Package 'BitBreedingSim'

May 14, 2025

Type Package

Title Fast Breeding Simulation

Version 0.1.0

Author Minoru Inamori

Maintainer Minoru Inamori <inamori@ut-biomet.org>

Description Use bit operations to speed up breeding simulations.

License MIT License

Imports Rcpp (>= 1.0.5)

LinkingTo Rcpp

RoxygenNote 7.3.2

Encoding UTF-8

Suggests roxygen2

Roxygen list(markdown = TRUE)

Contents

`add_pop_to_VCF` *Add a Population object to a VCF object*

Description

This function adds a Population object to an existing VCF object and creates a new VCF object that includes the population information.

Usage

```
add_pop_to_VCF(vcf, pop)
```

Arguments

`vcf`	A VCF object. This should be an object created using the relevant VCF functions in the package.
`pop`	A Population object. This should be an object created using the relevant Population functions in the package.

Value

A new VCF object that includes the added Population information.

Examples

```
# Assume 'vcf' is a valid VCF object and 'pop' is a valid Population object
new_vcf <- add_pop_to_VCF(vcf, pop)
summary(new_vcf) # Inspect the new VCF object
```

add_trait_A *Add Trait with Additive Effects to BaseInfo*

Description

This function adds a trait with additive effects to the BaseInfo object.

Usage

```
add_trait_A(
  info,
  name,
  mean,
  h2,
  sd = NULL,
  a = NULL,
  loci = NULL,
  num_loci = 1
)
```

Arguments

info	External pointer to BaseInfo object
name	Name of the trait
mean	Phenotype mean
h2	Heritability
sd	Optional. Phenotype standard deviation
a	Optional. Numeric vector of additive effects

loci	Optional. List of loci in the form of a data frame with two columns: 'chrom' and 'marker'. For example:
	`chrom <- c(3, 5, 10)`
	`marker <- c(1, 2, 1000)`
	`loci <- data.frame(chrom, marker)`

num_loci	Optional. Number of loci (default is 1)

Examples

```
# Assume 'info' is a valid BaseInfo object
add_trait_A(info, "Trait1", mean=100.0, h2=0.6, sd=10.0, num_loci = 2)
trait <- get_trait(info, 1)
summary(trait)
```

add_trait_AD *Add Trait with Additive and Dominance Effects to BaseInfo*

Description

This function adds a trait with additive and dominance effects to the BaseInfo object.

Usage

```
add_trait_AD(
  info,
  name,
  mean,
  sd = NULL,
  h2 = NULL,
  H2 = NULL,
  a = NULL,
  d = NULL,
  loci = NULL,
  num_loci = 1
)
```

Arguments

info	External pointer to BaseInfo object
name	Name of the trait
mean	Phenotype mean
sd	Optional. Phenotype standard deviation
h2	Optional. Narrow-sense heritability (proportion of variance due to additive genetic effects)
H2	Optional. Broad-sense heritability (proportion of variance due to all genetic effects, can be NULL)
a	Optional. Numeric vector of additive effects
d	Optional. Numeric vector of dominance effects

loci	Optional. List of loci in the form of a data frame with two columns: 'chrom' and 'marker'. For example:
	`chrom <- c(3, 5, 10)`
	`marker <- c(1, 2, 1000)`
	`loci <- data.frame(chrom, marker)`

num_loci	Optioal. Number of loci (default is 1)

Examples

```
# Assume 'info' is a valid BaseInfo object
add_trait_AD(info, "Trait1", mean=100.0, h2=0.6, H2=0.7, sd=10.0, num_loci = 2)
trait <- get_trait(info, 1)
summary(trait)
```

`check_dataframe` *Validate the structure and content of the data frame used for crossing information*

Description

This function checks if the input data frame `df` meets the required structure for the `cross_by_table` function. It ensures that `df` is a valid `data.frame`, contains the required columns (`mat`, `pat`, `num`), and verifies that the `num` column has positive numeric values. If any of these checks fail, the function stops execution and displays an error message.

