Using the epinetr package

Dion Detterer, Paul Kwan, Cedric Gondro

2021-11-09

Contents

Introduction **1**

Constructing the initial population **4**

Constructing a population using a genotype matrix 6

Modifying an existing population 7

Attaching effects to the population **8**

Attaching an epistatic network to the population 9

Supplying a user-defined network 15

Creating purely epistatic QTL 18

Calculating effects **21**

Inspecting additive, epistatic and environmental components 23

Deriving additive and epistatic components 25

Running the simulation **27**

Using the pedigree dropper 34

Conclusion **35**

References **35**

Introduction

epinetr is a forward-time genetic simulation package that includes the ability to construct epistatic networks of arbitrary complexity. Applications for *epinetr* include the testing of methods for the detection and selection of epistasis, as well as assessing the impact of epistasis on prediction and the extent to which epistasis is captured by additive models, all under varying degrees of epistatic complexity.

There are four broad steps in the workflow:

1. Construct the initial population with necessary parameters

2. Attach additive effects to the population
3. Attach an epistatic network to the population and visualise the network
4. Run a forward-time simulation of the population and plot the simulation run

In Section 1, we look at the various ways of constructing a population in *epinetr*. In Section 2, we briefly discuss how to attach additive effects to a population before diving into the options for constructing epistatic networks. Section 3 shows how *epinetr* calculates the components of each individual's phenotype, and how to recreate that calculation. Finally, in section 5, we demonstrate how to run the simulation.

As a preview, here's an example workflow:

```
library(epinetr)

# Build a population of size 1000, with 50 QTL, broad-sense heritability of 0.4,
# narrow-sense heritability of 0.2 and overall trait variance of 40.
pop <- Population(popSize = 1000, map = map100snp, alleleFrequencies = runif(100),
                  QTL = 50, broadH2 = 0.4, narrowh2 = 0.2, traitVar = 40)

# Attach additive effects
pop <- addEffects(pop)

# Attach an epistatic network
pop <- attachEpiNet(pop)

# Plot the network
plot(getEpiNet(pop))
```

```
# Inspect initial phenotypic components
head(getComponents(pop))
```

```
##   ID Sire Dam   Additive  Epistatic Environmental    Phenotype        EBV Sex
## 1  1    0   0 -2.0579466  3.9726690    -1.6529863   0.26173610 -11.573699   F
## 2  2    0   0 -0.5153622 -5.4447487     5.9432675  -0.01684344  -8.859594   F
## 3  3    0   0 -0.7533478 -0.1786125     0.7087766  -0.22318368 -11.132483   M
## 4  4    0   0  2.1115509  1.6284715     2.1953339   5.93535624  -4.400067   M
## 5  5    0   0 -1.2236780 -5.5874433    -5.8088298 -12.61995116  -6.889174   M
## 6  6    0   0  3.5130259  0.2177563     2.8232224   6.55400449  -1.010882   M
```

```
# Run a simulation across 250 generations
pop <- runSim(pop, generations = 250, truncSire = 0.1, truncDam = 0.5)

# Plot the simulation run
plot(pop)
```

```
# Get the allele frequencies
af <- getAlleleFreqRun(pop)

# Get the phased genotypes of the resulting population
geno <- getPhased(pop)

# Get a subset of the resulting population
ID <- getComponents(pop)$ID
```

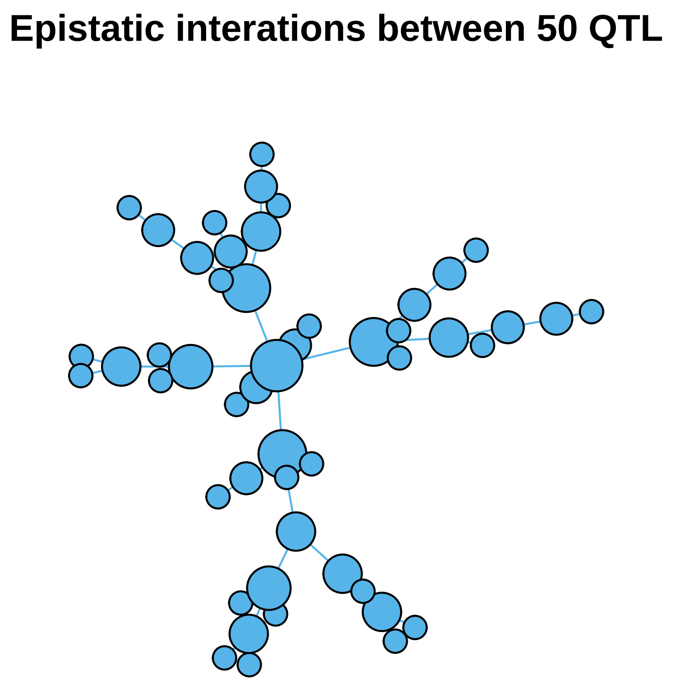

Figure 1: An epistatic network generated between 50 QTL.

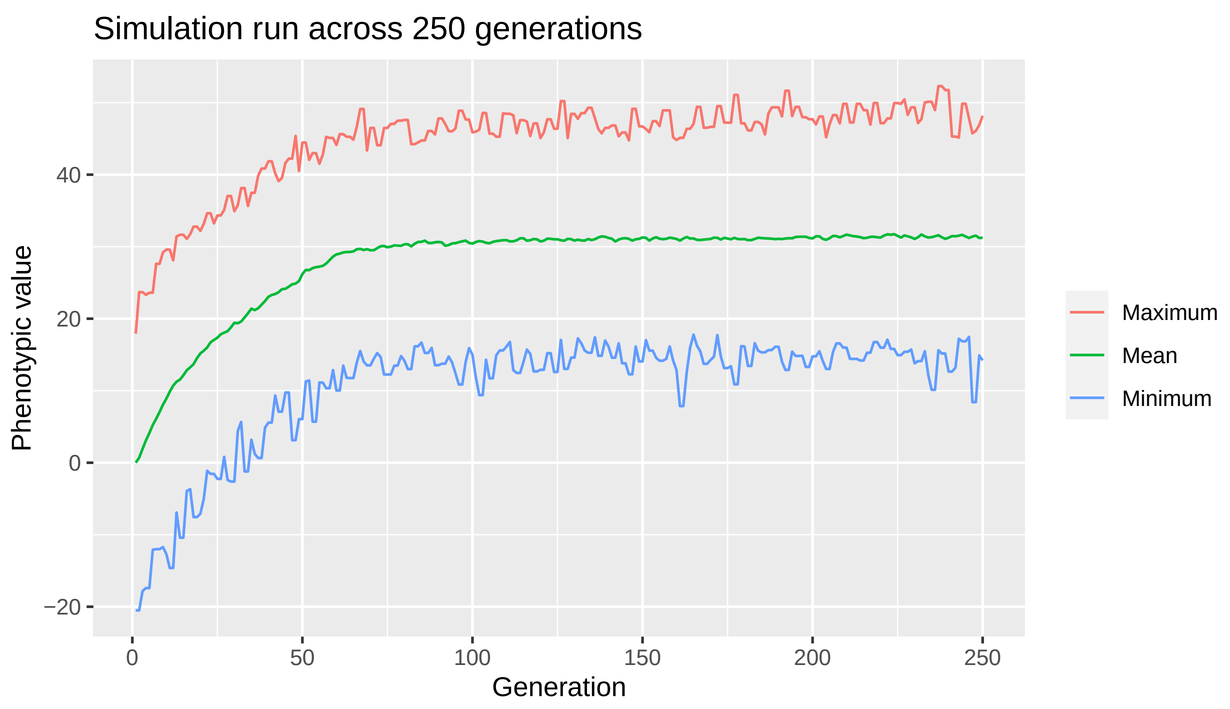

Figure 2: A graphical representation of a simulation run across 250 generations.

```r
ID <- sample(ID, 50)
pop2 <- getSubPop(pop, ID)
```

Constructing the initial population

Constructing the initial population is done via the *Population* function. The only data you will absolutely need is a *map*, either via a variant call format (VCF) file or directly via a data frame.

If you are directly supplying a map data frame, the first column should list the single nucleotide polymorphism (SNP) IDs, the second column should list the chromosome IDs for each SNP and the third column should list the position of each SNP on its chromosome in base pairs.

For example:

```r
head(map100snp)
```

```
##   V1 V2        V3
## 1  1  1  31433512
## 2  2  1  61930663
## 3  3  1  95513586
## 4  4  1 130263477
## 5  5  1 166021598
## 6  6  1 201083562
```

```
nrow(map100snp)
```

```
## [1] 100
```

```
length(unique(map100snp[, 2]))
```

```
## [1] 22
```

There are 100 SNPs across 22 chromosomes in this map data frame.

Given this map, we can now construct a population. We'll generate 200 individuals using allele frequencies selected from a uniform distribution, we'll select 20 QTL at random and we'll give the phenotypic trait under examination a variance of 40, a broad-sense heritability of 0.9 and a narrow-sense heritability of 0.6.

```
pop <- Population(popSize = 200, map = map100snp, QTL = 20,
                  alleleFrequencies = runif(100),
                  broadH2 = 0.9, narrowh2 = 0.6, traitVar = 40)
pop
```

```
## Population of size 200
## Specified initial trait variance: 40
## Initial broad-sense heritability: 0.9
## Initial narrow-sense heritability: 0.6
## Using 100 SNPs with 20 QTL:
## 8  23  24  25  28  29  38  40  44  49  54  58  61  64  69  70  73  80  82  95
```

We can fall back on the built-in defaults for *broadH2*, *narrowh2* and *traitVar* of 0.5, 0.3 and 1, respectively:

```
pop <- Population(popSize = 200, map = map100snp, QTL = 20,
                  alleleFrequencies = runif(100))
pop
```

```
## Population of size 200
## Specified initial trait variance: 1
## Initial broad-sense heritability: 0.5
## Initial narrow-sense heritability: 0.3
## Using 100 SNPs with 20 QTL:
## 6  9  17  18  20  27  30  33  39  41  57  62  63  64  85  87  92  93  94  95
```

epinetr estimates a breeding value for each individual in addition to supplying a true genetic value (TGV). The estimated breeding value (EBV) is calculated by first estimating the heritability using genomic relationship matrix (GRM)[1] restricted maximum likelihood (GREML),[2] then using the heritability estimate to in turn estimate additive SNP effects via gBLUP.[3,4] The EBVs are thus a window into how the model generated by *epinetr* appears under an assumption of additivity.

