MOBPS

Modular Breeding Program Simulator

Torsten Pook[1,2,3], Martin Schlather[3,4], Henner Simianer[2,3]

1: Wageningen University & Research, Animal Breeding and Genomics Group

2: University of Goettingen, Animal Breeding and Genetics Group

3: University of Goettingen, Center for Integrated Breeding Research

4: University of Mannheim, Stochastics and Its Application Group

1. DEZEMBER 2022

MoBPS v1.10.06

Table of Contents

1 General

MoBPS is an R-package to simulate complex and large-scale breeding programs with a focus on livestock and crop populations. Simulations are performed on an individual basis. MoBPS is a versatile tool, providing standard procedures applied in animal and plant breeding like GBLUP and OGC, but also allowing to use own selection schemes while still controlling the simulation of phenotypes, meiosis and costs of the simulated scheme.

In the following, the underlying grouping of individuals will be described in section 3. Next, the two key steps of the simulation procedure, namely the creation of a starting population (section 4) and the simulation of breeding processes (section 54.5) are described. As it is our goal to provide a lot of flexibility while performing the simulation, there is a need of many parameters – luckily only a few of those will be needed for most simulations. For a better overview of potential scenarios to use MoBPS a variety of relatively plant and animal breeding programs are provided in section 6.

Next, a variety of utility functions to export/import specific information directly or generate standardized plots like a PCA or derive inbreeding rates (section 7, 8, 9). Sections 10 & 11 provide technical detail on the storage structure and computing times. Sections 12, 13, 14 provide a glossary on all available input parameter in *creating.diploid()* and *breeding.diploid().* Section 15 provides technical detail on the underlying methods to simulate traits.

Lastly, a brief introduction on our graphical interface (available at www.mobps.de) is given in section 0. Note that this interface is still in active development and not part of the MoBPS paper. The use of the interface is still encouraged, but for major projects, we highly recommend close collaboration for further development or the direct use of the overall more flexible R-package itself.

For questions regarding the tool or how to set up your simulation, feel free to contact me directly (Torsten.pook@uni-goettingen.de). We are always happy for questions as it really helps improve the tool and its documentation. For quick reply, it would help to provide a small example of our problem.

2 Installation

The current version of MoBPS requires R 3.0 or higher. To ensure fast computing times we would additionally recommend the use of the R-package RandomFieldsUtils and miraculix.

Step 1 (only Windows)

On some Windows System installation of Rtools is required. This is no R-package, but separate program that can be downloaded at https://cran.r-project.org/bin/windows/Rtools/.

Step 2 (Optional)

RandomFieldsUtils and miraculix are R-package to enhance the computational performances. Please do **not** install these packages via CRAN, but download them from github: https://github.com/tpook92/MoBPS. It is important to first install RandomFieldsUtils and then miraculix.

For Linux use RandomFieldsUtils 1.0.6 and miraculix 1.0.5.

For Windows use RandomFieldsUtils 0.6.6 and miraculix 1.0.0.1.

Code in the packages is highly dependent of the respective other package. So please do not use version of the package that are not compadable with each other, but just the combinations suggested here!

Step 3

Install the MoBPS R-package itself. You can install MoBPS via CRAN, however, we could highly encourage to use the version of available on GitHub (https://github.com/tpook92/MoBPS) as we are providing basically weekly update on the framework while the CRAN version will only be updated once every couple of months.

Either download the package directly from GitHub or use the R-package devtools to install from your R prompt:

```
devtools::install_github("tpook92/MoBPS", subdir="pkg")
```

The usage of Mac OS has not been tested by us, but should also work.

Step 4 (optional)

In addition to the R-package itself, we are providing a package containing some commonly used genetic maps from Ensembl (Zerbino et al. 2018). You can install the maps package in the same way as MoBPS itself:

```
devtools::install_github("tpook92/MoBPS", subdir="pkg-maps")
```

3   Individual grouping

One of the key challenges, for any simulation program is to provide the user with the necessary flexibility to perform certain breeding actions on specific groups of individuals. The conceptual idea behind MoBPS is orientation along the gene-flow concept (Hill 1974) which is using the term of a cohort as a group of individuals with same characteristics like age, sex, and genetic origin. Note that internally all simulations are based on single individuals and gains are based on individual simulations and no expected gene flow.

3.1   gen/database/cohorts

Whenever a group of individuals is generated, the user can assign a specific and unique name – these kinds of group will later be referred to with the keyword **cohort**. Whenever a new cohort is generated it will automatically be assigned to a generation – to selected all cohorts of a certain generation, one can select the group of individuals via the parameter **gen.** Lastly, groups of individuals can be selected via the parameter **database**. A database contains four input information:

1. Generation
2. Sex (1 – male, 2- female)
3. Number of the first individual to consider (default: first individual of that generation/sex)
4. Number of the last individual to consider (default: last individual of that generation/sex)

The number of the individuals is assigned based on the order of generation in each specific generation.

As an example, consider the following selected input:

> gen = 4:5, database = matrix(c(3,1,21,50), ncol=4), cohorts="Founder"))

This parameter input will lead to MoBPS using all individuals of generations 4 and 5, as well as the male individuals 21 to 50 from generation 3 and the individuals from the cohort "Founder".

Basically all breeding actions later introduced (section 5) will rely on this to selected with groups to use for breeding value estimation (**bve.gen**, **bve.database**, **bve.cohorts**), which cohorts to phenotype (**phenotyping.gen**, **phenotyping.database**, **phenotyping.cohorts**) or to just export information from the simulated population (section 0).

3.2   Sex of individuals

Although each individual on generation will be assigned to a sex, this is not binding for downstream operations. E.g., an individual stored as a female can be used as the father (or more generally as the first parent).

In particular, in plant breeding and aquaculture, the sex of an individual might even not be unique. Sex can still be used to store individuals with more structure (e.g. each sex for a different gene pool). You can also deactivate the use of two sexes at the beginning of you simulation and store all individuals as part of the first (and then only) sex in *creating.diploid().*

4 Creation of the starting population (*creating.diploid()*)

The input for the simulation of a breeding process is a population list. This list is created via *creating.diploid()*.

We provide exemplary genetic maps for some common species, which can be selected via the parameter **template.chip**. Note that primary the number of chromosomes and their genetic length is imported at this step (especially not real markers with known allele frequencies, effects, or base pairs). The maps provided via **template.chip** are "cattle" (Ma et al. 2015), "pig" (Rohrer et al. 1994), "chicken" (Groenen et al. 2009), "sheep" (Prieur et al. 2017), and "maize" (Lee et al. 2002). Alternatively, a genomic map can be inserted via the parameter **map** with exemplary maps being provided in the package itself or via import from Ensembl (section 149.10).

4.1 Importing/Generating of a genetic dataset

In case one has haplotype data for the founders/starting population this can be imported via the parameter **dataset** in form of a haplotype dataset:

```
> dataset
       Indi1Haplo1 Indi1Haplo2 Indi2Haplo1 Indi2Haplo2 Indi3Haplo1 Indi3Haplo2 Indi4Haplo1 Indi4Haplo2
SNP1            1           1           1           1           0           1           1           0
SNP2            0           1           1           0           1           0           1           1
SNP3            0           0           0           1           1           0           0           1
SNP4            1           1           1           1           0           0           1           1
SNP5            0           1           0           1           0           1           1           1
SNP6            0           1           0           0           0           1           1           1
SNP7            1           0           1           1           1           1           1           0
SNP8            0           0           0           1           0           1           0           1
SNP9            1           1           0           0           0           0           0           1
SNP10           0           1           0           0           1           1           1           0
```

Datasets can also be imported via entering the path of the vcf file in the parameter **vcf**. The R-package vcfR is needed for this. Otherwise, a dataset can be generated by setting the number of SNPs (**nsnp**) and individuals (**nindi**) – we offer four possible modes to simulate starting haplotypes ("all0", "allhetero", "random", "homorandom") leading to haplotypes (000…/000…, 000…/111…,X_1 X_2 X_3… /X_4 X_5 X_6 … with X_i~B(**freq**) X_1 X_2 X_3 … /X_1 X_2 X_3… with X_i~B(**freq**)). On default "random" is used. The allele frequencies for each marker are entered via the parameter **freq** and are sampled from a beta distribution with **beta_shape1** and **beta_shape2** (default: 1,1; leading to a uniform distribution of allele frequencies. When generating a second cohort of founder animals, one can use the same allele frequencies as used for previously generated individuals by setting the **freq** parameter to "same".
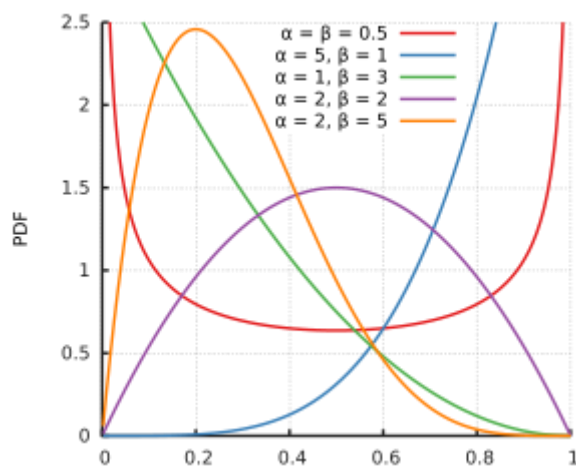
Figure 1: https://en.wikipedia.org/wiki/Beta_distribution

To generate an LD and haplotype structure without using real data we recommend starting with one of the simple datasets and simulate some random/non-random mating generations using *breeding.diploid()* (cf. section 54.5). The function *founder.simulation()* provides an automated way to generate such LD structure for the founder / **dataset**-input. For more details see section 9.7.

If a vcf file is used for data import or a map is provided, the chromosome, marker name, and base pair position are automatically imported. Alternatively, those can be provided via **chr.nr**, **bp** and **snp.name**. By doing this, multiple chromosomes can be inputted jointly. If no input for **chr.nr** is provided all markers are assumed to be on the same chromosome (for more on the usage of a genetic map, we refer to section 4.2). To limit the number of markers imported from a vcf-file set the parameter **vcf.maxsnp** accordingly.

In case more markers are to be added to an existing population list set **add.chromosome** to TRUE and repeat the previous process with the population list as an additional input.

To specify the sex of each sample either assign each individual a probability to be female (**sex.quota**) or alternatively use a vector (**sex.s**) assigning each individual its sex (M=1, F=2).

You can activate a mode in which all individuals are automatically assign to the same sex by setting **one.sex.mode** to TRUE. This will also automatically adapt **breeding.size**, **sex.quota** etc.

On generation, each individual can be assigned into a **class**. Classes allow to later limited breeding actions to specific classes (e.g. only select / phenotype individuals from class 1). On default all individuals will be assigned with class 0 on generation with all breeding actions affecting class 0 default. Culled animals will switch into class -1. More on this in section 5.

In addition to classes, individuals can also be assigned to genetic pools via the parameter **founder.pool**. Defining founder pools allow for the backtracking of which area of the genome stems from which founding population, which can be of interested in cross breeding. Additional QTL effects can be design to only be active for allele that stem from a specific founder pool and by that model breed-specific QTL effects.

4.2   Importing a genetic map

The user can provide a genetic map of up to five columns via the parameter **map**. The first column contains the chromosome of the respective marker, the second column the name of the marker, the third column the position in Morgan, the fourth column the physical position in base pairs, and the last column the allele frequency in the population. All values not provided are automatically set to NA and values are used as input for **chr.nr**, **snp.name**, **bp**, **snp.position**, **freq**.

Alternatively, maps can be imported via *ensembl.map()*. For more on this, we refer to section 14.

4.3   Simulating/Generating the genetic architecture underlying each trait

As the manual input of effects can be tiring, we provide some automated procedures to simulate some common effect structures (additive, dominant, qualitative, and quantitative epistasis) – if you do not need more, you can just skip to section 4.3.2). Technical details on the generation of correlated traits are given in section 15.

Note that this is the generation of an actual genetic value that is underlying each individual of the population. In reality, you cannot observe this, as traits will be caused by far more complex interactions, and effects are not known. This, on the other hand, enables opportunities to evaluate a model fit given a known structure (e.g. GWAS hits can be compared to actual effect markers instead of previously identified markers or similar). Traits can be named via the parameter **trait.name**. To obtain a given mean and genomic variance for a trait one can set the parameters **mean.target** and **var.target**. To make sure that the effect of the 0 variant is zero one can set **set.zero** to TRUE. This

scaling is performed via use of the function *bv.standardization()* on the resulting population list after generating the trait (section 9.10).

4.3.1 Custom-made genetic architectures

To simulate a custom-made genetic architecture we allow for effects caused by one (**real.bv.add**), two (**real.bv.mult**) or more SNPs (**real.bv.dice**). These effects can either be added directly while using *creating.diploid()* or added later using *creating.trait()*. To delete previously existing effects set **replace.real.bv** to TRUE. For multiple traits use lists as inputs for all parameters in this section with each list element containing information for one trait.

Input structure for the first two is a matrix with each row coding a single effect:

```
> real.bv.add
     SNP chromosome Effect 0 effect 1 effect 2
[1,] 120           1       -1.0      0.0       1
[2,] 42            5        0.0      0.0       2
[3,] 17            22       0.1      0.1       0
```

real.bv.add should be able to model any additive or dominance effects of single markers.

```
> real.bv.mult
     First SNP First chromosome Second SNP Second chromosome effect 00 effect 01 effect 02 effect 10 effect 11 effect 12 effect 20 effect 21 effect 22
[1,] 144       1                145        1                 1.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
[2,] 6         3                188        5                 0.37      0.16      1.33      1.49      1.58      2.51      0.38      2.12      0.98
[3,] 5         17               1          10                1.18      2.60      0.18      1.74      0.69      1.39      -1.21     0.96      1.94
```

real.bv.mult should be able to model any epistatic interaction between two markers.

To simulate even more complex effect structures use the parameter **real.bv.dice** allowing the modelling of effects caused by more than two SNPs.

Input for **real.bv.dice** is a list containing a list of all locations and a list of all effects:

```
> real.bv.dice
$location
$location[[1]]
     SNP chromosome
[1,] 11            1
[2,] 12            1
[3,] 16            4

$location[[2]]
     SNP chromosome
[1,] 14            2
[2,] 77            6
[3,] 15            9


$effects
$effects[[1]]
 [1]  1.8212212  1.5939013  1.9189774  1.7821363  1.0745650 -0.9893517  1.6198257  0.9438713  0.8442045 -0.4707524  0.5218499  1.4179416  2.3586796
[14]  0.8972123  1.3876716  0.9461950 -0.3770596  0.5850054  0.6057100  0.9406866  2.1000254  1.7631757  0.8354764  0.7466383  1.6969634  1.5566632
[27]  0.3112443

$effects[[2]]
 [1]  0.29250484  1.36458196  1.76853292  0.88765379  1.88110773  1.39810588  0.38797361  1.34111969 -0.12936310  2.43302370  2.98039990  0.63277852
[13] -0.04413463  1.56971963  0.86494540  3.40161776  0.96076000  1.68973936  1.02800216  0.25672679  1.18879230 -0.80495863  2.46555486  1.15325334
[25]  3.17261167  1.47550953  0.29005357
```

Each network of interacting markers is giving in the first list (location) and their effects are given in a second list (effects). Effects are sorted in the following order: 0...0, 0...01, ... 2...2 – resulting in a vector with 3^n elements, where n is the number of markers involved in the effect.

Marker effects assign to positions that currently do not exist (e.g. SNP 100 on chromosome 1 in case chromosome 1 only contains 50 SNPs) are automatically removed from the stored effects unless **remove.invalid.qtl** is set to FALSE.

4.3.2 Predefined genetic architectures

In the case of a predefined genetic architecture, all markers are assigned with the same probability to be drawn as effect markers. To exclude markers use a parameter **exclude.snps** containing a vector of all numeric positions of excluded markers. Numbering is consecutively starting with chromosome 1. Note that only markers, that are already included in the dataset, can be chosen as effect markers – so in case of more than one chromosome with no generation via **chr.nr** the effects should be added using *creating.trait()* (cf. section 9.99.3) or in the last run of *creating.diploid()*.

The number of additive (**n.additive**) and dominant (**n.dominant**) QTL as well as effects caused by qualitative (**n.qualitative**) and quantitative (**n.quantitative**) epistasis can be included directly. On default, each variance is set to 1 and effects are drawn from a Gaussian distribution. To assign the variance, one can input a vector containing the variance for each effect (**var.additive, var.dominant,…**). To generate additive or dominant QTLs which all have the same effect size use **n.equal.additive** and **n.equal.dominant**. The size of those effect can be chosen via **effect.size.equal.add** and **effect.size.equal.dom** with the default being 1. To make sure that all dominant effects are positive (e.g. to have positive effects of hybrid breeding) set the parameter dominant.only.positive to TRUE.

Qualitative epistasis is simulated by drawing 3 random effects for both involved markers, taking the absolute values from those, sorting them from low (0) to high (2), and multiplying those effects with each other. By this, we obtain the lowest effect for 00 and the highest for 22 with selection for the alternative allele to be beneficial in all cases.

Quantitative epistasis is simulated by drawing 9 random effects and assigning the absolute values of two of those to the corner 02 and 20. All other combinations are assigned the minus absolute values of the drawn random number.

To simulate more than one trait use vectors for **n.**XXX and lists for **var.**XXX instead.

4.3.3 Correlated Traits

To generate correlated quantitative traits selected the traits that should be correlated via **shuffle.traits** and provide the needed correlation in **shuffle.cor**. Note that QTLs are then assigned to the same markers to get correlations independent of the underlying LD structure. To set a correlation for traits with no underlying QTL, use **new.breeding.correlation**. As the simulation is done via sampling from a Gaussian distribution and genetic traits do not fulfill all requirements of a dependent multivariate Gaussian distribution (which is here used to model dependency), the obtained resulting correlations can be different from the correlation set in **new.breeding.correlation** if non-QTL traits have correlations with QTL-traits. We are currently working on alternatives for this.

4.3.4 Non-gaussian distributed traits

All underlying genomic values generated via methods given in section 4.3.2 are using Gaussian effect size. Especially for discrete distributions it is potentially beneficial to apply a transformation function on the generated phenotypes. This can be done by the use of the function creating.phenotypic.transform() – for more on that, we refer to section 15. E.g. to generate a binary trait use:

```
function(x){y = x>10; return(y)}
```

To transform all phenotypic observations higher than 10 to 1 and all below to 0. Thresholds to obtain certain probabilities of each phenotype must be manually derived according to the trait mean and variance set via bv.standardization() and the entered heritability.

4.3.5 Maternal / paternal effects

In both *creating.diploid()* and *creating.trait()* one can indicate if a trait is caused by maternal or paternal effects via the parameter **is.maternal** and **is.paternal**. Setting this to TRUE will lead to the genotype of the respective parent to be used in the evaluation of QTL effects. For founder individuals, the genotype of the individual itself will be used.

4.3.6 Traits as linear combination of other traits

The function *add.combi()* can be used add traits with underlying true genomic value equal to the combination of other traits. The trait to be considered can be selected via the parameter **trait** and the weightings can be chosen via **combi.weights**. The trait can be assigned with a name by the use of the **trait.name** parameter.

Note that phenotyping for such a trait is independent of phenotyping and heritablity for each of the other traits that is combined. Typically, only phenotyping of either the combined or the singular traits will be required. For an example on the use, we refer to section 6.10.

4.3.7 Fixed effects

Fixed effects can be added via the *add.fixed.effects()* function. Fixed effects are provided via the **fixed.effects** parameter as a p x k matrix with p being the number of traits and k being the number of fixed effects. To remove already existing fixed effects set replace to TRUE – otherwise newly added fixed effects will be added to the list of effects. Whenever new individuals are generated the realization of fixed effects can subquently be chosen via **fixed.effects.p** in *breeding.diploid().* For more on this see Section XX.

4.3.8 Breed-specific traits & Crossbreeding

On default, QTLs are evaluated on the raw genotype of the individual. To assign effect specific to a certain subpopulation set the parameter **trait.pool**. This requires setting up founder pools in *creating.diploid().*

4.4 Position of Markers

For our simulations, the physical position in base pairs does not really matter as we are interested in a position in Morgan internally. We assume points of recombination to be distributed according to a Poisson distribution. On default, we assume markers to be equidistant with the total length of the chromosome in Morgan given by the parameter **chromosome.length** (default: 5). When performing a joint generation of multiple chromosomes with different sizes enter a vector instead. If non of the

following options for the position of each marker are provided, markers are assumed to be equidistant. This will also minimize computation time and therefore should only be changed if needed.

Based on the physical positions entered in the parameter **bp** the position in Morgan can be derived by providing a conversion rate in **bpcm.conversion**. Note that tools like BEAGLE (Browning et al. 2018) assume 100.000.000 bp per Morgan. For chicken, we would recommend the use of 30.000.000 bp per Morgan.

Another way of entering the genetic position (in M) is via the parameter **snp.position** manually. Scaling can be performed internally by activating the parameter **position.scaling** and the **chromosome.length**. In addition, one should input a value for the number of base pairs before and after the last position (**length.before**, **length.behind**). Both should be chosen to be larger zero (default: 5) if scaling is performed.

For some applications, the recombination rate might not be the same for all individuals (e.g. male/female differences). To input an additional recombination map enter alternative positions in the parameter **add.architecture**. You can select which architecture is used for every parent in the actual simulation process via **gen.architecture.m** & **gen.architecture.f** in *breeding.diploid()*.

4.5   Related founder individuals

On default, pairwise founder kinships are zero as no pedigree information for previous generations can be entered. To manually enter a kinship matrix for founders of the population list the function *add.founder.kinship()* can be used. The kinship matrix can be provided via the parameter **founder.kinship**. Alternatively, on can also directly derive a kinship matrix according to (VanRaden 2008) by setting **founder.kinship** to "vanRaden". On default, the founder matrix is entered for generation 1. If a kinship matrix for another generation is supposed to be entered use the parameter **gen**.

4.6   Add genotyping arrays

As different genotyping arrays are typically used for different individuals, it is possible to add specific genotyping arrays via *add.array()* with the parameter **marker.included** providing binary (TRUE/FALSE) information if the respective SNP is on the array. On default, one genotyping array that is containing all markers is already added by using *creating.diploid().* In each subsequent step to generate genotypes the used array can be selected. For more on this, we refer to section 5.2.

4.7   Size.scaling

For testing, it is often time helpful to consider smaller individual numbers for fast simulations. This typically requires changing various parameters within a simulation script. Use of the parameter **size.scaling** in the first generation of individuals will automatically scale down all individual number by this factor. This will not work for complex simulations but can be helpful for relatively easy simulations.

5 Simulation of breeding processes (*breeding.diploid()*)

To perform the actual simulation of matings the function *breeding.diploid()* is used. Especially for that step, the sheer amount of different options can be deterring. In reality, only a few parameters will actually be relevant. In this section, we will first discuss absolutely necessary parameters, their default options and afterwards discuss possible deviations. For exemplary simulation, we refer to section 6. Note that you most likely can skip through some of the sections if you are not interested in changes in that dimension. There will be a lot of cases where there is the same parameter for the male and female part of the breeding program. We will limit ourselves here to the male parameter (parametername**.m**) – usage of the female version (parametername**.f**) will always be the same. The default setting for the female parameter is to be the same as the male parameter (<u>NOT</u> the default of the male parameter). An exception to this is when groups of individuals are selected to be used as parents and similar (e.g. **selection.f.cohorts**).

5.1 General setup

The output of *breeding.diploid()* is an updated population list. All newly generated individuals are added as an additional generation – to add them to a previously existing generation set **add.gen** to the generation you want to add to.

The number of newly generated individuals can be chosen via **breeding.size**. Input for this is a numeric value and sex of each offspring is randomly determined via **breeding.sex** (probability of a male offspring). To remove randomness set **breeding.sex.random** to FALSE or input a vector containing the number of new male/female individuals in **breeding.size** instead. To limit the number of litters to generate, the parameter **breeding.size.litter** can be used which will then lead to a stochastic size of the breeding population.

To control which individuals are used in the mating procedure use the parameter **selection.size** (vector of size 2, containing the number of used male/female individuals). In case **selection.size** is not provided it will automatically be set to use all available individuals. By default, only the individuals of the last generation and class 0 (this is usually all and you will realize when this is not the case) are used. Classes can be used to model migration to store groups of different genetic origin but same sex and generation or to just remove individuals from the pool of individuals considered for selection.

To control which individuals are used every cohort generated can be named via **name.cohort** in *breeding.diploid()* & *creating.diploid()*. The individuals used in the selection procedure can be chosen via **selection.m.gen**, **selection.m.database** and **selection.m.cohorts** (paternal side) and the same syntax for the maternal side. In the old version of the code, this is equivalent to the use **of best1.from.cohorts** and **best1.from.groups** that are still alternative input parameters for **selection.m.database** and **selection.m.cohorts**.

To combine individuals to a new cohort of individuals set **combine** to TRUE. This will generate a new cohort of all selected individuals - do not combine male and females individuals! To generate an exact copy (including genotyping/phenotyping information) of all selected male individuals set **copy.individual.m** to TRUE. An example of the use of **copy.individual.m** is provided in section 6.5. The use of **copy.individual** provides slightly more flexibility as it will just generate a copy of the paternal parent of a mating. As a side note: a hack to get the same functionality as **copy.individual.m** would be to set **selfing.mating** to TRUE, **selfing.sex** to 0, **max.offspring** to 1 and **breeding.size** to **selection.size**.

On default, the selection of individuals is done at random (For more on this we refer to 3.4).

To select the individuals of which class to consider in the selection procedure, use the parameter **class.m**, containing a vector of all usable classes. To control the class of the new individuals is assigned to, use the parameter **new.class** (default 0). Classes of all cohorts added are automatically added to the vector of considered classes (**class.m**) unless **add.class.cohorts** is set to FALSE.

Both the time of the generation of new individuals (**time.point**) and the type of the mating (**creating.type**) can be stored. Both parameters are mostly used internally in the web-based application and are automatically tracked internally.

5.2   Litter size and repeat of matings

To generate multiple offspring from the same dam/sire pair set **repeat.mating** to the desired number. In case offspring are generated via copying existing individuals use **repeat.mating.copy**. To use flexible litter size, you can provide a matrix (two columns: first: litter size, second: the probability of that litter size) instead of a numeric. The last value entered as the litter size will be saved and uses as the new default for all subsequent generations of offspring. To manually provide how often a mating is repeated / what size each litter has use repeat.mating.s.

The litter size be related to a trait. To do this set **repeat.mating** to "genetic" and provide which trait should be liked to the litter size via **repeat.mating.trait**. For computational reasons please provide **repeat.mating.max** with an upper boundary on the number of matings from a specific mating. Please make sure that the litter size will only have discrete measurements. This can for example be obtained by implementing a phenotypic transformation function. The size of each litter will be a phenotypic observation for the second / female parent of the mating.

5.3   Control of heritability, breeding values, genotypes and phenotypes

For each individual, an underlying true genetic value is calculated for each trait. Based on this, phenotypes can be generated. For which individuals to generate new phenotypes can be controlled via **phenotyping.gen**, **phenotyping.database** and **phenotyping.cohorts**. For a quick input of all individuals previously not phenotyped set **phenotyping** to "all", "non_obs", "non_obs_m" or "non_obs_f" for all or all previously not phenotyped individuals (potential of only one sex). To only generate phenotypes for a share of the selected gen/database/control use **share.phenotyped**. By default, for each individual at most one phenotype is generate. Set **multiple.observation** to TRUE to allow for more than one observation per individual. To generate multiple observations in a single run of *breeding.diploid()* set **n.observation** to that number. Observations are handled as repeated measurements with the **repeatability** controlling by how much the residual variance of the trait is reduced. In case more observations are generated, prior phenotypic observations are saved and explicitly <u>not</u> deleted. To model a correlation between the environmental variances for different traits set provide the desired correlation matrix via **new.residual.correlation** (this can also be done in *creating.diploid()*). For the simulation of correlated genetic values, we refer to section 4.3.3.

The environmental variance σ_e^2 can be controlled by the using of setting are target **heritability** and providing a list of individuals to calculate the genetic variance from via **sigma.e.gen**, **sigma.e.database** and **sigma.e.cohort**. Note that on default the genetic variance is calculated without any correction for

population structure set **variance.correction** to "parental.mean" or "generation.mean". Alternatively, it is also provide to provide a target value for the environmental starndard deviation in **sigma.e**. A manual change of the genetic variance **sigma.g** is not recommended but in principle possible (this will only affect the breeding value estimation not the genomic values themselves – to change the overall genetic variance use *bv.standardization()*). On default, it is estimated using all individuals used in the current breeding value estimation (set **forecast.sigma.g** to FALSE to deactivate).

For newly created individuals the phenotype is set to 0. Alternatively, one can it to the mean of the parents or create an observation by setting **new.bv.child** to "mean" or "obs" instead of "zero". In case of generating individuals via **copy.individual** one can also use "addobs" to import existing observations but also potentially generate additional ones via **n.observation**. Estimated breeding values are also kept unless **copy.individual.keep.bve** is set to FALSE.

To select the share of individuals genotyped use the parameter **share.genotyped** or select it manually via **genotyped.s** (in concordance to **sex.s** in *creating.diploid()*). To add additional genotypic data after generation the parameter **genotyped.gen**, **genotyped.database**, **genotyped.cohorts** can be used to set the share of genotyped individual to **genotyped.share**. In case an individual is generated via **copy.individual** the genotyping state is kept and the share of previously not genotyped individuals that are now genotyped can be controlled via **added.genotyped**. In case multiple genotyping arrays were generated (section 4.6), the array used for genotyping can be selected via **genotyped.array** with the default being an array containing all SNPs. Default is that individuals are not genotyped.

To remove the status of an individual to be genotyped use **genotyped.remove.gen**, **genotyped.remove.database**, **genotyped.remove.cohorts**. This is mostly suggested in case genotyped in done by accident. To only remove the genotyping state of a specific copy of an individual set **genotyped.remove.all.copy** to FALSE.

In some applications, the genetic value of the individual itself is not of importance and instead the performance of its offspring is of relevancy. To select for which individuals to import offspring phenotypes use **offpheno.parent.gen**, **offpheno.parent.database** and **offpheno.parent.cohorts**. Unless specified in **offpheno.offspring.gen**, **offpheno.offspring.database** and **offpheno.offspring.cohorts** all offspring are considered here.

For better comparison of and between breeding values it is possible to standardizing breeding values before the first generation by activating **standardize.bv**. By this the average breeding value is set to **standardize.bv.level** (default: 100) – for the calculation of this, the average of the individuals in generation **standardize.bv.gen** (default: 1) is used.

In case fixed effects are used in the simulation, one can choose the realisation of the fixed effects via the **fixed.effects.p**. In case multiple different realisations are possible it is also possible to provide a matrix as input with each row containing one realisation and probabilities of each realisation given in **fixed.effects.freq**.

Scaling in case of index selection with multiple traits is performed in the selection process itself.

5.4 Breeding value estimation

To perform selection one can perform breeding value estimation. To activate this set **bve** to TRUE. In the simplest case, one has to input which groups to use in the breeding value estimation via the parameters **bve.gen**, **bve.database** and **bve.cohorts**.

MoBPS will automatically collect information from all individuals if they are genotyped and/or phenotyped and perform the breeding value estimation accordingly. Different software can be used for the breeding value estimation.

1. Mixed Model via MoBPS-solver that assumed known heritability and direct solving of the mixed model without REML variance component estimation
2. Mixed Model via MiXBLUP
3. Bayesian approaches implemented in BGLR (Pérez und de los Campos 2014)
4. Mixed Model via EMMREML (Akdemir und Okeke 2015)
5. Mixed Model via sommer (Covarrubias-Pazaran 2016)
6. Mixed Model via rrBLUP (Endelman 2011)
7. Marker assisted selection
8. Parent/Grandparent mean
9. Own function

In any case estimated breeding values are entered for all individuals unless **bve.insert.gen**, **bve.insert.database** or **bve.insert.cohorts** directly classifies for which groups breeding values are to be entered. The accuracy of the breeding value estimation is automatically reported unless **report.accuracy** is set to FALSE. In case a breeding value estimation should only be performed for some of the traits the parameter **bve.ignore.traits** can be used to exclude selected traits.

On default, genomic information of all previously genotyped individuals is used. If different genotyping arrays are used, it is assumed that all individuals are imputed to the density of the individual with the highest number of genotyped markers. To include imputing errors use bve.**imputation.errorrate**. To only use markers availabe in all individuals set **bve.imputation** to FALSE. Alternatively, the markers used in the BVE can be selected via the **bve.array** parameter to select a set of markers used that was previously defined with *add.array().*

Fixed effects on default are estimated in rrBLUP, sommer and BGLR. In the direct estimation they are assumed to be known. To exclude them from the prediction model use the parameter **bve.exclude.fixed.effects**.

For the calculation of G we offer multiple methods with **relationship.matrix**="vanRaden" being the default (VanRaden 2008). Alternatives include "kinship", "CM", "CE" (Martini et al. 2017) and the non-Z-standardized version of the vanRaden method ("non_stand"). Unless "kinship" is used, all individuals with available genotypes will be used as genotyped individuals. When "kinship" is used a pedigree based evaluation is performed, even when individuals are genotyped. In case "kinship" is selected the depth of the pedigree has to be provided via the parameter **depth.pedigree** (default: 7). Note that these individuals are just used to calculate the A matrix. Individuals used in the actual breeding value estimation still need to be selected via **bve.gen**, **bve.database** and **bve.cohorts**. Individuals with no observed phenotype start with a value of 0. Internally all phenotypes that are exactly 0 are handled as an NA – suppress this by setting **bve.0isNA** to FALSE (note that only methods 1./2./4. are able to handle NAs in the data). To deactive the use of the single-step relationship matrix set **singlestep.active** to FALSE – this way non-genotyped individuals are not considered in the breeding value estimation. Similar can be done by the use of **remove.non.genotyped** (Legarra et al. 2014).