Usage

```
check_dataframe(df)
```

Arguments

`df`	A data frame. Must include the following columns: - `mat`: Names of the maternal parents for each cross. - `pat`: Names of the paternal parents for each cross. - `num`: Number of offspring to generate for each cross. Must contain positive numeric values.

Value

No return value. If the validation is successful, the function proceeds silently. If the validation fails, the function stops execution and displays an error message.

Examples

```
# Valid input example
df <- data.frame(mat = c("mat1", "mat2"), pat = c("pat1", "pat2"), num = c(1, 2))
check_dataframe(df) # Passes silently

# Invalid input example: Missing 'num' column
df_invalid <- data.frame(mat = c("mat1", "mat2"), pat = c("pat1", "pat2"))
check_dataframe(df_invalid) # Throws an error
```

check_parent_existance

Check Parent Existence in Population

Description

This function checks whether the maternal and paternal names in the given cross table are present in the specified maternal and paternal populations.

Usage

```
check_parent_existance(df, mat_pop, pat_pop)
```

Arguments

df	A data.frame representing the cross table containing 'mat' (maternal names) and 'pat' (paternal names) columns.
mat_pop	An external pointer to the maternal Population object.
pat_pop	An external pointer to the paternal Population object.

Value

This function does not return a value. It outputs messages if any maternal or paternal names in the cross table are not found in the respective populations.

Examples

```
# Assuming 'df', 'mat_pop', and 'pat_pop' are valid objects
check_parent_existance(df, mat_pop, pat_pop)
```

create_base_info

Create a BaseInfo object

Description

This function creates a BaseInfo object. If the seed is set to -1, a random seed is generated, resulting in different outcomes each time the function is called. If a specific seed is provided, the random number generation will be based on that seed, ensuring reproducible results.

Usage

```
create_base_info(
  positions = NULL,
  num_markers = 1000,
  bp = 1e+08,
  chrom_maps = NULL,
  num_chroms = 10,
  cM = 100,
  seed = -1
)
```

Arguments

positions	Optional. A list of numeric vectors, where each vector represents the marker positions for a chromosome in base pairs. The number of elements in positions defines the number of chromosomes and overrides num_chroms. If positions is provided, num_markers is ignored.
num_markers	Optional. An integer. Number of markers per chromosome. Ignored if positions is provided. Default is 1000.
bp	Optional. An integer. Length of each chromosome in base pairs. Ignored if positions is provided. Default is 100000000.
chrom_maps	Optional. A list of data.frames, each representing a chromosome map. Each data.frame should have two columns: 'cM' for centiMorgans and 'position' for base pair positions. The list should be named, with each name corresponding to a chromosome identifier (e.g., "chr1", "chr2", etc.). For a given cM value, the corresponding position can be interpolated or extrapolated, and vice versa. This effectively represents the mapping as a piecewise linear approximation of the chromosome's relationship between cM and base pair positions. If chrom_maps is provided, the parameter num_chroms is ignored.
num_chroms	Optional. An integer. Number of chromosomes. Ignored if either chrom_maps or positions is provided. Default is 10.
cM	Optional. A numeric. Length of each chromosome in centiMorgans. Ignored if chrom_maps is provided. Default is 100.
seed	Optional. An integer. A seed for random number generation. Default is -1, which generates a random seed.

Value

An external pointer to a BaseInfo object.