We can bypass GREML by supplying our own heritability estimate using *h2est*:

```
pop <- Population(popSize = 200, map = map100snp, QTL = 20,
                  alleleFrequencies = runif(100), h2est = 0.6)
```

We can also specify the QTL by listing their SNP IDs:

```
pop <- Population(popSize = 200, map = map100snp,
                  QTL = c(62, 55, 92, 74, 11, 38),
                  alleleFrequencies = runif(100),
                  broadH2 = 0.9, narrowh2 = 0.6, traitVar = 40)
pop
```

```
## Population of size 200
## Specified initial trait variance: 40
## Initial broad-sense heritability: 0.9
## Initial narrow-sense heritability: 0.6
## Using 100 SNPs with 6 QTL:
## 11  38  55  62  74  92
```

(Note that the QTL will be displayed only if there are no more than 100 QTL, so as not to flood the screen.)

A full list of QTL can also be displayed like so:

```
getQTL(pop)
```

```
##   ID Index
## 1 11    11
## 2 38    38
## 3 55    55
## 4 62    62
## 5 74    74
## 6 92    92
```

Note that any map supplied to the constructor will be sorted, first by chromosome and then by base pair position.

Constructing a population using a genotype matrix

For greater control, we can supply a matrix of phased biallelic genotypes to the constructor, either directly or via a VCF file using the *vcf* parameter. (Using a VCF file will also supply a map.)

If given directly, the matrix should be in individual-major format, with each allele coded with either a 0 or a 1 and no unknown values. For example, `geno100snp` is a genotype matrix of 100 SNPs across 500 individuals. It's already in the necessary format, which uses one individual per row and two columns per SNP.

```
dim(geno100snp)
```

```
## [1] 500 200
```

Examining the first 5 SNPs for the first individual we find the following:

```
geno100snp[1, 1:10]
```

```
##  V1  V2  V3  V4  V5  V6  V7  V8  V9 V10
##   1   0   1   1   1   1   1   1   0   0
```

That is, SNP 1 is heterozygous with genotype 1|0, SNPs 2-4 are homozygous with genotype 1|1 and SNP 5 is homozygous with genotype 0|0.

We can supply a phased genotype matrix to the constructor like so:

```
pop <- Population(popSize = nrow(geno100snp), map = map100snp, QTL = 20,
                  genotypes = geno100snp,
                  broadH2 = 0.9, narrowh2 = 0.6, traitVar = 40)
```

Supplying a phased genotype matrix allows us to directly specify the initial genotypes in the population. (The assumption is that the first allele for each SNP is inherited from the sire and the second allele for each SNP is inherited from the dam.) If we supply a population size to the constructor that does not match the number of rows in the genotype matrix, the genotypes will be used only to suggest allele frequencies for newly generated genotypes.

If we wish to use the genotypes to suggest allele frequencies while still maintaining the population size, we can set the *literal* flag to `FALSE`.

```
pop <- Population(popSize = nrow(geno100snp), map = map100snp, QTL = 20,
                  genotypes = geno100snp, literal = FALSE,
                  broadH2 = 0.9, narrowh2 = 0.6, traitVar = 40)
```

Modifying an existing population

The *Population* constructor can also be used to modify an existing population. We can, for example, adjust the heritability:

```
pop <- Population(pop, broadH2 = 0.7, traitVar = 30)
pop
```

```
## Population of size 500
## Specified initial trait variance: 30
## Initial broad-sense heritability: 0.7
## Initial narrow-sense heritability: 0.6
## Using 100 SNPs with 20 QTL:
## 22  23  25  26  27  28  32  33  37  44  46  53  54  58  60  83  84  89  90  96
```

We can also adjust the population size; this will necessarily generate a new set of genotypes based on the same allele frequencies:

```
pop <- Population(pop, popSize = 800)
pop
```

```
## Population of size 800
## Specified initial trait variance: 30
## Initial broad-sense heritability: 0.7
## Initial narrow-sense heritability: 0.6
## Using 100 SNPs with 20 QTL:
## 22  23  25  26  27  28  32  33  37  44  46  53  54  58  60  83  84  89  90  96
```

Similarly, we can adjust the allele frequencies, which will necessarily also generate a new set of genotypes:

```r
pop <- Population(pop, alleleFrequencies = runif(100))
pop
```

```
## Population of size 800
## Specified initial trait variance: 30
## Initial broad-sense heritability: 0.7
## Initial narrow-sense heritability: 0.6
## Using 100 SNPs with 20 QTL:
## 22  23  25  26  27  28  32  33  37  44  46  53  54  58  60  83  84  89  90  96
```

Where possible, the population features are preserved while adjusting only the parameters specified.

Attaching effects to the population

Because we have specified a non-zero narrow-sense heritability for our population, we now need to attach additive effects. This is done using the *addEffects* function.

```r
pop <- addEffects(pop)
pop
```

```
## Population of size 800
## Specified initial trait variance: 30
## Initial broad-sense heritability: 0.7
## Initial narrow-sense heritability: 0.6
## Additive variance in population: 18
## Using 100 SNPs with 20 QTL:
## 22  23  25  26  27  28  32  33  37  44  46  53  54  58  60  83  84  89  90  96
```

As expected, 60% of the phenotypic variance is attributable to additive effects.

By default, effects are selected from a normal distribution; we can, however, supply a different distribution function.

```r
pop <- addEffects(pop, distrib = runif)
```

Alternatively, we can supply our own additive effects for the QTL.

```r
effects <- c( 1.2,  1.5, -0.3, -1.4,  0.8,
              2.4,  0.2, -0.8, -0.4,  0.8,
             -0.2, -1.4,  1.4,  0.2, -0.9,
              0.4, -0.8,  0.0, -1.1, -1.3)
pop <- addEffects(pop, effects = effects)
getAddCoefs(pop)
```

```
##  [1]  1.8825760  2.3532200 -0.4706440 -2.1963387  1.2550507  3.7651520
##  [7]  0.3137627 -1.2550507 -0.6275253  1.2550507 -0.3137627 -2.1963387
## [13]  2.1963387  0.3137627 -1.4119320  0.6275253 -1.2550507  0.0000000
## [19] -1.7256947 -2.0394573
```

Note that the additive effects are scaled so as to guarantee the initial narrow-sense heritability. This is evident by adjusting the narrow-sense heritability within the population:

```
pop <- Population(pop, narrowh2 = 0.4)
getAddCoefs(pop)
```

```
##  [1]  1.5371169  1.9213961 -0.3842792 -1.7933030  1.0247446  3.0742337
##  [7]  0.2561861 -1.0247446 -0.5123723  1.0247446 -0.2561861 -1.7933030
## [13]  1.7933030  0.2561861 -1.1528377  0.5123723 -1.0247446  0.0000000
## [19] -1.4090238 -1.6652099
```

Attaching an epistatic network to the population

If broad-sense heritability is higher than narrow-sense heritability in the population, you will need to attach epistatic effects. The simplest way to do this is to use the *attachEpiNet* function with the default arguments, supplying only the population. This will generate a random epistatic network with the QTL as nodes.

```
pop <- attachEpiNet(pop)
pop
```

```
## Population of size 800
## Specified initial trait variance: 30
## Initial broad-sense heritability: 0.7
## Initial narrow-sense heritability: 0.4
## Additive variance in population: 12
## Epistatic variance in population: 9
## Using 100 SNPs with 20 QTL:
## 22  23  25  26  27  28  32  33  37  44  46  53  54  58  60  83  84  89  90  96
```

Note that the epistatic variance is as expected and the additive variance has been preserved.

We can visualise the network that was generated by using the *getEpiNet* function to retrieve the network before plotting it.

```
epinet <- getEpiNet(pop)
plot(epinet)
```

We can use the *scaleFree* flag to generate a network using the Barabasi-Albert model.[5]

```
pop <- attachEpiNet(pop, scaleFree = TRUE)
plot(getEpiNet(pop))
```

The Barabasi-Albert model for constructing networks assumes a connected graph that then adds a node at a time, with the probability that an existing node is connected to the new node given by $\frac{d_i}{\sum_j d_j}$, where d_i is the degree of node i. In *epinetr*, nodes are added in a random order (by shuffling the initial list) so as to ensure that no bias is introduced due to the initial ordering of nodes.

The "random" network model in *epinetr* acts as a control case: it uses the same algorithm as the Barabasi-Albert model but gives a uniform probability of connection to all existing nodes. This ensures that there are the same number of interactions in both cases.

If we want a number of QTL to only have additive effects applied, we can use the *additive* argument, giving the number of QTL not to be included in the epistatic network.

Epistatic interations between 20 QTL

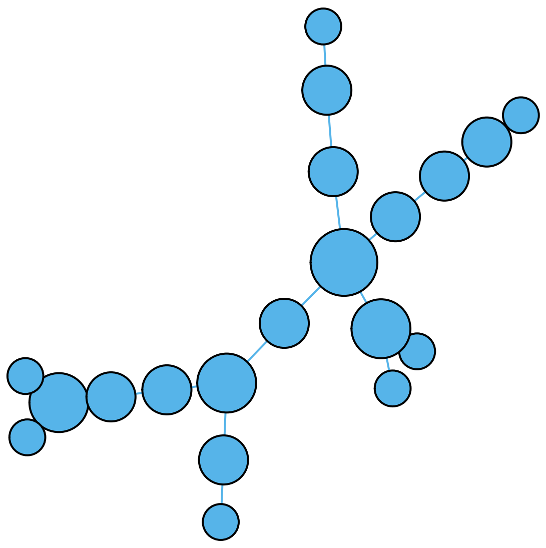

Figure 3: A random epistatic network generated between 20 QTL.

