CiBreed
Center for Integrated Breeding Research

MOBPS

Modular Breeding Program Simulator

Torsten Pook, Martin Schlather, Henner Simianer

14. FEBRUAR 2019

UNIVERSITY OF GOETTINGEN
Animal Breeding and Genetics Group

Table of Contents

1 General

MoBPS is an R-package to simulate complex and large scale breeding programs with 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. The actual process of the simulation can be split up into two steps: the creation of a starting population and the simulation of breeding processes.

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 understanding of the workflow we refer to section 14 for exemplary simulations. For a list of all input parameters and possible initializations, we refer to section 11.

Note that this package is under current development and still contains bugs. Simulations itself (meiosis, breeding values, estimation etc.) are well tested and should be trustworthy – it is more about typos in recently added parts that are crashing the tool and similar.

As MoBPS is still in development process all results should be regarded **with caution** and **we do not take any liability for errors in the simulation nor guaranty warranty.**

For questions regarding the tool or how to set up your simulation contact me (Torsten.pook@uni-goettingen.de). We are always happy for questions as it really helps improve the tool – documentation can always be improved. For quick reply it would help to provide a small example of our problem (ideally the population-list as a .RData object)

2 Installation

MoBPS requires R 3.3 or higher. We highly recommend the use of the packages RandomFieldsUtils (version 0.4.0+) and miraculix (0.3.2+) as they significantly reduce computation time when working with a high number of markers/individuals. All functionality **should** be able to run without both packages. For installation of the package we recommend the usage of the R function *install.packages* (under windows set type="source", repo=NULL). Usage was tested on Linux and Windows. The usage on Mac OS is currently not recommended.

For Windows the installation of Rtools is required. Some machines additionally require devtools.

3 Citation

There is currently no paper published on our R-package. This will hopefully soon change. For so long we suggest using following to citations for the R-packages MoBPS and miraculix:

```
  @Manual{,
    title = {MoBPS: Simulation of breeding programs},
    author = {Torsten Pook},
    year = {2018},
    note = {R package version 1.0.0},
  }

  @Manual{,
    title = {miraculix: Statistical Functions for Animal Breeding},
    author = {Malena Erbe and Martin Schlather and Florian Skene and Imran
S. Haque},
```

```
    year = {2018},
    note = {http://www.uni-goettingen.de/de/129970.html,
http://ms.math.uni-mannheim.de},
  }
```

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 the 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).

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 **vcfpath**. 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(0.5) X_1 X_2 X_3 … /X_1 X_2 X_3… with X_i~B(0.5)). To use different allele frequencies, provide them in a vector starting from the first to the last marker in the parameter **freq**. To draw allele frequencies from a uniform/beta distribution set **freq** to "unif" / "beta". Shape parameters for the beta distribution can be set via **beta_shape1** & **beta_shape2** (default: 1 & 1 – this is a uniform distribution but simulation is slightly slower).

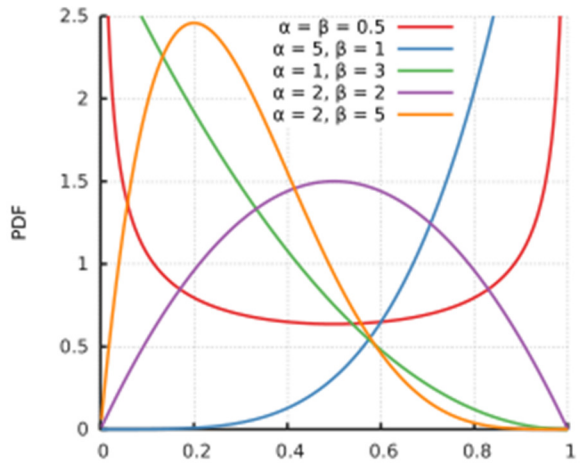


Figure 1: https://en.wikipedia.org/wiki/Beta_distribution

To generate an LD and haplotype structure without using real data we recommend to start with one of the simple datasets and simulate some random/non-random mating generations using *breeding.diploid()* (cf. Section 5). Wrapper functions for the automatic generation of those base populations in MoBPS are planned but not yet implemented.

If using a vcf file, the chromosome, marker name and base pair position are automatically exported. 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.

In case more markers are to be added later, 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).

4.2 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 don't need more you can just skip to section 4.2.24.2.2)

Note that this is the generation of an actual genetic value that is underlying each individual of the population. Normally you cannot observe this in reality, you typical 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).

4.2.1 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 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.

4.2.2 Predefined genetic architectures

In case of a predefined genetic architecture all markers are assigned 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 9.3) or in the last run of *creating.diploid()*.

The number of additive (**n.additive**) and dominate (**n.dominant**) QTL as well as effects caused by qualitative (**n.qualitative**) and quantitative (**n.quantitative**) epistasis can be included directly. To assign the variance, one can input a vector containing the variance for each effect (**var.additive, var.dominant,…).** On default each variance is set to 1 and effects are drawn from a Gaussian distribution.

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 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 other samples.

To simulate more than one trait use vectors for **n_**XXX and lists for **var_**XXX instead.

4.2.3 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 simulation is done via sampling from a Gaussian distribution and genetic traits do not fulfil all requirements of a dependent

multivariate Gaussian distribution (which is here used to model dependency), the obtained resulting correlations can be different to the correlation set in **new.breeding.correlation** if non-QTL traits have correlations with QTL-traits. We are currently working on alternatives for this.

4.2.4 Position of Markers

For our simulations, the physical position in base pairs does not really matter – instead we are interested in a position in Morgan. 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). This would typically be the case when using base pair positions in **snp.positions**. For joint generation of multiple chromosomes enter a vector instead.

Another way of entering the genetic position (in M) 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.

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. 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 (NOT the default of the male parameter).

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 previous 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.sex** instead.

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). 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 for example be used to model migration in store groups of different genetic origin but same sex and time of existence (generation).

To control which individuals are used every cohort generated can be named via **name.cohort** in *breeding.diploid()* & *creating.diploid()*. The cohorts to use in the selection process can be chosen by putting cohort names as inputs in **best1.from.cohort** (first (usually male) parent) and **best2.from.cohort** (second (usually female) parent). Note that the share of individuals to pick from which of these cohorts is currently not possible. To use groups (generation + sex) use **best1.from.groups** & **best2.from.groups** instead.

To combine individuals to a new cohorts of individuals set **combine** to TRUE. This will generate a new cohort of all selected individuals - do not combine male and females individuals!