Examples

```
# Create a BaseInfo object with a random seed
base_info_random <- create_base_info()
get_info(base_info_random)

# Create a BaseInfo object with a specific seed for reproducible results
```

```
base_info_reproducible <- create_base_info(seed = 123)
get_info(base_info_reproducible)

# Create marker positions manually
num_chr <- 10
position <- sapply(1:100, function(i) i * 1000000)
positions <- replicate(10, position, simplify = FALSE)

# Create a chromosome map with 100 cM and 100 Mbp
cM <- c(14.0, 27.9, 40.2, 48.3, 50.0, 51.7, 59.8, 72.1, 86.0, 100.0)
chr_position <- sapply(1:10, function(i) i * 10000000)
chrom_map <- data.frame(cM, chr_position)
chrom_maps <- replicate(10, chrom_map, simplify = FALSE)
names(chrom_maps) <- paste0("chr", 1:10)
info <- create_base_info(positions = positions, chrom_maps = chrom_maps, seed = 123)
get_info(info)
```

`create_HaploArray_from_pop` *Create a 3-dimensional HaploArray from a Population object*

Description

This function converts a Population object into a 3-dimensional array (HaploArray). The resulting HaploArray contains dimensions representing individuals, markers, and alleles (Maternal and Paternal). The HaploArray has the following dimensions:

dim1 Number of individuals.

dim2 Number of markers.

dim3 Size is 2, representing Maternal and Paternal alleles.

The array also includes dimension names such as individual names, marker names, and allele labels (Maternal and Paternal).

Usage

```
create_HaploArray_from_pop(pop)
```

Arguments

pop	An external pointer to a Population object. This object must be properly initialized.

Details

Note: This function is the reverse of `create_pop_from_HaploArray`, which creates a Population object from a HaploArray.

Value

A 3-dimensional array (HaploArray) containing the following dimensions:

Individuals Individual names as rows.

Markers Marker names as columns.

Alleles Maternal and Paternal alleles as the third dimension.

Examples

```
info <- create_base_info(num_chroms=2, num_markers=10, cM=100, bp=1e6, seed=2)
add_trait_A(info, "Trait1", mean=100.0, h2=0.6, sd=10.0, num_loci = 2)
trait <- get_trait(info, 1)

haploArray <- array(
  data = rbinom(n = 120, size = 1, prob = 0.3),
  dim = c(3, 20, 2),
  dimnames = list(
    paste0("Ind_", 1:3),
    paste0("Mrk_", 1:20),
    c("Maternal", "Paternal")
  )
)
pop <- create_pop_from_HaploArray(haploArray, info)
prog <- cross_randomly(pop, pop, num_inds = 5, name_base = "prog_")
haploArrayProg <- create_HaploArray_from_pop(prog)
```

---

create_info_pop_from_VCF

Create BaseInfo and Population from a VCF file

Description

This function takes a VCF object and a seed value, and returns both a Population object and its associated BaseInfo object. It reads the input VCF and initializes the data accordingly. The seed value is used to initialize the pseudo-random number generator for the BaseInfo object. If the seed is set to -1, an appropriate value will be automatically chosen.

Usage

```
create_info_pop_from_VCF(vcf, seed = -1)
```

Arguments

vcf	An external pointer to a VCF object.
seed	An integer. The seed value for initializing the BaseInfo object's pseudo-random number generator. Defaults to -1, which automatically selects a suitable seed.

Value

A list containing two elements:

info An external pointer to a BaseInfo object.

pop An external pointer to a Population object.

Examples

```
# Assuming 'vcf_file' is a valid VCF file
vcf <- read_VCF(vcf_file)
result <- create_info_pop_from_VCF(vcf, seed = 42)
summary(result$info)
summary(result$pop)
```

---

create_origins *Create origins for a Population object*

Description

Create origins for a Population object

Usage

```
create_origins(info, names = NULL, num_inds = NULL, name_base = NULL)
```

Arguments

info	An external pointer to a BaseInfo object.
num_inds	An integer. The number of individuals.
name_base	A string. The base name for individuals.

Value

An external pointer to a Population object.