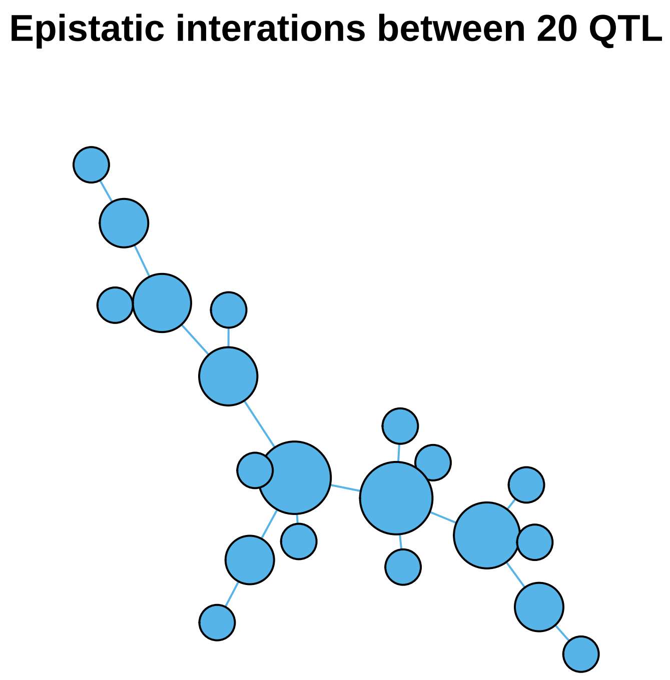

Figure 4: An epistatic network generated using the Barabasi-Albert model between 20 QTL.

```r
pop <- attachEpiNet(pop, scaleFree = TRUE, additive = 7)
plot(getEpiNet(pop))
```

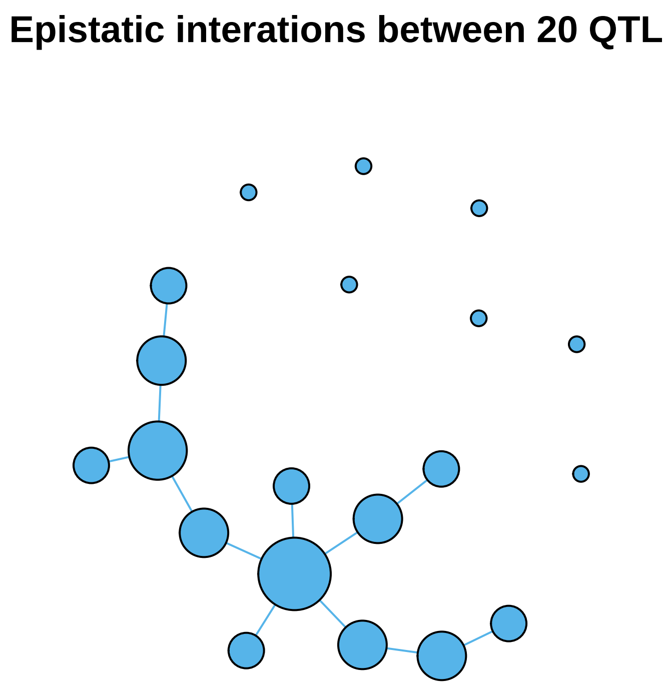

Figure 5: An epistatic network where 7 QTL have no epistatic effects.

The minimum number of interactions per QTL can be given with the m argument.

```r
pop <- attachEpiNet(pop, scaleFree = TRUE, additive = 7, m = 2)
plot(getEpiNet(pop))
```

There are three points to note with the m argument. The first point is that m specifically refers to the number of interactions and not the order of the interactions (which can be set with the k parameter; see below). The second point is that it has no impact on QTL designated as additive-only. The third point is that a minimal connected graph is initially constructed prior to m being strictly applied. (This means that you may still see some QTL with fewer than m interactions due to the value of k.)

We can also include higher-order interactions using the k argument, which accepts a vector specifying the orders of interaction to include:

```r
pop <- attachEpiNet(pop, scaleFree = TRUE, additive = 7, m = 2, k=2:7)
plot(getEpiNet(pop))
```

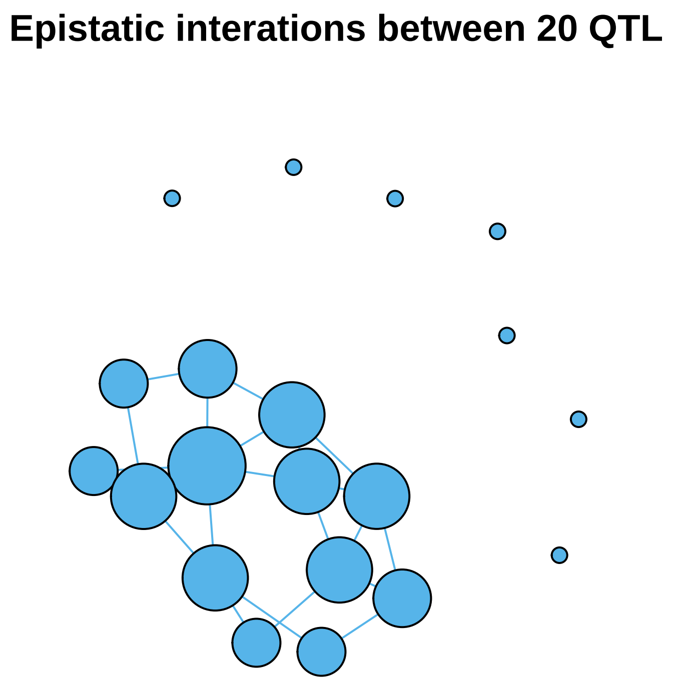

Figure 6: An epistatic network with a minimum of 2 interactions per epistatic QTL.

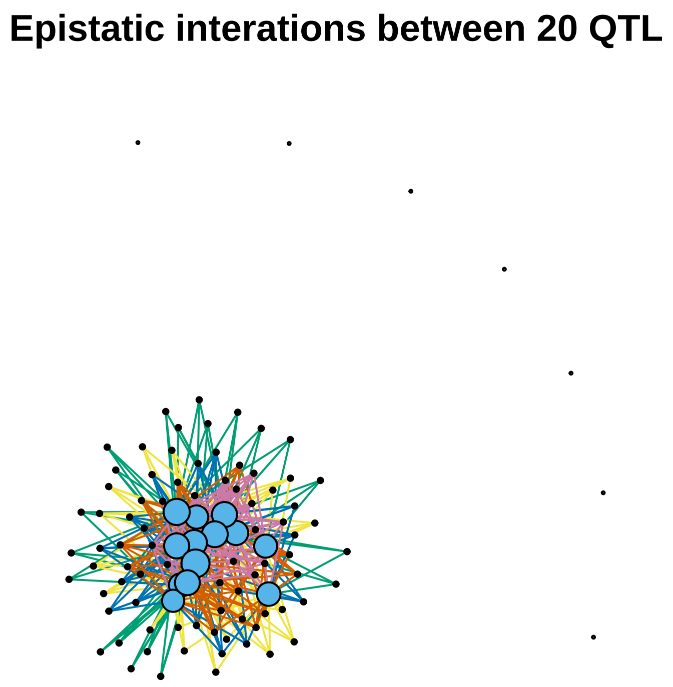

Figure 7: An epistatic network featuring 2-way to 7-way interactions

In cases where $m > 1$, any initial minimal connected graph is created by selecting n nodes such that n is the smallest integer satisfying the equation $\binom{n-1}{k-1} \geq m$. (In the default case of pairwise interactions, this reduces to $n \geq m + 1$.)

Where $k > 2$ we have extended the Barabasi-Albert model such that the *attachEpiNet* function adds interactions in order of increasing complexity: first, the function adds all 2-way interactions, then all 3-way interactions, *etcetera*. The important point to note is that, for networks generated using the Barabasi-Albert model, probabilities of connectedness are based on all previous interactions; for example, 3-way interactions are based on all previously established 2- and 3-way interactions. In this way, networks generated using the Barabasi-Albert model with multiple orders of interaction "layer up", such that the degrees from lower-order interactions contribute to the preferential attachment for higher-order interactions.

All auto-generated networks are highly connected. More diffuse networks can, however, be manually generated, as detailed in the next section.

Supplying a user-defined network

Internally, the network is stored as an incidence matrix, where the ith row corresponds to the ith QTL and the jth column corresponds to the jth interaction.

We can inspect the complete set of interactions using the *getIncMatrix* function.

```
inc <- getIncMatrix(pop)
dim(inc)
```

```
## [1]  20 102
```

There are currently 102 interactions in the population. Let's examine the first five.

```
inc[, 1:5]
```

```
##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    0    0    0    0    0
##  [2,]    0    0    0    0    0
##  [3,]    0    0    0    0    0
##  [4,]    0    0    0    0    0
##  [5,]    1    1    0    0    0
##  [6,]    0    0    0    0    0
##  [7,]    0    0    0    0    0
##  [8,]    0    0    0    0    0
##  [9,]    0    0    0    0    0
## [10,]    1    0    0    1    1
## [11,]    0    0    0    0    0
## [12,]    0    0    1    1    0
## [13,]    0    0    0    0    0
## [14,]    0    0    0    0    0
## [15,]    0    0    0    0    0
## [16,]    0    0    0    0    0
## [17,]    0    0    0    0    0
## [18,]    0    0    0    0    0
## [19,]    0    0    0    0    1
## [20,]    0    1    1    0    0
```

Here we can see that the first interaction is between QTL 5 and QTL 10, and that QTL 10 is included in 3 of the first 5 interactions.