Alternatively, the old implementation can still be used. This mode provides slightly more flexibility but the input structure requires a deeper understanding of the storage system of individuals and by this is more prune to errors in the input for inexperienced users. To control which generations are used in the mating, use the parameter **used.generations.m**. This is a vector controlling what share of **selection.size** is assign to each generation. The last element of the vector is the share of the last generation of the population – in case the vector is shorter than the number of generations all previous values are set to 0 (e.g. c(20,0,30,50)) would lead to 50% of the selected individuals coming from the last generation, 30% of the second to last and 20% of the forth to last.

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 be used, use the parameter **class.m**, containing a vector of all usable classes.To control the class of the new individuals 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.

To generate multiple offspring from the same dam/sire pair set **repeat.mating** to the desired number.

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 to track the simulation procedure.

5.2 Control of heritability, breeding values and phenotypes

For each individual, an underlying true genetic value is calculated for each trait. Based on this, phenotypes can be generated. For with individuals to generate new phenotypes can be controlled via **new.bv.observation.gen**, **new.bv.observation.database** and **new.bv.observation.cohorts**. For a quick input of all individuals previously not phenotyped set **new.bv.observation** to “all” or “non_obs” for all or all previously not phenotyped individuals. On 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. Note that the number of times an observation for an individual is generated does matter since the environmental variance will be reduced with each observation previous observations will be used). To model a correlation between the environmental variances for different traits set provide the desired correlation matrix via **new.phenotyp.correlation** (this can also

be done in *creating.diploid()*). For the simulation of correlated genetic values we refer to section 4.2.3.

The environmental variance $\sigma_e$can be controlled by the usage of **sigma.e**. This can either be set to a fixed numeric value or be estimated to fulfill a target heritability. For the second possibility, the genetic variance is calculated based on the individuals specified in **sigma.e.gen**, **sigma.e.database** and **sigma.e.cohorts** and the needed environmental variance is then calculated by the usage of a predefined **heritability**. You can also use the environmental variance of the previous simulation by setting **use.last.sigma.e** to TRUE. A change in the genetic variance **sigma.g** is not recommended but in principle possible (this will only affect the breeding value estimation). On default it is estimated using all individuals used in the current breeding value estimation (set **forecast.sigma.s** to FALSE to deactivate). To specify with groups to use to estimate σ_g use **sigma.g.gen**, **sigma.g.database**, **sigma.g.cohorts**.

For newly created individuals the phenotype is set to the mean of the phenotypes of both parents. Alternatively, one can set it to zero or create an observation by setting **new.bv.child** to “zero” or “obs” instead of “mean”.

In some applications, the genetic value of the individual itself is not of importance – instead the performance of its offspring is of relevancy. To use the average of all offspring of an individual as its phenotype activate **bve.childbase** and the computation is performed for all groups in **bve.childbase.parents**. To exclude offspring set **bve.childbase.children** to those groups that should be used.

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.

Scaling in case of index selection with multiple traits can only be performed in the selection process itself.

5.3 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**.

As there are a lot of different ways to perform breeding value estimation we implemented multiple variants:

1. GBLUP with assumed known heritability and direct solving of the mixed model without REML variance component estimation
2. Bayesian approaches implemented in BGLR
3. GBLUP using EMMREML
4. GBLUP using sommer
5. Pedigree-based BLUP via breedR
6. Own function

In any case estimated breeding values are entered for all individuals unless **bve.insert.gen**, **bve.insert.database** or **bve.cohorts** directly classifies for which groups breeding values are to be entered.

For the calculation of G we offer multiple methods with **computation.A**="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"). 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 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.

5.3.1 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. Note that this is still an empirical measure that can change when using different individuals in the estimation process. Especially for bigger populations heritability estimation should not be problematic and is not performed in each breeding value estimation in practice as well. To instead estimate the additive genetic variance using a parental model activate **estimate.add.gen.var**.

5.3.2 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 is used – alternatively one can use BayesA, BayesB, BayesC by setting **BGLR.method** to "BayesA", "BayesB", "BayesC" instead of "RKHS". (Currently only RKHS!...)

To control the number of the burn-in and iterations use **BGLR.burnin** and **BGLR.iteration**. To deactivate printing of results of 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 tents to crash when the same path is used multiple times. Activating **BGLR.save.random** will hinder this.

5.3.3 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.

5.3.4 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.

5.3.5 Pedigree-based (breedR)

Traditional breeding value estimation using pedigree data. To activate set **breedR.bve** to TRUE – especially for bigger populations this is much faster computation wise. Have to double-check estimation technique used – heritability seems to be underestimated!

Alternativly the pedigree-based relationship matrix can also be used in all other methods by setting computation.A to “kinship”. Takes about the same time as with other relationship matrices then (No usage of high number of zeros in the dataset).

5.3.6 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 ones 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.database = cbind(4, c(1,2)),
                               new.bv.observation=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 6. For details on importing function, we refer to section 7.

5.3.7 Calculating marker effects & GWAS

For some applications (e.g. gen editing) it is necessary to identify causal markers. Although they 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 5.3.1 (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 same as for 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 really basic implementation of GWAS with no correction for population structure or similar.

5.4 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 totally random. To use estimated breeding values as a selection criteria 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 store details on which individuals were selected, which mating 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.type** to “pheno”, “bv” or “bve” (default: “bve”).

5.4.1 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** 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**.

5.4.2 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 ratio is the ratio between the frequency 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 being the first one.

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

5.4.3 Maximum number of offspring per individual

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 sexs or a vector with the first value coding the maximum for male and the second the one for female.

5.4.4 Plant breeding (no-sexes & selfing & dh-production & cloning)

For some applications the sex of an individuals is not relevant (or and individual does not even have a sex). Even though a sex is still stored internal it might be neglectable for the application at hand. In this case one can allow matings between individuals from the same sex by usage of **same.sex.activ**. The probability to select a female individuals as a parent can be set via **same.sex.sex** (default=0.5). To additionally allow for selfing set **same.sex.selfing** to TRUE. 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 and exact copy of an individual active **copy.individual**. Instead of simulating meiosis both chromosome sets of the selected first parent (usually 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).

5.4.5 Targeted/Fixed mating/Manual selection of individuals

If none of the previously described methods works for your simulation, you can also just 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 coding the likelihood of the offspring being female.

To provide a manual list of selected individuals use the parameters **best1.from.group** and **best2.from.group**. If only one of the two is provided the other one will be calculated automatically.

5.4.6 Gene-Editing

With increasing popularity of methods like CRISPR/Cas 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 usage of the predictions via rrBLUP/GWAS in 3.3.5. We only count actually performed edits – if an allele is already beneficial the next best marker is edited instead. Although in practice not possible 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 will work, all possible offspring will survive the procedure and traits are as simple as designed (usually single marker QTL).

5.5 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**. 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^-5). A mutation back to the reference is assigned the probability of **remutation.rate** (default: 10^-5). Those values tend to be a lot higher than what you would expect in nature. Depending on the base pair one would expect something around 10^-8 for a random loci.