Examples

```
# Example 1: Specify individual names directly
pop <- create_origins(info, names = c("p1", "p2"))
get_individual_names(pop)

# Example 2: Generate individual names using name_base and num_inds
pop2 <- create_origins(info, num_inds = 2, name_base = "q")
get_individual_names(pop2)
```

create_pop_from_HaploArray *Create a Population object from a HaploArray*

Description

This function takes a 3-dimensional array (HaploArray) and a BaseInfo object to create a Population object. The HaploArray should have dimensions representing individuals, markers, and alleles (Maternal and Paternal). Allele values must be binary (0 or 1). A value of 0 indicates that the allele matches the reference genome, while a value of 1 indicates the alternative allele. The function validates the input data and initializes the Population object accordingly.

Usage

```
create_pop_from_HaploArray(haploArray, info)
```

Arguments

haploArray	A 3-dimensional array where: **dim1** Number of individuals. **dim2** Number of markers. **dim3** Size must be 2, representing Maternal and Paternal alleles. Allele values must be binary (0 or 1), based on the reference genome.
info	An external pointer to a BaseInfo object.

Value

An external pointer to a Population object.

Examples

```
# Create a HaploArray
haploArray <- array(
  data = rbinom(n = 30, size = 1, prob = 0.3),
  dim = c(3, 5, 2),
  dimnames = list(
    paste0("Ind_", 1:3),
    paste0("Mrk_", 1:5),
    c("Maternal", "Paternal")
  )
)

# Assuming 'info' is a valid BaseInfo object
pop <- create_pop_from_HaploArray(haploArray, info)
summary(pop)
```

`cross_by_table` *Cross populations according to a table*

Description

This function performs crossing of individuals from two Population objects (maternal and paternal parents) based on specified crossing information provided in a table. Each row of the table defines a specific cross, including the number of offspring to generate for that cross. You can specify either `name_base` or provide a complete `names` vector for the offspring.

Usage

```
cross_by_table(
  df,
  mat_pop,
  pat_pop,
  names = NULL,
  name_base = NULL,
  num_threads = 0
)
```

Arguments

`df`	A data frame containing crossing information. It should include the following columns: `mat` Names of the maternal parents for each cross. `pat` Names of the paternal parents for each cross. `num` Number of offspring to generate for each cross.
`mat_pop`	A Population object representing the maternal parents. This should be an object created using the relevant Population functions in the package.
`pat_pop`	A Population object representing the paternal parents. This should be an object created using the relevant Population functions in the package.
`names`	Optional. A character vector containing specific names for individuals. If provided, the length of `names` should match the total number of offspring specified in `df`.
`name_base`	Optional. A character string used as the base name for generating individual names. For example, if `name_base = "prog_"`, the generated names will follow the format `c("prog_1", "prog_2", ...)`.
`num_threads`	Optional. A positive integer representing the number of threads to use. If not specified or set to 0, the maximum number of available threads is used.

Value

A Population object with offspring individuals.

Examples

```
# Example using name_base
df <- data.frame(mat = c("mat1", "mat2"), pat = c("pat1", "pat2"), num = c(1, 2))
new_population <- cross_by_table(df, mat_pop, pat_pop, name_base = "prog_")

# Example using predefined names
names_vector <- c("child1", "child2", "child3")
new_population <- cross_by_table(df, mat_pop, pat_pop, names = names_vector)

# Summary of the new Population object
summary(new_population)
```

---

cross_randomly *Cross Population randomly*

Description

This function performs random crossing of individuals from two Population objects (maternal and paternal parents) to generate a new Population object. For each offspring, a parent is randomly selected from each Population (one from the maternal Population and one from the paternal Population). You can specify either `name_base` and `num_inds` or provide a complete `names` vector for the offspring.