On default, the phenotypes of the individuals themselves ("own") will be used in the breeding value estimation. Alternatively, the average offspring phenotype ("off") or an average between own and offspring phenotype ("mean", "weighted") can be used and providing it in the input parameter **bve.input.phenotype**. To calculate the mean offspring performance, provide the cohorts to calculate

this for in **offpheno.parents.gen/database/cohorts**. This will automatically analyze all potential offspring – to save computing time, one can directly provide a list of the individuals to consider as offspring in **offpheno.offspring.gen/database/cohorts**.

In case a high number of individuals is used in the breeding value estimation, exact solving of the mixed model equations can be computationally challenging. Instead of the direct inversion, it is also possible to use a solver. MoBPS includes a preconditioned conjugate gradident (PCG)- solver (Fletcher und Reeves 1964) from the R-package cPCG (Zhuang 2019) that is using a Jacobi preconditioner (**bve.solve** = "pcg"). One can also enter a function to replace solve() / chol2inv(chol()) in the BVE in **bve.solve**.

To not perform statistical breeding value estimation but instead using the phenotypes as breeding value estimates set **phenotype.bv** to TRUE.

As the presence of true effect markers in the dataset might be a strong assumption one can set **remove.effect.position** to not use markers associated with any traits in the breeding value estimation.

To only included individuals with a certain class set **bve.class** to a vector containing all classes to consider.

In case individuals were generated via **copy.individual** (this is especially relevant for the web-based application) each individual is only used at most once. To consider the same individual multiple times set **bve.avoid.duplicates** to FALSE. Note that cloning will not lead to the same ID.

5.4.1 Direct approach with known heritability

Main advantage of a direct estimation is a massive improvement in computation time as the usually necessary REML estimation takes most of the time. In practice, it might not be realistic but since genetic values are known it is possible. Especially for bigger populations heritability estimation should not be problematic and is not performed in each breeding value estimation in practice as well. Note that this is still an empirical measure that can change when using different individuals in the estimation process with no correction for population structure. To correct for generation or parental average use the parameter **variance.correction**. To instead estimate the additive genetic variance using a parental model activate **estimate.add.gen.var**. Or provide estimates for for the residual and genetic variance via **sigma.e** and **sigma.g**.

In case of missing phenotypes, estimates will be based on (VanRaden 2008) method 2. This will also be used in case of the use of single step. Alternatively, one can use rrBLUP based estimates by setting **bve.direct.est** to FALSE. Note that this second version is slower and requires the presence of individuals that are phenotyped and genotyped.

In case multiple observations are generated for some individuals, the residual variance in the mixed model will automatically be adapted accordingly. To deactivate this feature set **bve.per.sample.sigma.e** to FALSE.

5.4.2 MiXBLUP

This functionally is only available when the user has a licsence for the commercial software MiXBLUP (https://www.mixblup.eu/index.html) and can be activated by setting mixblup.bve to TRUE. The path of the MiXBLUP.EXE has to be provided in **mixblup.path** which will can be directly called from within the MoBPS simulations.

All required files for MiXBLUP will automatically be generated in the backend in the path provided in **mixblup.files**. One can also manually provide input files for MiXBLUP via **mixblup.path.pedfile/parfile/datafile/inputfile/genofile** or deactivate writting of those files via **mixblup.pedfile/parfile/datafile/inputfile/genofile**. Additional input parameters in MiXBLUP can be provided via **mixblup.apy, mixblup.apy.core, mixblup.alpha, mixblup.beta, mixblup.omega**.

5.4.3 Bayesian approaches (BGLR)

For performing Bayesian methods we are using the R-package BGLR. To activate the usage of BGLR set **BGLR.bve** to TRUE. On default, a Reproducing-kernel-hilbert-space ("RKSH") is used – alternatively one can use BayesA, BayesB, BayesC by setting **BGLR.model** to "BayesA", "BayesB", "BayesC", "BRR", or "BL". Other parameter values will all be chosen according to the defaults of the BGLR package. If you want to test alternative parameter settings or use other methods implemented in BGLR either do the estimation manually (Chapter 5.4.9) or contact me to add it to the package.

To control the number of the burn-in and iterations use **BGLR.burnin** and **BGLR.iteration**. To deactivate the printing of results of the interim steps set **BGLR.print** to FALSE (equal to verbose=FALSE in BGLR). On default, BGLR will generate some internal files in its computations. To select a path of where to store them chose it via **BGLR.save**. Especially when parallelizing thousands of simulations BGLR can crash when the same path is used multiple times (So be warned when running 100+ simulations in parallel). Activating **BGLR.save.random** will hinder this.

5.4.4 GBLUP (EMMREML)

Traditional GBLUP including variance component estimation using REML is performed by using the package EMMREML. To activate the usage set **emmreml.bve** to TRUE. EMMREML does not support missing phenotypes and therefore can only be used if phenotypes for all individuals in the BVE are available (if not use the direct approach, sommer or rrBLUP).

5.4.5 GBLUP (sommer)

Traditional GBLUP including variance component estimation using REML is performed by using the package sommer. To activate the usage set **sommer.bve** to TRUE. Sommer does support missing phenotypes.

To activate the use of the multi-trait model implemented in sommer use **sommer.multi.bve** to TRUE. Note that this will take substantially longer than single trait models.

5.4.6 GBLUP (rrBLUP)

Traditional GBLUP including variance component estimation using REML is performed by using the package rrBLUP. To activate the usage set **rrBLUP.bve** to TRUE. rrBLUP is about 2.5 times as fast as sommer for breeding value estimation.

5.4.7 Marker-assisted selection

Marker-assisted selection can be activated by setting **mas.bve** to TRUE. The markers used and their effects can be provided via **mas.markers** and **mas.effects**. If **mas.effects** is not provided the marker effects will be estimated using a simple linear regression. In case **mas.markers** are not provided the **mas.number** (default = 5) markers with the highest single marker effects will be chosen as the markers to use.

5.4.8 Parent/Grandparent mean

To use the mean performance of the parents / grandparents as the breeding value use **bve.parent.mean** / **bve.grandparent.mean**. On default breeding value estimates for the parents are used and if those are not available phenotypes. Alternatively on can select to use breeding values only ("bve"), phenotypes only ("pheno"), or genomic values ("bv") via the parameter **bve.mean.between**.

5.4.9 Own function

Instead of performing breeding value estimation inside of *breeding.diploid()* one can implement his own methodology by exporting all information needed to those computations and inserting own breeding values estimates via the function *insert.bve().*

According code could look like this:

```
# Simulate Phenotypes for generation 4 with heritability 0.4
population <- breeding.diploid(population, heritability = 0.4,
                              sigma.e.gen = 4,
                              phenotyping.gen = 4)
# Export genotypes and phenotypes for generation 4
genos <- get.geno(population, gen = 4)
phenos <- get.pheno(population, gen = 4)

# Here you perform your own method to assign breeding values to each individual
bve <- runif(ncol(genos)) # This is probably not the best technique for this =)

# Import breeding values estimated for generation 4
bves <- cbind(colnames(genos), bve)
population <- insert.bve(population, bves=bves)
```

For details on exporting functions, we refer to section 0. For details on importing function, we refer to section 8.

5.4.10 Calculating marker effects & GWAS

For some applications (e.g. gene editing) it is necessary to identify causal markers. Although marker effects are known in a simulation, in practice one has to identify them. Options here are either a direct calculation of the effect size of each marker based on the computations performed in 5.4.1 (rrBLUP) or the performance of a GWAS-study without correction for population structure. Methods can be activated by setting **estimate.u** or **gwas.u** to TRUE.

In case of a GWAS study one can additionally select the groups used in the study by setting **gwas.gen**, **gwas.database** and **gwas.cohorts** (default is to use the same groups as for the breeding value estimation). As a value for y, one can use the phenotype ("pheno"), true breeding value ("bv") or the

estimated breeding value ("bve"). Additionally, it might be necessary to standardize the y value by the mean of the group by activating **gwas.group.standard**. Note that this is a basic implementation of GWAS with no correction for population structure or similar.

5.4.11   Calculation of reliabilities

Reliabilities are not derived in any of the used R-packages. In the direct approach (Chapter 5.4.1), they can be derived by setting **calculate.reliability** to TRUE according to (VanRaden 2008).

5.5   Selection techniques & mating strategies

Selection of the individuals for matings in the following generations is of key importance for any breeding program. Especially here, one is limited to the techniques that work in the species one wants to simulate. On default settings, the selection of the new founders is done at random. To use estimated breeding values as the selection criterion set **selection.m** to "function". To ignore the best selected individuals set **ignore.best** to that value – note that this value will be internally subtracted from **selection.size**. E.g. to simulate mating between the top 100 female individuals with the third and fourth best male individual set **selection.size** = c(4,100) and **ignore.best**=c(2,0). To exclude specific individuals from the set of individuals to select from using **reduced.selection.panel.m**. The vector should contain all individuals to use (e.g. 1:10 when selecting from the first ten individuals).

To store details on which individuals were selected, which matings were performed and the currently estimated breeding values activate **store.breeding.totals**.

Selection can be performed based on the phenotype, genetic value or the breeding value estimates. To select what to use set **selection.criteria** to "pheno", "bv", "bve" or "offpheno" (default: "bve"). To select those individuals with the lowest breeding value set **selection.highest** to FALSE. To randomly select individuals set **selection.criteria** to "random". All individuals will then be selected with the same chance unless a specific probability per individual is provided in **selection.m.random.prob**.

To export a list of all selected individuals set **export.selected** to TRUE. The first list contains the database position of all selected males and the second list the database position of all selected females.

5.5.1 Multiple traits

When working with multiple traits, the selection of the best individuals is typically done by the use of a combination of those traits. All single values can either be added up directly (**multiple.bve**="add") or one can use a selection index just accounting for the ranking (**multiple.bve**="ranking"). To reduce scaling problems for different traits one can use **multiple.bve.scale.m** to standardize the variance in each trait. Note that this scaling in the cohort mode is for all individuals together whereas in the old selection modes it is done per group (! – needs to be the same!!!). Additionally, each trait can be assigned a weighting via **multiple.bve.weights.m**.

To derive the ideal index based on phenotypic/genotypic variance, reliabilities and economic gains per unit according to (Miesenberger 1997) set **selection.m.miesenberger** to TRUE. Economic gains can be provided in **multiple.bve.weights.m**. On default, the gain has to be provided per standard deviation of the breeding value estimations. Alternatives can be provided in **multiple.bve.scale.m** (default: "bve_sd") and are per unit ("unit") and per phenotypic standard deviation ("pheno_sd").

In case reliabilities are not derived (Chapter 5.4.11), they need to be estimated. On default, this is done by dividing the standard deviation of the breeding value estimation by the standard deviation of the phenotypes. Alternatives can be entered in **selection.miesenberger.reliability.est**, as the direct use of the heritability ("heritability") or the actual calculation according to the correlation between breeding value estimates and true underlying genomic values ("derived") which is of course not possible in practice but should be the most accurate.

5.5.2 Higher procreation of genetically favored individuals

Genetically favored individuals tend to procreate more often. To model this set a ratio between the likelihood of the best individual to mate compared to the worst individual (in the group of selected individuals) in **best.selection.ratio.m**. This parameter describes the ratio of the frequency of use between the best selected individual and the worst selected individual. All other frequencies are then calculated linearly. E.g. in a group of selected individuals with breeding values 105, 103 and 100 with a ratio of 6 the relative frequencies are of 6,4,1 (this just is a linear function – comp for individual 2: (103 − 100)/ (105-100) * (6-1) +1). Criteria behind can be either "bv", "bve" or "pheno" and can be entered in **best.selection.criteria.m**. To manually enter the probability of each individual in the group of selected individuals input a vector with frequencies for each individual in **best.selection.manual.ratio.m**. Individuals selected are sorted with the individual with the highest estimated breeding value beginning the first one. To instead use the order as provided in the database set **best.selection.manual.reorder** to FALSE.

This does not require breeding value estimation and can also be used to simulate slow natural selection processes over thousands of generations.

Higher procreation is also relevant for optimum genetic contribution theory and the use of **ogc** will automatically change these parameters accordingly (section 5.5.10).

5.5.3 Maximum number of offspring per individual & use of same mating pairing

To control the maximum number of times each individual is used for reproduction set **max.offspring**. Either enter a numeric if that boundary is for both sexes or a vector with the first value coding the maximum for male and the second the one for female. Similarly, **max.litter** can be used to limit the number of newly generated litters that are stemming from a given individual.

To avoid the to generate multiple offspring from the same pairing of individuals, the maximum number of matings from a specific combination of sires/dams can be limited via the parameter **max.mating.pair**.

5.5.4 Avoiding full-sibling / half-sibling matings

To not generate offspring from full siblings set **avoid.mating.fullsib** to TRUE. To avoid the use of full or half siblings for matings set **avoid.mating.halfsib** to TRUE.

5.5.5 Selection thresholds

Threshold selection can be performed in two different types of ways. First, the goal can be to only select from the pool of selection candidates but have the ultimate goal to select the number of individuals provided in **selection.size**. This can be done via **threshold.selection.index/value/sign/criteria**. Here, **threshold.selection.index** is the index to consider for the threshold selection (e.g. c(0,0,0,1,0) to apply threshold selection in trait 4). **Threshold.selection.value** contains the value that needs to be exceeded (or equalled / fallen below – depending on the input in **threshold.selection.sign** with values "<", "=", "<=", ">=" being possible). **Threshold.selection.criteria** then provided the selection criteria, which usually will be estimated breeding values ("bve"), but can also be true genomic values ("bv") or phenotypes ("pheno").

To apply a threshold selection on the group of already selected individuals use **threshold.selection** and **threshold.sign**. This will directly be applied on the used selection index and result in potentially less individuals than provided in **selection.size** being selected. Note that this threshold is applied on the selection index in case multiple traits are considered.

5.5.6 Plant breeding (no-sexes & selfing & DH-production & cloning)

For some applications, the sex of an individual is not relevant (or in particular for applications in plant breeding an individual does not even have a sex). Even though a sex is still stored internally, it might be neglectable for the application at hand. In this case, one can allow matings between individuals from the same sex by the usage of **same.sex.activ**. The probability to select a female individual as the parent can be set via **same.sex.sex** (default=0.5). To additionally allow for selfing set **same.sex.selfing** to TRUE. The probability for this mating is the same as any other mating combination.

To perform exclusively selfings, activate **selfing.mating** and selected the probability to use a female parent via **selfing.sex**.

To generate doubled haploid lines active **dh.mating** and selected the probability to use a female parent via **dh.sex**.

To generate an exact copy of an individual to **copy.individual** to TRUE. Instead of simulating the meiosis both chromosome sets of the selected first parent (usually the father) will be copied (to copy female individuals use **same.sex.activ** and set the probability to use females to 1 (**same.sex.sex**=1) or use the **copy.individual.m / copy.individual.f**.

5.5.7 Generate offspring for all sire combination

To generate offspring from each possible parental combination set **breeding.all.combination** to TRUE. In case only individuals from one sex are selected sex is ignored when deriving potential matings. **breeding.size** still has to be set.

5.5.8 Targeted/Fixed mating/Manual selection of individuals

If none of the previously described methods works for your simulation, you can also manually enter a list of all matings that should be performed. For this, use the parameter **fixed.breeding**. To perform

targeted mating in the group of the best individuals use **fixed.breeding.best**. Here each row just contains the sex and position in the list of selected individuals. In both cases, an additional column can be added that is coding the likelihood of the offspring being female.

5.5.9 Gene-Editing

With the increasing popularity of methods like CRISPR/Cas9 one might be interested in performing gene editing to increase the genetic gain. Gene editing can be activated by setting **gene.editing** to TRUE. The number of edits can be controlled via **nr.edits** and effect markers are picked via the usage of the predictions via rrBLUP/GWAS in section 5.4.10. We only count actually performed edits – if an allele is already beneficial, the next best marker is edited instead. Although such a procedure is not possible in practice, the first integrated way of editing is to edit all selected individuals – this technique is also performed in the approach PAGE (Jenko et al. 2015) and our counter version (Simianer et al. 2018).

As a more realistic scenario, we also allow for the editing of offspring via **gene.editing.offspring**. To only perform editing on male or female individuals set **gene.editing.offspring.sex** / **gene.editing.best.sex** to 1 (male) or 2 (female).

Note that traditionally modelled effects often neglected strongly deleterious mutations and we are here assuming that 100% of all edits to work, all possible offspring will survive the procedure and traits are as simple as designed (usually single marker QTL).

5.5.10   Optimum genetic contribution

To use optimum genetic contribution theory (Meuwissen 1997) from the group of selected individuals set **ogc** to TRUE. This implementation used the R-package optiSel (Wellmann 2017, 2019), with parameter options according to the optiSel package.

To select the target of selection use the parameter **ogc.target** (default: "min.sKin"; minimizing average kinship). Constrains can be entered via **ogc.uniform/ub/lb/ub.sKin/lb.BV/ub.BV/eq.BV**. Furthermore, MoBPS is providing the option to set a maximum gain in kinship / minimum gain in genomic value via **ogc.ub.sKin.increase** and **ogc.lb.BV.increase**. Details on the optiSel framework can be accessed by entering browseVignettes('optiSel') in the R-console.

One can also provide on weightings via **best.selection.manual.ratio.m** (Chapter 5.5.2) and/or **fixed.breeding** (Chapter 5.5.8). We are also willing to implemented alternatives if they are needed/wanted.

5.6 Genetic architecture

When simulating meiosis, we are accounting for recombination and mutation. We are assuming that recombination points are Poisson distributed with one expected point of recombination per 1 Morgan. To change this, set **recombination.rate** to the needed value. To not use, a fixed value but a step-function instead use **recom.f.indicator**. This will however only be useful in exceptional cases as the plain use of an accurate genetic map in *creating.diploid()* should be sufficient for basically all use cases. Additional genetic architectures can be added the same way as in *creating.diploid()* via **add.architecture**. To select the genetic architecture of recombination for set **gen.archicture.m** to the architecture that should be used for males.

Regarding mutation rates we are assuming that each marker has the same probability for a mutation – this can be changed via **mutation.rate** (default: 10^{-8}). A mutation back to the reference is assigned the probability of **remutation.rate** (default: 10^{-8}).

Duplications are implemented but the modelling is absolutely adhoc and probably needs refinement – talk to me if you plan to do something in that direction!

5.7   Other

5.7.1 Culling / Death

5.7.1.1   Culling module (Web-interface)

The new culling module is mainly intended for use in the web-interface. Here, parameters settings will take care of themselves and culling actions are automatically executed when a given time point is reached. To manually execute this in R use **culling.gen**, **culling.database** and **culling.cohort** to select for which groups to execute the module. The age of the individual can be provided in **culling.time**. The name of the culling reason can be provided in **culling.name**. Additional one can provide two breeding values (**culling.bv1**, **culling.bv2**) and two culling probabilities (**culling.share1**, **culling.share2**). For all other genomic values the probability of culling is then derived with linear extension.

An index of weighting between traits can be selected via **culling.index** (similar to **multiple.bve.weights.m** – Chapter 5.5.1). On default, no genomic influence is assumed and all individuals are culled with the same probability (**culling.share1**).

On default, the culling module is applied on all copies (generated via **copy.individual**) of the individuals selected. To deactivate this set **culling.all.copy** to FALSE. To not apply the culling module on all individuals of a selected group provide a vector in **culling.single** indicating in TRUE/FALSE if the module is to be applied on this specific individual.

For cohorts the stats on how many individuals have been killing due to each culling reason is also provided in population$info$culling.stats.

5.7.1.2   Old module

Especially for cost calculation it might be necessary to know the time of death for each individual. A group of individuals can be reduced to **reduce.group** with each row coding generation, sex, number of individuals to keep, class. To set the selection criteria use **reduce.group.selection** (default: "random").

5.7.2 Allele-frequency per generation

To store the frequency of each allele per generation activate **store.effect.freq**.

5.7.3 Set a random seed

For repeatability it might be helpful to set a random seed in R. This can be done via the parameter **randomSeed** or directly performed in R using *set.seed().*

5.7.4 save.recombination.history

To store the time of occurrence of each point of recombination activate **save.recombination.history**. This has to be done starting with the first generation and currently crashes after setting a new founder population (Currently nobody needs it! – but should be an easy fix!)

6 Exemplary scripts

6.1 Basic breeding actions

The following basic script is designed to give a brief overview on some basic breeding actions in MoBPS, according to the example given in our G3 Paper (Pook et al. 2020)

```
# Generation of a founder population
# with 100 individuals and 10,000 SNPs
# The genome contains 5 chromosomes with a length of 2 Morgan each
# Generation of one trait with 60 underlying QTL
# of which 50 are purely additive and 10 have a dominate effect
# The genomic variance of the trait simulated to be 1
# The generated cohort is named "Founder"

pop <- creating.diploid(nsnp=10000, nindi=100,
                        chr.nr=5, chromosome.length=2,
                        n.additive=50, n.dominant=10,
                        name.cohort="Founder",
                        var.target = 1)

# Generate phenotypic observations for all individuals
# Residual variance is set to result in a heritablity of 0.5
pop <- breeding.diploid(pop, heritability=0.5,
                        phenotyping="all")

# Perform a breeding value estimation using all individuals
# of generation 1 (which are all).
pop <- breeding.diploid(pop, bve=TRUE,
                        bve.gen = 1)

# Generate 100 offspring
# Use the top 20 male and top 20 female based on their BVE
# All male individuals from the "Founder" cohort are used
# as potential sires.
# All female individuals from the "Founder cohort are used
# as potential dams.
# The resulting cohort in named "Offspring".
pop1 <- breeding.diploid(pop, breeding.size=100,
                         selection.size=c(20,20),
                         selection.criteria = "bve",
                         selection.m.cohorts="Founder_M",
                         selection.f.cohorts="Founder_F",
                         name.cohort="Offspring")

# Same procedure, just with a higher selection intensity
# on the male side.
pop2 <- breeding.diploid(pop, breeding.size=100,
                         selection.size=c(5,20),
                         selection.m="function",
                         selection.m.cohorts="Founder_M",
                         selection.f.cohorts="Founder_F",
                         name.cohort="Offspring")
```

6.2 Averaging results of the basic breeding scheme over multiple simulations

Usually running a simulation just once will not produce reliable results. Running the simulation presented in section 6.1 100 times could for example look like this. Subsequent plots are generated using 10.000 simulations as done in the first MoBPS paper (Pook et al. 2020).

```
library(MoBPS)
gain1 <- gain2 <- inbreeding1 <- inbreeding2 <- numeric(100)
```

```r
for(index in 1:100){
  pop <- creating.diploid(nsnp=10000, nindi=100,
                            chr.nr=5, chromosome.length=2,
                            n.additive=50, n.dominant=10,
                            name.cohort="Founder",
                            var.target = 1)

  pop <- breeding.diploid(pop, heritability=0.5,
                            phenotyping="all")

  pop <- breeding.diploid(pop, bve=TRUE,
                            bve.gen = 1)

  pop1 <- breeding.diploid(pop, breeding.size=100,
                            selection.size=c(20,20),
                            selection.criteria = "bve",
                            selection.m.cohorts="Founder_M",
                            selection.f.cohorts="Founder_F",
                            name.cohort="Offspring")

  pop2 <- breeding.diploid(pop, breeding.size=100,
                            selection.size=c(5,20),
                            selection.m="function",
                            selection.m.cohorts="Founder_M",
                            selection.f.cohorts="Founder_F",
                            name.cohort="Offspring")

  # store the resulting genomic gains per run
  gain1[index] <- mean(get.bv(pop1, gen=2) - mean(get.bv(pop, gen=1)))
  gain2[index] <- mean(get.bv(pop2, gen=2) - mean(get.bv(pop, gen=1)))

  temp1 <- kinship.emp(population=pop1, gen=2)
  temp2 <- kinship.emp(population=pop2, gen=2)

  # Calculate the average of all off-diagonal values (avg. kinship)
  inbreeding1[index] <- mean(temp1-diag(diag(temp1))) * 100/99
  inbreeding2[index] <- mean(temp2-diag(diag(temp2))) * 100/99
}
```

The results could then be plotted like in Figure 1 of the MoBPS G3 Paper:

```r
par(mfrow=c(2,2))
par(mar=c(4.5,1.5,0.5,2))
hist(gain1, xlim=c(0.2,1.65), main="", axes = FALSE, xlab="", breaks = 50)
abline(v=mean(gain1), col="red", lwd=2)
axis(side=4)
axis(side=1, at=seq(0.4,1.6, by=.3))
title(ylab="Scenario 1", line=0, cex.lab=1.5)
hist(inbreeding1, xlim=c(0.004,0.019), main="", axes = FALSE, xlab="", breaks = 25)
abline(v=mean(inbreeding1), col="red", lwd=2)
axis(side=4, ylab="Scenario 1")
axis(side=1, at=seq(0.005,0.018, by=.003))
hist(gain2, xlim=c(0.2,1.65), axes=FALSE, main="", xlab="", breaks = 50)
abline(v=mean(gain2), col="red", lwd=2)
axis(side=4)
axis(side=1, at=seq(0.4,1.6, by=.3))
title(ylab="Scenario 2", line=0, cex.lab=1.5)
title(xlab="Genomic gain", cex.lab=1.5)
hist(inbreeding2, xlim=c(0.004,0.019), axes=FALSE, main="", xlab="", breaks = 40)
abline(v=mean(inbreeding2), col="red", lwd=2)
axis(side=4)
axis(side=1,at=seq(0.005,0.018, by=.003))
title(xlab="Gain in avg. kinship", cex.lab=1.5)
```
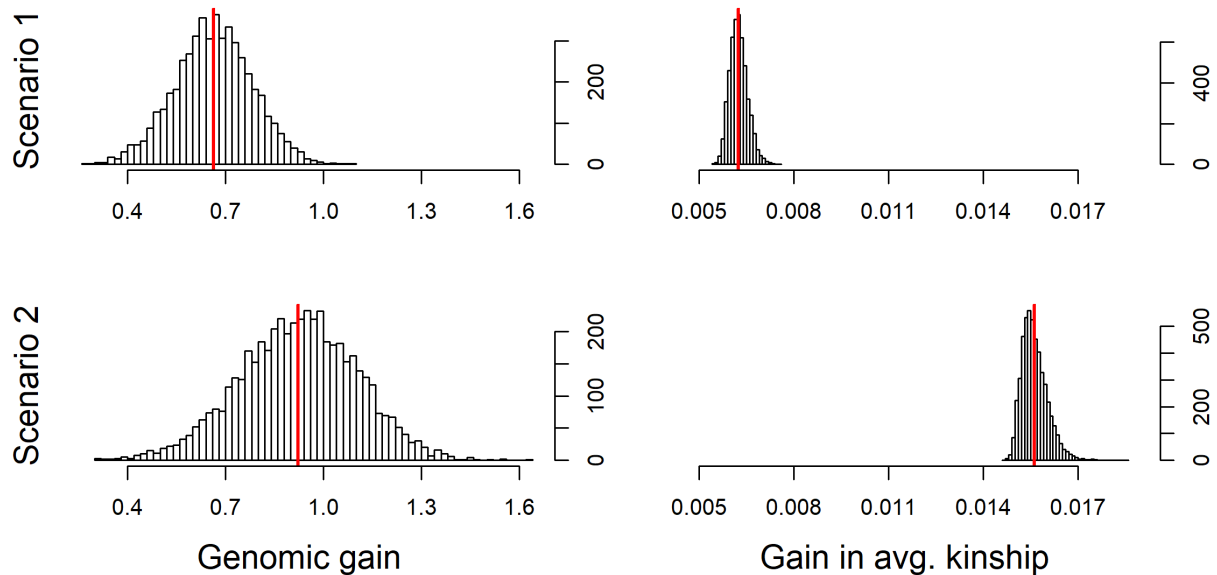
6.3 Generation of a founder population with real genomic data (vcf-file)

```
# Generation of an exemplary VCF-file to load it
# File should contain: 500 SNPs, 30 Individuals
# SNPs are placed on two chromosome
population <- creating.diploid(nsnp = 500, nindi = 30,
                                chr.nr = c(rep(1,250), rep(2, 250)),
                                bp = c(1:250*1000000, 1:250*50000))
get.vcf(population, path="import_test", gen=1)


# Convert provided base-pair position into positions in Morgan
# 1 Morgan = 100.000.000 bp is a common conversion rate
# This can differ between species (e.g. chicken: ~30.000.000 bp)
# or be dependent between telomere / centromere
population <- creating.diploid(vcf="import_test.vcf", bpcm.conversion = 1000000)
summary(population)
# Assume markers to be equidistant and generate a genome of fixed size:
population <- creating.diploid(vcf="import_test.vcf", chromosome.length = c(2,1))
summary(population)
```

6.4 Generation of a founder population with real genomic data (ped/map-file)

```
# Generation of an exemplary Ped/map-files to load it
# File should contain: 500 SNPs, 30 Individuals
# SNPs are placed on two chromosome
population <- creating.diploid(nsnp = 500, nindi = 30,
                                chr.nr = c(rep(1,250), rep(2, 250)),
                                bp = c(1:250*1000000, 1:250*50000))
get.pedmap(population, path="import_test", gen=1)


#
map <- as.matrix(read.table("import_test.map"))
ped <- as.matrix(read.table("import_test.ped"))
# Order in MoBPS map-files is not the same as in regular PedMap
# This will be fixed soon!
map[,3:4] <- map[,4:3]

# Convert PED-file into haplotype dataset (one haplotype per colum)
nsnp <- (ncol(ped)-6)/2
haplo1 <- ped[,1:nsnp*2+6-1]
haplo2 <- ped[,1:nsnp*2+6]
haplo <- t(rbind(haplo1, haplo2)[c(0,nrow(haplo1)) + sort(rep(1:nrow(haplo1),2)),])

population <- creating.diploid(dataset = haplo, map = map,  bpcm.conversion = 1000000)
```

```
summary(population)
# Assume markers to be equidistant and generate a genome of fixed size:
population <- creating.diploid(dataset = haplo, map = map,  chromosome.length = c(2,1))
summary(population)
```

6.5 Genotyping and Phenotyping of Subcohorts

Here we show you to generate genotype and/or phenotype data of selected individuals of a cohort.
The procedure we recommend using here is to generate a new cohort of individuals that is
containing copies of those individuals that are intended to be genotyped and/or phenotyped. In any
breeding value estimation of copies selected to be used in a breeding value estimation are accessed
the genotype information is used as long as at least one copy is genotyped. Phenotype information of
the copy that is phenotyped the most often is used. When using the copy function existing
phenotypic information is also copies but later generated phenotypic information is not
automatically added.

```
# Generate a starting population with 5000 SNPs and 500 male individuals
# 3 Traits with 500 purely additive QTL each
# No genotypes / phenotypes are generate
population <- creating.diploid(nsnp=5000, nindi = 500,
                               n.additive = c(500,500,500),
                               share.genotyped = 0,
                               sex.quota = 0,
                               name.cohort="Founder")

# Generate a copy of those individuals that are supposed to be
# genotyped and/or phenotyped
population <- breeding.diploid(population, selection.size = c(250,0),
                               copy.individual.m = TRUE,
                               selection.m.cohorts = "Founder",
                               name.cohort = "PartlyPhenotyped")

population <- breeding.diploid(population, selection.size = c(100,0),
                               copy.individual.m = TRUE,
                               selection.m.cohorts = "PartlyPhenotyped",
                               name.cohort = "Phenotyped+Genotyped")

# Generate phenotypes for trait 1 & 2. Each trait is assumed to have a heritabilty of 0.5
population <- breeding.diploid(population, phenotyping.cohorts = "PartlyPhenotyped",
                               n.observation = c(1,1,0), heritability = c(0.5,0.5,0.5),
                               sigma.e.cohorts = "Founder")

# Generate genotypes / phenotypes for all traits
population <- breeding.diploid(population, phenotyping.cohorts = "Phenotyped+Genotyped",
                               genotyped.cohorts = "Phenotyped+Genotyped")

# MoBPS default BVE: GBLUP (all non-genotyped individuals are ignored)
# BVE only for the individuals in "Phenotyped+Genotyped"-cohort
# Genotype / Phenotype information from all copies entered in bve.gen/database/cohorts is
collected
population <- breeding.diploid(population, bve=TRUE,
                               bve.cohorts = c("Founder", "PartlyPhenotyped",
                                               "Phenotyped+Genotyped"))

# BVE with pedigree BLUP
# All 500 individuals are used with 250/250/100 providing phenotype information
population <- breeding.diploid(population, bve=TRUE,
                               bve.cohorts = c("Founder", "PartlyPhenotyped",
                                               "Phenotyped+Genotyped"),
                               relationship.matrix = "kinship")

# BVE with single-step GBLUP
# Usually slightly higher prediction accuracy than pedigree BLUP
population <- breeding.diploid(population, bve=TRUE, singlestep.active = TRUE,
                               bve.cohorts = c("Founder", "PartlyPhenotyped",
                                               "Phenotyped+Genotyped"))
```