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!

5.6 Other

5.6.1 Culling / Death

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”).

5.6.2 Parameter for target-mating with J.W.R. Martini

Don’t think anyone needs documentation here – code is specific to the planned paper with Johannes and does not generalize (quick and dirty - https://github.com/Droogans/unmaintainable-code)

martini.selection / Special.comp / Special.comp.add / Max_auswahl / Predict.effects / SNP.density / Use.effect.markers / Use.effect.combination

5.6.3 Allele-frequency per generation

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

5.6.4 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().

5.6.5 save.recombination.history

To store the time of occurance 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!)

5.7 Storage & computation time

To store the time needed in each step of the simulation tool activate **store.comp.times**. For more details of the breeding value estimate you can additionally activate **store.comp.times.bve**. Activation of **Rprof** can provide even more information.

5.7.1 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 of things is not recommended –use following settings in *breeding.diploid()* instead:

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.ursprung: 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.

5.7.2 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 by setting the parameter **miraculix** to TRUE when creating the founder population in *creating.diploid()*. For more on this we refer to section 10.

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 but doMPI is supported as well – we highly recommend the use of miraculix instead (if it is working on your system)!

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.

5.7.3 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 similar computation time – but also includes screening for semi definite matrices and an automatically changed the algorithm, if needed, and thus proceeding without error.

Coming soon… Improvement from 1/3 n^3 to 1/6 n^3 possible.

5.7.4 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 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!

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

A lot of the data stored in a population list is highly compressed since saving haplotypes of all individuals of the dataset for all generations would in most cases 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 extent the package) it might be useful to understand the data structure behind. For that refer to section 5. In most cases using our predefined export functions should be enough:

In what individuals one is interested in can be controlled by usage of the parameter **gen**, **database** and/or **cohorts**. Here **cohorts** and **gen** are vectors containing all generations/cohort to include whereas **database** contains a matrix with each row coding a group to export:

```
> database
     Generation sex
[1,]          1   2
[2,]          5   1
```

This **database** will export the information for all female individuals from generation 1 and all male individuals from generation 5.

6.1 get.genos

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.

```
> 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
```

6.2 get.haplos

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.

```
> 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
```

6.3 get.bv / get.bve / get.pheno

These functions will export the true underlying breeding value ("bv"), the estimated breeding value ("bve") and the phenotype ("pheno").

```
> 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
```

6.4 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 our example 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"
```

6.5 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=12)
> ped[1:10,]
      offspring father      mother
 [1,] "M1_12"   "M2214_11" "W1776_11"
 [2,] "M2_12"   "M2052_11" "W1125_11"
 [3,] "M3_12"   "M1529_11" "W1904_11"
 [4,] "M4_12"   "M1712_11" "W1256_11"
 [5,] "M5_12"   "M221_11"  "W326_11"
```

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=12)
> ped[1:5,]
     offspring grandfatherf grandmotherf grandfatherm grandmotherm
[1,] "M1_12"   "M734_10"    "W2135_10"   "M2342_10"   "W313_10"
[2,] "M2_12"   "M1188_10"   "W1436_10"   "M489_10"    "W539_10"
[3,] "M3_12"   "M1702_10"   "W1860_10"   "M413_10"    "W1234_10"
[4,] "M4_12"   "M1560_10"   "W1211_10"   "M1649_10"   "W254_10"
[5,] "M5_12"   "M1649_10"   "W2093_10"   "M1593_10"   "W2390_10"
> ped <- get.pedigree3(population, gen=12)
> ped[1:5,]
     offspring father     mother     grandfatherf grandmotherf grandfatherm grandmotherm
[1,] "M1_12"   "M2214_11" "W1776_11" "M734_10"    "W2135_10"   "M2342_10"   "W313_10"
[2,] "M2_12"   "M2052_11" "W1125_11" "M1188_10"   "W1436_10"   "M489_10"    "W539_10"
[3,] "M3_12"   "M1529_11" "W1904_11" "M1702_10"   "W1860_10"   "M413_10"    "W1234_10"
[4,] "M4_12"   "M1712_11" "W1256_11" "M1560_10"   "W1211_10"   "M1649_10"   "W254_10"
[5,] "M5_12"   "M221_11"  "W326_11"  "M1649_10"   "W2093_10"   "M1593_10"   "W2390_10"
```

6.6 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] "Founder_M"      "Founder_W"      "Cows"           "Bulls"          "Selected_bulls"
> get.cohorts(population, extended=TRUE)[1:5,]
     name            generation male individuals female individuals class position first male position first female time point creating.type
[1,] "Founder_M"     "1"        "50"             "0"               "0"   "1"                "0"                    "0"       "0"
[2,] "Founder_W"     "1"        "0"              "50"              "0"   "0"                "1"                    "0"       "0"
[3,] "Cows"          "2"        "0"              "50"              "1"   "1"                "1"                    "1"       "2"
[4,] "Bulls"         "2"        "50"             "0"               "1"   "1"                "51"                   "1"       "2"
[5,] "Selected_bulls" "3"       "10"             "0"               "2"   "1"                "1"                    "1"       "1"
```

6.7 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
    0     0     0     0     0     0     0     0     0     0     0     0     0
M14_1 M15_1 M16_1 M17_1 M18_1 M19_1 M20_1 M21_1 M22_1 M23_1 M24_1 M25_1 M26_1
    0     0     0     0     0     0     0     0     0     0     0     0     0
```

…

```
W42_1 W43_1 W44_1 W45_1 W46_1 W47_1 W48_1 W49_1 W50_1  M1_2  M2_2  M3_2  M4_2
    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
    1     1     1     1     1     1     1     1     1     1     1     1     1
M18_2 M19_2 M20_2 M21_2 M22_2 M23_2 M24_2 M25_2 M26_2 M27_2 M28_2 M29_2 M30_2
    1     1     1     1     1     1     1     1     1     1     1     1     1
```

6.8 get.time.point

This function 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=1:2)
 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 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
```

…

```
W41_1 W42_1 W43_1 W44_1 W45_1 W46_1 W47_1 W48_1 W49_1 W50_1  M1_2  M2_2  M3_2  M4_2  M5_2
    0     0     0     0     0     0     0     0     0     0     1     1     1     1     1
 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 M19_2 M20_2
    1     1     1     1     1     1     1     1     1     1     1     1     1     1     1
```

6.9 get.creating.type

This function extracts the creating.type of each individuals – 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 - Reproduction_Selfing

5 - DH_Gene

6 - Cloning

7 - Combine

8 - Aging

```
> 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
```

6.10 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
```

6.11 get.vcf

Function to export genomic data in an vcf-file (currently using the synbreed-package but more efficient implementation including stored bp etc. is planned). Set **path** to the path you want to write to:

```
##fileformat=VCFv4.1
##filedate= 17980
##source="write.vcf of R-synbreed"
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
#CHROM  POS ID  REF ALT QUAL    FILTER  INFO    FORMAT  M1_3_set2   M10_3_set2  M2_3_set2   M3_3_set2
chr1    .   Chr1SNP1    A   G   .   PASS    .   GT  0|0 0|1 1|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0
chr1    .   Chr1SNP2    A   G   .   PASS    .   GT  1|0 0|1 1|1 0|1 0|1 0|1 0|0 0|0 1|0 0|0
chr1    .   Chr1SNP3    A   G   .   PASS    .   GT  0|1 1|0 0|0 1|0 0|0 1|0 1|1 0|1 0|1 0|0
chr1    .   Chr1SNP4    A   G   .   PASS    .   GT  1|0 0|1 1|0 1|1 0|1 1|1 0|0 1|0 0|0 0|0
chr1    .   Chr1SNP5    A   G   .   PASS    .   GT  0|0 0|1 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0
```

6.12 get.database

Function to merge gen, database and cohorts –info into a joint database. This is only needed internally – as it's the only internal get.X function it is still mentioned here for completeness.

```
> 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
```

7 Importing information to the population-list

7.1 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. 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)
```

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

8 Data structure of the population list

All information regarding the breeding program are 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 6.

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

8.1 $info

$info contains all general information concerning genetic architecture, size of the program and internal information needed to perform the simulations. Each entry is names 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
bv.mult.factor	(bv) * this
bv.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*
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
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)
effect.p	Markers involved as QTL in any trait
store.effect.freq	Frequency of each marker in each generation
last.sigma.e	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)

8.2 $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.]]))

8.2.1 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)
$breeding[[generation]][[13,14]]	Creating type of generation for males(13) and females (13) This is only relevant for the web-based application

```
> str(population$breeding[[1]])
 .. [list output truncated]
$ : num [1, 1:100] 46 48.9 31.5 35.7 49.6 ...
$ : num [1, 1:1000] 67.6 57.4 68.8 53 39.6 ...
$ : int [1:100] 0 0 0 0 0 0 0 0 0 0 ...
$ : int [1:1000] 0 0 0 0 0 0 0 0 0 0 ...
$ : num [1, 1:100] 33.4 47.5 31.9 59.1 38 ...
$ : num [1, 1:1000] 77.79 54.65 63.42 54.49 7.45 ...
$ : num [1, 1:100] -22.23 67.24 -9.34 -16.06 101.5 ...
$ : num [1, 1:1000] 80.4 48.1 96.2 37.1 30.5 ...
```

8.2.2 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). – for more see section 4.1 get.recombi
[[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]]	Was the individual used in any BVE (genotyping needed for cost-calculation)
[[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

```
> str(population$breeding[[3]][[1]][[1]])
List of 20
 $ : num [1:92] 0 0.218 0.352 0.619 1.072 ...
 $ : num [1:87] 0 0.414 0.668 1.677 2.573 ...
 $ : int [1:5] 108623 151503 201893 235772 330816
 $ : int [1:4] 21310 106105 172924 317461
 $ : int [1:91] 33557241 33557240 616 33557241 33557240 33557241 33557240 33557241
 $ : int [1:86] 24 25 33555017 33555016 25 24 25 24 25 24 ...
 $ : num [1:5] 2 1 60 94.7 106.7
 $ : num [1:5] 2 2 1131 96.7 105.9
 $ : NULL
 $ : NULL
 $ : NULL
 $ : NULL
 $ : NULL
 $ : NULL
 $ : num 1
 $ : num 1
 $ : NULL
 $ : NULL
 $ : num 0.323
 $ : num 0.507
```

9 Utility functions

Since the inclusion of miraculix, I did not need many utility functions any more – older version of function to show development of allele frequencies and others still exists but need modifications. Just tell me if you wish to have a certain function to help you with the package.

9.1 compute.costs

To calculate the costs of the currently simulated breeding program use compute.costs. Currently implemented cost factors include.

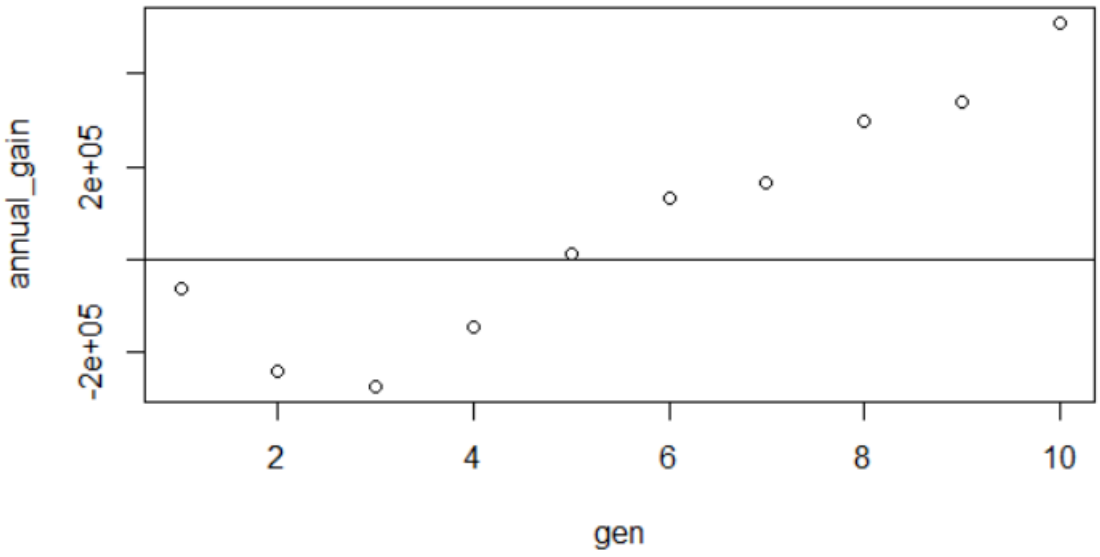
Cost factor	MoBPS -Parameter	Default
Phenotyping	phenotyping.costs	10
Genotyping	genotyping.costs	100
Fixed costs	fix.costs	0
Annual costs	fix.costs.annual	0
Profit per BV	profit.per.bv	1

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)