Usage

```
cross_randomly(
  mat_pop,
  pat_pop,
  names = NULL,
  num_inds = NULL,
  name_base = NULL,
  num_threads = 0
)
```

Arguments

`mat_pop`	A Population object representing the maternal parents. This should be an object created using the relevant Population functions in the package.
`pat_pop`	A Population object representing the paternal parents. This should be an object created using the relevant Population functions in the package.
`names`	Optional. A character vector containing specific names for individuals. If provided, the length of `names` determines the number of offspring.
`num_inds`	Optional. A positive integer representing the number of offspring. Required if `name_base` is specified.

name_base	Optional. A character string used as the base name for generating individual names. For example, if name_base = "p" and num_inds = 3, the generated names will be c("p1", "p2", "p3"). This parameter must be used in combination with num_inds.
num_threads	Optional. A positive integer representing the number of threads to use. If not specified or set to 0, the maximum number of available threads is used.

Value

A Population object with offspring individuals.

Examples

```
# Example using name_base and num_inds
new_population <- cross_randomly(mothers, fathers, num_inds = 100, name_base = "offspring_")

# Example using predefined names
names_vector <- c("child1", "child2", "child3")
new_population <- cross_randomly(mothers, fathers, names = names_vector)

# Summary of the new Population object
summary(new_population)
```

---

extract_map_data *Extract chromosome and position data from a Map object*

Description

This function retrieves detailed data from a Map object, including chromosome names and marker positions, and returns the data as a data.frame for easier manipulation.

Usage

```
extract_map_data(map)
```

Arguments

map	An external pointer to a Map object. This object must be properly initialized and created using appropriate functions.

Value

A data.frame containing the following columns:

chrom A character vector containing the names of chromosomes.

position An integer vector specifying the positions of markers along the chromosomes.

Examples

```
# Assuming 'map' is a valid Map object
map_data <- extract_map_data(map)
head(map_data) # Display the first few rows of the data.frame
```

---

generate_names *Generate names for individuals based on a base name and number of individuals*

Description

This function creates a vector of names for individuals when `names` is not provided. It uses a base name (`name_base`) and a specified number of individuals (`num_inds`) to generate names in the format `name_base1`, `name_base2`, ..., up to the total number.

Usage

```
generate_names(name_base, num_inds)
```

Arguments

`name_base`	A character string used as the base for generating names. For example, `name_base = "p"` will generate names like `"p1", "p2", ...`.
`num_inds`	A positive integer indicating the number of names to generate.

Value

A character vector of generated names.

Examples

```
# Generate 3 names with base "p"
generated_names <- generate_names("p", 3)
print(generated_names) # Output: "p1", "p2", "p3"
```

---

get_genotypes *Get genotypes from a Population object*

Description

This function retrieves the genotypes from a given Population object. The genotypes are represented in a matrix format where rows correspond to individuals and columns correspond to markers.

Usage

```
get_genotypes(pop)
```

Arguments

pop	An external pointer to a Population object.

Details

Genotype encoding:

- 0/0 is encoded as -1
- 0/1 is encoded as 0
- 1/1 is encoded as 1

Value

A matrix of genotypes where rows are individuals and columns are markers.

Examples

```
info <- create_base_info()
pop <- create_origins(info, num_inds = 2, names = c("p1", "p2"))
geno <- get_genotypes(pop)
geno[, 1:5]
```

---

get_individual_names *Get name data from a Population object*

Description

Get name data from a Population object

Usage

```
get_individual_names(pop)
```

Arguments

pop	An external pointer to a Population object.

Value

A data.frame containing the following columns:

name Names from the Population object

mat Maternal names from the Population object

pat Paternal names from the Population object

Examples

```
# Assuming 'pop' is a valid Population object
name_data <- get_individual_names(pop)
print(name_data)
```

`get_info` *Retrieve key values from a BaseInfo object*

Description

This function extracts essential information from a BaseInfo object. A BaseInfo object stores metadata related to genetic simulations, such as the number of chromosomes, traits, and markers. This function is useful for summarizing or verifying the content of a BaseInfo object during analysis.

Usage

```
get_info(info)
```

Arguments

`info`	An external pointer to a BaseInfo object. This object must be properly initialized and created using the appropriate BaseInfo constructor or relevant functions.