We can define our own network in the same way. `rincmat100snp` is an example of a user-defined incidence matrix, giving 19 interactions across 20 QTL:

```
rincmat100snp
```

```
##       V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19
##  [1,]  0  1  0  0  0  0  0  0  0   0   0   0   1   0   0   0   0   0   0
##  [2,]  0  0  0  0  0  0  1  0  0   0   0   0   0   0   0   0   0   0   0
##  [3,]  0  0  0  0  0  0  0  0  0   0   0   0   0   0   0   0   0   0   0
##  [4,]  0  0  0  0  1  0  0  0  0   0   0   0   0   0   0   0   0   0   0
##  [5,]  0  0  0  0  0  0  0  0  0   0   1   0   0   0   0   0   1   0   0
##  [6,]  0  0  0  0  0  0  0  0  1   0   0   1   0   0   1   0   1   1   0
##  [7,]  0  0  0  0  0  0  0  0  0   0   0   0   0   0   0   0   0   0   0
##  [8,]  0  0  0  0  0  0  0  0  0   0   1   0   0   0   0   0   0   0   0
##  [9,]  1  0  0  0  0  0  0  0  0   0   0   0   0   0   0   0   0   0   0
## [10,]  0  0  0  0  1  0  0  0  0   0   0   0   1   1   0   1   0   0   1
## [11,]  0  0  0  1  0  0  0  0  0   1   0   0   0   0   0   0   0   0   0
## [12,]  0  0  0  0  0  0  0  0  0   0   0   0   0   0   0   0   0   0   0
## [13,]  0  0  0  0  0  0  0  0  0   1   0   0   0   0   1   0   0   0   1
## [14,]  0  1  0  0  0  0  0  0  0   0   0   0   0   0   0   0   0   0   0
## [15,]  1  0  0  0  0  0  0  0  0   0   0   0   0   0   0   0   0   0   0
## [16,]  0  0  0  0  0  1  1  1  0   0   0   1   0   0   0   0   0   0   0
## [17,]  0  0  0  0  0  0  0  0  1   0   0   0   0   0   0   1   0   0   0
## [18,]  0  0  1  1  0  1  0  0  0   0   0   0   0   1   0   0   0   1   0
## [19,]  0  0  1  0  0  0  0  1  0   0   0   0   0   0   0   0   0   0   0
## [20,]  0  0  0  0  0  0  0  0  0   0   0   0   0   0   0   0   0   0   0
```

We can attach an epistatic network based on this incidence matrix to our population using the *incmat* argument:

```
pop <- attachEpiNet(pop, incmat = rincmat100snp)
```

Let's visualise the subsequent network:

```
plot(getEpiNet(pop))
```

As per the interaction matrix, 4 of the 20 QTL are not part of any interactions. (Rows 3, 7, 12 and 20 only contain 0s.)

We can create a 3-way interaction by modifying the matrix such that QTL 20 is included in the first interaction:

```
# Include the 20th QTL in the first interaction
mm <- rincmat100snp
mm[20, 1] <- 1
pop <- attachEpiNet(pop, incmat = mm)
```

Again, let's visualise the subsequent network.

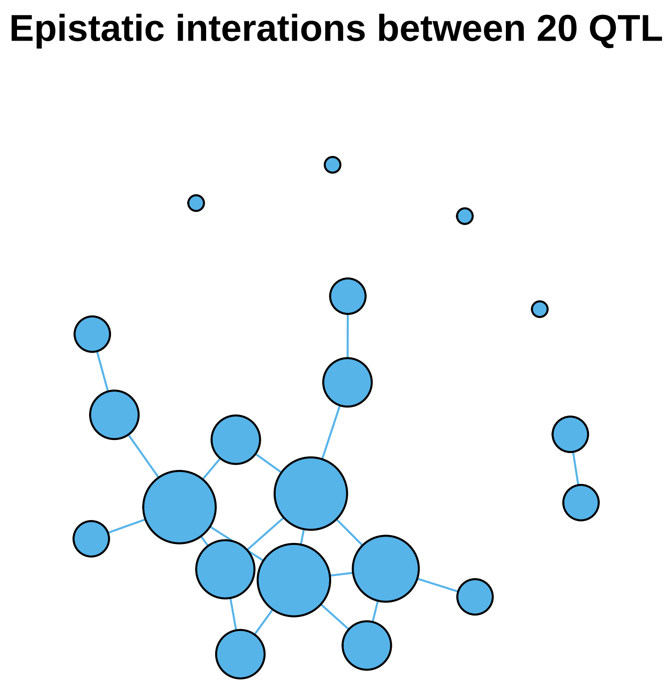

Figure 8: An epistatic network derived from a user-defined incidence matrix.

```r
plot(getEpiNet(pop))
```

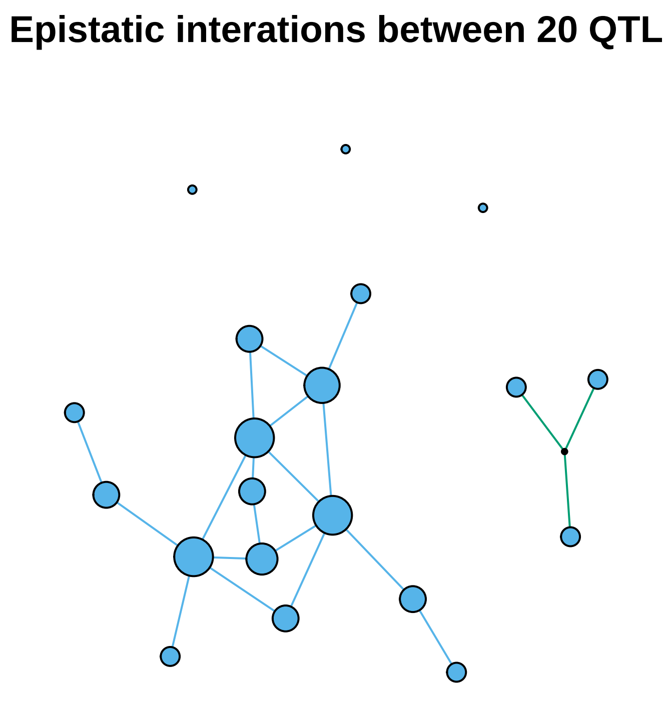

Figure 9: A user-defined epistatic network featuring a single 3-way interaction.

The network now includes a single 3-way interaction.

Creating purely epistatic QTL

As already shown, it is possible to create purely additive QTL; similarly, it is also possible to create purely epistatic QTL. This involves specifying the additive coefficients explicitly, where at least one coefficient is 0; for example:

```r
# 20 coefficients, 3 of which are 0
coefs <- sample(c(rep(0, 3), rnorm(17)), 20)
pop <- addEffects(pop, effects = coefs)
getAddCoefs(pop)
```

```
##  [1]  0.00000000  0.00000000  0.00000000  0.40190355 -0.58757429  4.28253863
##  [7] -0.32596999 -0.89358098  0.20130553  0.41005458 -3.12425064 -0.05213444
## [13]  0.73179985  0.54476970  1.60360722  1.22896418  1.11410139  0.25023685
## [19]  1.34597914  1.62324526
```

We can see here that QTL 1, 2 and 3 all have additive coefficients of 0, making them purely epistatic (since the population includes epistatic interactions).

If we wish to have both purely additive and purely epistatic QTL in our model, we can explicitly give the SNP IDs of the QTL we want to be purely additive to *attachEpiNet*.

Suppose we have 15 QTL overall. For the sake of simplicity, we'll make the 15 QTL the first 15 SNPs in the map:

```
pop <- Population(pop, QTL = 1:15)
```

Let's make the first 5 QTL additive-only by giving their SNP IDs to the additive parameter of *attachEpiNet*:

```
pop <- attachEpiNet(pop, additive = 1:5)
```

As we can see in the incidence matrix, the first 5 of the 15 QTL have no epistatic effects:

```
getIncMatrix(pop)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##  [1,]    0    0    0    0    0    0    0    0    0
##  [2,]    0    0    0    0    0    0    0    0    0
##  [3,]    0    0    0    0    0    0    0    0    0
##  [4,]    0    0    0    0    0    0    0    0    0
##  [5,]    0    0    0    0    0    0    0    0    0
##  [6,]    0    0    0    0    0    0    0    0    1
##  [7,]    0    0    1    0    1    0    1    0    0
##  [8,]    1    0    0    0    0    1    0    0    0
##  [9,]    0    0    0    1    0    0    0    0    0
## [10,]    0    0    0    0    1    0    0    0    1
## [11,]    0    0    0    0    0    0    0    1    0
## [12,]    0    0    0    0    0    0    1    0    0
## [13,]    0    0    0    0    0    1    0    0    0
## [14,]    1    1    0    0    0    0    0    0    0
## [15,]    0    1    1    1    0    0    0    1    0
```

Now, we can plot the network in order to visualise the incidence matrix:

```
plot(getEpiNet(pop))
```

Let's make QTL 6-10 epistatic-only by explicitly giving their coefficients as 0 to *addEffects*:

```
coefs <- rnorm(15)
coefs[6:10] <- 0
pop <- addEffects(pop, effects = coefs)
getAddCoefs(pop)
```

```
##  [1]  1.8163999 -2.3241301  2.0367146 -1.7171268 -1.2660893  0.0000000
##  [7]  0.0000000  0.0000000  0.0000000  0.0000000 -1.1025921 -0.7227369
## [13] -3.2974631 -2.8813620  1.6818694
```

We now have a population where 5 of the QTL are purely additive, 5 are purely epistatic and 5 are both additive and epistatic.