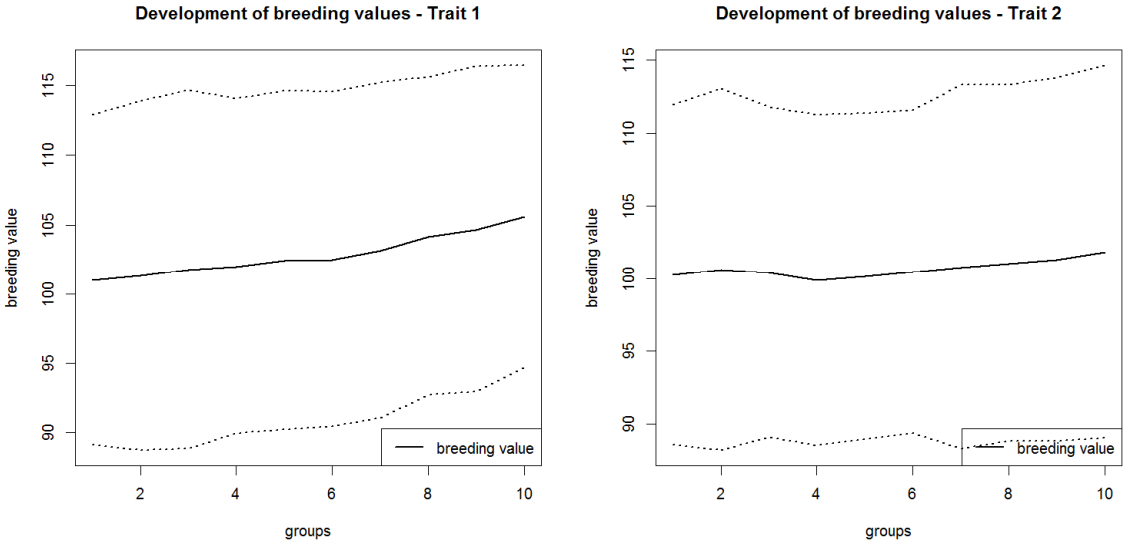
```
> compute.costs(population,gen=1:10)
 [1]  -63014.02 -242252.85 -274059.21 -144566.68   12944.92  132774.58  165916.47  299725.18  339646.66  510074.31
```



9.2 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:

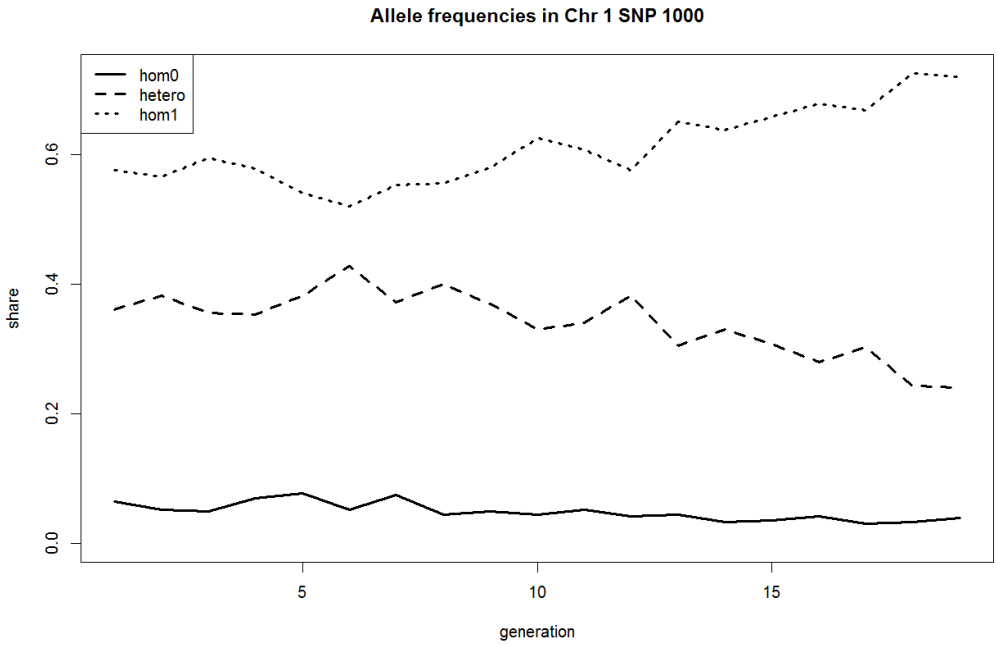
Which groups to display is selected via the parameters **gen**, **database** & **cohorts** (similar to get.X in section 6). Confidence bands are drawn for “bv” (1), “bve” (2) & “pheno”(3) – to change the quantile use the parameter **quantile** (default: 0.95), to exclude selected the for with to draw 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.



9.3 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. On default, both sexes are used to derive allele frequencies. To exclude on set include accordingly (male – 1, female -2, default c(1,2)).

```
> analyse.population(population,1,1000)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14] [,15] [,16] [,17] [,18] [,19]
[1,]   26   21   20   28   31   21   30   18   20    18    21    17    18    13    14    17    12    13     8
[2,]  144  153  142  141  153  171  149  160  148   132   136   153   122   132   123   112   121    97    48
[3,]  230  226  238  231  216  208  221  222  232   250   243   230   260   255   263   271   267   290   144
```



9.4 new.base.generation

With 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**.

9.5 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.

It is planned to incorporate this in an improved version of this in *creating.diploid()* - Coming soon...

10 Memory and computation times

Critical parts of MoBPS concerning memory requirements and computation times can be performed using the linked 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 commutation 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 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.

11 List of all parameters with exemplary inputs for all parameters

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^-5	Value between 0 and 1
remutation.rate	10^-5	Value between 0 and 1
recombination.rate	1	Any positive numeric
selection.m	"random"	"function"
selection.f	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
used.generation.m	1	Vector with at most generation elements
used.generation.f	used.generation.m	So.
compute.snps	FALSE	Don't touch – will be removed
relative.selection	FALSE	Don't touch – will be removed
class.m	0	Vector with all classes to consider
class.f	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!)
recom.f.polynom	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; 187 is reserved for dead individuals) https://en.wikipedia.org/wiki/187_(slang)
bve	FALSE	TRUE
bve.gen	NULL	1:3
bve.database	NULL	Generation sex [1,] 1 2 [2,] 5 1
bve.cohorts	NULL	c("Founder_M", "F1")
sigma.e	100	Numeric value above 0
sigma.s	100	Numeric value above 0
new.bv.observation	NULL	"all" for all individuals "non_obs" for all previously not observed
new.bv.observation.gen	NULL	1:3
new.bv.observation.database	NULL	Generation sex [1,] 1 2 [2,] 5 1
new.bv.observation.cohorts	NULL	c("Founder_M", "F1")
new.bv.child	"mean"	"zero", "obs"
computation.A	"vanRaden"	"kinship", "CE", "CM", "non_stand"
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, sex.probability) with 1 being father, 2 being 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)
store.breeding.totals	FALSE	TRUE
forecast.sigma.s	TRUE	FALSE

multiple.bve	“add”	“ranking”
multiple.bve.weights	1	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.criteria	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	NEEDS REWORK!
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	FALSE	TRUE
use.last.sigma.e	FALSE	TRUE
save.recombination.history	FALSE	TRUE
martini.selection	FALSE	TRUE
BGLR.bve	FALSE	TRUE
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
copy.individual	FALSE	TRUE
dh.mating	FALSE	TRUE
dh.sex	0.5	Numeric value between 0 and 1
bve.childbase	FALSE	TRUE
bve.childbase.parants	NULL	Generation sex [1,] 1 2 [2,] 5 1
bve.childbase.children	NULL	Generation sex [1,] 1 2 [2,] 5 1