Value

A list containing the following elements:

num_chroms An integer representing the total number of chromosomes contained in the BaseInfo object.

num_traits An integer indicating the number of traits managed by the BaseInfo object.

num_markers An integer specifying the total number of markers tracked in the BaseInfo object.

Examples

```
# Assume 'info' is a valid BaseInfo object
values <- get_info(info)
print(values$num_chroms)  # Print the number of chromosomes
print(values$num_traits)  # Print the number of traits
print(values$num_markers) # Print the number of markers
```

get_map *Retrieve Genetic Map Information*

Description

This function retrieves the genetic map information from a `BaseInfo` object.

Usage

```
get_map(info)
```

Arguments

`info`	An object of class `BaseInfo`. This object contains the genetic map information.

Value

A list of data frames, each representing a chromosome. Each data frame contains two columns:

- `cM`: The centiMorgan positions of the markers.
- `position`: The base pair positions of the markers.

Examples

```
## Not run:
  # Assuming `info` is a valid BaseInfo object
  map <- get_map(info)
  print(map)

## End(Not run)
```

get_phased_genotypes *Get phased genotypes from a Population object*

Description

This function retrieves the genotypes from a given Population object. The genotypes are represented in a matrix format where rows correspond to markers and columns correspond to individuals.

Usage

```
get_phased_genotypes(pop)
```

Arguments

`pop`	An external pointer to a Population object.

Details

Genotype is 0|0, 0|1, 1|0, or 1|1

Value

A matrix of genotypes where rows are individuals and columns are markers.

Examples

```
info <- create_base_info()
pop <- create_origins(info, num_inds = 2, names = c("p1", "p2"))
geno <- get_phased_genotypes(pop)
geno[, 1:5]
```

---

get_phased_int_genotypes

Get phased integer genotypes from a Population object

Description

This function retrieves the genotypes from a given Population object. The genotypes are represented in a matrix format where rows correspond to individuals and columns correspond to markers. Each individual has two rows: the first row represents the maternal allele and the second row represents the paternal allele.

Usage

```
get_phased_int_genotypes(pop)
```

Arguments

pop	An external pointer to a Population object.

Details

Genotype is represented as integers: 0 or 1

Value

A matrix of genotypes where rows are individuals and columns are markers. Each individual has two rows: the first row is the maternal allele and the second row is the paternal allele.

Examples

```
info <- create_base_info()
pop <- create_origins(info, num_inds = 2, names = c("p1", "p2"))
geno <- get_phased_int_genotypes(pop)
geno[, 1:5]
```

get_phenotypes *Get phenotypes from a Population object*

Description

Get phenotypes from a Population object

Usage

```
get_phenotypes(pop, i)
```

Arguments

pop	An external pointer to a Population object.
i	An integer index representing the trait for which phenotypes are to be retrieved. The index should be between 1 and the total number of traits available in the Population object.

Value

A vector of phenotypes.

Examples

```
# Assuming 'pop' is a valid Population object and trait index 1 is valid
phenotypes <- get_phenotypes(pop, 1)
print(phenotypes)
```

get_pop_info *Get information for a Population object*

Description

This function retrieves detailed information about a Population object, including the number of individuals, the number of chromosomes, and the number of markers in the population.

Usage

```
get_pop_info(pop)
```

Arguments

pop	A Population object. This should be an object created using the relevant Population functions in the package.

Value

A list containing: - num_individuals: The number of individuals in the population. - num_chromosomes: The number of chromosomes in the population. - num_markers: The number of markers in the population.

Examples

```
# Create base information
info <- create_base_info()
# Create a population with 2 individuals
pop <- create_origins(info, num_inds = 2, names = c("p1", "p2"))
# Retrieve population information
get_pop_info(pop)
```

---

get_trait *Get a trait from a BaseInfo object*

Description

Get a trait from a BaseInfo object

Usage

```
get_trait(info, i)
```

Arguments

info	An external pointer to a BaseInfo object.
i	An integer. The index of the trait to retrieve.