6.6 Simulation of a base population with a hard sweep

We here show how to perform an exemplary simulation to generate a base population and a hard sweep.

```
# Generate a starting population with 5000 SNPs and 200 individuals
# and a single chromosome of length 2 Morgan.
population <- creating.diploid(nsnp = 5000, nindi = 200, chromosome.length = 2)

# LD build up via 100 generations of random mating
# Each generation contains 200 individuals
for(index in 1:100){
  population <- breeding.diploid(population, breeding.size = 200,
                                 selection.size = c(100,100))
}

# Derive allele frequency and check LD for the last generation:
genotype.check <- get.geno(population, gen = length(population$breeding))
p_i <- rowMeans(genotype.check)/2
ld.decay(population, genotype.dataset = genotype.check, step = 10, max = 500)
```
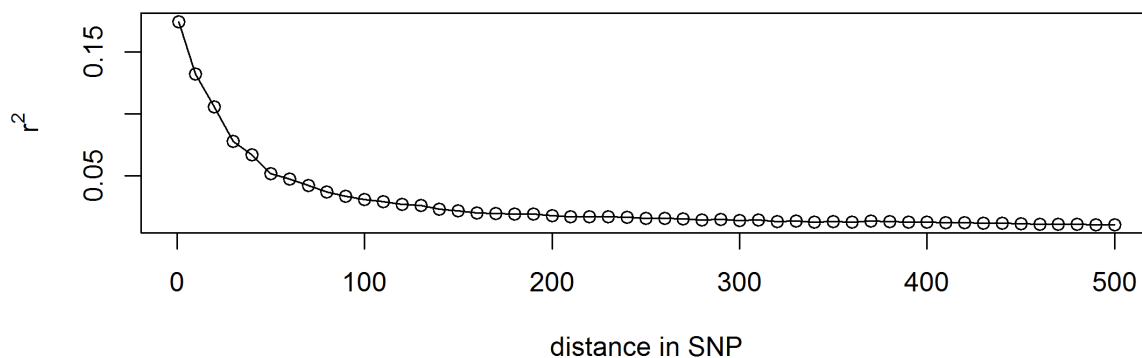


```
# Simulate a favorable mutation in a previously fixed marker
fixated_markers <- which(p_i==0) # Which markers are fixated
qtl_posi <- sample(fixated_markers, 1) # Selected a fixated marker in A
trait <- cbind(qtl_posi, 1, 0, 1, 2) # SNP, Chromosome, Effect AA, Effect AB,
Effect BB
population <- creating.trait(population, real.bv.add = trait)

# Generate a mutation in the first male individual
population <- mutation.intro(population, 101, 1, 1, qtl_posi)

# Simulate generations with selection pressure
# Individuals with the favorable SNP are picked 5 times as often
for(index in 1:25){
  population <- breeding.diploid(population, breeding.size = 200,
                                 selection.size = c(100,100),
                                 best.selection.ratio.m = 5,
                                 best.selection.ratio.f = 5)
}

analyze.population(population, gen = 98:115, chromosome = 1, snp = qtl_posi)
```
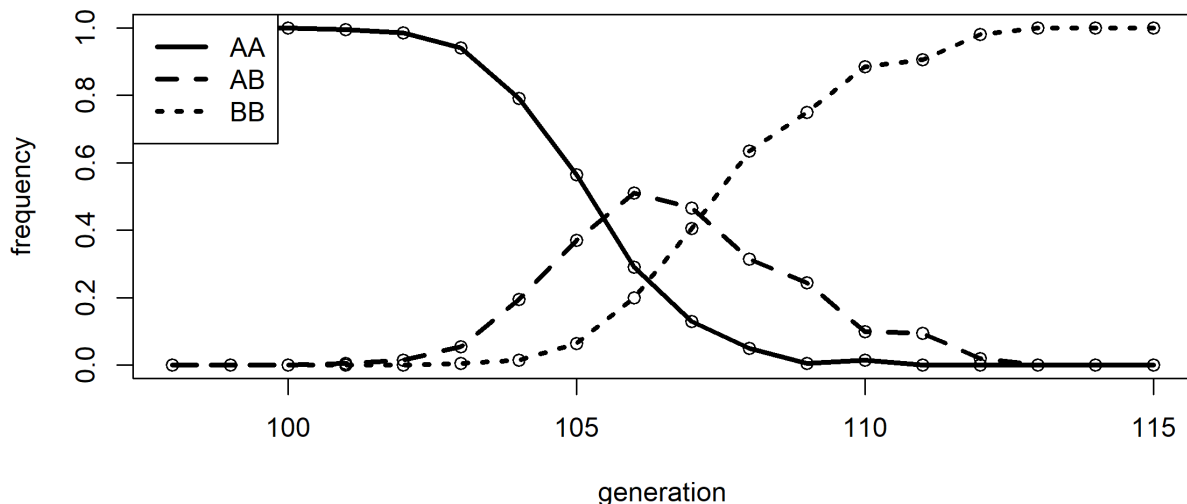
6.7 Generation of traits based on genomic & phenotypic data

Genotyping and phenotypic data can be used to generate a realistic trait architecture for a real-world population. For this the function effect.estimate.add provide an rrBLUP estimator to assign linear effects all markers.

```r
# Load in some data (Ideally real-world but here from an simulated MoBPS population
data(ex_pop)
pheno <- get.pheno(ex_pop, gen=1:5)
geno <- get.geno(ex_pop, gen=1:5)
haplo <- get.haplo(ex_pop, gen=1:5)
map <- get.map(ex_pop, use.snp.nr=TRUE)

# Estimate marker effects
real.bv.add <- effect.estimate.add(geno, pheno, map)

# Generate new population (or in this case exactly the same) with estimated marker effects
population <- creating.diploid(dataset = haplo, map = map, real.bv.add = real.bv.add)

# First run of breeding.diploid to calculate true genomic values
population <- breeding.diploid(population)

# Check how good the approximation works here:
cor(get.bv(population, gen=1)[1,], get.pheno(ex_pop, gen=1:5)[1,])
cor(get.bv(population, gen=1)[1,], get.bv(ex_pop, gen=1:5)[1,])
```

6.8 Generation of traits with breed specific QTLs

Some allele only have an effect on a trait when they stem from a specific breed. For this, it is possible to generate breed specific QTLs. The use of founder pools not only allows for modelling of breed specific traits, but also to backtrack from which founder pool each allelic variant stem from originally.

```r
# Generation of a small population with 10 individuals from two distinct
pools
population = creating.diploid(nsnp=100, nindi=4, founder.pool = 1)
population = creating.diploid(population = population , nsnp=100, nindi=4,
                              founder.pool = 2)

# Generation of a trait that will only have variablity in pool 1
population = creating.trait(population, n.additive = 100, trait.pool = 1)
```

```
# Generation of a trait that will only have variablity in pool 2
population = creating.trait(population, n.additive = 100, trait.pool = 2)

old_bv = get.bv(population, gen = 1)
# Extraction of QTL effects to combine the traits into a joint trait
qtl_effects = get.qtl.effects(population)
real.bv.add = rbind(qtl_effects[[1]][[1]], qtl_effects[[1]][[2]])

# Generate a trait that combines the two pool specific traits into a joint
trait
# Replace the old two traits
population = creating.trait(population, real.bv.add = real.bv.add,
                            replace.traits = TRUE)

# Simulate a couple of generations of random mating
population = breeding.diploid(population, breeding.size = 50)
population = breeding.diploid(population, breeding.size = 50)
population = breeding.diploid(population, breeding.size = 4)

new_bv = get.bv(population, gen=1)

old_bv
#M1_1        M2_1        M3_1        M4_1        F1_1        F2_1        F3_1        F4_1
#Trait 1   95.33272   97.05488 100.00000 100.00000   94.98303   82.02923
100.00000 100.00000
#Trait 2 100.00000 100.00000   82.48126   82.76736 100.00000 100.00000
99.78461   83.69153

new_bv
#M1_1        M2_1        M3_1        M4_1        F1_1        F2_1        F3_1        F4_1
#Trait 1 95.33272 97.05488 82.48126 82.76736 94.98303 82.02923 99.78461
83.69153

# Which segment stems from which founder pool?
pools = get.pool(population, gen = 4, plot = TRUE)
```


```
34
```
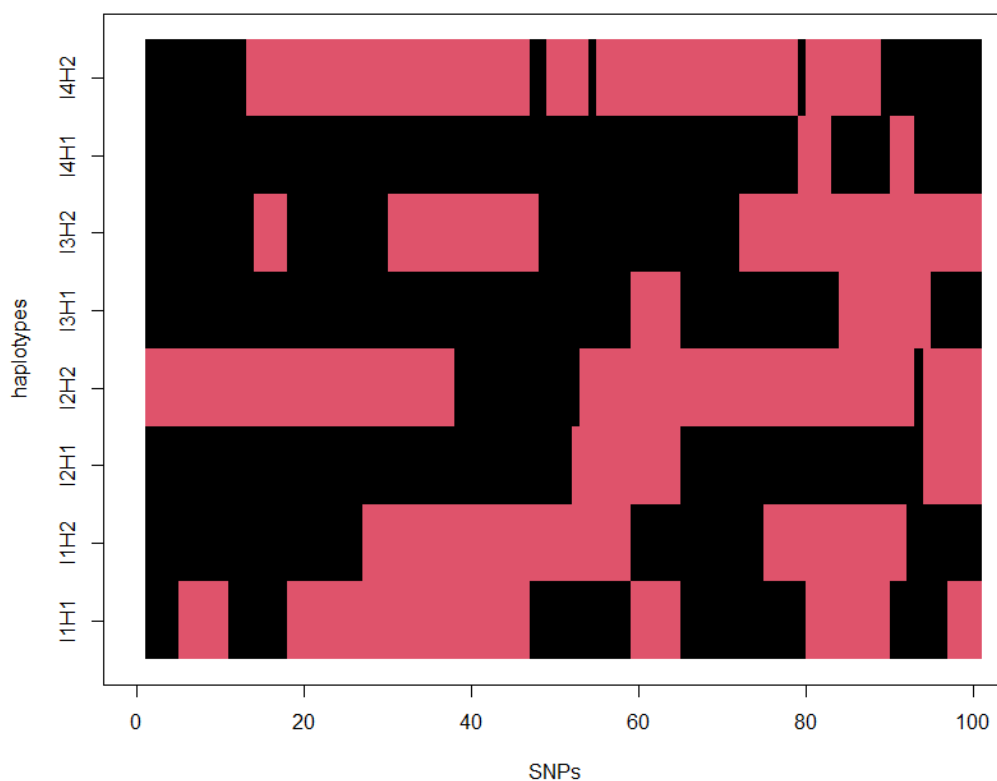
6.9 Generation of inbred lines

```
# Generation of an inbred founder population
# All plants are saves as sex 0 (male)
dhfounder_population <- creating.diploid(nindi = 200, nsnp = 500, dataset = "homorandom",
                                         sex.quota = 0)

# Generation of a heterozygous founder population
population <- creating.diploid(nindi = 50, nsnp = 500, sex.quota = 0)

# Generation of inbreds via DH technology
dh_population <- breeding.diploid(population, breeding.size = c(200,0), dh.mating = TRUE)

# Generation of inbreds via five generation of selfing
selfing_population <- population
for(index in 1:5){
  selfing_population <- breeding.diploid(selfing_population, breeding.size = c(200,0),
                                         selfing.mating = TRUE)
}

table(get.geno(dhfounder_population, gen=1))
table(get.geno(dh_population, gen=2))
for(gen in 1:6){
  print(table(get.geno(selfing_population, gen=gen)))
}
# Share of heterozygous markers in the selfing population reduces by ~50% per cycle
```

6.10 Generation of a trait with maternal component

For some traits part of the trait expression depends on the genotype of a parent (e.g. mother supplying messenger RNA or proteins to the egg). The following example shows how to generate such a trait with the direct and maternal effect being separated into sub-traits. All following analysis would then be executed just based on the combined trait:

```
# 1. Trait is direct effect
# 2. Trait is maternal effect
population <- creating.diploid(nsnp = 1000, nindi=10,
                              n.additive = c(100,100),
                              var.target = c(100,20),
                              shuffle.traits = 1:2,
                              shuffle.cor = matrix(c(1,0.5, 0.5, 1), nrow=2),
                              is.maternal = c(FALSE, TRUE),
                              trait.name = c("direct_effect", "maternal_effect"))

# 3. Trait is the sum of both previous traits
population <- add.combi(population, trait = 3, combi.weights = c(1,1), trait.name =
"combined_trait")

# Generate some offspring
population <- breeding.diploid(population, breeding.size = 100)

# Looking at trait 2 and mother id
bvs <- get.bv(population, gen=2)
pedi <- get.pedigree(population, gen=2)
cbind(pedi[,3], bvs[2,])
```

6.11 Simulation of a MAGIC population in maize

We here show how to perform an exemplary simulation of a MAGIC population in maize with a mating scheme given in (Zheng et al. 2015) – cf. adjacent Figure.

Since default settings in MoBPS are to always use the last generation anyway the needed code is quite short even without cohort mode. For the sake of completeness, we provide a cohort version for the script as well.

In term of computation time this simulation with a 15.3M genome, 31k SNPs and a total of 780 individuals took 1.6 seconds on one core of my local maschine.

Non-cohort-modus:

```
# Generation of 20 fully-homozygous founders lines
# All plants are stored as male individuals (sex=0)
population <- creating.diploid(nindi = 20, sex.quota = 0, template.chip = "maize",
                              dataset = "homorandom")

# Simulate matings between all founders.
# Each plan is involved in exactly 19 matings.
population <- breeding.diploid(population, breeding.size = c(190,0),
                              breeding.all.combination = TRUE,
                              selection.size = c(20,0), max.offspring = 19)

# Simulate matings between plants of the last generation.
# Each plant is involved in exactly 2 matings.

population <- breeding.diploid(population, breeding.size = c(190,0),
                              selection.size = c(190,0), same.sex.activ = TRUE,
                              same.sex.sex = 0, max.offspring = 2)
population <- breeding.diploid(population, breeding.size = c(190,0),
                              selection.size = c(190,0), same.sex.activ = TRUE,
                              same.sex.sex = 0, max.offspring = 2)
population <- breeding.diploid(population, breeding.size = c(190,0),
                              selection.size = c(190,0), same.sex.activ = TRUE,
                              same.sex.sex = 0, max.offspring = 2)
```

Cohort-modus:

```
# Generation of 20 fully-homozygous founders lines
# All plants are stored as male individuals (sex=0)
population <- creating.diploid(nindi = 20, sex.quota = 0, template.chip = "maize",
                              dataset = "homorandom", name.cohort = "F0")

# Simulate matings between all founders.
# Each plan is involved in exactly 19 matings.
population <- breeding.diploid(population, breeding.size = c(190,0),
                              breeding.all.combination = TRUE,
                              selection.size = c(20,0),
                              selection.m.cohort = "F0", name.cohort = "F1")

# Simulate matings between plants of the last generation.
# Each plant is involved in exactly 2 matings.

population <- breeding.diploid(population, breeding.size = c(190,0),
                              selection.size = c(190,0), same.sex.activ = TRUE,
                              same.sex.sex = 0, max.offspring = c(2,0),
                              selection.m.cohort = "F1", name.cohort = "F2")
population <- breeding.diploid(population, breeding.size = c(190,0),
                              selection.size = c(190,0), same.sex.activ = TRUE,
                              same.sex.sex = 0, max.offspring = c(2,0),
                              selection.m.cohort = "F2", name.cohort = "F3")
```
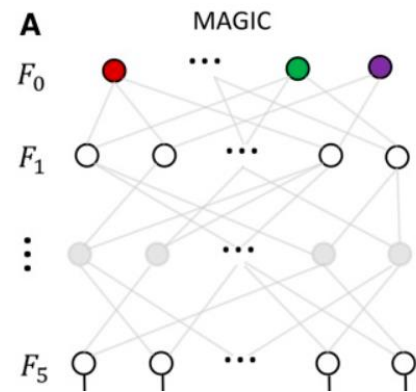
```r
population <- breeding.diploid(population, breeding.size = c(190,0),
                               selection.size = c(190,0), same.sex.activ = TRUE,
                               same.sex.sex = 0, max.offspring = c(2,0),
                               selection.m.cohort = "F3", name.cohort = "F4")
```

6.12 Simulation of Introgression on blue eggshell QTL in chicken

We here show how to perform an exemplary simulation of a breeding scheme to perform introgression of a single QTL. In terms of computing time this simulation with a 5M genome, 5k SNPs and a total of 520 individuals took 1.2 seconds on one core of my local machine without the usage of miraculix.

```r
# Generate an input SNP-dataset
# 10 White-Layer (0) (20 haplotypes, 5'000 SNPs)
# 10 Wild population (1) (20 haplotypes, 5'000 SNPs)
dataset1 <- matrix(0, nrow = 5000, ncol = 20)
dataset2 <- matrix(1, nrow = 5000, ncol = 20)

# Generation of a trait
# Colums code: SNP, chromosome, effect 00, effect 01, effect 11
# Blue Eggshell QTL is positioned on SNP 2000, chromosome 1
major_qtl <- c(2000, 1, 0, 10000, 20000)
# In all other positions the white layer genome is assumed to be favorable
# All marker effects combiened are smaller than the blue eggshell QTL
rest <- cbind(1:5000, 1, 1, 0.5, 0)
trait <- rbind(major_qtl, rest)

# Generation of the base-population
# First 10 individuals are female (sex=2)
# Next 10 individuals are male (sex=1)
population <- creating.diploid(dataset = cbind(dataset1, dataset2),
                               real.bv.add = trait, name.cohort = "Founders",
                               sex.s = c(rep(2,10), rep(1,10)))

# Simulate random mating:
population <- breeding.diploid(population, breeding.size = c(100,100),
                               selection.size = c(10,10),
                               selection.m.cohorts = "Founders_M",
                               selection.f.cohorts = "Founders_F",
                               name.cohort = "F1")

# Simuation of matings with selection:
# Top 50 cocks are mated to the 10 founder hens
# Selection of the cocks based on their genomic value ("bv")
# Target: Increase share of white layer while preserving blue egg shell QTL

population <- breeding.diploid(population, breeding.size = c(100,100),
                               selection.size = c(50,10),
                               selection.m.cohorts = "F1_M",
                               selection.f.cohorts = "Founders_F",
                               name.cohort = "BC1", selection.m = "function",
                               selection.criteria = "bv")
population <- breeding.diploid(population, breeding.size = c(100,100),
                               selection.size = c(50,10),
                               selection.m.cohorts = "BC1_M",
                               selection.f.cohorts = "Founders_F",
                               name.cohort = "BC2", selection.m = "function",
                               selection.criteria = "bv")
population <- breeding.diploid(population, breeding.size = c(100,100),
                               selection.size = c(50,10),
                               selection.m.cohorts = "BC2_M",
                               selection.f.cohorts = "Founders_F",
                               name.cohort = "BC3", selection.m = "function",
                               selection.criteria = "bv")
```

```
# Mating of cocks and hens that are heterozygous in blue egg shell QTL
# 25% of resulting offspring should be homozygous in blue egg shell QTL

population <- breeding.diploid(population, breeding.size = c(100,100),
                              selection.size = c(50,50),
                              selection.m.cohorts = "BC3_M",
                              selection.f.cohorts = "BC3_F",
                              name.cohort = "IC",
                              selection.criteria = "bv")

# Check genomic share of wild race in the final generation
genoIC <- get.geno(population, cohorts = "IC_F")
plot(rowSums(genoIC)/200, xlab = "genome", ylab = "frequency of wild allele", type
= "l")
abline(v = 2000, lwd = 2, col = "red")
```
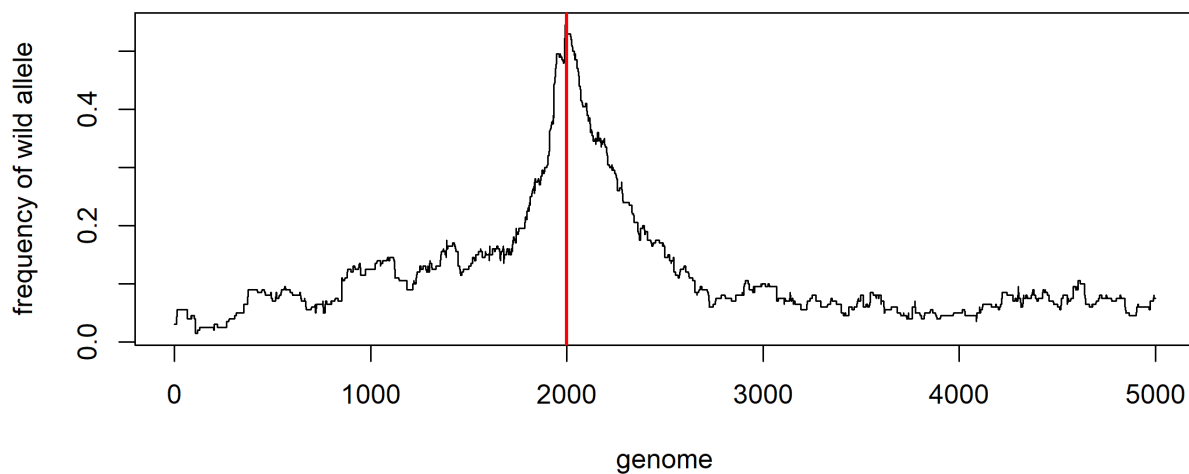

As expected, the frequency of genetic material stemming from the wild type is higher in the region of the QTL.

6.13 Simulation of a conservation breeding program using cock rotation in chicken

We here show how to perform an exemplary simulation of a conservation breeding scheme in chicken and compare a random mating environment with the use of a cock rotation mating scheme (Pook et al. 2017). In term of computing time these two simulations with a 5M genome, 5k SNPs and a total of 1092 individuals each took 5.7 seconds on one core of my local maschine without the usage of miraculix.

```r
# Generate initial boxes with 5 hens (sex=2) and 1 cock (sex=1) each
population <- NULL
for(index in 1:7){
  population <- creating.diploid(population = population, nindi = 6,
                                 nsnp = 5000, sex.s = c(1, 2, 2, 2, 2, 2),
                                 name.cohort = paste0("Box_", index, "gen_0"))
}

# Simulate 25 generations of matings.
# Hens are rotated by one box per generation.
# best1.from.cohort is the cohort used as sires
# best2.from.cohort is the cohort used as dams
for(gen in 1:25){
    for(index in 1:7){
    population <- breeding.diploid(population, breeding.size = c(1,5),
                                   selection.size = c(1,5),
              selection.m.cohorts = paste0("Box_",
                if(index==1){7} else {index-1},"gen_", gen-1,"_M"),
              selection.f.cohorts = paste0("Box_", index,"gen_", gen-1,"_F"),
              name.cohort = paste0("Box_", index, "gen_", gen),
              add.gen=gen+1
    )
  }
}

# Generate a population of same size without cock rotation
pop1 <- creating.diploid(nindi = 42, nsnp = 5000,
                         sex.s = c(rep(1,7), rep(2,35)))

# Simulate 25 generations of random mating
for(gen in 1:25){
  pop1 <- breeding.diploid(pop1, breeding.size = c(7,35),
                           selection.size = c(7,35))
}

kin <- kinship.development(population, gen = 1:26, ibd.obs = 1000, hbd.obs = 100)
kin1 <- kinship.development(pop1, gen = 1:26, ibd.obs = 1000, hbd.obs = 100)
```
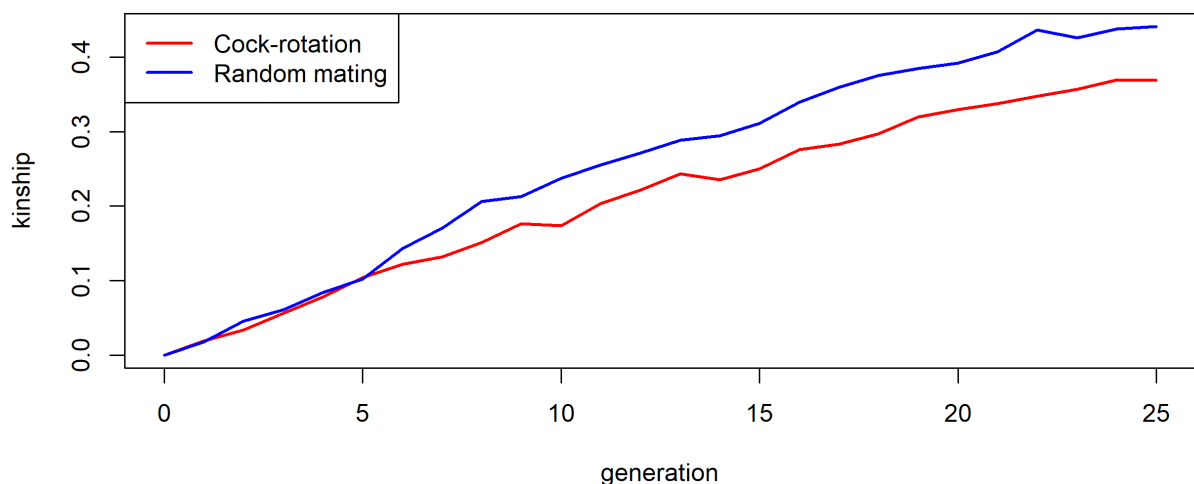
6.14 Simulation of gene editing in a cow breeding program

The following script can be used to simulate a breeding program that is utilizing genome editing. Design is chosen according to (Jenko et al. 2015; Simianer et al. 2018). Note that individual numbers are much smaller than in the two references to ensure low computation times. Simulation of 20 generations with 50'000 cows per generation would take 26.3 hours using 24 cores on the gwdg-hpc (Intel E5-2650 (2X12 core 2.2GHz)). This small example with a 5 Morgan chromosome, 5k SNPs and 4300 individuals took 7.4 seconds.

```r
# Generation of a base population:
# 1'000 Founder individuals
# 5'000 SNPs
# 100 additive single marker QTL
population <- creating.diploid(nindi = 1000, nsnp = 5000,
                               n.additive = 100, name.cohort = "Founders")

# Simulation of a random mating generation
# 100 bulls (sex=1), 1'000 cows (sex=2) are generated
population <- breeding.diploid(population, breeding.size = c(100,1000),
                               selection.size = c(500,500),
                               selection.m.cohorts = "Founders_M",
                               selection.f.cohorts = "Founders_F",
                               name.cohort = "Random")

# Generate 200 offspring of both from the top 5 bulls / 200 cows
# Heritability of the trait is set to 0.5
# only phenotypes previously unobserved cows are generated
population <- breeding.diploid(population, breeding.size = 200,
                               selection.size = c(5,200), bve = TRUE,
                               heritability = 0.5,
                               phenotyping = "non_obs_f",
                               selection.criteria = "bve",
                               name.cohort = "Top",
                               selection.m.cohorts = "Random_M",
                               selection.f.cohorts = "Random_F")

# Generate additional cows using all cows of the previous generation
# Cows are added to the same generation as the previous simulation
population <- breeding.diploid(population, breeding.size = c(0,900),
                               selection.size = c(5,1000),
                               selection.criteria = "bve",
                               name.cohort = "Sec_F",
                               selection.m.cohorts = "Random_M",
                               selection.f.cohorts = "Random_F",
                               add.gen = 3)

# Same cycle as before with additional genome editing
# Edits are chosen based on highest effects in rrBLUP
population <- breeding.diploid(population, breeding.size = c(100,100),
                               selection.size = c(5,200), bve = TRUE,
                               phenotyping = "non_obs_f",
                               selection.criteria = "bve",
                               name.cohort = "Top_Edit",
                               selection.m.cohorts = "Top_M",
                               selection.f.cohorts = c("Top_F","Sec_F"),
                               nr.edits = 20, estimate.u = TRUE)

population <- breeding.diploid(population, breeding.size = c(0,900),
                               selection.size = c(5,1000),
                               selection.criteria = "bve",
                               name.cohort = "Sec_Edit",
                               selection.m.cohorts = "Top_M",
```
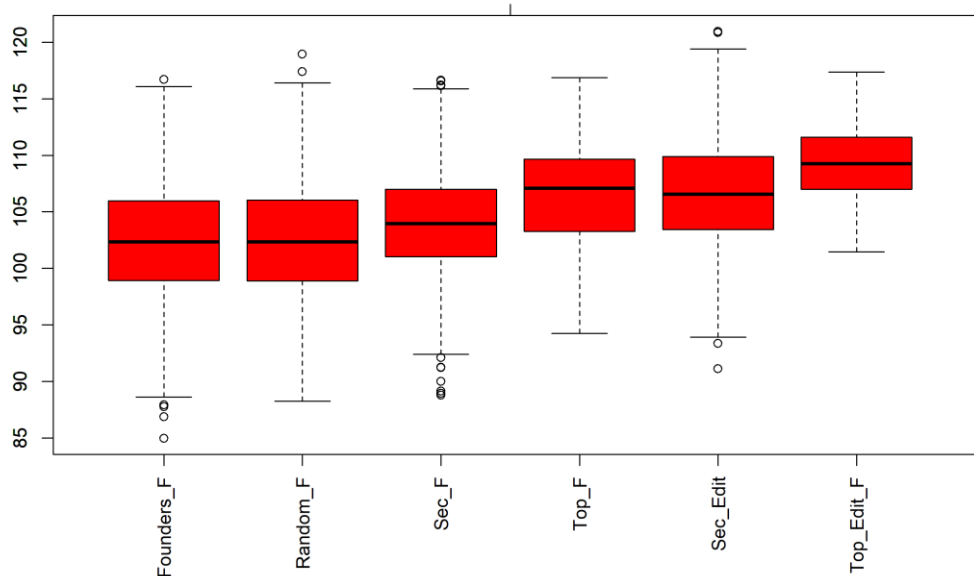
```
                              selection.f.cohorts = c("Top_F","Sec_F"),
                              add.gen = 4)


bv.development.box(population, cohorts = c("Founders_F", "Random_F", "Sec_F",
                              "Top_F", "Sec_Edit", "Top_Edit_F"), display = "bv")
```


6.15 Simulation of a simple GxE case

Depending on the species and use case the actual prediction model can be vastly different. We will
here showcast an example of where the observations in the two environments are still seen as
observations from the same trait with a single trait model being applied. To use a multitrait model,
one should not use **combine.traits** as done here and activate the multitrait model wanted (e.g.
sommer.multi.bve = TRUE)

```
# Generation of a baseline population
# Generation of a trait with slightly different genetic architecture in two
environments

population <- creating.diploid(nsnp=5000, nindi=100, n.additive = c(100,100),
                               shuffle.cor = matrix(c(1,0.9,0.9,1), ncol=2))

# Linking of the two traits (environments)
population <- combine.traits(population, combine.traits = 1:2)

# Collection of phenotypic data
# 50 lines are phenotypes in either of the two environments
population <- breeding.diploid(population, phenotyping.database = cbind(1,1),
                               heritability = 0.3, n.observation = c(1,0))
population <- breeding.diploid(population, phenotyping.database = cbind(1,2),
                               heritability = 0.3, n.observation = c(0,1))


# Breeding value estimation on the entire set
population <- breeding.diploid(population, bve=TRUE, bve.gen = 1)

# Details on the accuracy of the BVE on the individuals sets
#(for comparability later)
analyze.bv(population, database=cbind(1,1))
analyze.bv(population, database=cbind(1,2))
```

```
# Breeding value estimation when only using one of the two sets
population <- breeding.diploid(population, bve=TRUE, bve.database = cbind(1,1))
population <- breeding.diploid(population, bve=TRUE, bve.database = cbind(1,2))
```

6.16 Use of the culling module

```
# Generate a founder population
population <- creating.diploid(nsnp=1000, nindi = 100, n.additive = 100)

# Make sure the simulated trait has a set mean and variance
population <- bv.standardization(population, gen=1, mean.target = 100, var.target = 10)

# Generate offspring
population <- breeding.diploid(population, breeding.size = 500, selection.size = c(50,50))

### Plain application of culling with a set probability

# Apply culling with a probability of 0.3 to all individuals
pop1 <- breeding.diploid(population, culling.gen=2, culling.share1 = 0.3)

# Death animals put into class (-1)
# This means that these individuals on default will not be used for reproduction anymore
# These individuals can still be used in a breeding value estimation
table(get.class(pop1, gen=2))
#-1    0
#170 330

### Apply culling with a probability depending on the genomic value

pop2 <- breeding.diploid(population, culling.gen=2, culling.bv1 = 100, culling.share1 = 0.5,
culling.bv2 = 105, culling.share2 = 0.1,
                        culling.index = 1)

class <- get.class(pop2, gen=2)
bv <- get.bv(pop2, gen=2)[1,]

par(mfrow=c(2,1))
hist(bv[class==0], xlim=c(90,110), main="Genomic value distribution of alive individuals",
xlab="genomic value")
hist(bv[class==(-1)], xlim=c(90,110), main="Genomic value distrition of death individuals",
xlab="genomic value")
```