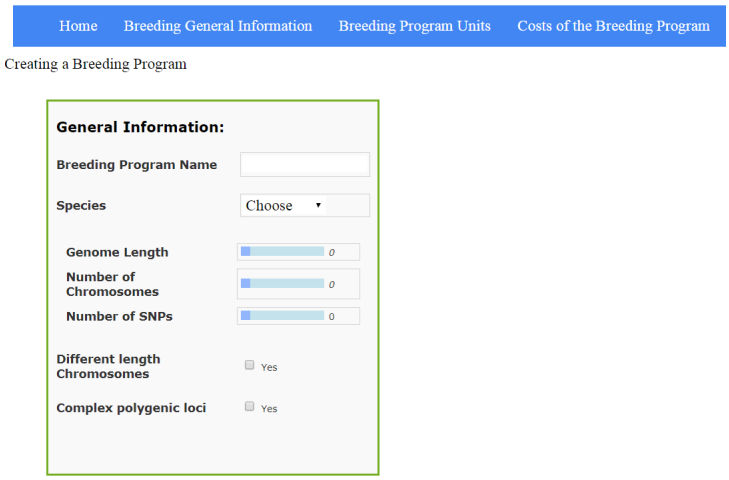
n.observation	1	Natural number
bve.0isNA	TRUE	FALSE
phenotype.bv	FALSE	TRUE
standardize.bv	FALSE	TRUE
standardize.bv.level	100	Numeric value
standardize.bv.gen	1	Natural number <= generation number
delete.same.ursprung	FALSE	TRUE
remove.effect.position	FALSE	TRUE
estimate.u	FALSE	TRUE
new.phenotyp.correlation	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
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
store.comp.times	TRUE	FALSE
store.comp.times.bve	TRUE	FALSE
special.comb	FALSE	Part of martini selection – dont use!
max.auswahl	Inf	Part of martini selection – dont use!
predict.effects	FALSE	Part of martini selection – dont use!
SNP.density	10	Part of martini selection – dont use!
use.effect.markers	FALSE	Part of martini selection – dont use!
use.effect.combination	FALSE	Part of martini selection – dont use!
import.position.calculation	NULL	Function f(cm_position) = Last previous SNP
special.comb.add	FALSE	Part of martini selection – dont use!
ogc	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"
gen.architecture.m	0	Natural number (select one of the previously stored architectures)
gen.architecture.f	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	best.selection.ratio.m	positive numeric value
best.selection.criteria.m	"bv"	"bve", "pheno"
best.selection.criteria.f	best.selection.criteria.m	"bve", "pheno"

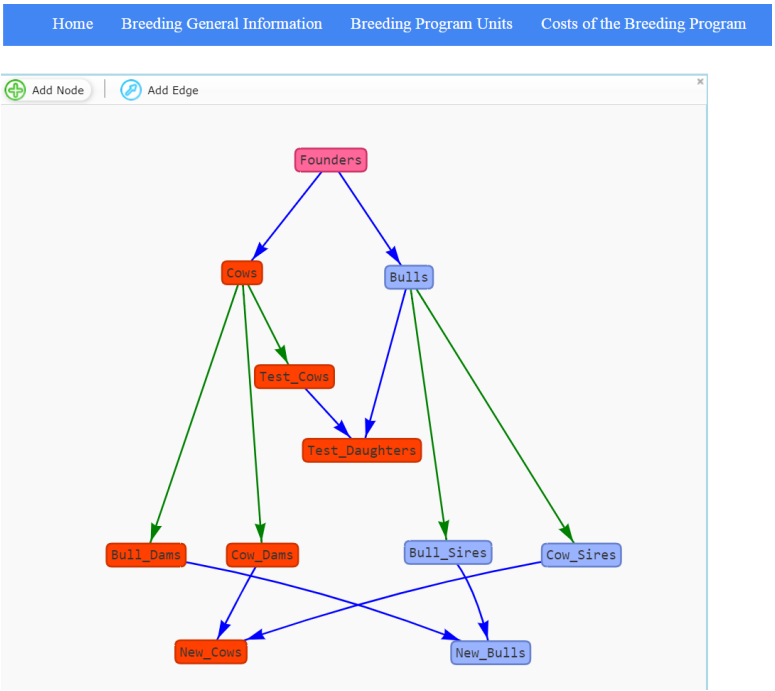
best.selection.manual.ratio.m	NULL	positive numeric value
best.selection.manual.ratio.f	best.selection.manual.ratio.f	positive numeric value
bve.class	NULL (take all!)	vector containing numeric values
Parallel.generation	FALSE	TRUE
Ncore.generation	1	Positive numeric value
Name.cohort	NULL	“Founders” 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
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 - Reproduction_Selfing # 5 - DH_Gene # 6 - Cloning # 7 - Combine # 8 - Aging

12 User-interface

As the manual input of a breeding program via R-functions can be quite tedious, the long-term plan is to, in additional to the R-package itself, release a web-based user interface. This user-interface will contain most commonly used variables as well as a tool to draw a specific breeding scheme containing of nodes (cohorts of individuals) and edges (breeding & selection processes).



Long-term simulations are supposed to be directly started via the web-interface. For this, the web-server is hosted from Goettingen but all computations needed have to run locally. For bigger simulations, one instead can automatically generate a script to run the particular simulation on a server cluster. As of now, we are mostly working on including more parameters as well as including reality-checks to avoid impossible inputs (or at least put out warnings in those cases). Additionally the direct link between the interface and R is still a concern – the translation from the JavaScript based JSON-file to R is of no issue.



The user interface is still not openly available but when interested in testing it contact me (Torsten.pook@uni-goettingen.de). Files here can be run in R via the function *json.simulation.*

Figure 2: Demo-Version of the slide to set General information about the breeding program. Genetic architecture, traits can be loaded in which pre-built templates for common species.

Figure 3: Demo-Version of the slide to input the breeding program one wants to simulate. Green edges code selection, blue edges code reproduction.