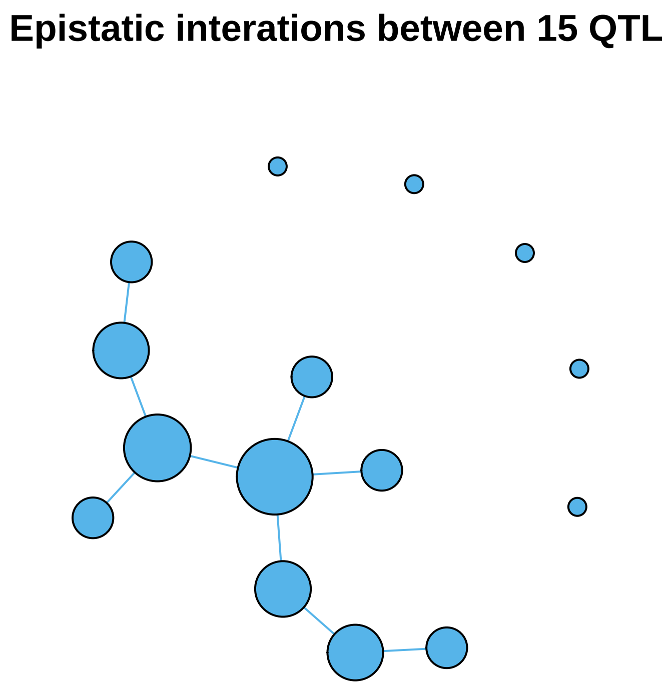

Figure 10: The epistatic network generated with the first 5 of the 15 QTL being purely additive.

Calculating effects

Now that we've seen how *epinetr* builds epistatic networks, it's time to turn our attention to the effects generated by these networks.

Each QTL can, of course, only have one of three genotypes per individual: the homozygous genotype coded 0|0, the heterozygous genotype and the homozygous genotype coded 1|1. For an interaction consisting of k QTL, this corresponds to 3^k possible genotypes within the interaction overall, and for this reason, *epinetr* assigns a set of 3^k potential epistatic effects drawn from a normal distribution to each interaction.

Let's create a new population with 20 QTL, additive effects and an epistatic network consisting solely of 3-way interactions:

```
pop <- Population(popSize = 200, map = map100snp, QTL = 20,
                  alleleFrequencies = runif(100),
                  broadH2 = 0.9, narrowh2 = 0.6, traitVar = 40)
pop <- addEffects(pop)
pop <- attachEpiNet(pop, k = 3)
```

We can plot the network in order to visualise these 3-way interactions:

```
plot(getEpiNet(pop))
```

Let's determine which QTL are part of the first interaction:

```
qtls <- which(getIncMatrix(pop)[, 1] > 0)
qtls
```

```
## [1]  2 16 19
```

The possible effects for this interaction can be found using *getInteraction*:

```
interaction1 <- getInteraction(pop, 1) # Return first interaction array
interaction1
```

```
## , , 1
##
##             [,1]      [,2]       [,3]
## [1,] -1.53362233 0.6315356  0.5237621
## [2,]  0.02921504 1.4146000  0.2436621
## [3,] -0.32396525 0.4293025 -1.3600177
##
## , , 2
##
##            [,1]       [,2]       [,3]
## [1,]  0.1410683 -0.7269974  0.8801060
## [2,]  3.5425986 -2.0599813  0.1500994
## [3,] -0.9598682 -0.7111304 -1.2629551
##
## , , 3
##
##            [,1]       [,2]       [,3]
## [1,] -0.7513985  0.0308959 -0.1408719
## [2,] -0.2865082  0.9165172  0.6292963
## [3,] -0.0315287 -1.4217879  1.8564355
```

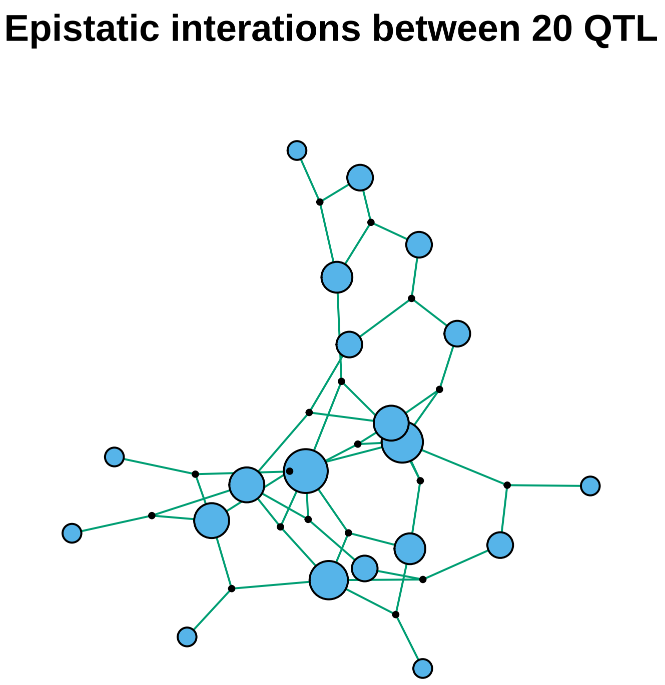

Figure 11: An epistatic network consisting of 3-way interactions.

As expected, this is a 3^3 array of possible effects: the first dimension maps to QTL 2, the second dimension maps to QTL 16 and the third dimension maps to QTL 19. Along each dimension, the first index maps to the homozygous genotype coded 0|0, the second index maps to the heterozygous genotype and the third index maps to the homozygous genotype coded 1|1.

Suppose for a particular individual QTL 2 has the heterozygous genotype. The possible effects for the interaction are thus given by the following:

```
interaction1[2, , ]
```

```
##            [,1]       [,2]       [,3]
## [1,] 0.02921504  3.5425986 -0.2865082
## [2,] 1.41459995 -2.0599813  0.9165172
## [3,] 0.24366206  0.1500994  0.6292963
```

Furthermore, suppose that QTL 16 has the homozygous genotype coded 0|0. The possible effects for the interaction are now further constrained to the following:

```
interaction1[2, 1, ]
```

```
## [1]  0.02921504  3.54259864 -0.28650820
```

Finally, suppose that QTL 19 has the homozygous genotype coded 1|1. We now have all we need in order to know the effect of this interaction on the phenotype:

```
interaction1[2, 1, 3]
```

```
## [1] -0.2865082
```

Thus the contribution of this particular interaction to this individual's overall phenotype is -0.2865082. (An offset, however, has yet to be applied: see the next section for details.)

Inspecting additive, epistatic and environmental components

Once any necessary additive and epistatic effects are attached to our population, we can inspect the phenotypic components of each individual, using the *getComponents* function:

```
components <- getComponents(pop)
head(components)
```

```
##   ID Sire Dam    Additive Epistatic Environmental   Phenotype       EBV Sex
## 1  1    0   0  -4.8150112  2.788248     0.4872345  -1.5395284 14.269578   F
## 2  2    0   0  -0.9732234  3.057018    -1.9050966   0.1786983  8.802091   M
## 3  3    0   0 -13.0667338 -5.727073    -2.2481768 -21.0419834  1.767167   F
## 4  4    0   0   4.9591609  6.722858    -2.6414225   9.0405961 29.529996   F
## 5  5    0   0  -3.1251000 -2.939172     0.8029818  -5.2612903  6.914923   F
## 6  6    0   0  -5.5489497  4.414932    -1.3918394  -2.5258567  7.016178   F
```

As we can see, GBLUP-based EBVs have also been calculated for each individual.

Inspecting the additive component, we can see its mean and variance are as expected:

```
mean(components$Additive)
```

```
## [1] -3.590471e-17
```

```
var(components$Additive)
```

```
## [1] 24
```

Similarly for the epistatic and environmental components:

```
mean(components$Epistatic)
```

```
## [1] -3.229621e-17
```

```
var(components$Epistatic)
```

```
## [1] 12
```

```
mean(components$Environmental)
```

```
## [1] -1.07729e-17
```

```
var(components$Environmental)
```

```
## [1] 4
```

The means are (effectively) 0 because the components are zero-centred using fixed offsets applied to the additive and epistatic components that are preserved across generations. (Only the initial generation's environmental component is zero-centred.) We can retrieve these offsets with the *getAddOffset* and *getEpiOffset* functions, respectively:

```
getAddOffset(pop)
```

```
## [1] -18.38241
```

```
getEpiOffset(pop)
```

```
## [1] -3.425918
```

For the overall phenotypic value, we find the following:

```
mean(components$Phenotype)
```

```
## [1] -9.714451e-17
```

```
var(components$Phenotype)
```

```
## [1] 40.00098
```

This approximation of the specified variance is due to a small amount of co-variance between components:

```
cov(components$Additive, components$Epistatic)
```

```
## [1] -0.004116437
```

```
cov(components$Additive, components$Environmental)
```

```
## [1] 0.4931992
```

```
cov(components$Environmental, components$Epistatic)
```

```
## [1] -0.4885939
```

However:

```
cor(components$Additive, components$Epistatic)
```

```
## [1] -0.0002425634
```

epinetr attempts to minimise these co-variances by selecting from within the random distributions for the environmental and epistatic components such that co-variances are minimal, given computational constraints. In particular, the epistatic component is optimised using a genetic algorithm.

Deriving additive and epistatic components

We are now in a position to derive the additive component for the population. First, we'll retrieve the population's unphased genotypes using the *getGeno* function:

```
geno <- getGeno(pop)
```

Alternatively, we could use the *getHaplo* function, which returns a list of the two haplotype matrices within the population; we would then need to sum the two haplotypes together.