Genomic value distribution of alive individuals

Genomic value distrition of death individuals
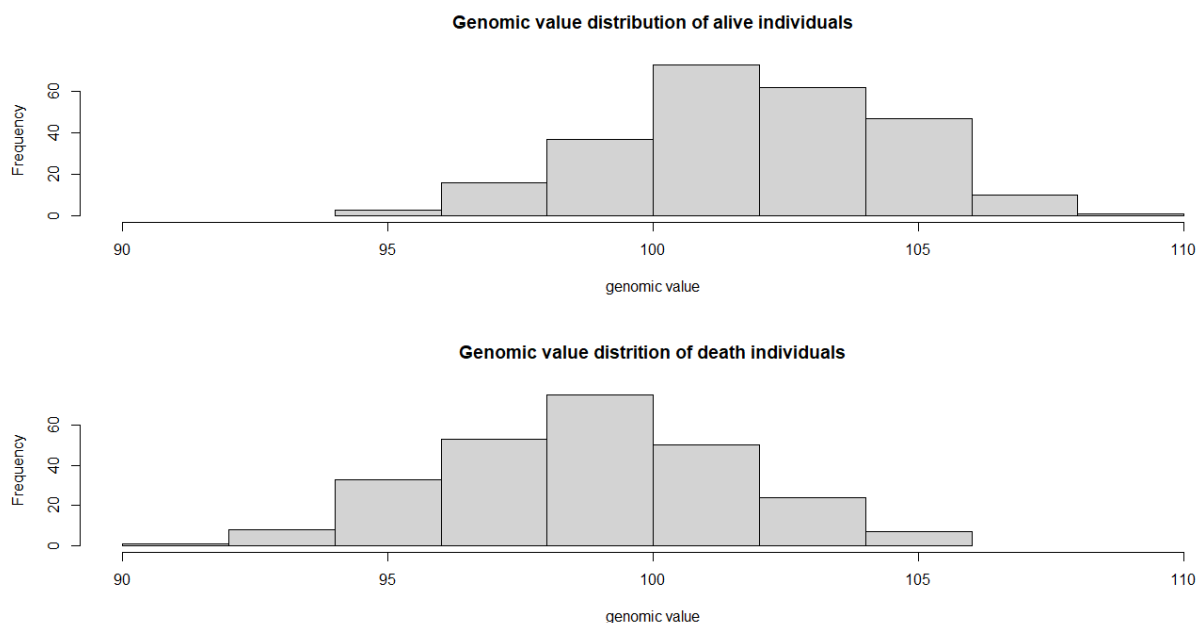
6.17 Threshold selection

Threshold selection will exclude all animals below predefined selection threshold to be used for selection.

```
# Generate a founder population with 5 traits
population = creating.diploid(nindi=50, nsnp=1000, n.additive = rep(100,5),
                              mean.target = 100, var.target = 25)

# Generate some phenotypes and perform a BVE
population = breeding.diploid(population, heritability = 0.3, phenotyping = "all")
population = breeding.diploid(population, bve = TRUE, bve.gen=1)

# Only consider individuals that
# 1. Have a minimum BVE of 99 for trait 1
# 2. Have a minimum underlying true genomic value of 98 for trait 3
# 3. Have a minimum BVE of 199 for trait 4 + trait 5
threshold.selection.index = matrix(c(1,0,0,0,0,
                                     0,0,1,0,0,
                                     0,0,0,1,1), ncol=5, byrow=TRUE)

threshold.selection.value = c(99, 98, 199)
threshold.selection.sign = rep(">", 3)
threshold.selection.criteria = c("bve", "bv", "bve")

# Generate 100 offspring from 3 males / females that fullfil these contrains
# selection otherwise is random
population = breeding.diploid(population,
                             breeding.size = 100,
                             selection.size = c(3,3),
                             selection.criteria = "random",
                      threshold.selection.index = threshold.selection.index,
                      threshold.selection.value = threshold.selection.value,
                      threshold.selection.sign = threshold.selection.sign,
                      threshold.selection.criteria = threshold.selection.criteria)

#Start selection procedure.
#Selection male size:
#  Threshold selection 1 is fullfilled by 21 out of 25.
#Threshold selection 2 is fullfilled by 22 out of 25.
#Threshold selection 3 is fullfilled by 14 out of 25.
#After threshold selection 11 individuals remain to select 3.
#Selection female size:
#  Threshold selection 1 is fullfilled by 19 out of 25.
#Threshold selection 2 is fullfilled by 19 out of 25.
#Threshold selection 3 is fullfilled by 16 out of 25.
#After threshold selection 10 individuals remain to select 3.

# Exemplary further analysis:
# Extract the used parents and look at their estimated breeding values
parent_ids = unique(as.numeric(get.pedigree(population, gen=2, id=TRUE)[,c(2,3)]))
parent_database = get.database(population, id=parent_ids)
get.bve(population, database = parent_database)


#M8_1      M24_1      M25_1       F2_1       F10_1      F24_1
#Trait 1  99.42298 100.46841  99.93170 103.15000 104.41123 101.03186
#Trait 2 100.02388 103.03817  96.80039  99.05030  98.33225 100.87097
#Trait 3 103.56059 101.58174 102.27021 100.29314 101.98670  98.70362
#Trait 4 101.32436 101.60426 101.98240  99.23875 101.63607 102.05797
#Trait 5  98.89409  98.72613  98.91435 102.66770 101.44858  99.52546
```

6.18 Controlling of the litter size

In case the litter size is not depending on any underlying genetics, it can be controlled by setting **repeat.mating** in *breeding.diploid().* This example reflects on how to get up a simulation with the litter size having underlying genetics.

```
# Setting up the initial population
population = creating.diploid( nsnp=500, chr.nr =5, nindi = 100)


# distribution of litter size
litter.size = matrix(c(1, 0.70,
                       2, 0.15,
                       3, 0.10,
                       4, 0.05
), byrow=TRUE, ncol=2)


# Construction of the fertility trait with discrete phenotypic realisations
# This is a trait with only purely dominate effects
# (to make it highly related to inbreeding)

population = creating.trait(population, trait.name = "Fertility",
                            n.additive = 250,
                            mean.target = 100, var.target = 100)

population = breeding.diploid(population, heritability = 1/5)

# The trait on default has a gaussian distribution
# here I am adding code to make phenotypic observations be discrete natural numbers

sigma_ferti = sqrt(500)
# This is the overall phenotypic variance of the fertility trait
prob_cum = cumsum(litter.size[,2])

litter.function = function(x){
  if(x < qnorm(prob_cum[1], mean=100, sd=sigma_ferti)){
    y = 1
  } else if(x < qnorm(prob_cum[2], mean=100, sd=sigma_ferti)){
    y = 2
  } else if(x < qnorm(prob_cum[3], mean=100, sd=sigma_ferti)){
    y = 3
  } else {
    y = 4
  }
  # IF you dont like If loops and want it efficiently do this instead
  # y = sum(  qnorm(prob_cum, mean=100, sd=sigma_ferti) < x) +1
  return(y)
}

# Traits that control the litter size
# should only have natural numbers as potential realisations

population = creating.phenotypic.transform(population,
                       phenotypic.transform.function = litter.function,
                                           trait = 1)
population = breeding.diploid(population,
                             repeat.mating = "genetic",
                             repeat.mating.trait = 1,
                             repeat.mating.max = 4)

# Simulate 100 years of random mating
for(generation in 2:100){
  population = breeding.diploid(population, breeding.size.litter = 30,
                             selection.criteria = "random",
                             selection.size = c(10,10))
}
```

```
# As individuals with better fertility have more offspring
# fertility increases over time

bv = numeric(100)
for(index in 1:100){
  bv[index] = mean(get.bv(population, gen=index))
}

# The increase in the trait reduces over time as the
# difference in litter size between animals reduces over time

plot(bv, type="l", xlab = "generation", ylab = "genomic value")
```
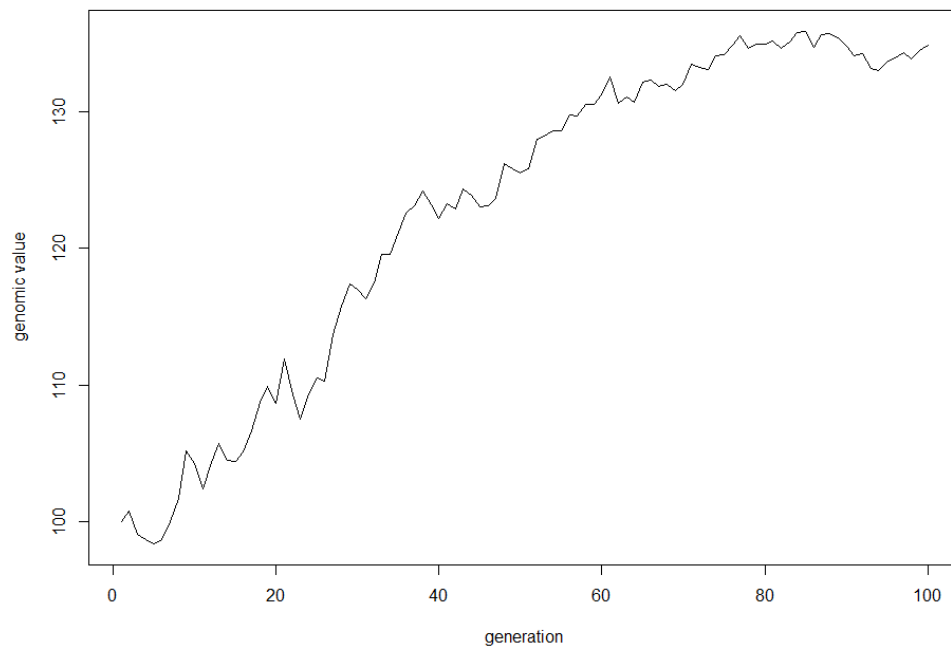


```
plot(rowSums(population$info$size)[-1]/30, main="Litter size",
     xlab = "generation", ylab = "avg. litter ")
```
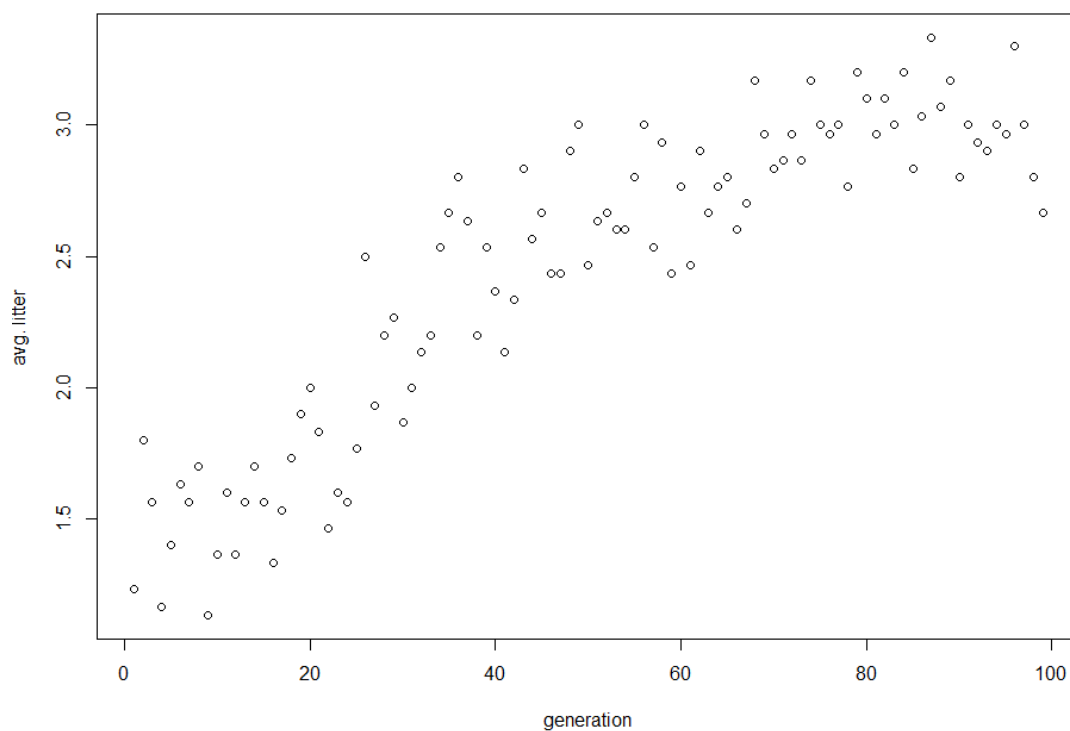
6.19 Simulation of an age structure

```r
# Generation of an initial breeding population
population <- creating.diploid(nsnp = 1000, nindi = 200, name="Breeding_Population_1")

for(index in 1:20){

  # Generation of new offspring
  population <- breeding.diploid(population, breeding.size = 100,
                  name=paste0("Offspring_",index),
                  selection.m.cohorts = paste0("Breeding_Population_", index, "_M"),
                  selection.f.cohorts = paste0("Breeding_Population_", index, "_F"),
                  time.point = index)

  # Calculate how many animals are alive
  alive_m <- sum(get.class(population,
                  cohorts = paste0("Breeding_Population_", index, "_M"))==0) + 50
  alive_f <- sum(get.class(population,
                  cohorts = paste0("Breeding_Population_", index, "_F"))==0) + 50

  # individuals from the offspring and breeding population are merged into a joint
  # breeding population for the next cycle

  population <- breeding.diploid(population, combine = TRUE,
                  selection.m.cohorts = c(paste0("Breeding_Population_", index, "_M"),
                                          paste0("Offspring_", index, "_M")),
                  name.cohort = paste0("Breeding_Population_", index+1, "_M"))

  population <- breeding.diploid(population, combine = TRUE,
                  selection.f.cohorts = c(paste0("Breeding_Population_", index, "_F"),
                                          paste0("Offspring_", index, "_F")),
                  name.cohort = paste0("Breeding_Population_", index+1, "_F"))

  # All individuals are culled with a probability of 0.3
  ### IF different culling probabilites per age are wanted culling could be
  ### applied on the Offspring cohorts
  ### if one copy of an individuals is culled, all other copies are culled as well
  ### culling.share1 can also be a vector to assign each individual will a different
  ### culling probability

  population <- breeding.diploid(population, culling.cohorts =
                          paste0("Breeding_Population_", index+1, c("_M", "_F")),
                  culling.share1 = 0.3)

}

# Extract the Age distribution in the final breeding population
age <- 21 - get.age.point(population,
                  cohorts=paste0("Breeding_Population_", 21, c("_M", "_F")) )
table(age)
# age
#1   2   3   4   5   6   7   8   9  10  11  16  17  21
#100 73  54  32  23  18  17   9   6   7   6   1   1   1
hist(age, main="Distribution of the age of individuals in the breeding population", nclass=50)
```


Distribution of the ages of individuals in the breeding population
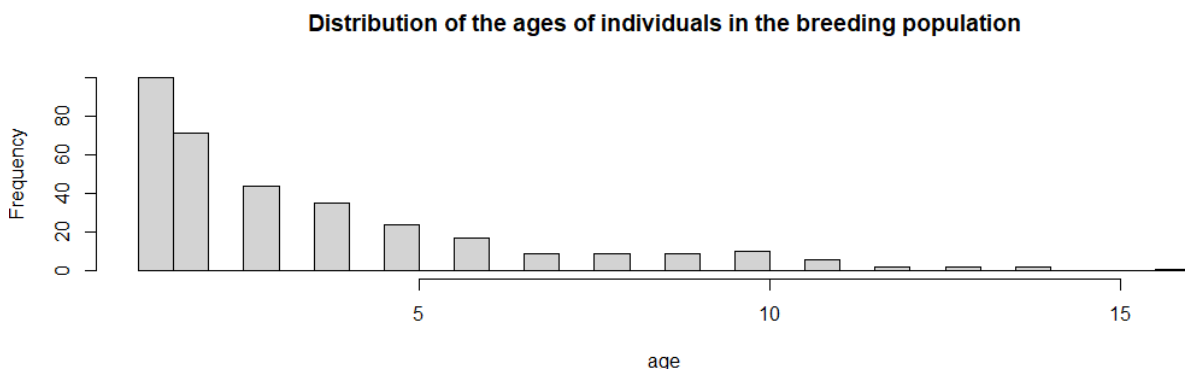
6.20 Import data from a ped-file

```
# Create an exemplary ped-file:
population <- creating.diploid(nsnp=1000, nindi=100)
get.pedmap(population, path="temp", gen=1)

# Transform PED -> Haplotypes per column
ped_file <- utils::read.table("temp.ped")
haplo12 <- t(ped_file[,-(1:6)])
haplo <- matrix(0, ncol = ncol(haplo12)*2, nrow=nrow(haplo12)/2)
for(index1 in 1:ncol(haplo12)){
  haplo[,index1*2+c(-1,0)] <- matrix(haplo12[,index1], ncol=2, byrow=TRUE)
}

# markers in exclude snps are not picked as QTL / no genotyping is performed
population = creating.diploid(dataset = haplo)
```

6.21 Generate QTL / and neural loci (e.g. not used in breeding value estimation)

```
# The goal here is to generate 1000 markers of which:
# 100 are QTLs not placed on an array
# 500 markers that are placed on an array
# 400 markers that are not on the array (neural)

# Pick array markers:
array_marker <- sort(sample(1:1000, 500))
array_mobps <- rep(FALSE, 1000)
array_mobps[array_marker] <- TRUE


# markers in exclude snps are not picked as QTL / no genotyping is performed
population = creating.diploid(nsnp=1000, nindi=100, n.additive = 100,
                             exclude.snps = array_marker, share.genotyped = 0)

# Add an additional genotyping array
# Default array (1) will always contain all markers
# Hence this is the second array (2)
# You can later selected with array to use for genotyping with genotyped.array
population <- add.array(population, marker.included = array_mobps)

# Extract QTL position and neural loci position
qtls <- sort(get.qtl(population))
neural_loci <- (1:1000)[-c(qtls, array_marker)]


# Perform a breeding value estimation based on the SNP subset
population <- breeding.diploid(population = population, genotyped.array = 2,
                              genotyped.gen = 1, genotyped.share = 0.5)
population <- breeding.diploid(population, phenotyping.gen = 1, heritability = 0.5)
population <- breeding.diploid(population, bve=TRUE, bve.gen = 1,
                              singlestep.active = TRUE)

# if less than all markers are used in the BVE you will always get a print of the
form:
# 500 markers survived filtering for BVE.

# To check which markers are genotyped for which individuals
get.genotyped.snp(population, gen=1)
```

7   Exporting information from the population-list (get.XXXX)

Most of the data stored in a population list is highly compressed since saving haplotypes of all individuals of the dataset for all generations would often exceed most local machines or even servers. For some applications (especially if one wants to perform his own fancy simulations without contacting the author and asking him to extend the package) it might be useful to understand the data structure behind it. For that, we refer to section 10. In most cases, using our predefined export functions should be enough:

To select for which individuals the required information is exported groups of individuals can be selected via **gen**, **database,** and **cohorts**. For details on the structure, we refer to section 3. On default, names of individuals will be chosen according to the MoBPS identifier name coding "Sex Number _ Generation". To instead use the underlying ID assigned upon generation set the parameter **use.id** to TRUE. Note that the other of individuals will be done according to position in the database (starting with the lowest generation), even when the input in **gen/database/cohorts** might suggest a different ordering!

7.1   get.geno

This function will export genotypes. To additionally output the base pair of the minor/major allele set the parameter **export.alleles** to TRUE. Each column contains one genotype with column names indicating sex, individual number and generation.

To replace all non-genotyped entries with NAs set **non.genotyped.as.missing** to TRUE.

```
> genos <- get.geno(population,gen=3)
> genos[1:5,1:10]
         M1_3 M2_3 M3_3 M4_3 M5_3 M6_3 M7_3 M8_3 M9_3 M10_3
Chr1SNP1    1    2    1    0    2    1    2    2    2     1
Chr1SNP2    1    0    1    0    2    1    1    2    1     0
Chr1SNP3    1    0    1    1    0    1    0    1    0     1
Chr1SNP4    0    0    1    1    1    1    0    0    0     1
Chr1SNP5    1    0    1    2    1    1    0    0    0     0
```

7.2   get.haplo

This function will export haplotypes. To additionally output the base pair of the minor/major allele set the parameter **export.alleles** to TRUE. Each column contains one haplotype with column names indicating sex, individual number, chromosome set (currently only diploid individuals) and generation. To replace all non-genotyped entries with NAs set **non.genotyped.as.missing** to TRUE.

```
> haplos <- get.haplo(population,gen=3)
> haplos[1:5,1:10]
         M1_3_set1 M1_3_set2 M2_3_set1 M2_3_set2 M3_3_set1 M3_3_set2 M4_3_set1 M4_3_set2 M5_3_set1 M5_3_set2
Chr1SNP1         0         1         1         1         0         1         0         0         1         1
Chr1SNP2         0         1         0         0         0         1         0         0         1         1
Chr1SNP3         1         0         0         0         1         0         0         1         0         0
Chr1SNP4         0         0         0         0         1         0         1         0         1         0
Chr1SNP5         1         0         0         0         1         0         1         1         1         0
```

7.3 get.bv / get.bve / get.pheno / get.reliability / get.selectionindex / get.npheno

These functions will export the true underlying breeding value ("bv"), the estimated breeding value ("bve"), the phenotype ("pheno"), the reliability for each breeding value estimation/trait ("reliability"), the selection index used in the last selection procedure ("selectionindex") and the number of times each trait has been phenotyped for the selected groups of individuals (**gen/database/cohort**).

```
> bv <- get.bv(population,gen=1)
> bve <- get.bve(population,gen=1)
> pheno <- get.pheno(population,gen=1)
> bve[1:5]
[1] 45.95833 48.85317 31.53675 35.73686 49.55310
> bv[1:5]
[1] 33.35516 47.52439 31.92523 59.13089 37.96659
> pheno[1:5]
[1] -22.230518  67.241256  -9.337895 -16.056936 101.497754
```

7.4 get.qtl / get.qtl.effects / get.qtl.variance

These function will provide information on the underlying simulated QTLs. With *get.qtl()* providing information on the positions (based on order in the genotype dataset) have effects. *get.qtl.effects()* additionally provide explicit information on the size of the effects. *get.qtl.variance()* will compute the additive genetic variance of all single marker QTLs (**n.additive** / **n.dominant / n.equal.additive / n.equal.dominant** / input of **real.bv.add**).

```
data(ex_pop)
get.qtl(ex_pop)
```

[1] 62 97 23 15 64 38 81 41 7 99

```
> data(ex_pop)
> effects <- get.qtl.effects(ex_pop)
> effects
[[1]]
[[1]][[1]]
       SNP Chromo  Effect AA Effect AB  Effect BB
  [1,]  62      1  0.7891847         0 -0.7891847
  [2,]  97      1  0.9209757         0 -0.9209757
  [3,]  23      1  0.3367262         0 -0.3367262
  [4,]  15      1 -0.1273978         0  0.1273978
  [5,]  64      1 -0.2388993         0  0.2388993
  [6,]  38      1 -0.8356816         0  0.8356816
  [7,]  81      1  2.0973001         0 -2.0973001
  [8,]  41      1  0.5510084         0 -0.5510084
  [9,]   7      1 -1.0976467         0  1.0976467
[10,]  99      1 -0.2680986         0  0.2680986


[[2]]
[[2]][[1]]
NULL


[[3]]
[[3]][[1]]
NULL


> effects <- get.qtl.variance(ex_pop)
> data(ex_pop)
> effects <- get.qtl.variance(ex_pop)
> effects
[[1]]
          add_snp add_chromo
Chr1SNP62      62          1 0.332831013
Chr1SNP97      97          1 0.102122822
Chr1SNP23      23          1 0.019774265
Chr1SNP15      15          1 0.008439706
Chr1SNP64      64          1 0.021094142
Chr1SNP38      38          1 0.269289086
Chr1SNP81      81          1 1.189399762
Chr1SNP41      41          1 0.123994438
Chr1SNP7        7          1 0.694462956
Chr1SNP99      99          1 0.026335678
```

7.5 get.recombi

This function will export all points of recombination and the genetic origin of each segment. The structure here is a list of 4 elements with elements 1 (paternal) and 2 (maternal) containing recombination points and elements 3 (paternal) and 4 (maternal) containing the genetic origin.

Each row is coding the genetic origin between two points (generation, sex, individual number, chromosome set). In the example provided this would mean that the segment between 0.000 and 0.218 of the paternal chromosome originates from the second chromosome set of the 532nd female individual of the first generation.

Note that in addition to all recombination points the start and end points of chromosomes are also exported.

```
> recombi <- get.recombi(population, gen=3)
> recombi[[1]][[1]]
 [1]  0.0000000  0.2183368  0.3517032  0.6187816  1.0718667  1.3089574  1.6053402  1.6390138  1.6627053  1.9827624  2.0846120
[12]  2.1364282  2.4399836  3.0289287  3.7351480  4.0304151  4.6348112  5.5724172  5.9576469  6.5091164  6.9677451  7.4578594
[23]  8.5493007  8.8105940  8.9809813  9.2018383 10.0248883 11.4693043 11.6175350 11.6398517 12.6182508 13.4061942 13.5190658
[34] 13.8967908 14.4158473 14.4829801 14.7182578 15.1508620 15.1545097 15.5787925 15.7266202 15.8199412 16.3273204 17.1619292
[45] 17.3799130 17.7438021 18.1580695 19.0026216 19.8908821 20.1429153 21.3074407 22.2018853 22.5145334 22.8492019 22.8980965
[56] 23.2768180 23.4729906 23.8671758 24.1965660 24.2540496 24.5550846 24.9169786 25.2573038 25.3671158 25.5634302 25.7719966
[67] 25.9989692 26.2255921 26.8175851 26.8552645 27.0569816 27.3228323 27.7300196 27.7441943 28.0331752 28.0909684 28.0925820
[78] 28.4657455 28.7984996 29.0263514 29.1274251 29.2751559 29.3417943 29.5015208 29.6375127 29.8279188 30.0385607 30.1115719
[89] 30.1383062 30.2662008 30.2851334 30.4432174
> recombi[[1]][[3]][1:5,]
     [,1] [,2] [,3] [,4]
[1,]    1    2  352    2
[2,]    1    2  352    1
[3,]    1    1   78    1
[4,]    1    2  352    2
[5,]    1    2  352    1
> recombi[[1]][[5]]
[1] "M1"
```

7.6   get.pedigree (1/2/3)

This function will export the pedigree. Individuals are coded by sex, individual number and generation.

```
> ped <- get.pedigree(population, gen=5)
> ped[1:5,]
     offspring father  mother
[1,] "M1_5"    "M12_4" "F10_4"
[2,] "M2_5"    "M2_4"  "F4_4"
[3,] "M3_5"    "M3_4"  "F1_4"
[4,] "M4_5"    "M2_4"  "F13_4"
[5,] "M5_5"    "M12_4" "F10_4"
> ped <- get.pedigree(population, gen=5, raw=TRUE)
> ped[1:5,]
     offspring.gen offspring.sex offspring.nr father.gen father.sex father.nr mother.gen mother.sex mother.nr
[1,]             5             1            1          4          1        12          4          2        10
[2,]             5             1            2          4          1         2          4          2         4
[3,]             5             1            3          4          1         3          4          2         1
[4,]             5             1            4          4          1         2          4          2        13
[5,]             5             1            5          4          1        12          4          2        10
```

Instead of a character string with "M"/"F" indicting sex, one can also directly export a table with 9 columns indicating Sex (1/2), Generation(1,2,3,…) and individual number (1,2,3,…) in a numeric format by setting **raw** to TRUE.

To export grandparents use *get.pedigree2()*, to get both *get.pedigree3()*. In *get.pedigree2()* one can additionally export the share of the genome inherited by which grandparent by setting the parameter **shares** to TRUE.

```
> ped <- get.pedigree2(population, gen=5)
> ped[1:5,]
     offspring paternal grandfather paternal grandmother maternal grandfather maternal grandmother
[1,] "M1_5"    "M19_3"              "F13_3"              "M6_3"               "F19_3"
[2,] "M2_5"    "M4_3"               "F6_3"               "M11_3"              "F13_3"
[3,] "M3_5"    "M19_3"              "F19_3"              "M6_3"               "F10_3"
[4,] "M4_5"    "M4_3"               "F6_3"               "M4_3"               "F19_3"
[5,] "M5_5"    "M19_3"              "F13_3"              "M6_3"               "F19_3"
> ped <- get.pedigree3(population, gen=5)
> ped[1:5,]
     offspring father  mother  paternal grandfather paternal grandmother maternal grandfather maternal grandmother
[1,] "M1_5"    "M12_4" "F10_4" "M19_3"              "F13_3"              "M6_3"               "F19_3"
[2,] "M2_5"    "M2_4"  "F4_4"  "M4_3"               "F6_3"               "M11_3"              "F13_3"
[3,] "M3_5"    "M3_4"  "F1_4"  "M19_3"              "F19_3"              "M6_3"               "F10_3"
[4,] "M4_5"    "M2_4"  "F13_4" "M4_3"               "F6_3"               "M4_3"               "F19_3"
[5,] "M5_5"    "M12_4" "F10_4" "M19_3"              "F13_3"              "M6_3"               "F19_3"
```

7.7   get.pedigree.visual

This function will generate a visualization of the pedigree of the individuals selected via **gen/database/cohorts**. The depth of the pedigree can be controled via the **depth.pedigee** parameter (default: 3). For this the R-package visPedigree is used with parameters from visPedigree also being

available within *get.pedigree.visual()* (**showgraph**, **outline**, **compact**). To use IDs instead of individual names set **id** to TRUE.

```
population = creating.diploid(nsnp=100, nindi=10)
population = breeding.diploid(population, breeding.size=10)
population = breeding.diploid(population,
          selection.m.database=cbind(c(1,2),1,1,5), breeding.size=10)
population = breeding.diploid(population,
          selection.m.database=cbind(c(2,3),1,1,5), breeding.size=10)
get.pedigree.visual(population, gen=4)
```
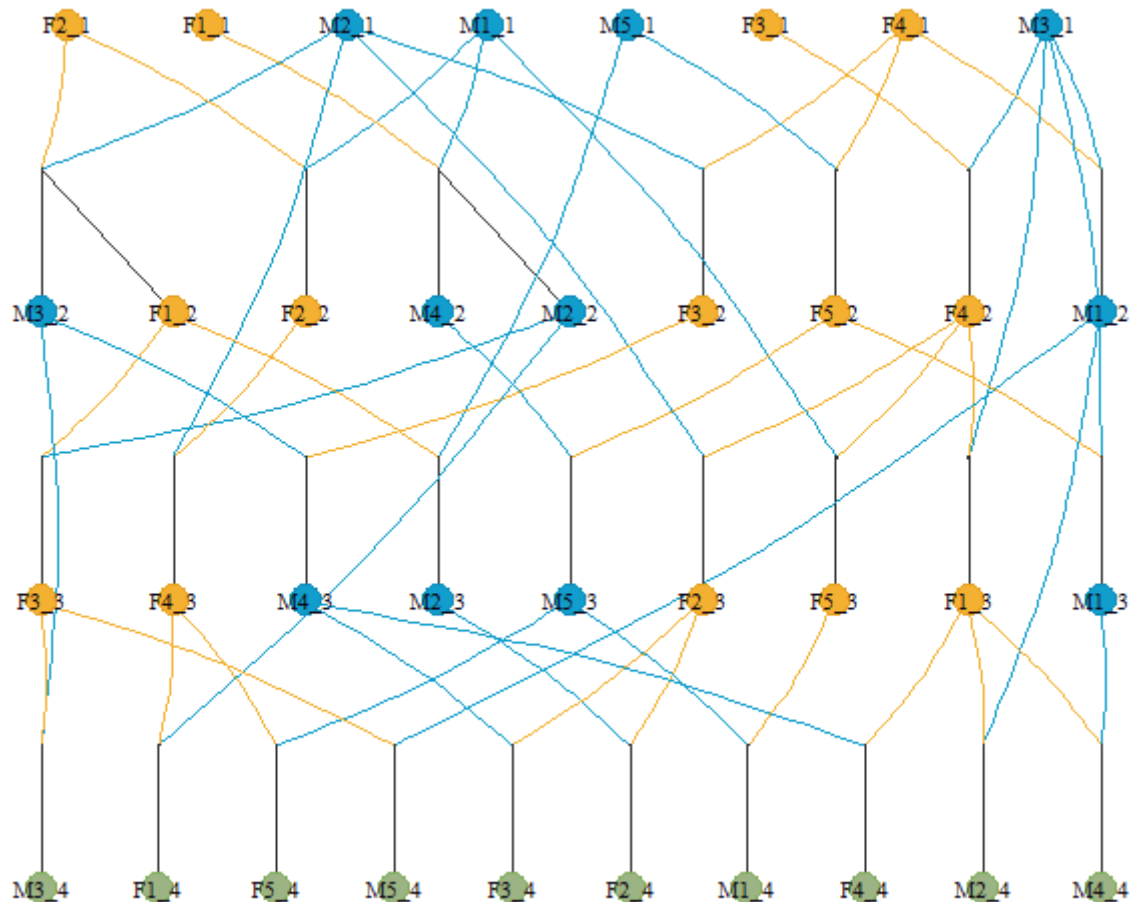

7.8 get.cohorts

This function extracts all existing cohorts from the population list. Set **extended** to TRUE to also extract further information on the cohorts:

```
> get.cohorts(population)[1:5]
[1] "Bull"           "Cows"           "Selected_Bulls" "New_bulls"      "New_cows"
> get.cohorts(population, extended=TRUE)[1:5,]
               name           generation male individuals female individuals class position first male position first female time point creating.type
Bull           "Bull"           "1"        "250"            "0"               "0"   "1"                 "0"                  "0"        "0"
Cows           "Cows"           "1"        "0"              "250"             "0"   "0"                 "1"                  "0"        "0"
Selected_Bulls "Selected_Bulls" "2"        "50"             "0"               "1"   "1"                 "1"                  "0"        "1"
New_bulls      "New_bulls"      "3"        "250"            "0"               "2"   "1"                 "1"                  "1"        "2"
New_cows       "New_cows"       "3"        "0"              "250"             "1"   "251"               "1"                  "1"        "2"
```

7.9 get.id

This function extracts the individual ids for individuals from the selected gen/database/cohorts.