13 Commonly used words definition

Group: Group of individuals with the same sex and belonging to the same generation

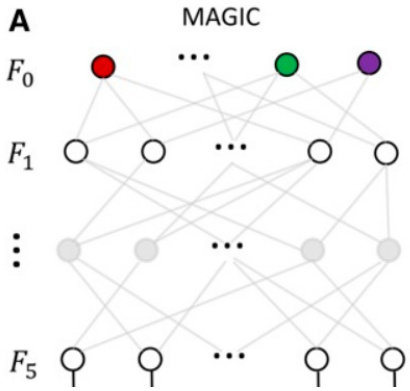
Cohort: Group of individuals with the same sex generated in a single run of *breeding.diploid()*

Class: Auxiliary variable to classify individuals in an additional dimension (besides sex & generation)

Founder: Founder individuals are the start-point of a simulation and all individuals in the population can be traced back to the founders. Because of this only for those individuals genotype/haplotype data has to be saved.

14 Exemplary scripts

14.1 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 15M genome, 32k SNPs and a total of 780 individuals took 2 seconds on one core of my local maschine without the usage of miraculix.

Non-cohort-modus:

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

# From each plant 19 offspring are generated (no-selfing, same.sex to allow for matings
# between individuals from the same sex (in this case MALE - same.sex.sex=0))
population <- breeding.diploid(population, breeding.size=c(190,0), selection.size=c(20,0),
                               same.sex.activ = TRUE, same.sex.sex=0, max.offspring = 19)

# Couple of generations of random mating between the plants of the last generation
# To keep diversity each plant is limited to 2 offspring.

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:

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

# From each plant 19 offspring are generated
# Selfing can acure here! - need to add something to it here!
population <- breeding.diploid(population, breeding.size=c(190,0), selection.size=c(20,0),
                               same.sex.activ = TRUE, same.sex.sex=0, max.offspring = c(19,0),
                               best1.from.cohort="F0", name.cohort = "F1")

# Couple of generations of random mating between the plants of the last generation
# To keep diversity each plant is limited to 2 offspring.

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),
                               best1.from.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),
                               best1.from.cohort = "F2", name.cohort = "F3")
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),
                               best1.from.cohort = "F3", name.cohort = "F4")
```

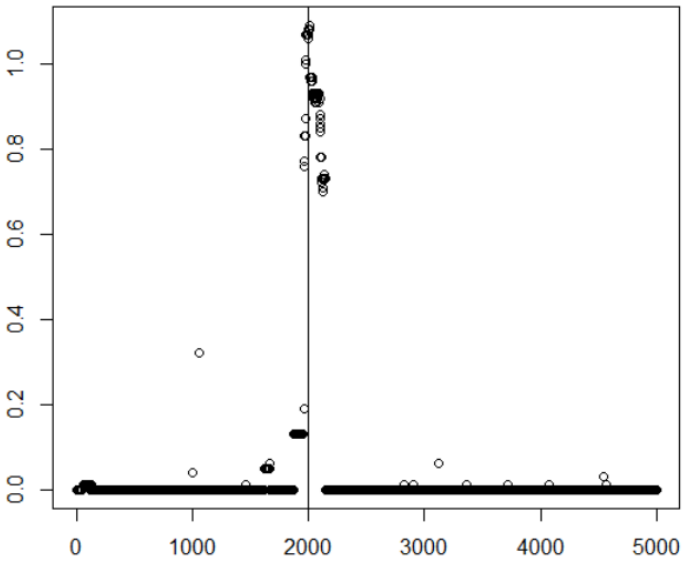
14.2 Simulation of Introgression on blue eggshell QTL

We here show how to perform an exemplary simulation of a breeding scheme to perform introgression of a single QTL.

In term of computation 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 maschine without the usage of miraculix.

```
library(MoBPS)
# Generate a starting population with allele 0 coding White-Layer, allele 1 coding wild-race
dataset1 <- matrix(1, nrow=5000, ncol=20)
dataset2 <- matrix(0, nrow=5000, ncol=20)

# Generation of traits
# Blue Eggshell QTL is on SNP 2000 on chromosome 1
# In all other positions the white layer genome is assumed to be favourable
# but all marker effects combiened are modelled as smaller as the blue eggshell QTL
trait <- c(2000,1, 0, 10000, 20000)
trait2 <- cbind(1:5000,1,1,0.5,0)

population <- creating.diploid(dataset=cbind(dataset1,dataset2), miraculix=FALSE,
                               real.bv.add=rbind(trait, trait2), name.cohort="Founders")

# Simulate mating. In each generation the top 10 cocks of the previous generation
# with the 10 hens of the foundering population
# Target: Increase share of white layer while preserving blue egg shell QTL


population <- breeding.diploid(population, breeding.size=c(100,100), selection.size=c(10,10),
                               best1.from.cohort="Founders_M", best2.from.cohort="Founders_W",
                               name.cohort="F1", selection.criteria.type = "bv")
population <- breeding.diploid(population, breeding.size=c(100,100), selection.size=c(10,10),
                               best1.from.cohort="F1_M", best2.from.cohort="Founders_W",
                               name.cohort="BC1", selection.m="function",
                               selection.criteria.type = "bv")
population <- breeding.diploid(population, breeding.size=c(100,100), selection.size=c(10,10),
                               best1.from.cohort="BC1_M", best2.from.cohort="Founders_W",
                               name.cohort="BC2", selection.m="function",
                               selection.criteria.type = "bv")
population <- breeding.diploid(population, breeding.size=c(100,100), selection.size=c(10,10),
                               best1.from.cohort="BC2_M", best2.from.cohort="Founders_W",
                               name.cohort="BC3", selection.m="function",
                               selection.criteria.type = "bv")
# Final generation: Use newly generated hens to get animals that are homozygous in blue egg shell QTL
```

```
population <- breeding.diploid(population, breeding.size=c(100,100), selection.size=c(10,10),
                               best1.from.cohort="BC3_M", best2.from.cohort="BC3_W",
                               name.cohort="IC", selection.m="function",
                               selection.criteria.type = "bv")