Next, we'll select only the QTL within the genotypes:

```
geno <- geno[, getQTL(pop)$Index]
```

Finally, we'll multiply the genotypes by the additive coefficients and apply the offset:

```
additive <- geno %*% getAddCoefs(pop) + getAddOffset(pop)
additive[1:5]
```

```
## [1]  -4.8150112  -0.9732234 -13.0667338   4.9591609  -3.1251000
```

Compare with the additive component for the first five individuals given by *getComponents*:

```
getComponents(pop)$Additive[1:5]
```

```
## [1]  -4.8150112  -0.9732234 -13.0667338   4.9591609  -3.1251000
```

The function *getEpistasis* returns a matrix:

```
head(getEpistasis(pop))
```

```
##            [,1]       [,2]      [,3]       [,4]       [,5]      [,6]       [,7]
## [1,] -0.1408719 -0.6363654 0.7329887  0.8684964  0.9954562  1.240152  0.7020566
## [2,]  0.6315356 -0.8451467 1.2845097  0.9954399 -1.0961423 -1.143580  2.1206373
## [3,]  0.5237621 -1.3491067 0.9292732 -0.5678992 -0.9042016  1.151988  0.2988748
## [4,]  0.8801060 -0.6363654 1.1690991  0.9954399  0.9954562 -1.143580  0.6925515
## [5,] -0.7269974 -1.1899959 0.5974305 -0.3707536  0.4858424 -1.580997 -0.5594304
## [6,]  0.5237621 -0.6363654 1.2845097  0.9954399  0.9954562 -1.143580  2.1206373
##            [,8]        [,9]      [,10]     [,11]      [,12]       [,13]
## [1,] -0.7438637  1.71370634 -0.1146389 0.8219348  0.6179330 -1.89633988
## [2,]  1.2990934  0.05420309  0.1945798 0.1557243  1.0539369 -0.03025653
## [3,] -0.4592987  0.99195104  1.2205574 0.9570476 -0.7100282  0.39565566
## [4,]  1.2990934 -0.07460584  0.7935462 1.4871272  1.0539369  1.02886123
## [5,]  0.3120463 -0.20160117 -0.5202665 0.9570476 -1.5442226  0.50324412
## [6,] -0.7003319 -0.07460584  0.7935462 0.5921773  1.0539369  1.02886123
##           [,14]      [,15]      [,16]       [,17]      [,18]
## [1,]  0.1824583  1.2282944  0.3954778  0.62131212 -0.3740203
## [2,] -0.8208557  1.5165094 -0.1250205 -0.02523949  1.2630087
## [3,] -1.1172612 -1.1908538 -0.6963776 -0.71702965 -1.0582076
## [4,]  0.2372694  1.4574113  0.1456302 -0.71702965  0.4848287
## [5,]  1.9153018 -0.3186575  1.6226142  0.62131212  0.4848287
## [6,] -0.8208557 -0.3186575 -0.1250205  1.92143614  0.3505049
```

The rows in this matrix are the individuals; the columns are the contributions of each interaction to the overall epistatic component. By summing the rows and applying the epistatic offset to each value, we can derive the contribution of epistasis to each individual's phenotype:

```
epistatic <- rowSums(getEpistasis(pop)) + getEpiOffset(pop)
epistatic[1:5]
```

```
## [1]  2.788248  3.057018 -5.727073  6.722858 -2.939172
```

We can similarly compare this result with the epistatic component for the first five individuals given by *getComponents*:

```
getComponents(pop)$Epistatic[1:5]
```

```
## [1]  2.788248  3.057018 -5.727073  6.722858 -2.939172
```

We can thus easily derive both the additive and epistatic components for each individual. This can be further replicated for individuals not in the population.

Suppose you have a matrix of genotypes (`geno2`) for five individuals not in the population:

```
geno2 <- geno2[, getQTL(pop)$Index]
geno2
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
## [1,]    1    1    0    2    1    0    1    1    0     1     0     1     1     1
## [2,]    1    1    2    2    0    1    2    0    0     1     2     1     1     2
## [3,]    1    2    2    2    0    0    1    0    1     2     0     1     1     2
## [4,]    2    2    0    2    1    2    1    1    2     2     1     1     0     0
## [5,]    1    0    2    1    0    1    1    2    1     1     1     0     2     1
##      [,15] [,16] [,17] [,18] [,19] [,20]
## [1,]     0     1     1     0     0     2
## [2,]     1     2     2     1     2     2
## [3,]     1     2     0     1     2     1
## [4,]     0     2     0     1     2     1
## [5,]     2     2     0     1     0     0
```

We can find their additive components using the following code:

```
additive2 <- geno2 %*% getAddCoefs(pop) + getAddOffset(pop)
additive2[1:5]
```

```
## [1]  -7.414510   2.752778  -7.998020   7.617690 -17.842772
```

Similarly, we can find their epistatic components using the following code:

```
epistatic2 <- rowSums(getEpistasis(pop, geno = geno2)) + getEpiOffset(pop)
epistatic2
```

```
## [1] -7.469675  5.097355 -1.081159 -3.502617 -8.270756
```

Note that this is achieved via the optional *geno* argument in the *getEpistasis* function.

Running the simulation

In order to run the simulation, we need to use the *runSim* function.

```
popRun <- runSim(pop, generations = 150)
```

The above command will iterate through 150 generations, with generation 1 being the initial generation supplied.

There are several optional arguments that can be supplied to the simulator to alter selection, recombination and mutation across generations:

- *selection* determines whether random selection (the default) is employed or linear ranking selection[6] is used (by supplying the string "ranking");
- *fitness* determines whether selection occurs based on phenotypic value (the default), true genetic value (by supplying the string "TGV") or estimated breeding value (by supplying the string "EBV");
- *truncSire* and *truncDam* give the proportion of sires and dams, respectively, to include in selection, when sorted by descending phenotype (defaulting to 1);

- *burnIn* determines how many initial generations will use random selection with no truncation (defaulting to 0);
- *roundsSire* and *roundsDam* give the maximum number of generations for sires and dams, respectively, to survive within the population, assuming there are enough offspring generated to fill the population (defaulting to 1);
- *litterDist* is a vector of probabilities for the size of each litter, starting with a litter size of 0 (defaulting to `c(0, 0, 1)`, i.e. each litter will always contain two offspring);
- *breedSire* is the maximum number of times a sire can breed within a single generation (defaulting to 10);
- *mutation* is the rate of mutation for each SNP;
- *recombination* is a vector of probabilities specifying the rate of recombination between consecutive SNPs in the *map* (obviously excepting consecutive SNPs on different chromosomes);
- *allGenoFileName* is a string giving the file name to optionally output the genotypes from all generations.

Each mating pair produces a number of full-sibling offspring by sampling once from the litter-size probability mass function given by *litterDist*. The vector can be of arbitrary length, such that in order to specify the possibility of up to n full-sibling offspring per mating pair, a vector of length $n + 1$ must be supplied.

Linear ranking selection is a form of weighted stochastic selection: if the individuals in a population of size n are each given a rank r based on descending order of fitness (i.e. the individual with the highest fitness is given the rank $r_1 = 1$ while the individual with the lowest fitness is given the rank $r_n = n$), the probability of an individual i being selected for mating is given by:

$$P(i \text{ is selected}) = \frac{2(n - r_i + 1)}{n(n+1)}$$

For a population of 10, we have the following probabilities of selection for the highest-to-lowest ranked individuals:

```r
n <- 10
pmf <- 2 * (n - 1:n + 1) / (n * (n + 1))
pmf
```

```
##  [1] 0.18181818 0.16363636 0.14545455 0.12727273 0.10909091 0.09090909
##  [7] 0.07272727 0.05454545 0.03636364 0.01818182
```

Fitness itself can be based on the phenotypic value, the true genetic value or the estimated breeding value, with the true genetic value simply being the sum of any additive and epistatic components of the phenotypic value; as stated previously, the estimated breeding value is calculated using gBLUP.

epinetr performs selection by first splitting the population into male and female sub-populations. Next, if the round is outside any initial burn-in period, each sub-population is truncated to a proportion of its original size per the values of *truncSire* and *truncDam*, respectively.

When linear ranking selection is used, females are then exhaustively sampled, without replacement, for each mating pair using their linear ranking probabilities, as given above; males are sampled for each mating pair using their linear ranking probabilities but with replacement, where they are each only replaced a maximum number of times as specified by *breedSire*. Random selection occurs in the same manner, but all probabilities are uniform. During any initial burn-in period, random selection is enforced.

As stated above, selection occurs based on fitness: this can be based on the phenotypic value, the true genetic value or the estimated breeding value

Finally, each mating pair produces a number of full-sibling offspring by sampling once from the litter-size probability mass function given by *litterDist* (with the default guaranteeing two full-sibling offspring per

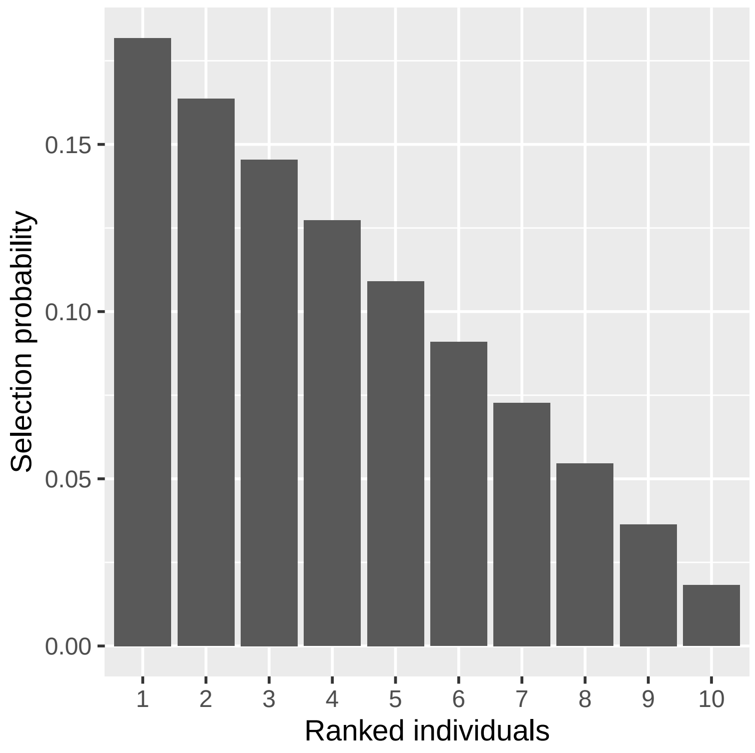

Figure 12: Probability mass function for linear ranking selection on a population of 10 individuals.

mating pair). Half-siblings occur when sires can mate more than once per round (as given by *breedSire*) or sires or dams survive beyond one round (as given by *roundsSire* and *roundsDam*, respectively). Note that in order to maintain the population size, the youngest sires and dams earmarked as reaching the end of their breeding lifespan may be kept in the population to the next round if there is a shortfall of offspring: this can also result in the appearance of half-siblings.