52

7.10 get.class

This function extracts the class of each individual:

```
> get.class(population, gen=1:2)
  M1_1   M2_1   M3_1   M4_1   M5_1   M6_1   M7_1   M8_1   M9_1  M10_1  M11_1  M12_1  M13_1  M14_1
     0      0      0      0      0      0      0      0      0      0      0      0      0      0
 M15_1  M16_1  M17_1  M18_1  M19_1  M20_1  M21_1  M22_1  M23_1  M24_1  M25_1  M26_1  M27_1  M28_1
     0      0      0      0      0      0      0      0      0      0      0      0      0      0
 M29_1  M30_1  M31_1  M32_1  M33_1  M34_1  M35_1  M36_1  M37_1  M38_1  M39_1  M40_1  M41_1  M42_1
     0      0      0      0      0      0      0      0      0      0      0      0      0      0
```

…

```
F241_1 F242_1 F243_1 F244_1 F245_1 F246_1 F247_1 F248_1 F249_1 F250_1   M1_2   M2_2   M3_2   M4_2
     0      0      0      0      0      0      0      0      0      0      1      1      1      1
  M5_2   M6_2   M7_2   M8_2   M9_2  M10_2  M11_2  M12_2  M13_2  M14_2  M15_2  M16_2  M17_2  M18_2
     1      1      1      1      1      1      1      1      1      1      1      1      1      1
 M19_2  M20_2  M21_2  M22_2  M23_2  M24_2  M25_2  M26_2  M27_2  M28_2  M29_2  M30_2  M31_2  M32_2
     1      1      1      1      1      1      1      1      1      1      1      1      1      1
```

7.11 get.pool

This function extract from which founder pool each allelic variant was inherited from.

7.12 get.genotyped

This function extracts if an individual is genotyped or not (mostly relevant for cost calculation and/or use in Single Step GBLUP (Legarra et al. 2014).

```
> get.genotyped(population, gen=6)
  M1_6   M2_6   M3_6   M4_6   M5_6   M6_6   M7_6   M8_6   M9_6  M10_6  M11_6  M12_6
 FALSE   TRUE  FALSE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE
 M13_6  M14_6  M15_6  M16_6  M17_6  M18_6  M19_6  M20_6  M21_6  M22_6  M23_6  M24_6
  TRUE  FALSE   TRUE  FALSE  FALSE   TRUE  FALSE   TRUE   TRUE   TRUE  FALSE   TRUE
 M25_6  M26_6  M27_6  M28_6  M29_6  M30_6  M31_6  M32_6  M33_6  M34_6  M35_6  M36_6
 FALSE  FALSE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE  FALSE  FALSE  FALSE
 M37_6  M38_6  M39_6  M40_6  M41_6  M42_6  M43_6  M44_6  M45_6  M46_6  M47_6  M48_6
  TRUE  FALSE   TRUE  FALSE  FALSE  FALSE   TRUE   TRUE   TRUE  FALSE   TRUE  FALSE
```

7.13 get.genotyped.snp

This function extracts for which SNPs the selected groups of individuals (**gen/database/cohorts**) are genotyped.

```
> data(ex_pop)
> genotyped.snps <- get.genotyped.snp(ex_pop, gen=2)
> genotyped.snps
          M25_2 M25_2 M25_2 M25_2 M25_2 M25_2 M25_2 M25_2 M25_2 M2
Chr1SNP1      1     1     1     1     1     1     1     1     1
Chr1SNP2      1     1     1     1     1     1     1     1     1
Chr1SNP3      1     1     1     1     1     1     1     1     1
Chr1SNP4      1     1     1     1     1     1     1     1     1
Chr1SNP5      1     1     1     1     1     1     1     1     1
Chr1SNP6      1     1     1     1     1     1     1     1     1
Chr1SNP7      1     1     1     1     1     1     1     1     1
Chr1SNP8      1     1     1     1     1     1     1     1     1
Chr1SNP9      1     1     1     1     1     1     1     1     1
```

7.14 get.time.point

This function to extract the time point of generation – this is mostly applicable when using the web-based application since there, the first possible time point of generation is automatically calculated.

```
> get.time.point(population , gen=1:3)
  M1_1   M2_1   M3_1   M4_1   M5_1   M6_1   M7_1   M8_1   M9_1  M10_1  M11_1  M12_1  M13_1  M14_1
     0      0      0      0      0      0      0      0      0      0      0      0      0      0
 M15_1  M16_1  M17_1  M18_1  M19_1  M20_1  M21_1  M22_1  M23_1  M24_1  M25_1  M26_1  M27_1  M28_1
     0      0      0      0      0      0      0      0      0      0      0      0      0      0
...

 M33_2  M34_2  M35_2  M36_2  M37_2  M38_2  M39_2  M40_2  M41_2  M42_2  M43_2  M44_2  M45_2  M46_2
     0      0      0      0      0      0      0      0      0      0      0      0      0      0
 M47_2  M48_2  M49_2  M50_2   M1_3   M2_3   M3_3   M4_3   M5_3   M6_3   M7_3   M8_3   M9_3  M10_3
     0      0      0      0      1      1      1      1      1      1      1      1      1      1
 M11_3  M12_3  M13_3  M14_3  M15_3  M16_3  M17_3  M18_3  M19_3  M20_3  M21_3  M22_3  M23_3  M24_3
     1      1      1      1      1      1      1      1      1      1      1      1      1      1
```

7.15 get.creating.type

This function extracts the creating type of each individual – this is mostly applicable when using the web-based application of the package. Here following coding is used:

0 – Founder

1 – Selection

2 – Reproduction

3 – Recombination

4 – Selfing

5 – DH-Production

6 – Cloning

7 – Combine

8 – Aging

9 – Split

```
> get.creating.type(population, gen=1:3)
  M1_1   M2_1   M3_1   M4_1   M5_1   M6_1   M7_1   M8_1   M9_1  M10_1  M11_1  M12_1  M13_1  M14_1  M15_1
     0      0      0      0      0      0      0      0      0      0      0      0      0      0      0
 M16_1  M17_1  M18_1  M19_1  M20_1  M21_1  M22_1  M23_1  M24_1  M25_1  M26_1  M27_1  M28_1  M29_1  M30_1
     0      0      0      0      0      0      0      0      0      0      0      0      0      0      0
...

 W31_2  W32_2  W33_2  W34_2  W35_2  W36_2  W37_2  W38_2  W39_2  W40_2  W41_2  W42_2  W43_2  W44_2  W45_2
     2      2      2      2      2      2      2      2      2      2      2      2      2      2      2
 W46_2  W47_2  W48_2  W49_2  W50_2   M1_3   M2_3   M3_3   M4_3   M5_3   M6_3   M7_3   M8_3   M9_3  M10_3
     2      2      2      2      2      1      1      1      1      1      1      1      1      1      1
```

7.16 get.cullingtime

This function extracts the time of culling of each individual – this is mostly applicable when using the web-based application of the package.

7.17 get.time.point

This function extracts the time the selected gen/database/cohorts were generated.

7.18 get.age.point

This function extracts the time THIS copy of the individual selected in gen/database/cohorts were generated. Usually the same as get.time.point unless copy.individual was used.

7.19 get.individual.loc

Function to derive the position in the stored population-list.

```
> get.individual.loc(population, gen=1)
     generation sex individual nr.
M1_1          1   1              1
M2_1          1   1              2
M3_1          1   1              3
M4_1          1   1              4
M5_1          1   1              5
```

7.20 get.pheno.off

Function to extract the average phenotype of the offspring of individuals selected in gen/database/cohorts.

7.21 get.pheno.off.count

Function to extract the number of observations used to derive the average phenotype of offspring in *get.pheno.off()*.

7.22 get.selectionbve

Function to extract the breeding value used for selection in the last breeding value estimation for selected gen/database/cohorts. Mostly used in case of multi-trait models

7.23 get.pca

This function will generate a PCA for the individuals in selected groups (**gen/database/cohorts**). On default the first two (1:2) principle components will be plotted. Other components can be selected via **components**. On default, individuals are colored via their group (**coloring** = "group"). Alternatively, coloring can be done according to the individual sex ("sex"), class ("class"), original class at the point of generation ("genclass") or just the same color for all ("plain"). Alternatively, it is also possible to provide a vector with the length of the number of individuals in coloring, which will then use these individually chosen colors for each respective individual.

```
data("ex_pop")
get.pca(ex_pop, gen=1:2)
```
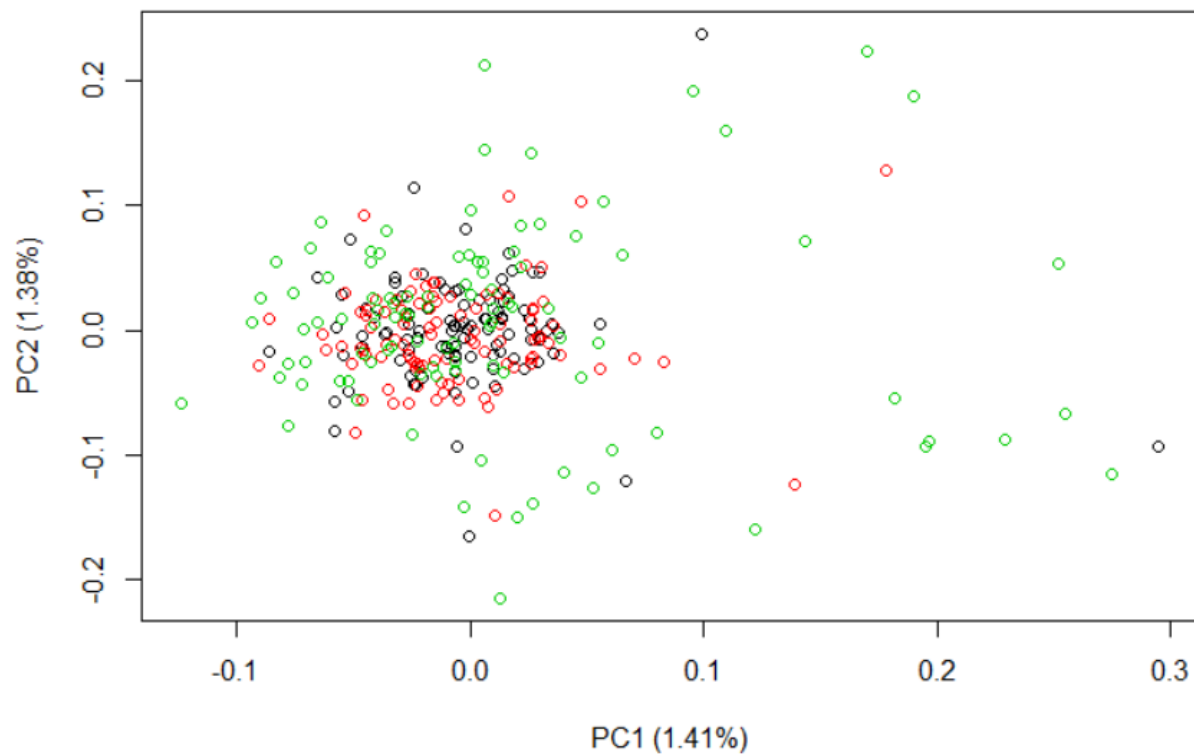
7.24 get.admixture

This function will generate an admixture plot for the individuals in selected groups (**gen/database/cohorts**). For this, the R-package alstructure is used. The number of dimensions to include is automatically estimated based on the alstructure algorithm, but can also be manually set via the parameter **d**. To sort individuals based on their contributions to the respective dimensions set **sort** to TRUE. To exclude individuals with low contributions from the sorting, use **sort.cutoff**.

```
data(ex_pop)
get.admixture(ex_pop, gen=4:6, d=3, sort=TRUE)
```
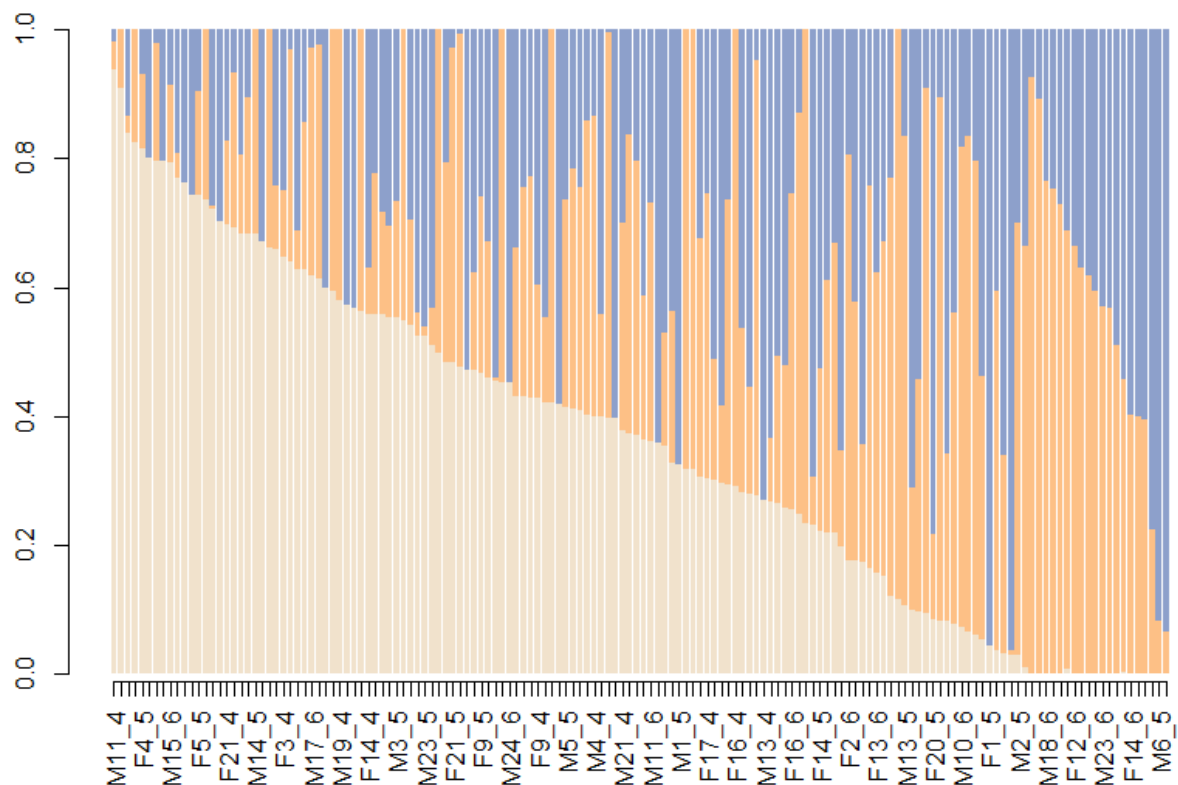
7.25 get.dendrogram

This function will generate a dendrogram plot for the individuals in selected groups (**gen/database/cohorts**). For this, the R-package NAM is used. The used genetic distance can be selected via the parameter method (default: "Nei", alt: "Rogers", "Prevosti", "Modified Rogers").

```
data(ex_pop)
get.dendrogram(ex_pop, gen=2)
```


Dendrogram based on Nei's genetic distances
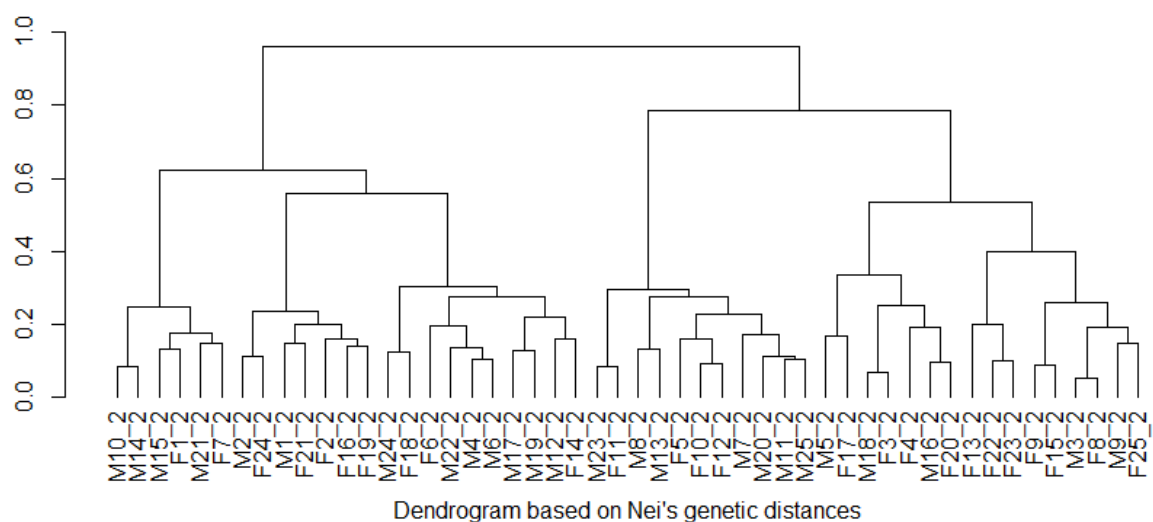
7.26 get.dendrogram.heatmap

This function will generate a dendrogram plot for the individuals in selected groups (**gen/database/cohorts**) including a heatmap for the different traits selected via the parameter **traits** and **type** (default: "pheno", alt: "bv", "bve"). For this, the R-package NAM is used. The used genetic distance can be selected via the parameter method (default: "Nei", alt: "Rogers", "Prevosti", "Modified Rogers").

```
population <- creating.diploid(nsnp=1000, nindi=40, n.additive = c(100,100,100),
            shuffle.cor = matrix(c(1,0.8,0.2,0.8,1,0.2,0.2,0.2,1), ncol=3),
            shuffle.traits = 1:3)
population <- breeding.diploid(population, phenotyping = "all", heritability = 0.5)
get.dendrogram.heatmap(population, gen=1, type="pheno")
```


7.27 get.dendrogram.trait

This function will generate a dendrogram plot for distribution of trait values in selected groups (**gen/database/cohorts**) for the different traits selected via the parameter **traits** and **type** (default: "pheno", alt: "bv", "bve"). For this, the R-package NAM is used. The used genetic distance can be selected via the parameter method (default: "Nei", alt: "Rogers", "Prevosti", "Modified Rogers").
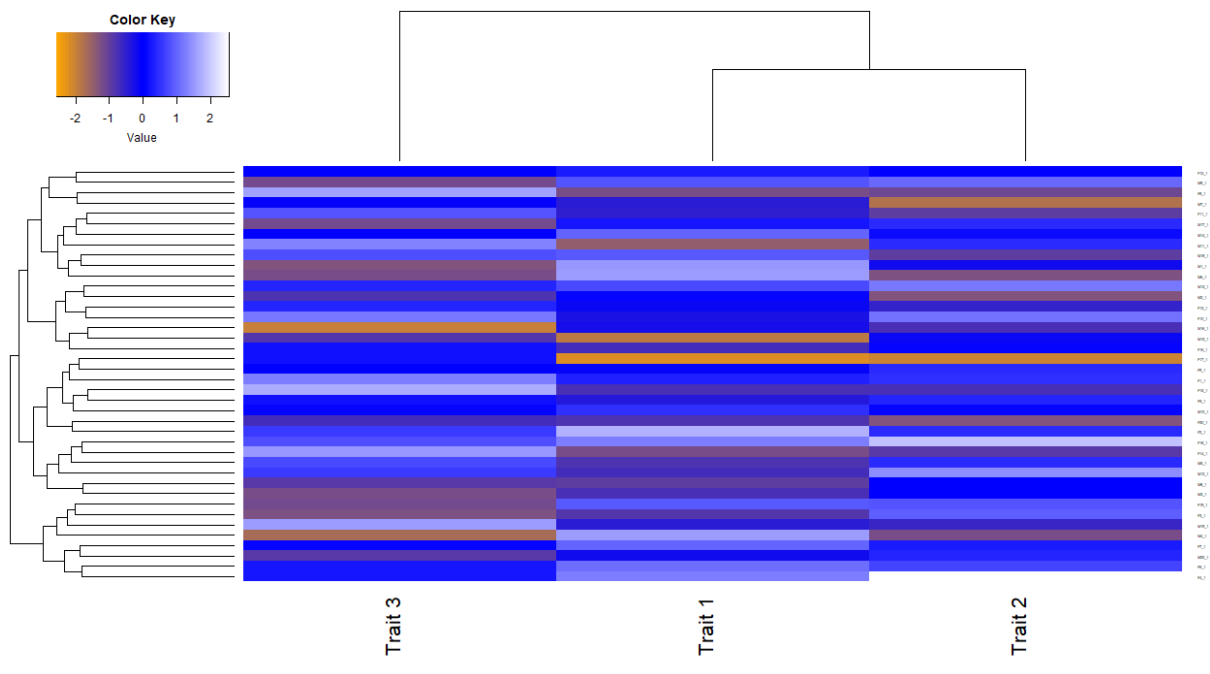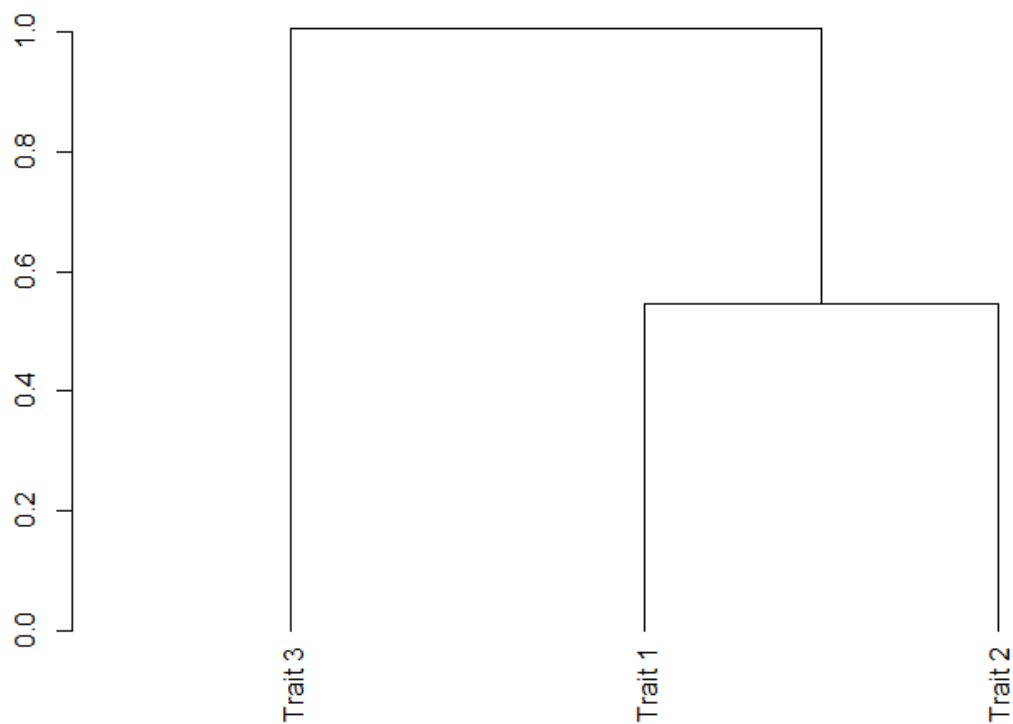
```
population <- creating.diploid(nsnp=1000, nindi=100, n.additive = c(100,100,100),
            shuffle.cor = matrix(c(1,0.8,0.2,0.8,1,0.2,0.2,0.2,1), ncol=3),
shuffle.traits = 1:3)
population <- breeding.diploid(population, phenotyping = "all", heritability = 0.5)
get.dendrogram.trait(population, gen=1, type="pheno")
```

Dendrogram based on phenotypic distances

7.28 get.phylogenetic.tree

This function will generate a phylogenetic tree for selected groups (**gen/database/cohorts**). For this, the R-packages NAM and phylogram are used. The distance measure used can be selected via the parameter **method** (default: "Nei", alt: "Rogers", "Prevosti", "Modified Rogers"). A circular layout tree can be build by setting the parameter **circular** to TRUE. Ward distance is used in the clustering.

```
data(ex_pop)
get.phylogenetic.tree(ex_pop, gen=1, circular=FALSE)
```
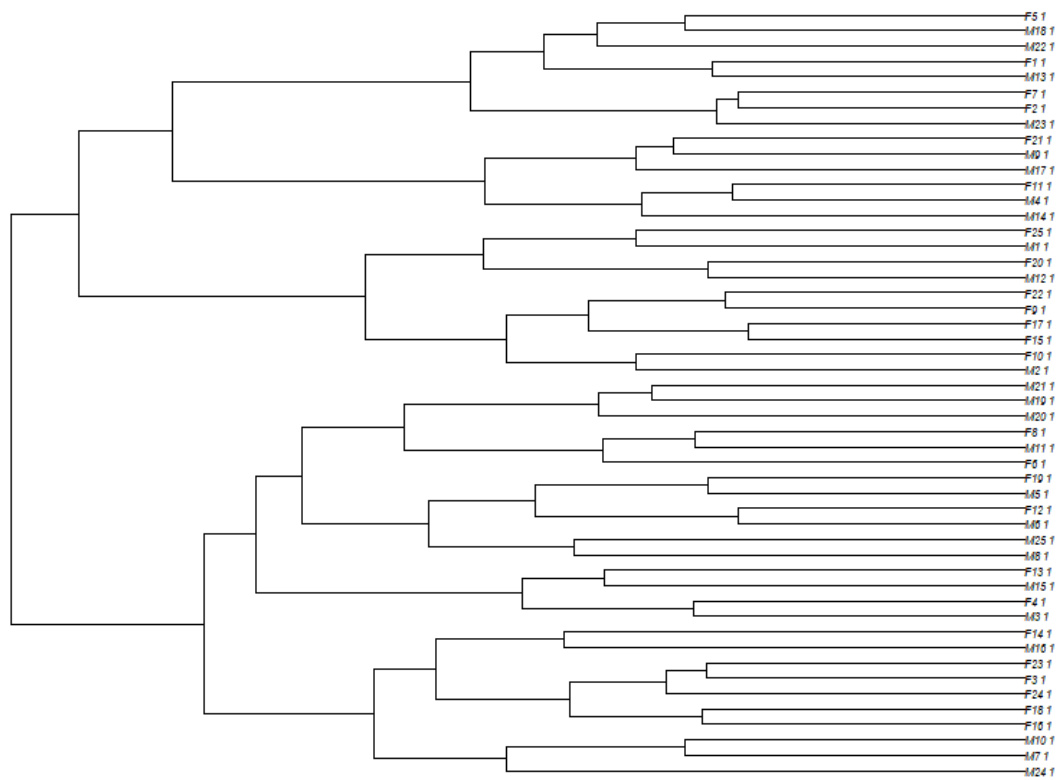
```
get.phylogenetic.tree(ex_pop, gen=1, circular=TRUE)
```
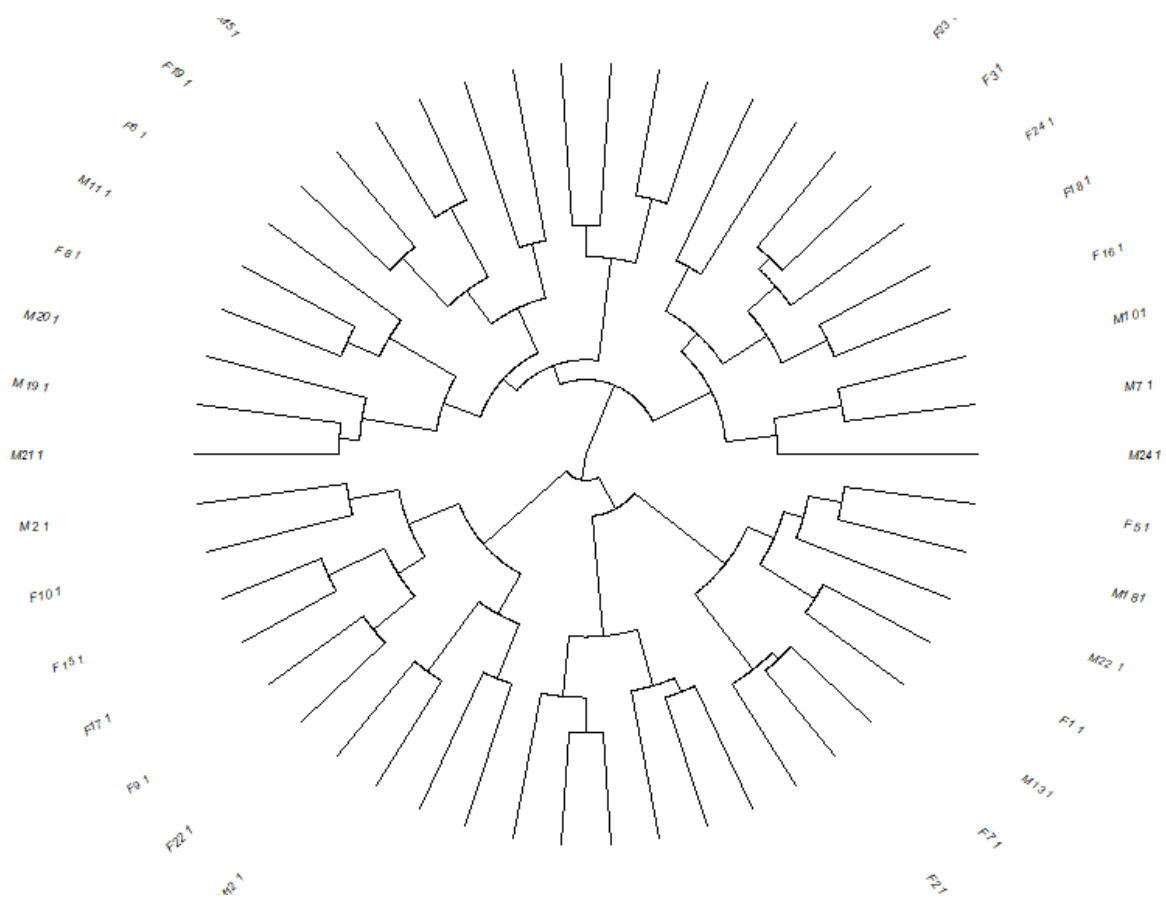
7.29 get.distance

This function allows to compute the genetic distance between two populations given via **gen1/database1/cohort1** and **gen2/database2/cohort2**. For more than two populations provide each database as a list element in **gen/database/cohorts.list**. The distance measure used can be selected via the parameter **type** (default: "nei", alt: "cavalla", "reynold", "nei_minimum", "nei_distance"). To selected which marker are used, the parameter **marker** can be used (default: "alt") or statistics can be provided **per.marker**.

7.30 get.effect.freq

To extract the allele frequency at all QTLs to a selected group of individuals (gen/database/cohorts) use this function. To sort markers along the genome set sort to TRUE.

7.31 get.selectionindex

Function to extract the last selection index used to form a breeding value for selected gen/database/cohorts. In particular relevant in case index weights are scaled by reliability factors as in (Miesenberger 1997).