# Check genomic share of wild race in the final generation
genoIC <- get.geno(population, cohort="IC_M")
plot(rowSums(genoIC)/100)
abline(v=2000)
```



As expected, most regions of the genome are fixated for the white layer allele. Around the region of the blue-eggshell QTL, the share of the wild race is higher.

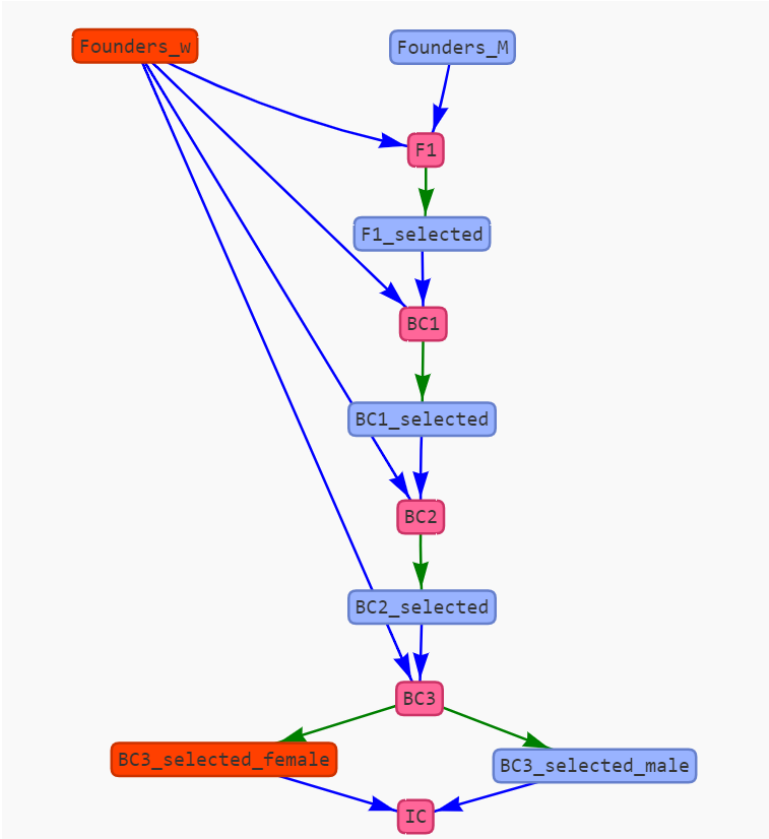
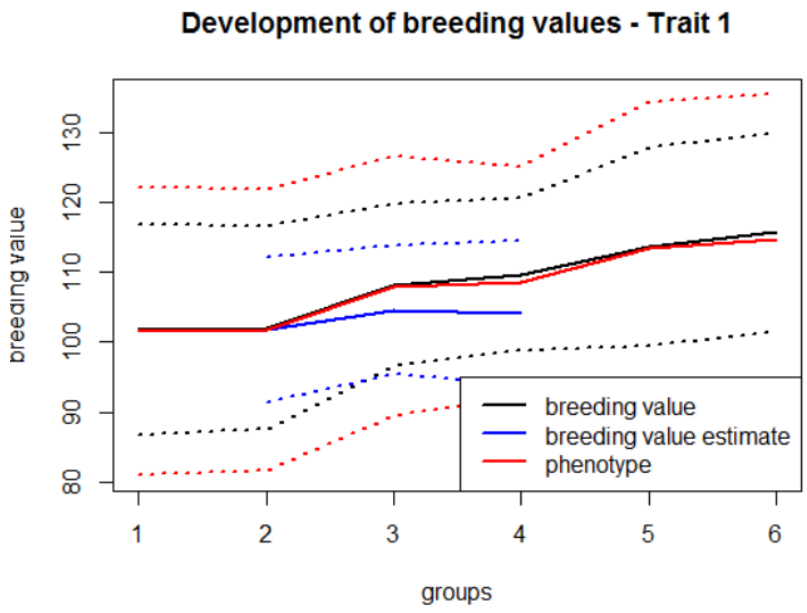


Figure 4: Mating Scheme for Introgression of the blue-egg-shell QTL. Graph is generated via user-interface in MoBPS

14.3 Simulation of gene editing in a cow breeding program

Here we show how to perform a simulation of a regular generation in a cattle breeding program, followed by a generation of using gene editing of the best sires 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 ~ 30 hours using 24 cores on the gwdg-hpc (Intel E5-2650 (2X12 core 2.2GHz)).

```
# Generation of a base population
population <- creating.diploid(dataset="random", nindi=1000, nsnp=5000,
                               n.additive =  100, name.cohort = "Founders")

# Generation of random mating
population <- breeding.diploid(population, breeding.size=c(100, 1000), selection.size = c(500,500),
                               best1.from.cohort = "Founders_M", best2.from.cohort = "Founders_F",
                               name.cohort = "Random")

# Generate bulls (and same number of cows) under high selection intensity.
# Heritability is set to 0.5, only phenotypes of cows are observed
population <- breeding.diploid(population, breeding.size=c(100, 100), selection.size=c(5, 200),
                               bve=TRUE, heritability = 0.5,
                               new.bv.observation = "non_obs", selection.m ="function",
                               new.bv.observation.sex=2, name.cohort = "Top",
                               best1.from.cohort = "Random_M", best2.from.cohort = "Random_F")
# Generate additional cows using all previous cows
population <- breeding.diploid(population, breeding.size=c(0, 900), selection.size=c(5, 500),
                               selection.m ="function", name.cohort = "Sec_F",
                              best1.from.cohort = "Random_M", best2.from.cohort = "Random_F",
                              use.last.sigma.e = TRUE, add.gen = 3)
# Additional perform gene editing on bulls using for mating
# 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, new.bv.observation = "non_obs", selection.m ="function",
                               new.bv.observation.sex=2, name.cohort = "Top_Edit",
                               best1.from.cohort = "Top_M", best2.from.cohort = c("Top_F", "Sec_F"),
                               nr.edits = 20, estimate.u = TRUE,
                               use.last.sigma.e = TRUE)

# Generate additional cows using all previous cows
population <- breeding.diploid(population, breeding.size=c(0, 900), selection.size=c(5, 500),
                              selection.m ="function", name.cohort = "Sec_Edit",
                              best1.from.cohort = "Top_M", best2.from.cohort = c("Top_F", "Sec_F"),
                              use.last.sigma.e = TRUE, add.gen=4)
# Just simulate some more phenotypes for the plot (alt. use ne)
population <- breeding.diploid(population, use.last.sigma.e = TRUE,
                              new.bv.observation = "non_obs")

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



Literaturverzeichnis

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.

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.

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.

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

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.

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.

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.

Zheng, Chaozhi; Boer, Martin P.; van Eeuwijk, Fred A. (2015): Reconstruction of genome ancestry blocks in multiparental populations. In: *Genetics*, genetics. 115.177873.

15 Acknowledgements

This package was developed in the context of the European Union's Horizon 2020 Research and Innovation Program under grant agreement n°677353 IMAGE

Additional thanks goes to the Research Training Group 1644 "Scaling Problems in Statistics" for financing travelling, BMBF project "MAZE – Accessing the genomic and functional diversity of maize to improve quantitative traits" (Grant ID 031B0195) and all members of the Animal Breeding and Genetics Group at the University of Goettingen for all the helpful advice to people with less genetic background and ideas of things to implement.