We can perform different simulation runs on the same population as follows:

```
popRunRank <- runSim(pop, generations = 150, selection = "ranking")
popRunBurnIn <- runSim(pop, generations = 150, burnIn = 50,
                       truncSire = 0.1, truncDam = 0.5,
                       roundsSire = 5, roundsDam = 5,
                       litterDist = c(0.1, 0.3, 0.4, 0.2),
                       breedSire = 7)
popRunTGV <- runSim(pop, generations = 150,
                    truncSire = 0.1, truncDam = 0.5,
                    fitness = "TGV")
popRunEBV <- runSim(pop, generations = 150,
                    truncSire = 0.1, truncDam = 0.5,
                    fitness = "EBV")
```

To visually compare these runs, we can plot them:

```
plot(popRun)
```

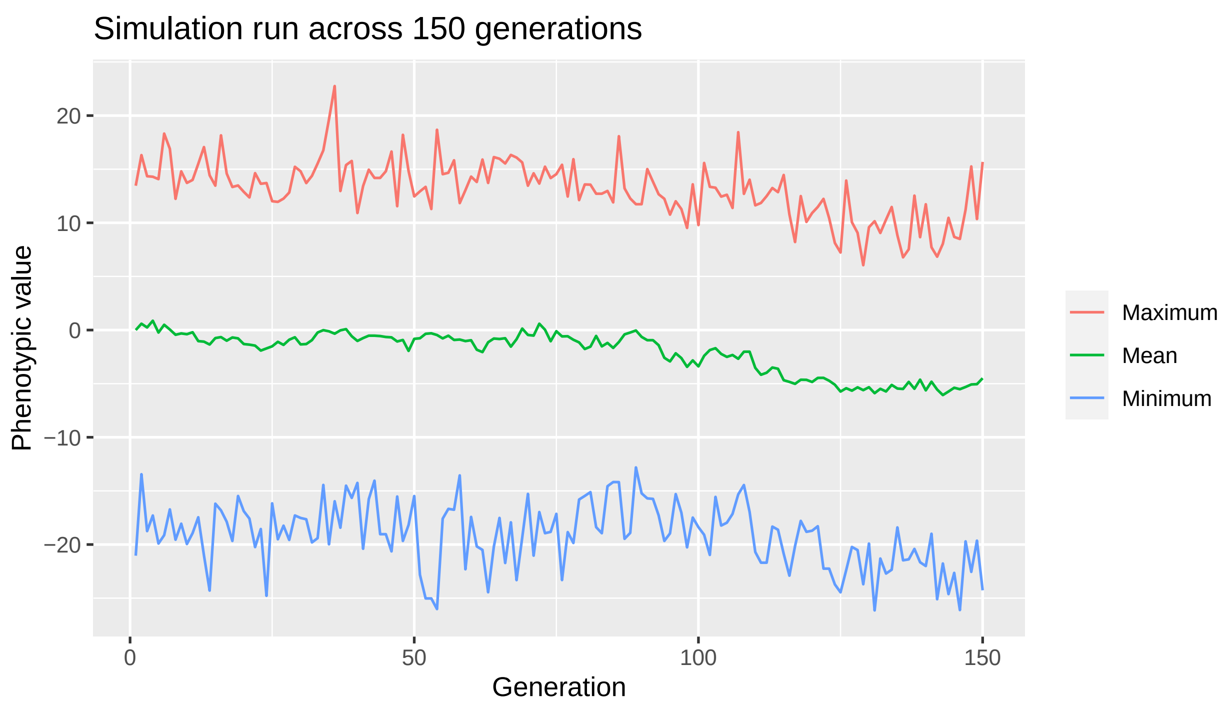

Figure 13: A graphical representation of a simulation run using the default parameters.

```
plot(popRunRank)
```

```
plot(popRunBurnIn)
```

```
plot(popRunTGV)
```

```
plot(popRunEBV)
```

Using the *allGenoFileName* argument in *runSim* allows the simulator to write a serialised file containing all the genotypes generated during the run. To retrieve such a file, we use the *loadGeno* function:

```
popRun <- runSim(pop, generations = 150, allGenoFileName = "geno.epi")
geno <- loadGeno("geno.epi")
```

This object is a matrix in the same phased format as the genotype matrices supplied to the constructor.

```
geno[1:5, 1:8]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    0    0    0    1    0    0    0    0
## [2,]    0    1    0    0    0    0    0    0
## [3,]    0    0    0    1    0    0    0    0
## [4,]    0    0    0    0    0    0    0    0
## [5,]    0    0    0    1    0    0    0    0
```

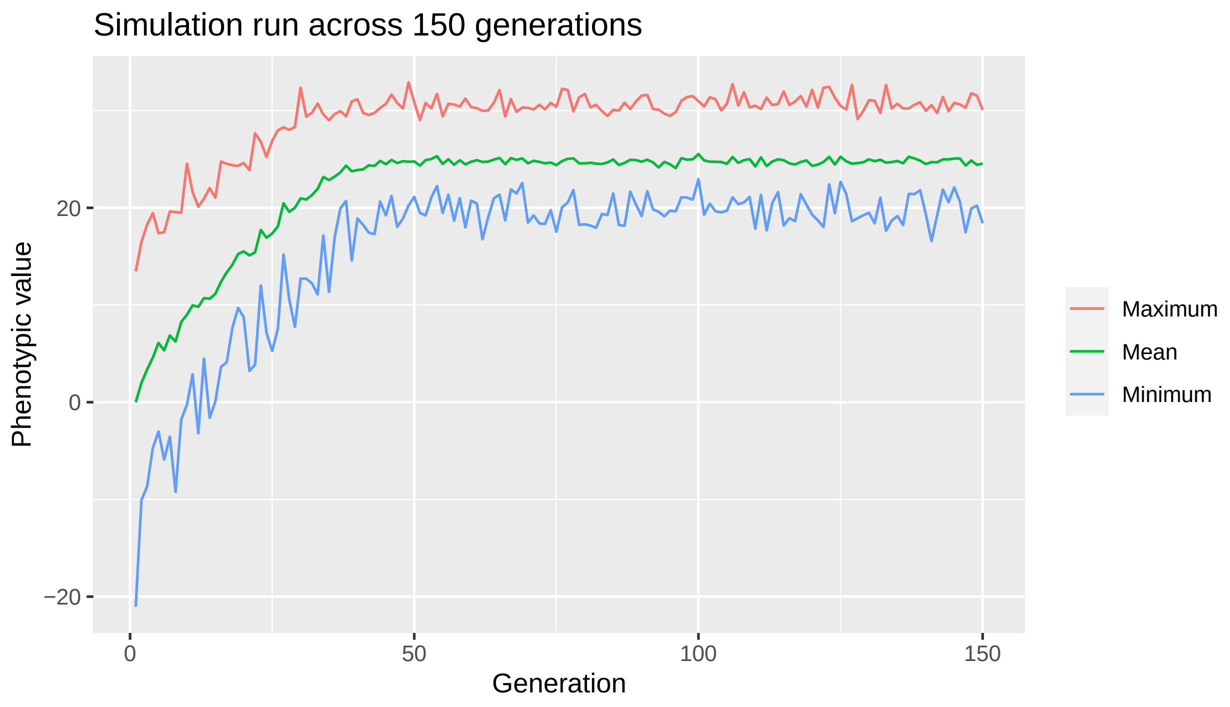

Figure 14: A graphical representation of a simulation run using linear ranking selection.

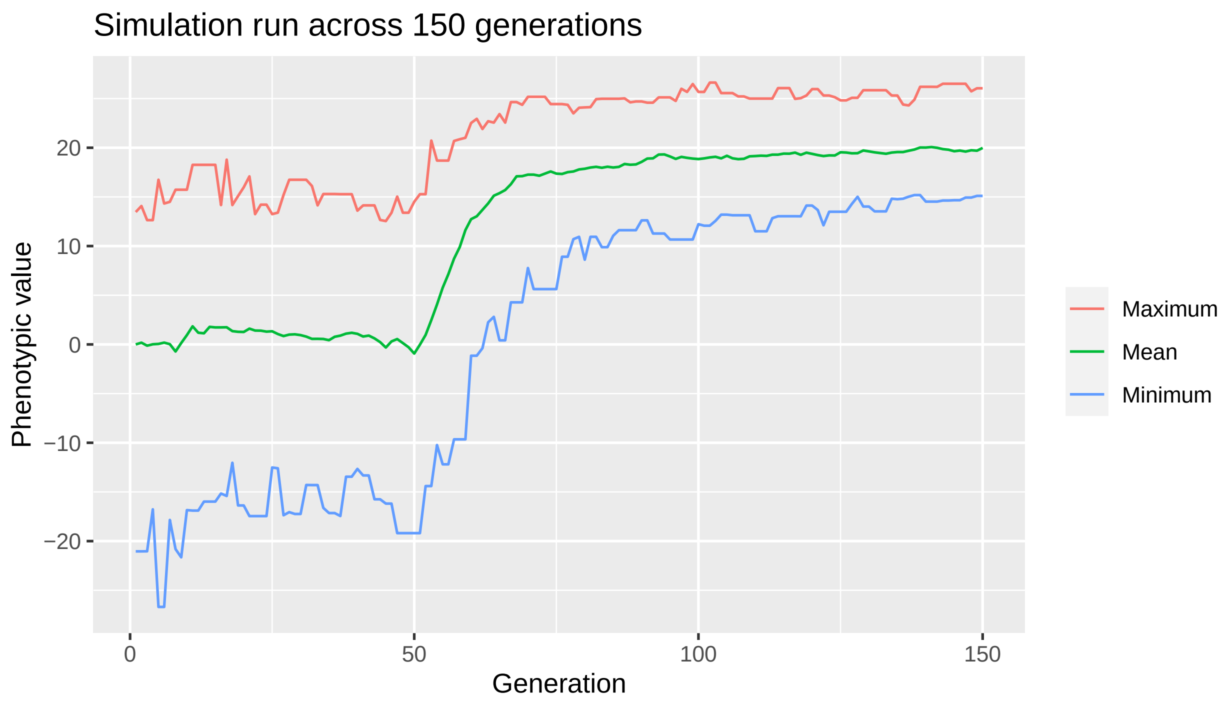

Figure 15: A graphical representation of a simulation run using truncation selection and a burn-in period of the first 50 generations.