7.32 get.vcf

Function to export genomic data in a vcf-file. The groups to include in the file can be selected via **gen/database/cohorts** and the path to write to is selected in **path**. To replace all non-genotyped entries with "./." set **non.genotyped.as.missing** to TRUE.

```
 1    ##fileformat=VCFv4.2
 2    ##filedate=20200203
 3    ##source='MoBPS_1.4.65'
 4    ##FORMAT=<ID=GT,Number=1,Type=String,Description='Genotype'>
 5    #CHROM  POS ID  REF ALT QUAL    FILTER  INFO    FORMAT  M1_2    M2_2    M3_2    M4_2    M5_2    M6_2
 6    1   500000  Chr1SNP1    A   C   .   PASS    .   GT  0|0 1|1 0|0 1|1 1|0 0|0 0|0 0|0 1|0 0|0 0|1 0|1 0|1
 7    1   1000000 Chr1SNP2    A   C   .   PASS    .   GT  1|1 0|1 1|0 0|1 1|0 1|0 1|1 1|1 0|1 0|0 1|1 1|1 1|1
 8    1   1500000 Chr1SNP3    A   C   .   PASS    .   GT  0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0
 9    1   2000000 Chr1SNP4    A   C   .   PASS    .   GT  1|0 1|1 1|1 0|1 1|0 0|1 1|1 0|0 1|1 1|1 1|1 0|1 0|1
10    1   2500000 Chr1SNP5    A   C   .   PASS    .   GT  1|1 1|1 1|0 0|1 1|1 1|1 1|0 1|0 1|1 0|1 1|0 0|0 1|0
11    1   3000000 Chr1SNP6    A   C   .   PASS    .   GT  1|1 1|0 1|1 1|1 1|1 1|1 1|1 1|0 0|0 0|1 1|0 1|1 0|0
12    1   3500000 Chr1SNP7    A   C   .   PASS    .   GT  1|1 0|0 0|0 0|1 0|1 1|1 1|0 1|1 0|1 0|1 0|0 0|0 1|0
13    1   4000000 Chr1SNP8    A   C   .   PASS    .   GT  1|1 1|1 1|1 1|1 1|1 1|1 1|1 0|1 1|1 1|1 1|1 1|1 1|1
14    1   4500000 Chr1SNP9    A   C   .   PASS    .   GT  0|0 0|1 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0
15    1   5000000 Chr1SNP10   A   C   .   PASS    .   GT  0|1 1|0 0|0 0|0 0|0 0|1 0|0 0|0 0|0 0|1 0|0 0|0 0|0
16    1   5500000 Chr1SNP11   A   C   .   PASS    .   GT  0|0 0|0 0|1 0|0 0|1 1|0 0|1 0|1 1|0 0|0 0|1 0|0 0|0
17    1   6000000 Chr1SNP12   A   C   .   PASS    .   GT  0|1 1|1 1|1 1|1 0|1 1|1 0|1 1|0 0|1 1|1 0|1 1|0 1|1
18    1   6500000 Chr1SNP13   A   C   .   PASS    .   GT  1|1 1|1 1|0 1|1 1|0 1|1 1|0 1|1 1|1 1|1 1|0 1|0 1|0
19    1   7000000 Chr1SNP14   A   C   .   PASS    .   GT  0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0
```

7.33 get.pedmap

Function to export genomic data in a ped and a map-file (PLINK format (Purcell et al. 2007)). The first of the two columns of each marker is representing the first haplotype of the individual. The groups to include in the file can be selected via **gen/database/cohorts** and the path to write to is selected in **path**. To replace all non-genotyped entries with N set **non.genotyped.as.missing** to TRUE.

```
1    1 M1_2 M12_1 F28_1 1 0 A A C C A A A C C C C C C C C C A A C A
2    1 M2_2 M46_1 F41_1 1 0 C C C A A A C C C C A C A A C C C A A C
3    1 M3_2 M15_1 F15_1 1 0 A A A C A A C C A C C C A A C C A A A A
4    1 M4_2 M1_1 F18_1 1 0 C C C A A A C A C A C C C A C C A A A A A
5    1 M5_2 M63_1 F51_1 1 0 A C A C A A A C C C C C C A C C A A A A
6    1 M6_2 M97_1 F65_1 1 0 A A A C A A C A C C C C C C C C A A C A
7    1 M7_2 M85_1 F28_1 1 0 A A C C A A C C A C C C A C C C A A A A
8    1 M8_2 M63_1 F7_1 1 0 A A C C A A A A A C A C C C C A A A A A C
9    1 M9_2 M2_1 F52_1 1 0 A C C A A A C C C C A A C A C C A A A A A
10   1 M10_2 M97_1 F27_1 1 0 A A A A A A C C C A C A C A C C A A C A
11   1 M11_2 M29_1 F88_1 1 0 C A C C A A C C A C A C A C A A C C A A A A
12   1 M12_2 M39_1 F25_1 1 0 C A C C A A C A A A C C A A C C A A A A
13   1 M13_2 M87_1 F71_1 1 0 C A C C A A C A A C A A A C C C A A A A
14   1 M14_2 M67_1 F92_1 1 0 A A C C A A A A A C C C C A C C A A C C
```

```
1    1 Chr1SNP1 0 500000
2    1 Chr1SNP2 0 1000000
3    1 Chr1SNP3 0 1500000
4    1 Chr1SNP4 0 2000000
5    1 Chr1SNP5 0 2500000
6    1 Chr1SNP6 0 3000000
7    1 Chr1SNP7 0 3500000
8    1 Chr1SNP8 0 4000000
9    1 Chr1SNP9 0 4500000
10   1 Chr1SNP10 0 5000000
11   1 Chr1SNP11 0 5500000
12   1 Chr1SNP12 0 6000000
13   1 Chr1SNP13 0 6500000
14   1 Chr1SNP14 0 7000000
15   1 Chr1SNP15 0 7500000
16   1 Chr1SNP16 0 8000000
17   1 Chr1SNP17 0 8500000
18   1 Chr1SNP18 0 9000000
19   1 Chr1SNP19 0 9500000
20   1 Chr1SNP20 0 10000000
21   1 Chr1SNP21 0 10500000
22   1 Chr1SNP22 0 11000000
```

7.34 get.map

Function to export a matrix with map-information (1st column: Chromosome, 2nd column: SNP-name, 3rd column: BP-position, 4th column: Morgan-position.

7.35 get.database

Function to merge **gen**, **database,** and **cohorts** –info into a joint database. On default, the resulting database will have as low number of rows as possible and by that put multiple cohorts into a joint row. To avoid this set **avoid.merging** to TRUE.

```
> get.database(population, gen=1, cohorts="Selected_bulls")
         start end
[1,] 1 1      1  50
[2,] 1 2      1  50
[3,] 3 1      1  10
```

Instead of providing **gen/database/cohort** it is also possible to provide individual IDs in the **id** parameter. Available animals will subsequently be screened on where selected IDs are located. On default, the first version of an individual will be given – to display all copies set **id.all.copy** to TRUE, to display the last copy set **id.last** to TRUE. This is only relevant if **copy.individual** is used as this allows the generation of multiple version of an individual with the same ID.

7.36 get.ngen

This function will calculate the number of generations in a population list.

7.37 get.size

This function will calculate the number of male and female individuals (1st / 2nd column) in each generation.

8 Importing information to the population-list

8.1 insert.bve

To manually insert breeding values (type="bve"), true genetic values (type="bv") or phenotypes (type="pheno") use the function insert.bve. Output is a modified population list. In case new phenotypes are observed this is counted as **count** observations. This will only increase the number of phenotypes generated for a trait unless **count.only.increase** is set to FALSE. In case bve are changed it is assumed that genotyping was necessary unless **count** is set to 0. This is only relevant for economic calculations.

New observations are entered in the parameter **bves** with the first column coding the individual and the others containing values for the traits:

```
> bves
     Individual Name Trait 1 Trait 2
[1,] "M1_1"          "101.5" "104.2"
[2,] "M2_1"          "102"   "98.9"
[3,] "M3_1"          "99.7"  "98.4"
[4,] "M4_1"          "98.2"  "101.2"
[5,] "M5_1"          "103.8" "101.1"
> population <- insert.bve(population, bves)
```

The structure of individual names is the same in all export functions (sex ["M"/"F"], individual number [1,2,…], and generation [1,2,…]). It is recommended to just use the names as they are exported via *get.geno()* etc..

8.2 set.class

This function can be used to manually change the class of selected **gen/database/cohorts** to the class **new.class** in an already existing population list.

```
population <- set.class(ex_pop, database=cbind(1,1), new.class = 2)
```

9 Utility functions & advanced features

In this section a set of utility function is described that should help you get a better overview of your population list. This list is constantly growing and we are always happy for suggestions on additional function to add to our package.

9.1 set.default

This function can be used to automatically initialize selected parameters in *breeding.diploid()*. The parameter to initialize is selected via **parameter.name** and the value is given in **parameter.value**. To remove a specific initialization use **parameter.remove** or remove all previously set parameters via setting **reset.all** to TRUE.
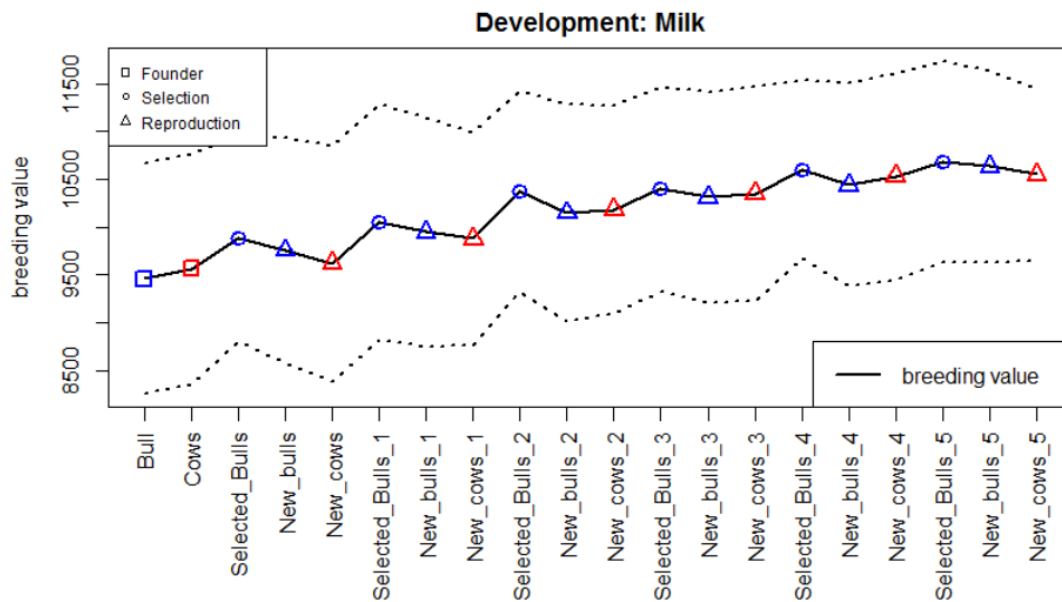
9.2 bv.development

This function will generate a plot showing the development of breeding values and phenotypes over generations. 95% confidence bands are included in a dotted line:

Which groups to display is selected via the parameters **gen**, **database** & **cohorts**. In case the user interface was used to generate the population list set **json** to TRUE to automatically display all selected cohorts. Confidence bands are drawn for "bv" (1), "bve" (2) & "pheno"(3) – to change the quantile use the parameter **quantile** (default: 0.95), to exclude selected the for with to draw a confidence band via the parameter **confidence** (default: c(1,2,3)). Groups with only zeros are ignored on default – if you want lines to be included for all selected cohorts set **ignore.zero** to FALSE.

To display the time point, the creating type, the sex, and cohort name set **display.time.point**, **display.creating.type**, **display.cohort.name**, **display.sex** to TRUE. In case the generating interface between groups is highly heterozygous it might be useful to use **equal.spacing** between displayed cohorts. To not display the line displaying a long-term trend of the breeding values set **display.line** to FALSE.

```
population <- json.simulation(total = ex_json)
bv.development(population, json=TRUE, bvrow=1, confidence = 1, development = 1,
               display.creating.type = TRUE, display.sex = TRUE,
               display.cohort.name = TRUE)
```

Development: Milk

9.3 bv.development.box

This function will generate a plot displaying the development of breeding values with a boxplot for each selected **gen**, **database**, **cohorts**. In case the user-interface was used to generate the population set **json** to TRUE to automatically display all selected cohorts. To only display a subset of trait set **bvrow** to those traits. To display phenotypes or breeding value estimations for the individuals instead of breeding values, set **display** to "pheno" / "bve".

In case the user interface was used to generate the population, one can display which cohorts where generated by which other cohort (via selection or reproduction) by setting **display.selection** and **display.reproduction** to TRUE.

```
population <- json.simulation(total = ex_json)
bv.development.box(population, json = TRUE, bvrow = 1)
```
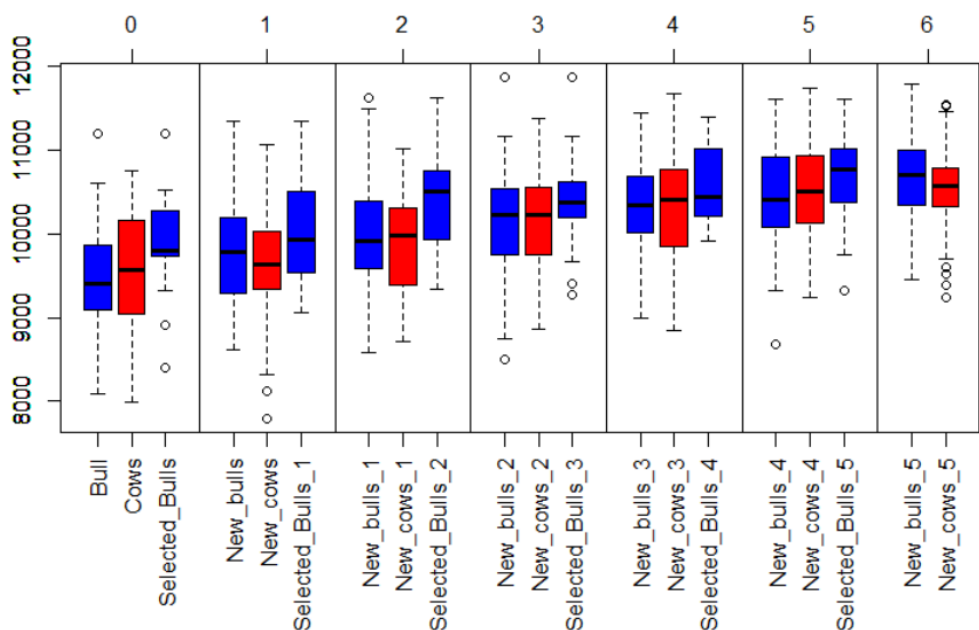
9.4 Relationship between individuals and inbreeding

9.4.1 Definition in MoBPS

The definition of kinship in MoBPS is in accordance with (Malécot 1948). This formally means the probability of a randomly chosen allele (or position in the genome) from two individuals to be in IBD. For both individuals which of the two chromosome pairs is chosen is also done at random. This at first might sound unnecessarily technical – thus let's consider a small example with a genome with 5 alleles. Instead of alleles for each marker, we will be using numbers to indicate from which founder a specific segment is stemming:

Individual 1 chromosome pair 1	1	1	1	1	1
Individual 1 chromosome pair 2	1	2	2	2	2
Individual 2 chromosome pair 1	1	3	3	3	3
Individual 2 chromosome pair 2	1	4	4	4	4

When looking at locus 1, independent of which of the chromosome pairs of both individuals is chosen, this will always result in a case of IBD. In locus 2 – 5 we will be never in IBD. As the chance to choose each allele is 20%, the resulting kinship is 0.2.

Secondly consider the following case:

Individual 1 chromosome pair 1	1	1	1	1	1
Individual 1 chromosome pair 2	2	2	1	2	2
Individual 2 chromosome pair 1	1	1	1	3	3
Individual 2 chromosome pair 2	4	4	4	4	4

In locus 1 and 2, we will have a case of IBD when looking at the first chromosome pair of both individuals 1 and 2, which will happen with a probability of 25% (0.5^2). In locus 3, it does not matter which of the chromosome pairs of individual 1 to look at as long as chromosome pair 1 of individual 2 is chosen. Thus, the probability here is 50%. Locus 4 and 5 will never result in a case of IBD. The resulting kinship is 0.2 * 0.25 + 0.2*0.25 + 0.2 * 0.5 + 0.2 * 0 + 0.2* 0 = 0.2.

In both cases, kinship is the same but the areas of the genome causing IBD are different. Areas under stronger selection will typically have a higher probability of IBD occurring, thus a kinship of 0.2 does not mean that 20% of the genome is in IBD, but is much more nuanced.

You can also calculate kinship within an individual. The kinship of an individual will always be at a minimum 0.5 as the probability to sample the same chromosome pair twice is 50%. The inbreeding coefficient of an individual can be calculated by calculating: 2* (kinship – 0.5).

In MoBPS there are different ways to calculate the kinship that will be explained in the following subsections:

9.4.2 kinship.exp

The *kinship.exp()* uses pedigree information to compute the expected kinship, e.g. a parent / offspring pair will have an average kinship of 0.25, a grandparent/offspring pair will have an average kinship of 0.125. Note that the expected kinship will be 0.5 – times the pedigree relationship matrix. Note that under selection, actually resulting kinships can differ, e.g. when 2 of 100 siblings are selected, these are typically more related – which can add up over multiple generations.

To control how many generations back are used, the parameter **prev.gen** can be used. On default, it is assumed all individuals before are unrelated unless specified by the use of *add.founder.kinship()*. Alternatively, one can provide a **kinship.matrix** for the individuals of the first generation via **first.individuals**. It should be noted that there are more efficient ways to derive a pedigree matrix than this as a pedigree matrix usually contains a high number of zeros and this will output the entire matrix but this should still do easily the job for < 100.000 individuals on most systems – alternatively, one can export the pedigree via *get.pedigree()* and use tools outside of R.

```
> kinship.exp(population = population, database = cbind(5,1,1,6))
Derive pedigree-matrix based for  6  individuals based on  77  individuals.
            1501       1502       1503       1504       1505       1506
1501 0.5000000 0.0000000 0.0000000 0.0078125 0.0156250 0.0000000
1502 0.0000000 0.5000000 0.0312500 0.0000000 0.0078125 0.0156250
1503 0.0000000 0.0312500 0.5000000 0.0312500 0.0078125 0.0078125
1504 0.0078125 0.0000000 0.0312500 0.5000000 0.0078125 0.0000000
1505 0.0156250 0.0078125 0.0078125 0.0078125 0.5000000 0.0078125
1506 0.0000000 0.0156250 0.0078125 0.0000000 0.0078125 0.5000000
```

9.4.3 inbreeding.emp / kinship.emp / kinship.emp.fast

As MoBPS does store all points of recombination it is possible to directly calculate the kinship, by backtracking of chromosome segments and comparing if and what segments of individuals are in IBD. In addition to the standard input of the **population** list and selected **gen**, **database**, **cohorts** it is also possible to provide the lists within the population list ($breeding) for each individual to consider via the **animals** parameter. This calculation is explicitly not limited to just the simulated SNPs but also considers everything in-between. This calculation does only consider the founder of each chromosome segment and explicitly not mutations (even after a mutation in the SNP founder stays the same!).

Inbreeding.emp() will calculate this for all individuals. *kinship.emp()* will calculate this for all pairwise relationships. As this procedure is computationally costly and a kinship matrix can become large, it can oftentimes be sufficient to not evaluate all pairs, but just randomly sample some pairs to evaluate. This can be done in *kinship.emp.fast()* and will evaluate **ibd.obs** pairwise relationships (off-diagonal) and **hbd.obs** relationship with the individual with itself (main-diagonal). Note that the defaults of both **ibd.obs** and **hbd.obs** are chosen very small (200/50) to ensure fast computations and will be unbiased – in particular for large cohorts, increasing will however help to reduce estimation variance!

```
> kinship.emp(population = population, database = cbind(5,1,1,6), sym = TRUE)
             M1_5        M2_5        M3_5        M4_5        M5_5        M6_5
M1_5 0.500000000 0.00000000 0.000000000 0.005644235 0.018530287 0.000000000
M2_5 0.000000000 0.50000000 0.038807169 0.000000000 0.012979156 0.012041486
M3_5 0.000000000 0.03880717 0.500000000 0.045487471 0.003175374 0.002525169
M4_5 0.005644235 0.00000000 0.045487471 0.500000000 0.017167699 0.000000000
M5_5 0.018530287 0.01297916 0.003175374 0.017167699 0.500000000 0.008303114
M6_5 0.000000000 0.01204149 0.002525169 0.000000000 0.008303114 0.500000000
```

```
> kinship.emp.fast(population = population, database = cbind(5,1,1,6))
[1] 0.01097741 0.50000000
```

9.4.4 kinship.development

Function to display the development of kinship over different **gen**, **database**, **cohorts**. Internally *kinship.emp.fast()* is used and the same optional parameters can be used to improve computation time / accuracy.

```
population <- json.simulation(total = ex_json)
kinship.development(population, json = TRUE, display.cohort.name = TRUE)
```

Kinship development
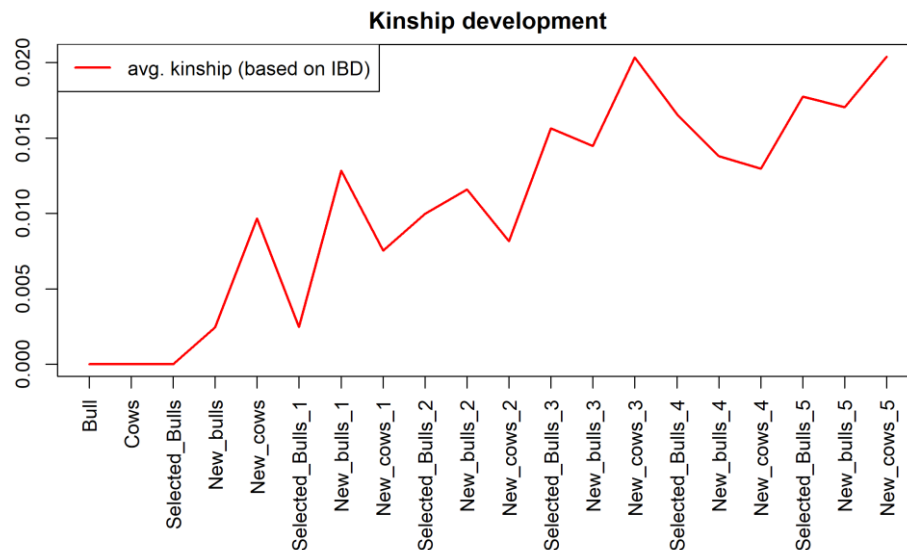
9.5 analyze.population

With this function, one can analyze the allele frequency of a specific marker over time. Select the marker to analyze via parameters **chromosome** & **snp**. To selected with generations to compare use **gen**, **database**, **cohorts**.

```
population <- json.simulation(total = ex_json)
analyze.population(population,5,2, gen=1:8)
```

```
> analyze.population(population,5,2, gen=1:8)
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]   198   27  220  200  207  151  162  143
[2,]   214   12  278  278  250  264  269  239
[3,]    88   11   52   72   93  135  119  118
```
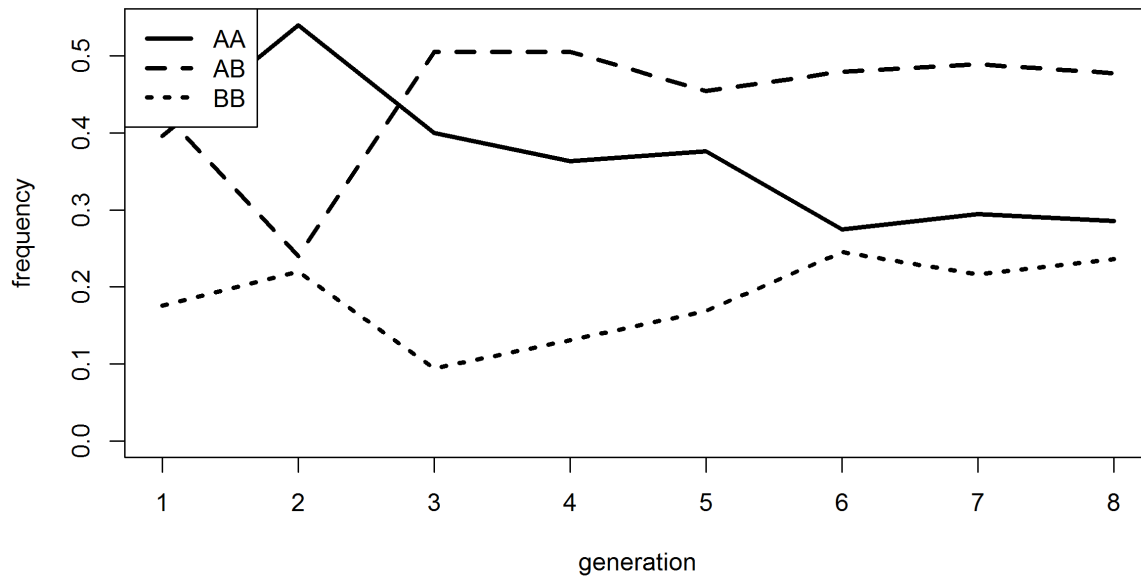
9.6 new.base.generation

With a rising number of generations, the number of points of recombinations to store is increasing. For efficient storing it can make sense to compute and store haplotypes for a later generation and use those individuals as a new founder generation. For this use *new.base.generation()* and select the new base generation via the parameter **base.gen**. To further reduce memory needs and computation time you can additionally delete data of previous generations via **delete.previous.gen**, **delete.breeding.totals** and **delete.bve.data**.

9.7 founder.simulation

This function can be used to simulate a random mating structure for a high number of generations and will lead to an output that can be used as an input for **dataset** in *creating.diploid()*. The number of individuals can be controlled via **nindi** and the number of SNPs to consider via **nsnp**. In case the number of individuals in the final dataset is supposed to be higher than in the build-up generations **nfinal** can be used to clarify. The share of female individuals in both **nindi** and **nfinal** is controlled via **sex.quota / sex.quota.final**. The total number of generations to simulated in controlled via **n.gen**. Furthermore, all parameters to design a genomic map are given in the same way as in *creating.diploid()*. In case not only the final haplotypes, but the population list itself is of interest set **big.output** to TRUE which will provide you with the genomic map, the population list and a pedigree relationship matrix for the final generation.

Finally, an automatic estimation of the LD-decay and allele frequency spectrum can be generated by setting the parameter **plot** to TRUE.

Note that this is "only" modelling a random mating architecture and in case a population structure with a more complex background is wanted, a manual generation with *creating.diploid()* and *breeding.diploid()* is required.

```
population <- founder.simulation(nindi=100, nsnp=1000, n.gen=100)
```

```
Successfully generated cohort: Cohort_101_M
Database position: 101 (gen), 1 (sex), 1 (first), 50 (last).
Successfully generated cohort: Cohort_101_F
Database position: 101 (gen), 2 (sex), 1 (first), 50 (last).

Start estimation of LD decay / effective population size.
403 of 1000 markers are fixated.
Effective population size is estimated to be around 119.
```
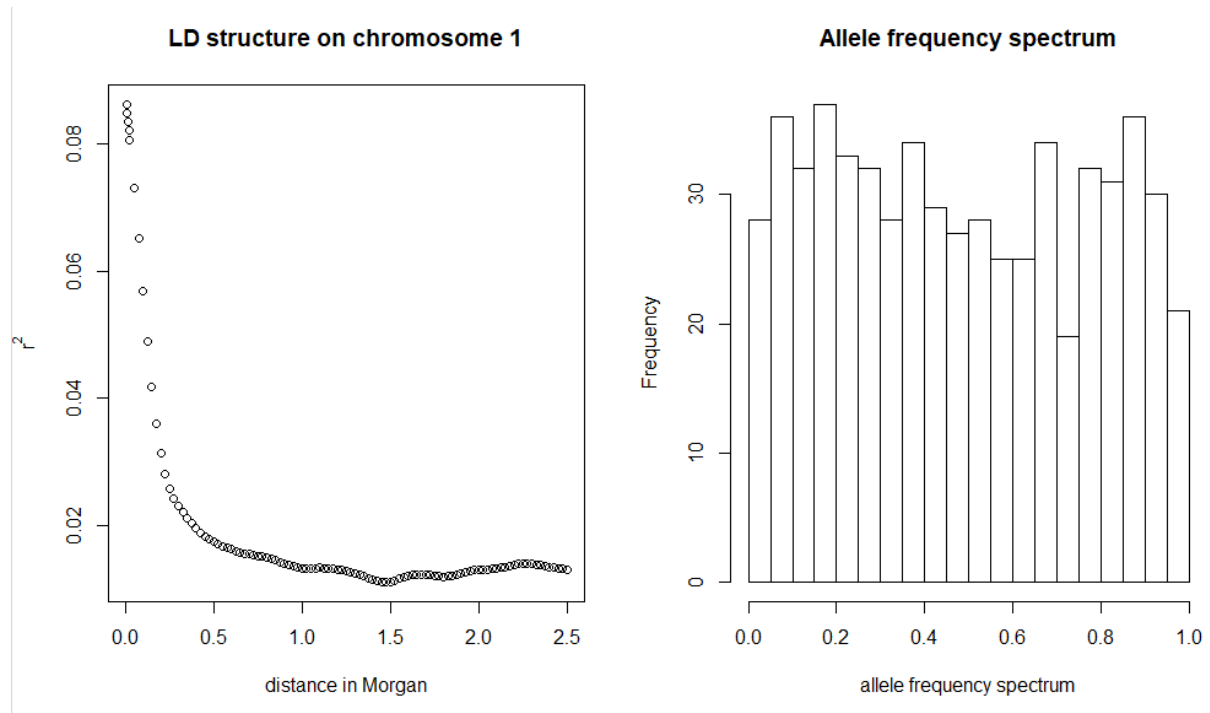

9.8   pedigree.simulation

This function can be used to simulate a given pedigree. In the pedigree the first column contains the ID of the offspring, second/third column the ID of the father / mother. Optionally one can provide the earliest generation of generation in the fourth column and the sex of each individual in the fifth column. If not specified all individuals will be generated at the earliest possible time and in a joined group per generation – default is all individuals are male. We do not perform any pedigree clean-up in this function!

All parameter in creating.diploid() regarding genome structure and traits are also available for pedigree.simulation().

```
pedigree <- matrix(c(1,0,0,
                     2,0,0,
                     3,0,0,
                     4,1,2,
                     5,1,3,
                     6,1,3,
                     7,1,3,
                     8,4,6,
```
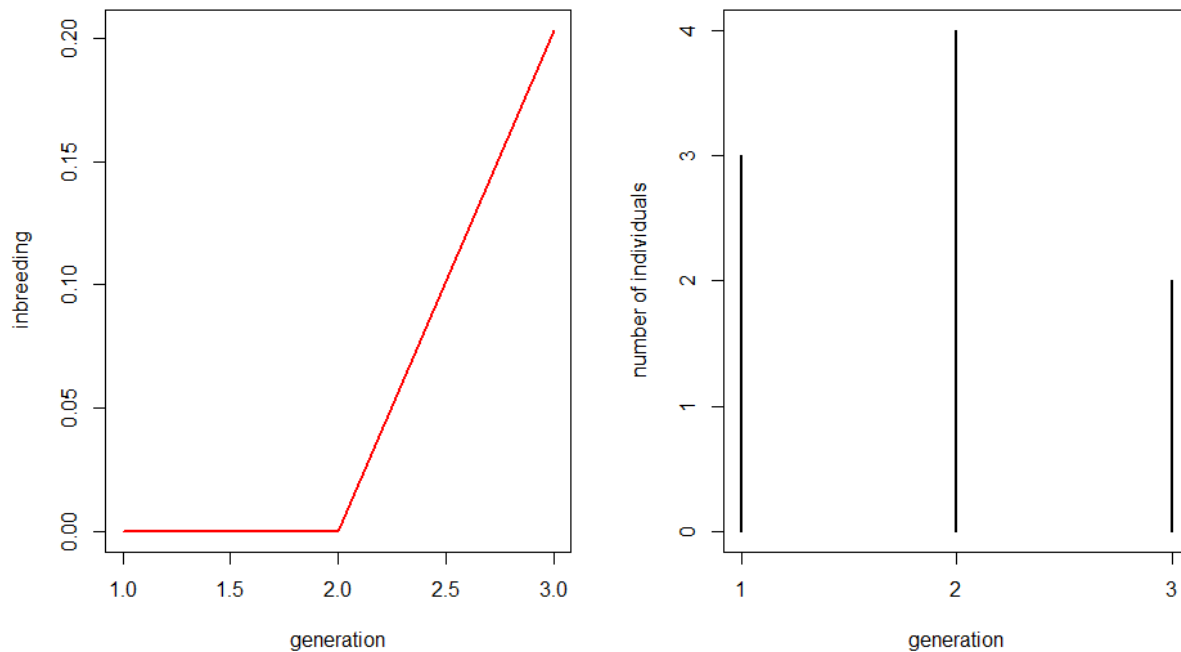
```
                    9,4,7), ncol=3, byrow=TRUE)

population <- pedigree.simulation(pedigree, nsnp=1000)
```


9.9 creating.trait

With this function one can generate additional traits for the base population without the need to add genetic datasets. Functionality is the same as *creating.diploid()* otherwise. For details on the use we refer to section 4.3.

9.10 combine.traits

With this function multiple traits can be linked and will then be handled as observations for the same trait in a breeding value estimation (e.g. for GxE or slightly different genetic backgrounds in different breeds).

9.11 bv.standardization

This function can be used to scale QTL effects to obtain a certain mean (**mean.target**) and genomic variance (**var.target**) in a set of preselected groups (**gen/database/cohorts**). On default, estimated breeding values and phenotypes are not scaled – to active this set **adapt.bve** and **adapt.pheno** to TRUE.

9.12 Merging.cohorts

It is not possible to add individual to an existing cohort via *breeding.diploid()*. This function allows to combine two cohorts into a joint cohort. The name of the new cohorts is provided via **name.cohort** and the cohorts that should be combined are provided via **cohorts**. Nonetheless, by design within MoBPS a cohort has to have uniform sex and generation. Thus, only cohorts in the same generation

that very generated directly after another can be merged. To combine other cohorts, use of the combine parameter in *breeding.diploid()* can be used instead. *Merging.cohorts()* avoids the generation of a new copy of the individuals in the cohorts that should be combined.

9.13 creating.phenotypic.transform

This function can be used to generate to automatically apply a transformation function on the generated phenotypes. The transformation function used has to be provided in **phenotypic.transform.function** and the trait to transform is selected via **trait** (default: 1). E.g. to generate a binary trait use:

function(x){y = x>10; return(y)}

To transform all phenotypic observations higher than 10 to 1 and all below to 0. Thresholds to obtain certain probabilities of each phenotype must be manually derived according to the trait mean and variance set via *bv.standardization()* and the entered heritability.

9.14 clean.up

This function can be used to delete recombination points between segments that stem from the same founder strand and therefore are usually uninformative. On default, all individuals will undergo this clean up procedure. To select specific groups for the clean up use **gen/database/cohorts**.

9.15 optimize.cores

The generation of new individuals can use multiple cores. This is however not efficient on all systems, depend on the genome & trait architecture and will lead to higher memory demands. One can run *optimze.cores()* on an existing population list to check the ideal number of cores to be used for the generation.

9.16 ensembl.map

Via this function, genetic maps provided in Ensembl (https://www.ensembl.org/index.html & http://plants.ensembl.org/index.html) can be imported. Internally the package biomaRt is used – for guidelines on how to install this package we refer to https://bioconductor.org/packages/release/bioc/html/biomaRt.html.

The naming of parameters is orientated according to the biomaRt package. Set **dataset** to the dataset you want to access (e.g. for cattle-SNPs: "btaurus_snp") – for a list of possible datasets run this function with **export.datasets** set to TRUE.

To import a subset of all markers use the parameter filter and **filter.values**. To limit the markers to a specific SNP-chip just set **filter.values** to the name of the chip (e.g. **filter.values**="Illumina BovineSNP50 BeadChip"). Names of potential filters for a dataset can be exported by setting **export.filters** to TRUE.