Value

The trait at the specified index.

Examples

```
# Assume 'info' is a valid BaseInfo object
trait <- get_trait(info, 1)
summary(trait)
```

get_VCF_info *Get information from a VCF object*

Description

This function retrieves detailed information about a VCF object, including the number of individuals and the number of markers contained within the VCF.

Usage

```
get_VCF_info(vcf)
```

Arguments

vcf	A VCF object. This should be an object created using the relevant VCF functions in the package.

Value

A list containing: - num_individuals: The number of individuals in the VCF object. - num_markers: The number of markers in the VCF object.

Examples

```
# Assume 'vcf' is a valid VCF object
vcf_info <- get_VCF_info(vcf)
print(vcf_info) # Displays the number of individuals and markers
```

join_pops *Join multiple Population objects*

Description

Join multiple Population objects

Usage

```
join_pops(...)
```

Arguments

...	External pointers to Population objects.

Value

An external pointer to a new Population object containing the combined individuals.

Examples

```
# Assuming 'pop1', 'pop2', and 'pop3' are valid Population objects
combined_pop <- join_pops(pop1, pop2, pop3)
print(combined_pop)
```

`print_trait` *Print detailed information about a Trait object*

Description

This function prints the detailed characteristics of a Trait object in a human-readable format. A Trait object represents a characteristic or feature in genetic simulations, and this function displays its metadata such as its name, type, mean, standard deviation, heritability (h2), associated loci, additive effects, and dominant effects (if applicable).

Usage

```
print_trait(trait, ...)
```

Arguments

`trait`	A Trait object containing the following elements: **name** The name of the trait as a character string. **type** The type of the trait (e.g., "quantitative" or "binary"). **mean** The mean value of the trait as a numeric. **sd** The standard deviation of the trait as a numeric. **h2** The narrow-sense heritability of the trait as a numeric (between 0 and 1). **H2** (Optional) The broad-sense heritability of the trait as a numeric (if applicable). **hasdominants** A logical value indicating whether the trait includes dominant effects. **loci** A list of loci associated with the trait, formatted as coordinate pairs. **additives** A numeric vector representing the additive effects of the trait. **dominants** (Optional) A numeric vector representing the dominant effects of the trait (if applicable).
`...`	Additional arguments (not currently used).

Details

The loci are formatted as coordinate pairs, representing their positions. Additive and dominant effects are printed as space-separated values for clarity.

Value

Prints the detailed information of the Trait object directly to the console.

Examples

```
info <- create_base_info(num_chroms=2, num_markers=10, cM=100, bp=1e6, seed=2)
addTraitA(info, "Trait1", mean=100.0, h2=0.6, sd=10.0, num_loci = 2)
trait <- getTrait(info, 1)
print_trait(trait)

# Alternatively, use the generic print() function
print(trait)
```

---

read_VCF *Read a VCF file and return a VCF object*

Description

This function reads a VCF file from the specified filename and returns a VCF object as an external pointer. If the input is not a valid character string, a message is displayed and NULL is returned.

Usage

```
read_VCF(filename)
```

Arguments

filename	A character string specifying the path to the VCF file.

Value

An external pointer to a VCF object, or NULL if an error occurs.

Examples

```
# Assuming 'example.vcf' is a valid VCF file
vcf <- read_VCF("example.vcf")
if (is.null(vcf)) {
  cat("Failed to read the VCF file.\n")
}
```

select_pop *Select individuals from a Population object*

Description

Select individuals from a Population object

Usage

```
select_pop(pop, indices)
```

Arguments

pop	An external pointer to a Population object.
indices	A vector of integer indices representing the individuals to be selected.

Value

An external pointer to a new Population object containing the selected individuals.

Examples

```
# Assuming 'pop' is a valid Population object and indices are valid
selected_pop <- select_pop(pop, c(1, 2, 3))
print(selected_pop)
```

set_individual_names *Set individual names for a Population object*

Description

This function assigns names to individuals in a Population object. The names argument must be a character vector, and pop must be an object of the "Population" class.

