 'd'), nrow=2)
```

```
try( validate_kinship( char_mat ) )

# and non-square matrices
non_kinship <- matrix(1:2, nrow=2)
try( validate_kinship( non_kinship ) )

# and non-symmetric matrices
non_kinship <- matrix(1:4, nrow=2)
try( validate_kinship( non_kinship ) )
# but example passes if we drop symmetry requirement this way
validate_kinship( non_kinship, sym = FALSE )

# instead of stopping, can get a logical value
# this returns FALSE
validate_kinship( non_kinship, logical = TRUE )
```

`weights_subpops` *Get weights for individuals that balance subpopulations*

Description

This function returns positive weights that sum to one for individuals using subpopulation labels, such that every subpopulation receives equal weight. In particular, if there are `K` subpopulations, then the sum of weights for every individuals of a given subpopulation will equal `1 / K`. The weight of every individual is thus inversely proportional to the number of individuals in its subpopulation. If the optional sub-subpopulation labels are also provided, then each sub-subpopulation within a given subpopulation is also weighted equally.

Usage

```
weights_subpops(subpops, subsubpops = NULL)
```

Arguments

`subpops`	The length-`n` vector of subpopulation assignments for each individual.
`subsubpops`	The optional length-`n` vector of sub-subpopulation assignments for each individual. Each sub-subpopulation must belong to a single subpopulation (a nested hierarchy) or an error is produced.

Value

The length-`n` vector of weights for each individual.

Examples

```
# if every individual has a different subpopulation, weights are uniform:
subpops <- 1:10
weights <- weights_subpops( subpops )
stopifnot( all( weights == rep.int( 1/10, 10 ) ) )

# subpopulations can be strings too
subpops <- c('a', 'b', 'c')
weights <- weights_subpops( subpops )
stopifnot( all( weights == rep.int( 1/3, 3 ) ) )

# if there are two subpopulations
# and the first has twice as many individuals as the second
# then the individuals in this first subpopulation weight half as much
# as the ones in the second subpopulation
subpops <- c(1, 1, 2)
weights <- weights_subpops( subpops )
stopifnot( all( weights == c( 1/4, 1/4, 1/2 ) ) )

# 2-level hierarchy example
subpops <- c(1, 1, 1, 2, 2)
subsubpops <- c('a', 'b', 'b', 'c', 'd')
weights <- weights_subpops( subpops, subsubpops )
stopifnot( all( weights == c( 1/4, 1/8, 1/8, 1/4, 1/4 ) ) )
```

Index