For us, the direct export in R via Ensembl was painfully slow and the package did not always do what we intended it to do. Therefore, we are providing exemplary map files for most common species in the associated R-package MoBPSmaps. For a list of maps that are already included in MoBPS and the associated data package MoBPS_maps we refer to section 14.

```
cattle_map <- ensembl.map(dataset="btaurus_snp", filter.values="Illumina BovineSNP50 BeadChip")
> cattle_map[1:10,]
      Chromosome SNP-ID        bp         M  freq
 [1,] "1"        "rs42778024"  "435963"   NA NA
 [2,] "1"        "rs41609588"  "776231"   NA NA
 [3,] "1"        "rs108982244" "907810"   NA NA
 [4,] "1"        "rs29026917"  "1073496"  NA NA
 [5,] "1"        "rs29015852"  "1150763"  NA NA
 [6,] "1"        "rs108981857" "1566539"  NA NA
 [7,] "1"        "rs108994381" "1695632"  NA NA
 [8,] "1"        "rs41635940"  "1929664"  NA NA
 [9,] "1"        "rs41255293"  "2082139"  NA NA
[10,] "1"        "rs41580909"  "2235492"  NA NA
```

9.17 compute.costs

To calculate the costs of the currently simulated breeding program use the function *compute.costs()*. Currently implemented cost factors include the following:

Cost factor	MoBPS -Parameter	Default
Phenotyping	phenotyping.costs	10
Genotyping	genotyping.costs	100
Housing/Field costs	Housing.costs	0
Fixed costs	fix.costs	0
Annual costs	fix.costs.annual	0
Profit per BV	profit.per.bv	1

Note that all default settings are basically chosen at random and should be modified when analyzing a real breeding program. In case costs/gains between sexes are different, use a vector. To separate between generations use a matrix with each row coding costs/gains per generation.

To only calculate the resulting costs of some generations/cohorts use database/gen.

To model an interest rate set **interest.rate** (default: 1 – meaning i = 0%. We here assume inputs of the form 1 + i) – with costs/gain changed according to a base generation (**base.gen** – default: 1)

```
> compute.costs(population,gen=1:10)
 [1]  -63014.02 -242252.85 -274059.21 -144566.68   12944.92  132774.58  165916.47  299725.18  339646.66  510074.31
```
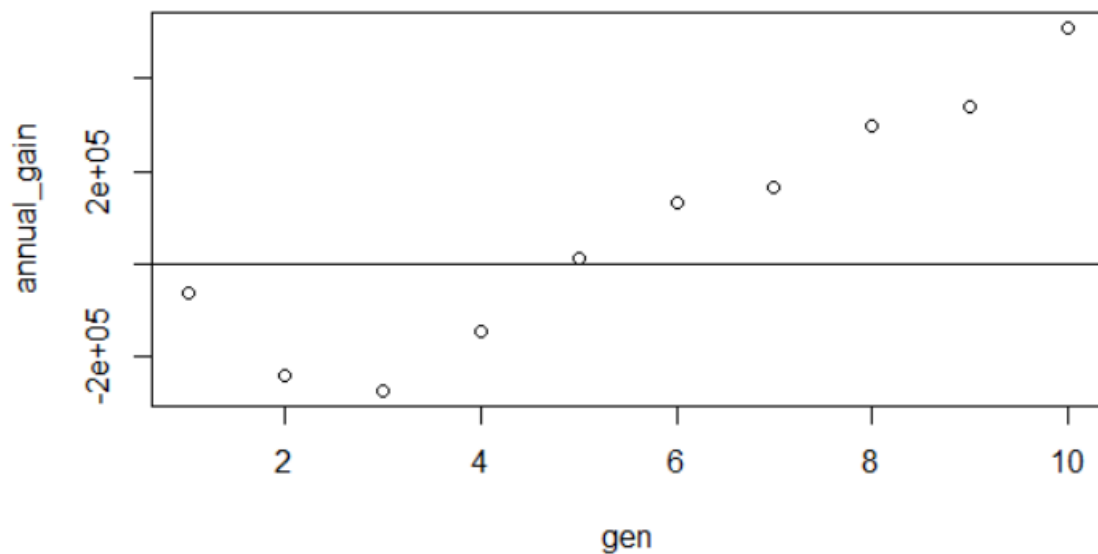
9.18 compute.costs.cohorts

The functionally of *compute.costs.cohorts()* is similar to *compute.costs()* with the added benefit of usability with input parameters provided in our user-interface. Input parameters include **phenotyping.costs**, **genotyping.costs**, **housing.costs**, **fix.costs**, **fix.costs.annual**, **profit.per.bv**, **interest.rate**. In case the user interface is provided considered groups (**gen**, **database**, **cohorts**) are automatically assigned to their time point of creation for discounting.
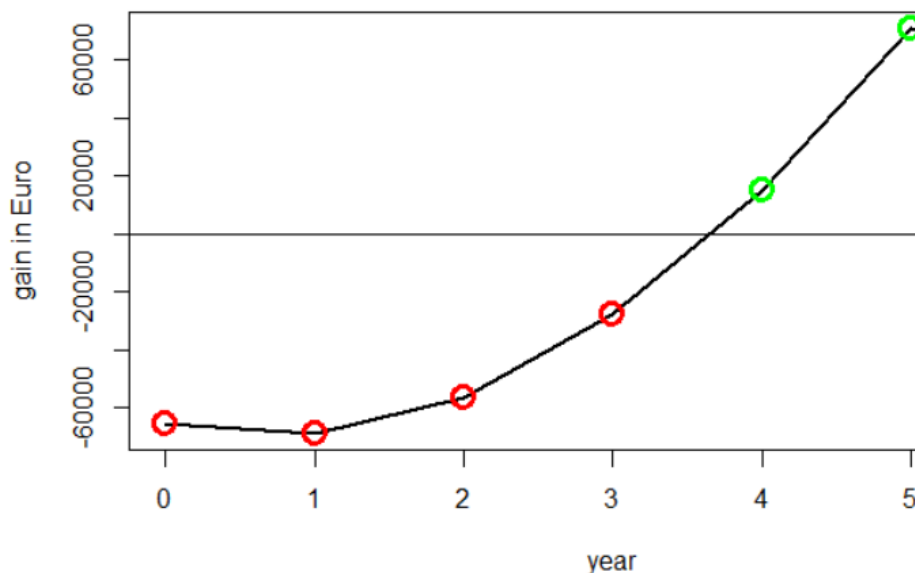
9.19 summary

The population list generated via *creating.diploid()* and *breeding.diploid()* is of class "population". Application of the generic function summary leads to an overview of the population list including the number of individuals, cohorts, the structure of the genome and traits:

```
> summary(pop)
Population size:
Total: 100 Individuals
Of which 50 are male and 50 are female.
There are 2 generations
and 4 unique cohorts.

Genome Info:
There are 5 unique chromosomes.
In total there are 10000 SNPs.
The genome has a total length of 10 Morgan.
No physical positions are stored.

Trait Info:
There is 1 modelled trait.
The trait has underlying QTL
The trait is named: Trait 1
```

9.20 pedmap.to.phasedbeaglevcf

The standard input for genomic data in MoBPS are haplotypes (not genotypes!). In the case of using your own genomic data, it is highly recommended to perform genomic phased before using the dataset as an input. In this function, a routine pipeline to generate a phased dataset is executed. In this pipeline BEAGLE 5.0 (https://faculty.washington.edu/browning/beagle/beagle.html) and PLINK 1.9 (https://www.cog-genomics.org/plink/1.9/) are used. Additional file are generated in a selected directory. Path for all three have to be provided in **beagle_jar**, **plink_dir**, **db_dir**. Input can either be a dataset in PLINK format (**ped_path**, **map_path**) or a vcf-file (**vcf_path**). Defaults are all chosen to work on the webserver for our web-interface (Chapter 0). Phasing can also be directly performed in the web-interface.

10 Data structure of the population list

All information regarding the breeding program is stored in a population list (R-object: list) which is modified by each run of *breeding.diploid()* and *creating.diploid()*. The population list contains matrices, inside of lists, inside of lists, inside of lists, inside of lists, inside of lists, inside of lists (you get the point!) – when understanding the structure behind it is actually not that bad, luckily you do not have to understand the structure behind for most applications since you can use exporting function discussed in section 0.

 The list contains two major parts - $info ((or [[1]])) and $breeding ((or[[2]]):

10.1 $info

$info contains all general information concerning genetic architecture, size of the program and internal information needed to perform the simulations. Each entry is named according to what it is supposed to contain.

schlather.slot1	Internal variable for miraculix (cre: M. Schlather)
chromosome	Number of chromosomes in the population list
snp	Number of SNPs per chromosome
position	Position (in Morgan) on the chromosome for each marker
snp.base	Major/Minor Allele (e.g. characters since internally 0/1 is used)
snp.position	Overall position in the genome (ongoing over chromosomes
length	Length of each chromosome
length.total	Cumulative length of chromosomes
func	It's just FALSE – placeholder for later
size	Size of each group (generation/sex)
bve	Coding if breeding values are simulated
bv.calculated	Coding if breeding values are calculated for the founders (will be after first run of *breeding.diploid()*)
breeding.totals	Recap of each run of *breeding.diploid()* (if stored)
bve.data	Recap of each breeding value estimation (if stored)
bve.nr	Number of traits (with QTLs behind) to consider
bv.random	Coding which traits have underlying QTLs behind
bv.random.variance	Genetic variance for traits with no QTLs behind
snps.equidistant	Are SNPs equidistant on every chromosome (speed up!)
origin.gen	List of founding generations (with stored haplotypes)
cumsnp	Cumulative sum of SNP number (just to save computational time)
bp	Physical position of each marker (bp)
snp.name	Name of each marker
next.animal	ID of the next individual to generate
bve.mult.factor	(bv) * this
bve.poly.factor	(bv) ^ this

base.bv	This + QTL_effects
bv.calc	Number of total traits (including those with no QTL behind)
real.bv.length	Traits with (additive/multiplicative/dice-effects)
sex	Sex of the founders added in *creation.diploid*
real.bv.add	Lists with an overview of all single marker QTLs for each trait
real.bv.mult	Lists with an overview of all two marker QTLs for each trait
real.bv.dice	Lists with an overview of all three+ marker QTLs for each trait
pheno.correlation	Correlation matrix of the environmental variance between traits
bv.correlation	Correlation matrix of the genetic variance between traits (only for non-QTL traits)
miraculix	Coding if miraculix was used to generate the data – only miraculix users will be able to work with those population lists
cohorts	List of all cohorts with name and position in the population list
effect.p.add	Markers involved as QTL in the single QTL proportion
effect.p.mult1	Markers involved as QTL in the two-SNP QTL proportion
effect.p.mult2	Markers involved as QTL in the two-SNP QTL proportion
effect.p	Markers involved as QTL in any trait
store.effect.freq	Frequency of each marker in each generation
last.sigma.e.database	Database to derive the last used environmental variance
last.sigma.e.value	Last used environmental variance
last.sigma.e.heritabilty	Heritability assumed to derive last used environmental variance
comp.times	Computation times needed in each use of *breeding.diploid()* (if stored)
comp.times.bve	Computation times needed in the breeding value estimation in each use of *breeding.diploid()* (if stored)
comp.times.generation	Computation times needed in for the generation of new individuals in each use of *breeding.diploid()* (if stored)
culling.stats	Information on the culling reason of each individual (mostly releveant for the web-interface)
phenotypic.transform	Indication on with traits undergo a transformation function when generating a phenotype
phenotypic.transform.function	Function to apply on the generated phenotype (e.g. for discrete distributions)
repeat.mating	Probabilities for the number of offspring per mating (litter size)
repeat.mating.copy	Probabilites for the number of copys of an individual generated in all copy/combine actions
is.maternal	TRUE/FALSE indication if each respective traits is caused by maternal effects
is.paternal	TRUE/FALSE indication if each respective traits is caused by paternal effects
is.combi	TRUE/FALSE indication if each respective traits is derived as a combination of other traits

combi.weights	Weights of each other traits for combined traits

10.2 $breeding

$breeding contains all relevant information concerning the individuals of the breeding scheme. For efficiency purposes, a lot of this is internally coded or computed on-the-fly.

Individuals are sorted according to generation, sex and individual number. In case data has to be stored for both male and female (or father/mother) there will be two entries with the first one being the male (Have to talk with the equality commissioner about that!).

$breeding[[generation]][[sex]][[individual nr.]] ((or [[2]][[generation]][[sex]][[individual nr.]]))

10.2.1   Storage per generation

$breeding[[generation]][[3,4]]	Estimated breeding values of males (3) and females (4)
$breeding[[generation]][[5,6]]	Class of males (5) and females (6)
$breeding[[generation]][[7,8]]	Underlying "true" genetic values of males (7) and females (8)
$breeding[[generation]][[9,10]]	Observed phenotypes for males (9) and females (10)
$breeding[[generation]][[11,12]]	Time point of generation for male (11) and females (12) – in case of copy.individual != time of birth!
$breeding[[generation]][[13,14]]	Creating type of generation for males(13) and females (13) This is only relevant for the web-based application
$breeding[[generation]][[15,16]]	Individual IDs for male (15) and females (16)
$breeding[[generation]][[17,18]]	Has been removed – currently not in use
$breeding[[generation]][[19,20]]	Reliability estimated for male (19) and females (20)
$breeding[[generation]][[21,22]]	Last applied selection index (mostly relevant for complex selection indices like (Miesenberger 1997)
$breeding[[generation]][[23,24]]	Time of birth for male(23) and female (24) individuals
$breeding[[generation]][[25,26]]	Time of culling for male(25) and female (26) individuals
$breeding[[generation]][[27,28]]	Average offspring phenotype for male(27) and female (28) individuals
$breeding[[generation]][[29,30]]	Number of offspring phenotypes used in 27/28 for male(29) and female (30) individuals
$breeding[[generation]][[31,32]]	Time point of death for male (31) and female (32) individuals

10.2.2   Storage per individual

$breeding[[generation]][[sex]][[individual nr.]]…

[[1,2]]	Points of recombination on the first (1) and second (2) chromosome set
[[3,4]]	Points of mutations
[[5,6]]	Efficiently stored origins of segments between two points of recombination. Decoding using *decodeOrigins()* (miraculix) / *decodeOriginsR()* (else).

	Output in *get.recombi()* is automatically decoded
[[7,8]]	Father / Mother
[[9,10]]	Efficiently stored haplotypes (if it is a founder – else empty)
[[11,12]]	Storage of duplications (long not used!)
[[13,14]]	Storage of history of recombinations
[[15]]	How often a phenotype was generated for the individual
[[16]]	Is the individual genotyped
[[17]]	True breeding value before gene editing
[[18]]	Generation of death and previous class
[[19]]	Share of the genetic material of the grandfather of the father inherited
[[20]]	Share of the genetic material of the grandfather of the mother inherited
[[21]]	List of all individuals with the same id.
[[22]]	Vector of array used for genotyping
[[23]]	Permanent residual
[[24]]	Temporary (newly samples for each phenotyping) residual

```
> str(population$breeding[[1]][[1]][[1]])
List of 21
 $ : num [1:6] 0 0.2 0.4 0.6 0.8 1
 $ : num [1:6] 0 0.2 0.4 0.6 0.8 1
 $ : NULL
 $ : NULL
 $ : int [1:5] 0 0 0 0 0
 $ : int [1:5] 1 1 1 1 1
 $ : num [1:3] 1 1 1
 $ : num [1:3] 1 1 1
 $ :Haplotype information at 1000 loci.
Attributes are a List of 3
 $ information: int [1:22] 0 1000 1 530385344 0 530385344 0 0 0 NA ...
 $ method     : int 8
 $ class      : chr "haplomatrix"
 $ : chr "Placeholder_Pointer_Martin"
 $ : NULL
 $ : NULL
 $ : NULL
 $ : NULL
 $ : num [1:21] 0 0 0 0 0 0 0 0 0 0 ...
 $ : int 0
 $ : NULL
 $ : NULL
 $ : NULL
 $ : NULL
 $ : num [1:2, 1:3] 1 2 1 1 1 63
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:3] "generation" "sex" ""
```

11 Memory and computation times

Critical parts of MoBPS concerning memory requirements and computation times can be performed using the associated R-package miraculix. By using SSE2 operations and bit-wise storing computation speed can be massively increased leading to about 10 times faster matrix multiplications than the regular R implementation while needing only 1/16 of the regularly needed memory.

To speed up the computation of the breeding value estimation, one can use multiple cores by the usage of **miraculix.cores** (default: 1) or in case miraculix is not active **ncore**. To parallelize the generation of new individuals set **parallel.generation** to TRUE and set the number of cores used via **ncore.generation**. This will only lead to significant improvement in computation time for the generation of a lot of individuals. Even when using a single core ~1'000 individuals are generated per second. The packages doParallel (Microsoft Corporation and Steve Weston 2018) and doRNG (Renaud Gaujoux 2018) are used for parallelization in R.

Computing time for the whole simulation (population$info$comp.times), the steps of the breeding value estimation (population$info$comp.times.bve) and the generation of new individuals (population$info$comp.times.generation) are stored in the population list. If this is not required you can deactivate this by setting **store.comp.times**, **store.comp.times.bve** and **store.comp.times.generation** to FALSE in *breeding.diploid()*. Unless **verbose** is set to FALSE you should automatically receive notifications on the current step of the algorithm and computing times of individual steps. Activation of **Rprof** can provide even more information.

11.1.1 Efficient BLAS / LAPACK

MoBPS uses various functions and operations that use standard R function like the matrix multiplication and inversion. There are highly efficent solutions provided in the context of Basic Linear Algebra Subprograms (BLAS) and Linear Algebra Package (LAPACK). I personally am using a conda container with libopenblasp-r0.3.12.so. Just for reference to computing speed / benchmarking consider running the following on your system:

```r
n <- 5000
T <- matrix(0, nrow=n, ncol=n)
A <- T %*% T
```

My garbage windows computer without optimization takes 100 seconds for this code. Our computing cluster with their standard single core BLAS 6 seconds, and an optimized computing cluster with OpenBLAS requires 2 seconds using a single core (or 0.6 seconds with 5 cores). You yourself should judge if your simulation take long enough that you need to take a look into this. Effects will not be 100-fold as in this example as most of the computations have at least a bit more efficiently implemnted version that run if your BLAS is not good expect substantial gain (not only for MoBPS related project!)

11.1.2   Reducing the size of the population list

Especially when simulating populations with lots of markers, individuals and/or generations, data storage can become a problem. As the internal structure of a population list is complex and manually deleting things is not recommended.

As MoBPS is storing recombination points and not the haplotype itself (on the fly calculation), the amount of storage needed increases with the number of meiosis since the last founder. To negate those issues one can declare individuals as "new" founder. Meaning that the haplotypes for those individuals are calculated and thus recombination points before do not need to be stored. For details on this we refer to section 9.6. In particular when working with large genomes and low number of SNPs, this can massively reduce storage needs.

To delete points of recombination between the same founder strand, the function *clean.up()* can be applied on the population list. For more on that, we refer to section 9.12.

Further reduction of the required memory can be obtained by deleting old information. In contrast to *clean.up()* this will lose information and is only recommended to be used on early generations that are not needed in any later steps or in the following analysis. Use following parameters in *breeding.diploid()*:

delete.haplotypes: Vector containing all generations for which haplotypes no longer need to be stored (note that only founder generations are stored anyway – everything else is calculated on-the-fly)

delete.same.origin: Merge two adjacent segments with the same founding haplotypes (deletion of a recombination point with no influence)

delete.individuals: Vector containing all generations for complete deletion– to only delete one sex use **delete.sex** (vector contain sex to delete – 1 (male), 2 (female). Especially when the number of recombinations stored per individual, becomes bigger this is of relevancy.

11.1.3   Reducing memory needs in the BVE

To calculate the genomic relationship matrix, one has to perform matrix multiplication of a matrix containing n*p entries (individuals x markers). Note that for every entry only 2 bits are needed when using miraculix. Nevertheless, this can become extremely big – to reduce this, one can perform the calculation of G sequentially and only load a part of Z into memory at any time.

To activate this, set **sequenceZ** to TRUE. The number of markers in memory can be controlled via **maxZ** (default: 5000). Alternatively, one can put **maxZtotal** to control the total number of entries instead. As this is increasing the computation time, we first recommend to activation of miraculix. As long as miraculix is installed this will happen automatically unless the parameter **miraculix** in *creating.diploid()* is set to FALSE.

To speed up commutation one can use multiple cores by the usage of **miraculix.cores** (default: 1) or in case miraculix is not active **ncore**. The backend outside of miraculix is using doParallel and mclapply but doMPI is supported as well – we highly recommend the use of miraculix instead.

Setting **fast.compiler** to TRUE will additionally activate a just-in-time-compiler (enableJIT(3)).

To save computation speed in the GWAS, one can use **approx.residuals** – this does not influence the order of the predicted effect markers but will influence p-values slightly.

11.1.4   Inverting G using miraculix

The inversion of $(G + I_n \cdot \lambda)$ can take a lot of time, is numerically unstable and might not even be possible at all if the matrix is not invertible at all. Instead of the standard cholesky procedure using *chol2inv(chol())*, the inversion can also be done in RandomFieldsUtils/miraculix by activating **chol.miraculix**. Leading to slightly reduced computing times – but also includes screening for semi definite matrices and an automatically changed algorithm, if needed, and thus proceeding without error.

Note that depending on your Blas, you might automatically fall back on the basic solve() function from the base package. E.g., when using OpenBlas this function is fully parallized and uses computational trips that are just not available to use. If this is needed is automatically checked when loading the RandomFieldsUtils package, so you do not need to worry about this! You should however worry about using an efficient Blas ;)

11.1.5   Working with a high number of traits

When working with an extremely high number of traits (e.g. due to multiple versions of a traits in GxE) the generation of individuals can take extended time because the genomic value for each trait needs to be calculated. To not evaluate all traits on generation use the parameter **bv.ignore.traits** including a vector of all traits to not evaluate. Whenever phenotypes of a trait are generated this evaluation is automatically done at that time – this is only problematic when you just want to export a trait later without ever needing the calculation of it during the simulation itself.

Similarly, selected traits can be skipped in the breeding value estimation via the parameter **bve.ignore.traits**.

11.1.6   On-the-fly calculation of haplotypes

To save memory, haplotypes are calculated on-the-fly. For this, the location of each recombination point (between which markers) has to be stored. In case one is working with equidistant markers, it basically takes no time. For other cM-positions it might increase computation speed to provide a function that derives the last marker in front of a certain position in Morgan. This function can be entered via **import.position.calculation**. Only in extreme cases (lots of markers) this should even matter!

12 List of input parameters in breeding.diploid()

For a description of each parameter we refer to the use of the help function in R (?*breeding.diploid*) and/or other sections of this Guidelines.

Parameter	Default	options
population	NULL	A previous population list
mutation.rate	10^-8	Value between 0 and 1
remutation.rate	10^-8	Value between 0 and 1
recombination.rate	1	Any positive numeric
selection.criteria	NULL	"bve", "bv", "pheno", "offpheno"
selection.m	NULL ("random" if selection.criteria = NULL; else "function"	"random", "function"
selection.f	NULL (selection.m)	"random", "function"
new.selection.calculation	TRUE	FALSE
selection.function.matrix	NULL	Don't touch – will be removed
selection.size	C(0,0)	Vector with two non-negative values
breeding.size	0	Positive number // 2 element vector (male/female)
breeding.sex	0.5	Value between 0 and 1
breeding.sex.random	FALSE	TRUE
class.m	0	Vector with all classes to consider
class.f	NULL (class.m)	Vector with all classes to consider
add.gen	0 (will lead to added generation)	Value between 1 and number of generations
recom.f.indicator	NULL	Not necessary (use modified marker position instead!)
duplication.rate	0	
duplication.length	0.01	Duplication modelling needs changes!
duplication.recombination	1	
same.sex.active	FALSE	TRUE
new.class	0	Numeric value (ideally positive integer; -1 is reserved for dead individuals)
bve	FALSE	TRUE
bve.solve	"exact"	"pcg", function to replace solve() in the BVE.

bve.array	NULL (which will use 1)	Numeric value above 0
bve.direct.est	TRUE	FALSE
bve.gen	NULL	1:3
bve.database	NULL	``` Generation sex [1,] 1 2 [2,] 5 1 ```
bve.cohorts	NULL	c("Founder_M", "F1")
bve.avoid.duplicates	TRUE	FALSE
Bve.exclude.fixed.effects	NULL	
Bve.beta.hat.approx	TRUE	FALSE
report.accuracy	TRUE	FALSE
sigma.e	NULL	Numeric value above 0
sigma.s	100	Numeric value above 0
bve.per.sample.sigma.e	TRUE	FALSE
share.phenotyped	1	Value between 0 and 1
phenotyping	NULL	"all" for all individuals "non_obs" for all previously not observed "non_obs_m" for all previously not observed male individuals "non_obs_f" for all previously not observed female individuals
phenotyping.gen	NULL	1:3
phenotyping.database	NULL	``` Generation sex [1,] 1 2 [2,] 5 1 ```
phenotyping.cohorts	NULL	c("Founder_M", "F1")
phenotyping.child	NULL	„zero", „mean", „obs"
Relationship.matrix	"vanRaden"	"kinship", "CE", "CM", "non_stand"
depth.pedigree	7	Positive Integer
delete.haplotypes	NULL	Vector of all generations to delete (natural number)
delete.individuals	NULL	Vector of all generations to delete (natural number)
fixed.breeding	NULL	matrix with each row containing (gen1,sex1,nr1, gen2,sex2,nr2, probability.female) with gen/sex/nr1 information on the first parent / father, gen/sex/nr2 information on the second parent / mother)
fixed.breeding.best	NULL	matrix with each row containing

		(sex1, nr1, sex2, nr2, sex.probability) chosen from the group of selected individuals
max.offspring	C(Inf,Inf)	vector with two natural numbers (first male, second female)
max.litter	C(Inf,Inf)	Vector with two natural numbers (first male, second female)
max.mating.pair	Inf	Numeric value (1-Inf)
store.breeding.totals	FALSE	TRUE
forecast.sigma.s	TRUE	FALSE
multiple.bve	"add"	"ranking"
multiple.bve.weights.m	1	Any weights – use a vector with length equal to number of traits
multiple.bve.weights.f	NULL (multiple.bve.weights.m)	Any weights – use a vector with length equal to number of traits
store.bve.data	FALSE	TRUE
fixed.assignment	FALSE	"bestworst", "worstbest"
reduce.group	NULL	Per row: Generation, Sex, Individuals to survive, class of individuals
reduce.group.selection	"random"	"function"
selection.highest	c(TRUE,TRUE)	C(FALSE/TRUE,FALSE/TRUE)
same.sex.sex	0.5	Numeric value between 0 and 1
same.sex.selfing	TRUE	FALSE
selfing.mating	FALSE	TRUE
selfing.sex	0.5	Numeric value between 0 and 1
praeimplantation	NULL	No use currently recommended
sigma.e.gen	NULL	1:3
sigma.e.database	NULL	``` Generation sex
[1,]   1   2
[2,]   5   1 ```
sigma.e.cohorts	NULL	c("Founder_M", "F1")
heritability	NULL	Numeric value between 0 and 1
multiple.bve.scale.m	FALSE	TRUE
multiple.bve.scale.f	NULL (multiple.bve.scale.m)	TRUE
save.recombination.history	FALSE	TRUE
martini.selection	FALSE	TRUE
BGLR.bve	FALSE	TRUE
BGLR.model	"RKHS"	"BRR", "BL", "BayesA", "BayesB", "BayesC"
BGLR.burnin	500	natural number
BGLR.iteration	5000	natural number

BGLR.save	"RKHS"	any path you want
BGLR.save.random	FALSE	TRUE
BGLR.print	FALSE	TRUE
mas.bve	FALSE	TRUE
mas.markers	NULL	Vector with SNP-positons
mas.number	5	Any natural number
mas.effects	NULL	Vector with numeric values
threshold.selection	NULL	Numeric value
threshold.sign	">"	"<","=","<=",">="
copy.individual	FALSE	TRUE
copy.individual.m	FALSE	TRUE
copy.individual.f	FALSE	TRUE
copy.individual.keep.bve	TRUE	FALSE
dh.mating	FALSE	TRUE
dh.sex	0.5	Numeric value between 0 and 1
offpheno.parents.gen	NULL	1:3
offpheno.parents.database	NULL	``` Generation sex [1,] 1 2 [2,] 5 1```
offpheno.parents.cohorts	NULL	c("Founder_M", "F1")
offpheno.offspring.gen	NULL	1:3
offpheno.offspring.database	NULL	``` Generation sex [1,] 1 2 [2,] 5 1```
offpheno.offspring.cohorts	NULL	c("Founder_M", "F1")
bve.parent.mean	FALSE	TRUE
bve.grandparent.mean	FALSE	TRUE
bve.mean.between	"bvepheno"	„bve", „pheno", „bv"
n.observation	1	Natural number
share.genotyped	0	Numeric value between 0 and 1
added.genotyped	0	Numeric value between 0 and 1
genotyped.gen	NULL	c("Founder_M", "F1")
genotyped.database	NULL	c("Founder_M", "F1")
genotyped.cohorts	NULL	c("Founder_M", "F1")
genotyped.share	1	Numeric value 0-1
genotyped.array	1	Natural number
genotyped.remove.gen	NULL	c("Founder_M", "F1")
genotyped.remove.database	NULL	c("Founder_M", "F1")
genotyped.remove.cohorts	NULL	c("Founder_M", "F1")
genotyped.remove.all.copy	TRUE	FALSE

remove.non.genotyped	TRUE	FALSE
singlestep.active	TRUE	FALSE
bve.0isNA	TRUE	FALSE
phenotype.bv	FALSE	TRUE
standardize.bv	FALSE	TRUE
standardize.bv.level	100	Numeric value
standardize.bv.gen	1	Natural number <= generation number
bve.input.phenotype	"own"	"off", "mean", "weighted"
delete.same.origin	FALSE	TRUE
remove.effect.position	FALSE	TRUE
estimate.u	FALSE	TRUE
fast.uhat	TRUE	FALSE
new.residual.correlation	NULL	Positive definite matrix
new.breeding.correlation	NULL	Positive definite matrix
recalculate.bv.var.correlation	FALSE	TRUE
new.bv.random.correlated	TRUE	FALSE
estimate.add.gen.var	FALSE	TRUE
estimate.pheno.var	FALSE	TRUE
selection.m.gen	NULL	1:3
selection.f.gen	NULL	1:3

selection.m.database	NULL	`      Generation sex`
		`[1,]          1   2`
		`[2,]          5   1`

selection.f.database	NULL	`      Generation sex`
		`[1,]          1   2`
		`[2,]          5   1`

selection.m.cohorts	NULL	c("Founder_M", "F1")
selection.f.cohorts	NULL	c("Founder_M", "F1")
reduced.selection.panel.m	NULL	Vector containing numeric values
reduced.selection.panel.f	NULL	Vector containing numeric values
store.comp.times	TRUE	FALSE
store.comp.times.bve	TRUE	FALSE
fixed.effects.p	NULL	
fixed.effects.freq	NULL	
ogc	FALSE	TRUE
ogc.target	"min.sKin"	"max.BV", "min.BV"
ogc.uniform	NULL	"male", "female"