Usage

```
set_individual_names(names, pop)
```

Arguments

names	A character vector containing the names of individuals to assign.
pop	A Population object where the individual names will be set.

Value

None. The function modifies the Population object directly.

Examples

```
# Create a Population object and set individual names
info <- create_base_info()
pop <- create_origins(info, num_inds = 2, name_base = "ind")
set_individual_names(c("sample1", "sample2"), pop)

# Access the updated individual names
get_individual_names(pop)
```

---

summary_map *Summarize the details of a Map object*

Description

This function extracts and displays key information from a Map object. A Map object represents the genetic map used in simulations, containing details about chromosomes and points (e.g., markers or loci) along the map. The summary provides an overview of the structure and size of the map.

Usage

```
summary_map(map, ...)
```

Arguments

map	An external pointer to a Map object. This object must be properly initialized and created using the appropriate functions.
...	Additional arguments (not currently used).

Value

Prints a formatted summary of the Map object directly to the console.

Examples

```
# Assuming 'map' is a valid Map object
summary_map(map)
```

summary_population *Summarize the details of a Population object*

Description

This function extracts and displays key information from a Population object. A Population object represents a collection of individuals, chromosomes, and genetic markers used in genetic simulations. The summary includes the number of individuals, chromosomes, and markers in the population, providing an overview of its structure and size.

Usage

```
summary_population(pop, ...)
```

Arguments

pop	An external pointer to a Population object. This object must be properly initialized and created using the appropriate functions or constructors.
...	Additional arguments (not currently used).

Value

Prints a formatted summary of the Population object directly to the console.

Examples

```
# Assuming 'pop' is a valid Population object
summary_population(pop)
```

summary_trait *Summarize the details of a Trait object*

Description

This function provides a formatted summary of a Trait object. A Trait object represents a characteristic or feature in genetic simulations, and contains metadata such as its name, type, mean, standard deviation, heritability (h2), and information about associated loci. If the trait includes dominant effects, the broad-sense heritability (H2) is also displayed.

Usage

```
summary_trait(trait)
```

Arguments

trait	A Trait object containing the following elements: **name** The name of the trait as a character string. **type** The type of the trait (e.g., "quantitative" or "binary"). **mean** The mean value of the trait as a numeric. **sd** The standard deviation of the trait as a numeric. **h2** The narrow-sense heritability of the trait as a numeric (between 0 and 1). **H2** (Optional) The broad-sense heritability of the trait as a numeric (if applicable). **hasdominants** A logical value indicating whether the trait includes dominant effects. **loci** A list of loci associated with the trait.

Value

Prints a summary of the Trait object directly to the console.

Examples

```
info <- create_base_info(num_chroms=2, num_markers=10, cM=100, bp=1e6, seed=2)
addTraitA(info, "Trait1", mean=100.0, h2=0.6, sd=10.0, num_loci = 2)
trait <- getTrait(info, 1)
summary_trait(trait)

# Using the generic summary() function
summary(trait)
```

---

write_VCF *Write VCF or Population object to a VCF file*

Description

This function writes either a VCF object or a Population object to a VCF file. If the input is a VCF object, it directly writes the VCF data to the specified file. If the input is a Population object, it generates VCF data from the Population and writes it to the file.

Usage

```
write_VCF(obj, filename)
```

Arguments

obj	An object of class VCF or Population. The data to write to the VCF file.
filename	A string specifying the path to the output VCF file.

Examples

```
# Example 1: Write a Population object to a VCF file
pop <- create_origins(info, num_inds = 2, names = c("p1", "p2"))
write_VCF(pop, "population_output.vcf")

# Example 2: Write a VCF object to a VCF file
vcf <- read_VCF("input.vcf")
write_VCF(vcf, "vcf_output.vcf")
```

Index