ogc.ub	NULL	Numeric value
ogc.lb	NULL	Numeric value
ogc.ub.sKin	NULL	Numeric value
ogc.lb.BV	NULL	Numeric value
ogc.ub.BV	NULL	Numeric value
ogc.eq.BV	NULL	Numeric value
Ogc.ub.sKin.increase	NULL	Numeric value
Ogc.lb.BV.increase	NULLL	Numeric value
Relationship.matrix.ogc	"kinship"	"vanRaden" (see relationship.matrix)
depth.pedigree.ogc	7	Positive Integer
avoid.mating.fullsib	FALSE	TRUE
avoid.mating.halfsib	FALSE	TRUE
emmreml.bve	FALSE	TRUE
nr.edits	0	any natural number
gene.editing	FALSE	TRUE
gene.editing.offspring	FALSE	TRUE
gene.editing.best	FALSE	TRUE
gene.editing.offspring.sex	c(TRUE,TRUE)	Vector with two boole variables
gene.editing.best.sex	c(TRUE,TRUE)	vector with two boole variables
gwas.u	FALSE	TRUE
approx.residuals	TRUE	FALSE
sequenceZ	FALSE	TRUE
maxZ	5000	Any natural number
maxZtotal	0	Any natural number
gwas.gen	NULL	1:3
| gwas.database | NULL | ```
          Generation sex
    [1,]            1   2
    [2,]            5   1
``` |
gwas.cohorts	NULL	c("Founder_M", "F1")
delete.sex	c(1,2)	1 (male), 2 (female)
gwas.group.standard	FALSE	TRUE
y.gwas.used	"pheno"	"bv", "bve"
culling.cohort	NULL	Any cohort name
culling.time	Inf	Numeric value
culling.name	"Not_named"	Any character string
culling.bv1	100	numeric value
culling.share1	0	Probability between 0 and 1
culling.bv2	110	numeric value
culling.share2	0	Probability between 0 and 1

culling.index	0	Any weights – use a vector with length equal to number of traits, "lastindex"
culling.all.copy	TRUE	FALSE
gen.architecture.m	0	Natural number (select one of the previously stored architectures)
gen.architecture.f	NULL (gen.architecture.m)	Natural number (select one of the previously stored architectures)
ncore	1	Natural number
Z.integer	FALSE	TRUE
store.effect.freq	TRUE	FALSE
backend	"doParallel"	„doMPI"
randomSeed	NULL	natural number
randomSeed.generation	NULL	Natural number
Rprof	FALSE	TRUE
miraculix	FALSE	TRUE (automatically activated when miraculix is used is *creating.diploid()*)
miraculix.mult	NULL (leading to FALSE)	TRUE / FALSE
fast.compiler	0	3 (For R >= 3.4 this is default in R)
miraculix.cores	1	natrual number
store.bve.parameter	FALSE	TRUE
print.error.sources	FALSE	TRUE
chol.miraculix	FALSE	TRUE
bve.insert.gen	NULL	1:3
bve.insert.database	NULL	``` Generation sex [1,] 1 2 [2,] 5 1 ```
bve.insert.cohorts	NULL	c("Founder_M", "F1")
best.selection.ratio.m	1	positive numeric value
best.selection.ratio.f	NULL (best.selection.ratio.m)	positive numeric value
best.selection.criteria.m	"bv"	"bve", "pheno"
best.selection.criteria.f	NULL (best.selection.criteria.m)	"bve", "pheno"
best.selection.manual.ratio.m	NULL	positive numeric value
best.selection.manual.ratio.f	NULL (best.selection.manual.ratio.m)	positive numeric value
best.selection.manual.reorder	TRUE	FALSE

bve.class	NULL (take all!)	vector containing numeric values
parallel.generation	FALSE	TRUE
ncore.generation	1	Positive numeric value
name.cohort	NULL	"Founders" or any other character string
add.class.cohorts	TRUE	FALSE
display.progress	TRUE	FALSE
ignore.best	C(0,0)	Any two element vector (first male, second female)
combine	FALSE	TRUE
repeat.mating	1	Positive numeric value OR matrix with two colums (litter size; probability of that litter size)
repeat.mating.copy	1	Positive numeric value OR matrix with two colums (litter size; probability of that litter size)
repeat.mating.overwrite	TRUE	FALSE
Repeat.mating.trait	NULL	
Repeat.mating.max	NULL (100)	
Repeat.mating.s	NULL	
time.point	0	Positive numeric value (this will be automatically processed in the web-based-application)
creating.type	0	This is automatically stored in the web-based-application # 0 – Founder # 1 – Selection # 2 – Reproduction # 3 – Recombination # 4 – Selfing # 5 – DH-Production # 6 – Cloning # 7 – Combine # 8 – Aging # 9 – Split
genotyped.gen	NULL	1:3
genotyped.database	NULL	``` Generation sex [1,] 1 2 [2,] 5 1 ```
genotyped.cohorts	NULL	c("Founder_M", "F1")

genotyped.share	1	
genotyped.array	1 (everything)	
bve.imputation	TRUE	FALSE
bve.imputation.errorrate	0	Value between 0 and 1
sex.s	NULL	Vector with same length as number of new individuals
new.phenotype.correlation	NULL	Positive definite matrix
bve.ignore.traits	NULL	Vector containing integers
bv.ignore.traits	NULL	Vector containing integers
Mixblup.bve	FALSE	TRUE
Mixblup.pedfile	TRUE	FALSE
Mixblup.parfile	TRUE	FALSE
Mixblup.datafile	TRUE	FALSE
Mixblup.path	"MixBLUP"	Linux "./MixBLUP.exe" or other file path
Mixblup.files	"MiXBLUP_files/"	directory
Mixblup.genofile	TRUE	FALSE
Mixblup.path.pedfile	NULL	File path
Mixblup.path.parfile	NULL	File path
Mixblup.path.datafile	NULL	File path
Mixblup.path.inputfile	NULL	File path
Mixblup.path.genofile	NULL	File path
Mixblup.lambda	1	
Mixblup.omega	NULL --> Mixblup.lambda	
Mixblup.alpha	1	
Mixblup.beta	0	
Store.sparse	FALSE	TRUE
Mixblup.apy	FALSE	
Mixblup.apy.core	Inf	
OLD PARAMETERS THAT ARE STILL THERE HAVE NEWER VERSION		
new.bv.observation	phenotyping	"all" for all individuals "non_obs" for all previously not observed

		"non_obs_m" for all previously not observed male individuals "non_obs_f" for all previously not observed female individuals
new.bv.observation.gen	phenotyping.gen	1:3
new.bv.observation.database	phenotyping.database	```\n Generation sex\n[1,] 1 2\n[2,] 5 1\n```
new.bv.observation.cohorts	phenotyping.cohorts	c("Founder_M", "F1")
new.bv.child	phenotyping.child	"zero", "mean", "obs"
best1.from.group	NULL	Matrix with one group per row
best2.from.group	NULL	Matrix with one group per row
best1.from.cohort	NULL	Vector containing names of cohorts
best2.from.cohort	NULL	Vector containing names of cohorts
computation.A	"vanRaden"	"kinship", "CE", "CM", "non_stand"
relationship.matrix.ogc	"kinship"	"vanRaden" (see computation.A)
export.selected	FALSE	TRUE
export.relationship.matrix	FALSE	TRUE

13 List of input parameters in creating.diploid()

For a description of each parameter we refer to the use of the help function in R (?*creating.diploid*) and/or other sections of this Guidelines.

Parameter	Default	options
population	NULL (will lead to "random")	A previous population list
dataset	"random"	SNP-dataset (One haplotype per colum), "random", "all0", "homorandom", "allhetero"
nsnp	0	Positive integer value
nindi	0	Positive integer value
vcf	NULL	Path to a vcf-file
vcf.maxsnp	Inf	Any integer value
vcf.chromosomes	NULL (all chromosomes)	vector with character values
vcf.VA	TRUE	FALSE
map	NULL	Matrix with up to 5 colums containing (chr.nr, snp.name, bp, position in Morgan, allele freq). Rest will be set to NULL/NA. For more see section 14
chr.nr	NULL (all markers on the same chromosome)	Vector containing the chromosome for each generated marker, or natural number with the number of chromosomes
bp	NULL	Vector containing the base-pair for each generated marker
snp.name	NULL	Vector containing the snp-name for each generated marker
bpcm.conversion	0	Recommendations: For human: 1.000.000 For chicken: 300.000
chromosome.length	NULL (will lead to 5M)	Positive numeric value
freq	"beta"	Numeric value or vector for each marker
beta.shape1	1	Positive numeric value
beta.shape2	1	Positive numeric value
sex.s	"fixed"	"random", vector containing the sex of each newly added individual.
share.genotyped	1	Numeric value between 0 and 1
genotyped.s	NULL	vector containing the sex of each newly added individual.
add.chromosome	FALSE	TRUE

generation	1	Positive integer value (no empty generations inbetween!)
class	0	Numeric value (positive integer recommended)
sex.quota	0.5	Numeric value between 0 and 1
snps.equidistant	NULL (will be TRUE if no other way to derive Morgan-position is provided)	FALSE/TRUE
position.scaling	FALSE	TRUE
length.before	5	Positive numeric value
length.behind	5	Positive numeric value
hom0	NULL (automatically derived)	Vector containing major allele for each generated marker.
hom1	NULL (automatically derived)	Vector containing minor allele for each generated marker.
miraculix	TRUE	FALSE
bit.storing	FALSE	TRUE (this is less efficient than miraculix but does not rely on C-code)
nbits	30	Integer value between 1 and 30
bv.total	0 (automatically set according to traits provided)	Integer value. If higher than the number of traits simulate traits based on pedigree/inbreeding rates
trait.name	NULL	Vector containing the names of the traits (e.g. "milk")
real.bv.add	NULL	List with each element containing effect matrices
real.bv.mult	NULL	List with each element containing effect matrices
real.bv.dice	NULL	List with each element containing effect lists
n.additive	0	Positive integer value
n.equal.additive	0	Positive integer value
n.dominant	0	Positive integer value
n.equal.dominant	0	Positive integer value
dominant.only.positive	FALSE	TRUE
n.qualitative	0	Positive integer value
n.quantitative	0	Positive integer value
var.additive.l	NULL	List containing a single numeric value or vector with variances for each trait
var.dominant.l	NULL	List containing a single numeric value or vector with variances for each trait
var.qualitative.l	NULL	List containing a single numeric value or vector with variances for each trait

var.quantitative.l	NULL	List containing a single numeric value or vector with variances for each trait
effect.size.equal.add	1	Positive numeric value
effect.size.equal.dom	1	Positive numeric value
exclude.snps	NULL	Vector containing marker positions with no simulated random effects
shuffle.traits	NULL	TRUE
shuffle.cor	NULL	Correlation matrix for the traits to shuffle
replace.real.bv	FALSE	TRUE
name.cohort	NULL	Character string
skip.rest	FALSE	TRUE (INTERNAL PARAMETER!)
randomSeed	NULL	Integer value
template.chip	NULL	"cattle", "chicken", "pig", "sheep", "maize"
time.point	0	Positive numeric value (this will be automatically processed in the web-based-application)
creating.type	0	This is automatically stored in the web-based-application # 0 – Founder # 1 – Selection # 2 – Reproduction # 3 – Recombination # 4 – Selfing # 5 – DH-Production # 6 – Cloning # 7 – Combine # 8 – Aging # 9 – Split
remove.invalid.qtl	TRUE	FALSE

14 List of datasets included in the package

MoBPS does contain a variety of maps that are pre-imported from Ensembl since the actual import takes quite long for bigger map files. In case you feel a certain map is missing feel free to contact us so we can add it to the tool. Maps are available in the associated R-package MoBPSmaps. Only map_chicken1, map_cattle1 and map_maize1 are included in MoBPS itself. To use a specific map use it as an input for the parameter **map** in *creating.diploid()*.

In addition to all those maps an exemplary Json-file (**ex_json**) generated by a recent version of our interface is included for text use in *json.simulation()* and other function that utilize datasets generated by *json.simulation()*. Note that this file is automatically generated via the user interface and you do not have to worry about its structure.

Dataset name	Corresponding Chip	Number of Markers	Contains: 1. Physical position 2. Morgan position 3. allele frequency		
map_pig1	Axiom Genotyping Array	590'318	green	pink	pink
map_pig2	GGP Porcine HD	63'113	green	pink	pink
map_pig3	GGP Porcine LD	8'624	green	pink	pink
map_pig4	Illumina_PorcineSNP60	55'684	green	pink	pink
map_chicken1	Affymetrix Chicken600K Array	547'024	green	pink	pink
map_chicken2	Affymetrix Chicken600K Array (diversity subset)	293'251	green	pink	pink
map_chicken3	Affymetrix Chicken600K Array (50k subset)	50'000	green	pink	pink
map_cattle1	Illumina BovineSNP50 BeadChip	45'613	green	pink	pink
map_cattle2	Illumina BovineHD BeadChip	727'605	green	pink	pink
map_cattle3	Illumina BovineLD BeadChip	6'600	green	pink	pink
map_cattle4	Genotyping chip variations	732'645	green	pink	pink
map_horse1	Illumina EquineSNP50 BeadChip	51'105	green	pink	pink
map_sheep1	IlluminaOvineHDSNP	575'256	green	pink	pink
map_sheep2	IlluminaOvineSNP50	46'545	green	pink	pink
map_sheep3	Genotyping chip variants	580'661	green	pink	pink
map_goat1	Illumina_GoatSNP50	55'050	green	pink	pink
map_human1	Affy GeneChip500K	483'418	green	pink	pink
map_human2	Illumina_1M-duo	1'122'013	green	pink	pink
map_human3	Illumina_HumanHap550	545'902	green	pink	pink
map_maize1	Affymetrix Axiom Maize Genotyping Array	501'124	green	pink	pink
map_wheat1	Subset of a 55k chip from (Liu et al. 2018)	12'109	pink	green	pink
map_wheat2	Subset of a 90K chip from (Wen et al. 2017)	29'692	green	green	pink

map_sorghum1	Subset of a 90k chip from (Bekele et al. 2013)	3'000				
map_soybean1	Song et al. 2016	60'701				
map_rice1	Morales et al. 2020	7'098				
map_barley1	Bayer et al. 2017	50'826				
map_oilseed1	Mason et al 2017	16'858				
map_salmon1	Tsai et al 2013 (male recombination rates)	96'396				
map_salmon2	Tsai et al 2013 (female recombination rates)	96'396				
map_tilapia1	Penaloza et al. 2020	68'910				
map_seabream1	Palaiokostas et al. 2016	12'085				

Ex_json – the first few rows. Do not bother trying to understand it from just the JSON-file and look at it via our interface or let j*son.simulation()* just do the job for you:

```json
"Nodes": [
  {
    "id": "Bull",
    "Number of Individuals": "100",
    "x": -152,
    "y": -165,
    "individualsVar": "100",
    "Founder": "Yes",
    "Path": "",
    "Proportion of Male": 1,
    "BV Plot": "Yes",
    "Sex": "Male",
    "Phenotyping Class": "Default PhenoC",
    "Housing Cost Class": "Male individuals",
    "Proportion of genotyped individuals": 1,
    "label": "Bull",
    "color": "#9acef4",
    "title": "Bull: 100 Ind"
  },
```

15 On the generation of traits

In this section, we provide additional background on how traits and phenotypes are generated to obtain provided architectures that were selected in section 4.3.

15.1 General

To obtain correlated traits with underlying QTL the Cholesky decomposition of the matrix provided in **shuffle.cor** is calculated. Next, QTL effect sizes are scaled to ensure the same genetic variance for all traits and existing traits are then combined into new traits by use of the entries of the upper triangular matrix resulting from *chol()*. Note that the number of underlying QTL is for each trait will therefore change! An added benefit of this way of generating traits is that any positive semi-definite matrix is a valid input for this approach and trait correlations are consist over time.

In case non-QTL based traits (X_1) are simulated the genetic variance of those traits is sampled of a multivariate Gaussian distribution with the QTL-based traits (X_2):

$$X = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \sim N\left(\begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \begin{pmatrix} A & B \\ B^t & C \end{pmatrix} \right)$$

$$X_1 | X_2 \sim N\left(\mu_1 + B\, C^{-1}(X_2 - \mu_2), A - B\, C^{-1}B^t \right)$$

Residual effects are generated by generation independent *N(0,1)* random variables and subsequent scaling with the Cholesky decomposition of the residual correlation matrix (**new.residual.correlation**). Note that the first observation of one trait will always be correlated to the first observation of the other traits, but not to other observations! In case a repeatability w^2 is provided the residual variance σ_e^2 will be separated in a permanent part σ_{PU}^2 that will be used for all observations and the temporary part σ_{TU}^2 that will be re-sampled for each observation with:

$$w^2 = \frac{\sigma_a^2 + \sigma_{PU}^2}{\sigma_a^2 + \sigma_{PU}^2 + \sigma_{TU}^2},$$

$$h^2 = \frac{\sigma_a^2}{\sigma_a^2 + \sigma_e^2},$$

$$\sigma_e^2 = \sigma_{PU}^2 + \sigma_{TU}^2.$$

The resulting residuals for the phenotypes are then generated via:

$$\epsilon = \frac{1}{n} \sum_{i=1}^{n} \sigma_{TU} L X_i + \sigma_{PU} L X_{PU},$$

with $X_i, X_{PU} \sim N(0_p, \mathbf{1}_p), A = L^T L$ and A being the target residual correlation matrix (**new.residual.correlation**).

15.2 Genotype – by – environment

To model GxE, we recommend the generation of multiple different traits. Usually, one should make sure that those traits have high correlation (e.g. via shuffle.cor or a specific real.bv.add) and similar scaling (mean.target / var.target / bv.standardization()). These separate traits can be linked via the function combine.traits(), which will lead to multiple traits being viewed as the same trait in a breeding value estimation. An example on the use can be found in section XXX.

16 User-interface

16.1 MoBPSweb

In addition to the R-package itself, we also developed a web-based interface that can be used an relatively easy way to get started with the framework. This have also been published as MoBPSweb (Pook et al. 2021). Nonetheless, the user-interface is still in active development and available at www.mobps.de. Extended documentation on the use of the web-interface can be found on the webpage directly.

When looking through the openly available code you will find code snippets that are only relevant for the interface (*json.simulation()*).

The main goal of the user interface is the usage of the R-package without the need for programming skills in R or knowledge of the details of the package to set up your simulation. Note that the interface will not be able to grasp the full functionality/efficiency of the R-package but the goal is to get close. Input parameters can be entered in a web-based application (java-script) – especially the breeding scheme can be entered in an intuitive way via nodes (cohorts of individuals) and edges (breeding & selection processes).

Simulations can be directly started via the web interface with a VM server hosted from Goettingen. At the current state we can provide computational resources for smaller breeding programs (20 CPU, 64 GB Memory), but it is also possible to download the JSON-file containing all information of the user interface and run the simulation via *json.simulation()* in R. When running simulation directly in R, additional parameter settings are provided to export the population at a give time point (**export.population** via **export.gen/timepoint**), scale the size of the cohorts (**size.scaling**) and similar in accordance to *breeding.diploid()*.

Student users are allowed to use 5 cores & 20 GB memory with a maximum run-time of 4 hours.

Professional users are allowed to use 5 cores & 30 GB memory with a maximum run-time of 48 hours.

If you feel you need more computational ressources or are interested in setting up your own MoBPS webserver contact use directly (Torsten.pook@uni-goettingen.de).

A schematic overview of the structure of the MoBPSweb interface:
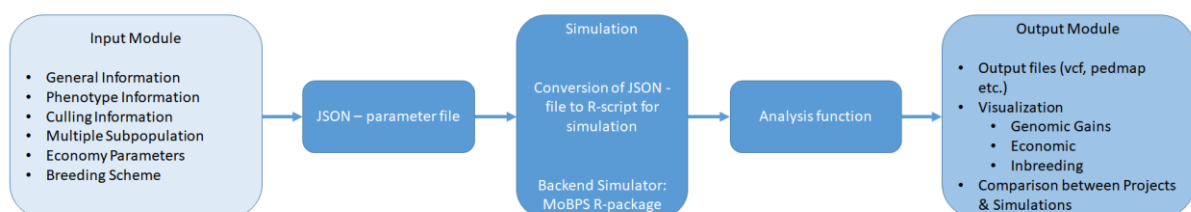
For most inputs, we provide implemented help buttons to briefly explain what kind of input is expected. The exemplary JSON-script provided in the R-package would look like this (note that all advanced parameter options are deactivated to not further complicate things):

16.2 Input modules

Phenotype Information ⓘ

Click here to read guildelines for Traits table, Upload Excel File, Correlation tables, SI, PC:

Add new phenotype | Show/Hide 3 phenotypes | Show/Hide QTLs | Show/Hide residual correlation | Show/Hide genetic correlation

Phenotype ⓘ	Unit ⓘ	Pheno. Mean ⓘ	Pheno. SD ⓘ	Heritability ⓘ	# Polygenic Loci ⓘ	Major QTL ⓘ	Value per unit (€) ⓘ	Show Cor	
Milk	liters	9300	900	0.35	1000	0	0,30	☑	X
Fat	%	3.9	0.4	0.4	1000	0	100	☑	X
Protein	%	3.4	0.3	0.38	100	0	100	☑	X

Click here to Upload Excel file for Correlation: ⓘ

Residual Correlation ⓘ

	Milk	Fat	Protein
Milk	1	0.1	0.3
Fat	0.1	1	-0.2
Protein	0.3	-0.2	1

Enter Phenotypic correlation instead of residual correlation ☐

Genetic Correlation ⓘ

	Milk	Fat	Protein
Milk	1	0.3	0.4
Fat	0.3	1	0.1
Protein	0.4	0.1	1

Breeding Scheme

Expand flash enviroment : ⓘ ☐

Click here to read the guildelines for Breeding Scheme:

Legends

Nodes:
- ☐ Male
- ☐ Female
- ☐ Both

Edges:
- ↗ Selection
- ↗ Reproduction
- ↗ Aging
- ↗ Combine
- ↗ Repeat
- ↗ Split
- ↗ Cloning
- ↗ Selfing
- ↗ DH-Production
- ↗ Semen-collection
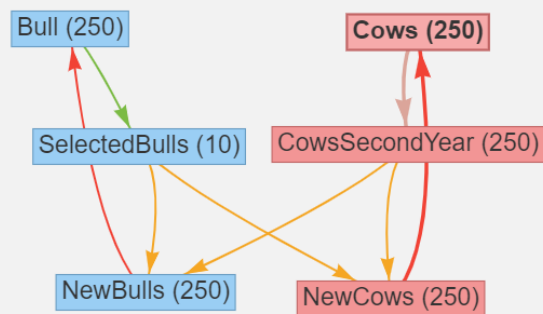
16.3 Output modules

After the simulations are executed, the resulting population-list can be download in R and then be manually analyzed. Alternatively, we also provide some basic evaluation functions. In case multiple

simulations are performed it is also possible to analyze the average between multiple runs of the simulation:

16.4 Compare Projects

The CompareProjects – module can be used to plot multiple different projects in a joint graph. All output modules available for single projects are also available for CompareProjects.

MoBPS Compare Projects

Select multiple projects to compare : ["Rinderbeispiel_v4", "ssBLUP_BVE", "Low_SelectionIntensity"]

```
456Simple_Cattle
Rinderbeispiel_v2
Rinderbeispiel_v3
Rinderbeispiel_v3_BVE
Rinderbeispiel_v3_gBVE
Rinderbeispiel_v3_short
Rinderbeispiel_v3a
Rinderbeispiel_v3a
Rinderbeispiel_v3_weights
Rinderbeispiel_v3_intensity
Rinderbeispiel_v4
Base-line
Base-line-single
ssBLUP_BVE
pedigree_BVE
Low_SelectionIntensity
Short_GInterval
Change_IndexWeigths
ssBLUP_BVE_single
Change_IndexWeights
Example_visualization
simianer_basic
```

Maximum number of repeats to analyze: ⓘ 2
Display 95% Confidence Intervals ☑ Display Legend ☑

Results: True Breeding Values

[Run Analysis]

Select plotting type: By Repeats ▾
Select cohorts (multiple selection possible): [Plot Results]

[Base-line_calf+ (25 Repeats) ×] [Low_SelectionIntensity_calf+ (25 Repeats) ×] [ssBLUP_BVE_calf+ (25 Repeats) ×] [× ▾]

RZM (Milk)

— Base-line_calf+
— Low_SelectionIntensity_calf+
— ssBLUP_BVE_calf+

Repeats
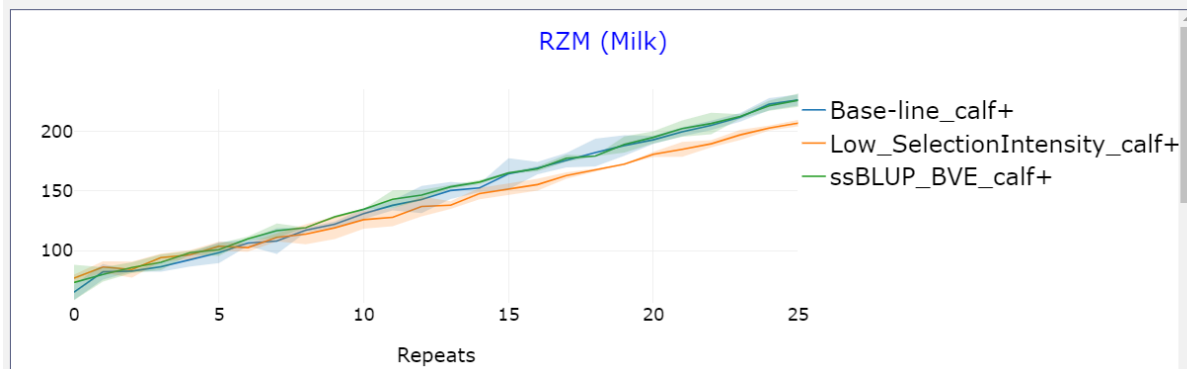
17 References

Akdemir, Deniz; Okeke, U. G. (2015): EMMREML. Fitting mixed models with known covariance structures. In: *R package version* 3 (1).

Bekele, Wubishet A.; Wieckhorst, Silke; Friedt, Wolfgang; Snowdon, Rod J. (2013): High-throughput genomics in sorghum. From whole-genome resequencing to a SNP screening array. In: *Plant biotechnology journal* 11 (9), S. 1112–1125.

Browning, Brian L.; Zhou, Ying; Browning, Sharon R. (2018): A One-Penny Imputed Genome from Next-Generation Reference Panels. In: *The American Journal of Human Genetics* 103 (3), S. 338–348.

Covarrubias-Pazaran, Giovanny (2016): Genome-assisted prediction of quantitative traits using the R package sommer. In: *PLOS ONE* 11 (6), e0156744.

Endelman, Jeffrey B. (2011): Ridge regression and other kernels for genomic selection with R package rrBLUP. In: *The Plant Genome* 4 (3), S. 250–255.

Groenen, Martien A.; Wahlberg, Per; Foglio, Mario; Cheng, Hans H.; Megens, Hendrik-Jan; Crooijmans, Richard PMA et al. (2009): A high-density SNP-based linkage map of the chicken genome reveals sequence features correlated with recombination rate. In: *Genome Research* 19 (3), S. 510–519.

Hill, William G. (1974): Prediction and evaluation of response to selection with overlapping generations. In: *Animal Science* 18 (2), S. 117–139.

Jenko, Janez; Gorjanc, Gregor; Cleveland, Matthew A.; Varshney, Rajeev K.; Whitelaw, C. Bruce A.; Woolliams, John A.; Hickey, John M. (2015): Potential of promotion of alleles by genome editing to improve quantitative traits in livestock breeding programs. In: *Genetics Selection Evolution* 47 (1), S. 55.

Lee, Michael; Sharopova, Natalya; Beavis, William D.; Grant, David; Katt, Maria; Blair, Deborah; Hallauer, Arnel (2002): Expanding the genetic map of maize with the intermated B73× Mo17 (IBM) population. In: *Plant Molecular Biology* 48 (5-6), S. 453–461.

Legarra, Andres; Christensen, Ole F.; Aguilar, Ignacio; Misztal, Ignacy (2014): Single Step, a general approach for genomic selection. In: *Livestock Science* 166, S. 54–65.

Liu, Jiajun; Luo, Wei; Qin, Nana; Ding, Puyang; Zhang, Han; Yang, Congcong et al. (2018): A 55 K SNP array-based genetic map and its utilization in QTL mapping for productive tiller number in common wheat. In: *Theoretical and Applied Genetics* 131 (11), S. 2439–2450.

Ma, Li; O'Connell, Jeffrey R.; VanRaden, Paul M.; Shen, Botong; Padhi, Abinash; Sun, Chuanyu et al. (2015): Cattle sex-specific recombination and genetic control from a large pedigree analysis. In: *PLoS genetics* 11 (11), e1005387.

Martini, Johannes W. R.; Gao, Ning; Cardoso, Diercles F.; Wimmer, Valentin; Erbe, Malena; Cantet, Rodolfo J. C.; Simianer, Henner (2017): Genomic prediction with epistasis models: on the marker-coding-dependent performance of the extended GBLUP and properties of the categorical epistasis model (CE). In: *BMC bioinformatics* 18 (1), S. 3.

Meuwissen, T. H. E. (1997): Maximizing the response of selection with a predefined rate of inbreeding. In: *Journal of animal science* 75 (4), S. 934–940.

Microsoft Corporation and Steve Weston (2018): doParallel: Foreach Parallel Adaptor for the 'parallel' Package. Online verfügbar unter https://CRAN.R-project.org/package=doParallel.

Miesenberger, Josef (1997): Zuchtzieldefinition und Indexselektion für die österreichische Rinderzucht: na.

Pérez, Paulino; de los Campos, Gustavo (2014): Genome-wide regression & prediction with the BGLR statistical package. In: *Genetics*, 483-495.

Pook, T.; Büttgen, L.; Ganesan, A.; Ha, N. T.; Simianer, H. (2020): MoBPSweb. A web-based framework to simulate and compare breeding programs. In: *bioRxiv*, 2020.07.08.193227. DOI: 10.1101/2020.07.08.193227.

Pook, Torsten; Weigend, Steffen; Simianer, Henner (2017): A generalized approach to calculate expectation and variance of kinship in complex breeding schemes. #26908. In: *Annual Meeting of EAAP*.

Prieur, Vincent; Clarke, Shannon M.; Brito, Luiz F.; McEwan, John C.; Lee, Michael A.; Brauning, Rudiger et al. (2017): Estimation of linkage disequilibrium and effective population size in New Zealand sheep using three different methods to create genetic maps. In: *BMC Genetics* 18 (1), S. 68.

Purcell, Shaun; Neale, Benjamin; Todd-Brown, Kathe; Thomas, Lori; Ferreira, Manuel A. R.; Bender, David et al. (2007): PLINK. A tool set for whole-genome association and population-based linkage analyses. In: *The American Journal of Human Genetics* 81 (3), S. 559–575.

Renaud Gaujoux (2018): doRNG: Generic Reproducible Parallel Backend for 'foreach' Loops. Online verfügbar unter https://CRAN.R-project.org/package=doRNG.

Rohrer, Gary A.; Alexander, Leeson J.; Keele, John W.; Smith, Tim P.; Beattie, Craig W. (1994): A microsatellite linkage map of the porcine genome. In: *Genetics* 136 (1), S. 231–245.

Schlather, Martin (2020): Efficient Calculation of the Genomic Relationship Matrix. In: *bioRxiv*, 2020.01.12.903146. DOI: 10.1101/2020.01.12.903146.

Simianer, Henner; Pook, Torsten; Schlather, Martin (2018): Turning the PAGE - the potential of genome editing in breeding for complex traits revisited. In: *World Congress on Genetics Applied to Livestock*, S. 190.

VanRaden, Paul M. (2008): Efficient methods to compute genomic predictions. In: *Journal of Dairy Science* 91 (11), S. 4414–4423.

Wen, Weie; He, Zhonghu; Gao, Fengmei; Liu, Jindong; Jin, Hui; Zhai, Shengnan et al. (2017): A high-density consensus map of common wheat integrating four mapping populations scanned by the 90K SNP array. In: *Frontiers in plant science* 8, S. 1389.

Zerbino, Daniel R.; Achuthan, Premanand; Akanni, Wasiu; Amode, M. Ridwan; Barrell, Daniel; Bhai, Jyothish et al. (2018): Ensembl 2018. In: *Nucleic acids research* 46 (D1), D754-D761.

Zheng, Chaozhi; Boer, Martin P.; van Eeuwijk, Fred A. (2015): Reconstruction of genome ancestry blocks in multiparental populations. In: *Genetics* 200 (4), 1073-1087.

18 Citation

The R-package MoBPS is published in G3 Genes, Genomes, Genetics.

For miraculix we recommend to cite our preprint at biorvix miraculix (Schlather 2020):

```
@article{Pook.2020,
 author = {Pook, Torsten and Schlather, Martin and Simianer, Henner},
 year = {2020},
 title = {MoBPS - Modular Breeding Program Simulator},
 pages = {g3.401193.2020},
 issn = {2160-1836},
 journal = {G3: Genes, Genomes, Genetics},
 doi = {10.1534/g3.120.401193}
}

@article {Schlather.2020,
        author = {Schlather, Martin},
        title = {Efficient Calculation of the Genomic Relationship Matrix},
        elocation-id = {2020.01.12.903146},
        year = {2020},
        doi = {10.1101/2020.01.12.903146},
        publisher = {Cold Spring Harbor Laboratory},
        URL = {https://www.biorxiv.org/content/early/2020/01/14/2020.01.12.903146},
        eprint = {https://www.biorxiv.org/content/early/2020/01/14/2020.01.12.903146.full.pdf},
        journal = {bioRxiv}
}
```

19 Acknowledgements

MoBPS has been adopted by Wageningen Livestock Research with all development since 09/2022 happening from here.
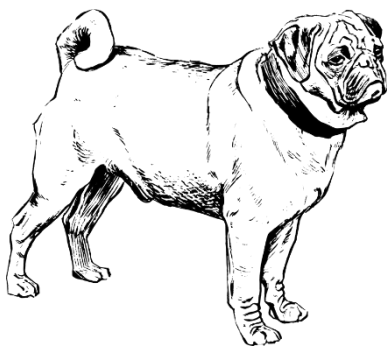
References

Fletcher, Reeves; Reeves, Colin M. (1964): Function minimization by conjugate gradients. In: *The computer journal* 7 (2), S. 149–154.

Malécot, Gustave (1948): Mathématiques de l'Hérédité. Paris: Masson et Cie.

Meuwissen, T. H. E. (1997): Maximizing the response of selection with a predefined rate of inbreeding. In: *Journal of animal science* 75 (4), S. 934–940.

Pook, Torsten; Büttgen, Lisa; Ganesan, Amudha; Ha, Ngoc-Thuy; Simianer, Henner (2021): MoBPSweb: A web-based framework to simulate and compare breeding programs. In: *G3: Genes, Genomes, Genetics* 11 (2). DOI: 10.1093/g3journal/jkab023.

Pook, Torsten; Schlather, Martin; Simianer, Henner (2020): MoBPS - Modular Breeding Program Simulator. In: *G3: Genes, Genomes, Genetics* 10 (6), 1915-1918. DOI: 10.1534/g3.120.401193.

Wellmann, Robin (2017): optiSel: optimum contribution selection and population genetics. In: *R Package. version 0.9.1*.

Wellmann, Robin (2019): Optimum contribution selection for animal breeding and conservation: the R package optiSel. In: *BMC Bioinformatics* 20 (1), S. 1–13.

Zhuang, Yongwen (2019): cPCG: Efficient and Customized Preconditioned Conjugate Gradient Method for Solving System of Linear Equations 2019. Online verfügbar unter https://github.com/styvon/cPCG.