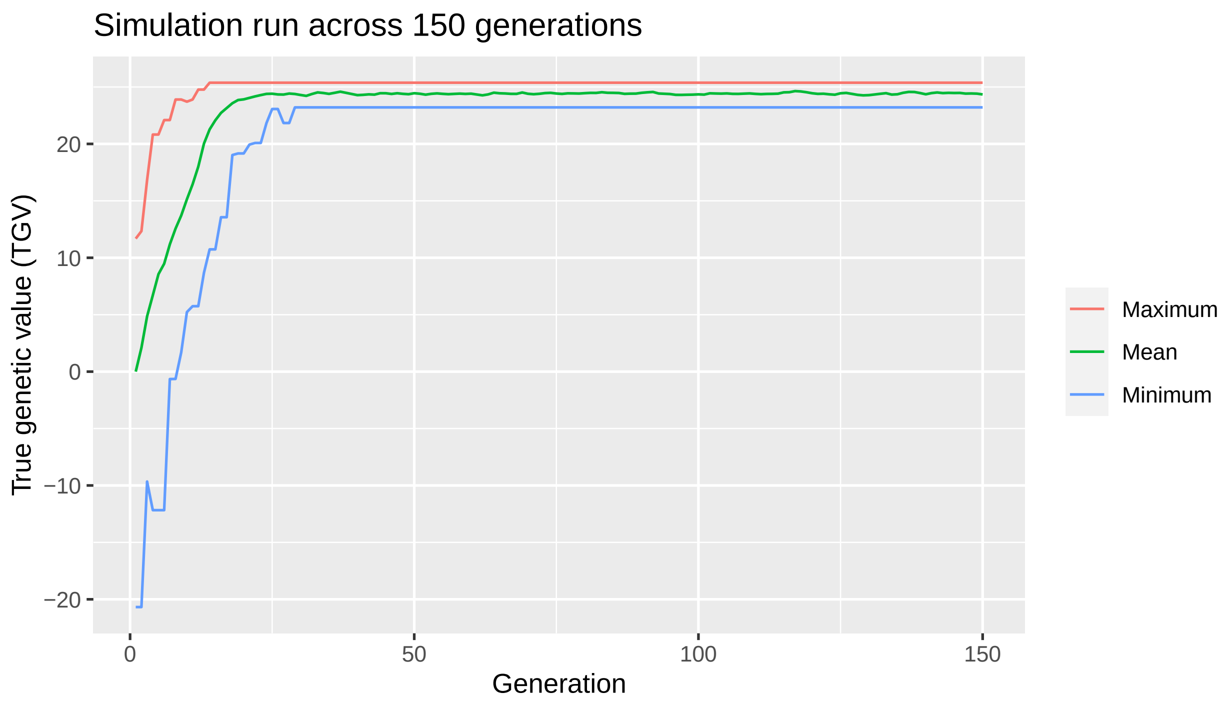

Figure 16: A graphical representation of a simulation run using truncation selection based on true genetic values.

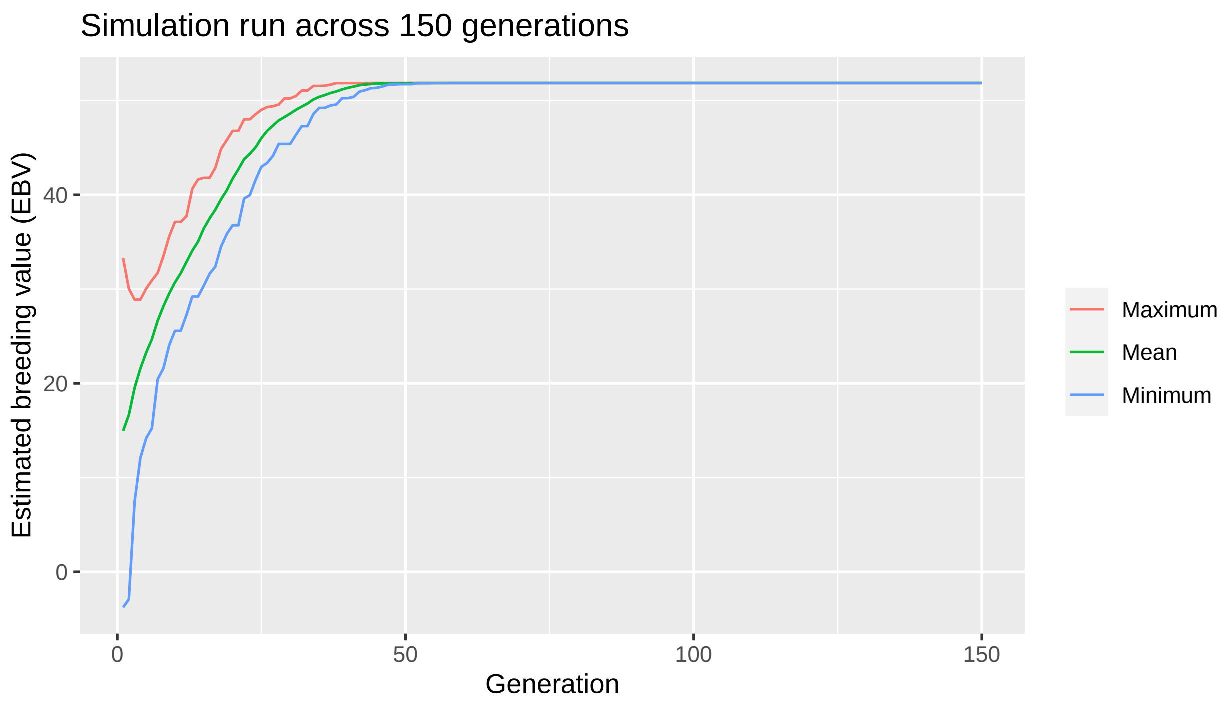

Figure 17: A graphical representation of a simulation run using truncation selection based on estimated breeding values.

To begin retrieving data from a simulation run, we can use the *getPedigree* function to return the pedigree data frame for the population across the entire run. For example:

```
ped <- getPedigree(popRun)
ped[512:517, ]
```

```
##       ID Sire Dam   Additive  Epistatic Environmental   Phenotype       EBV Sex
## 512 512  331 338 -2.7412185 -0.2401316    0.06327522  -2.9180748 12.432650   F
## 513 513  327 362 -4.3968503 -7.1025908    0.90913574 -10.5903054 12.614679   F
## 514 514  327 362  0.8707220  0.4362563   -1.75451140  -0.4475331 15.066130   F
## 515 515  220 266 -5.3492652  3.9855881    1.87366674   0.5099897 12.525046   F
## 516 516  220 266 -6.5683768  1.7462007   -0.50293537  -5.3251115  8.725033   F
## 517 517  248 359 -0.4417693  6.2158731   -2.20283087   3.5712728 16.896806   F
##     Round
## 512     3
## 513     3
## 514     3
## 515     3
## 516     3
## 517     3
```

Note that the originating round is included in the pedigree of each individual.

The *getAlleleFreqRun* function returns the allele frequencies for each SNP per generation. For example:

```
qtl <- getQTL(popRun)$Index
af <- getAlleleFreqRun(popRun)
af[, qtl[1]]
```

```
##   [1] 0.4575 0.4575 0.4725 0.4875 0.4675 0.4875 0.4550 0.4425 0.4600 0.4950
##  [11] 0.5375 0.5250 0.5100 0.4575 0.4675 0.4575 0.5000 0.5075 0.4875 0.4425
##  [21] 0.4525 0.3650 0.3200 0.3400 0.3500 0.3175 0.3200 0.3050 0.3125 0.3100
##  [31] 0.3200 0.2725 0.2650 0.3025 0.3125 0.3050 0.3200 0.3300 0.2925 0.2725
##  [41] 0.2850 0.2875 0.3325 0.3650 0.3450 0.3550 0.3625 0.3750 0.3775 0.3925
##  [51] 0.3775 0.4050 0.4375 0.4250 0.4400 0.4025 0.4200 0.3900 0.3500 0.3475
##  [61] 0.3175 0.3050 0.3050 0.3100 0.2800 0.2500 0.2425 0.2275 0.2400 0.2150
##  [71] 0.2350 0.2375 0.2150 0.2375 0.2450 0.2275 0.2325 0.2350 0.2650 0.2175
##  [81] 0.2075 0.2175 0.2150 0.1950 0.1550 0.1200 0.1325 0.1425 0.1575 0.1550
##  [91] 0.1425 0.1375 0.1425 0.1475 0.1625 0.1700 0.1825 0.2100 0.1900 0.1675
## [101] 0.1625 0.1575 0.1250 0.1100 0.1225 0.1500 0.1325 0.1225 0.1250 0.1100
## [111] 0.1425 0.1900 0.1650 0.1625 0.1575 0.1425 0.1025 0.1050 0.0925 0.0800
## [121] 0.0900 0.0850 0.0850 0.1025 0.1100 0.1200 0.1050 0.1000 0.1200 0.1100
## [131] 0.1025 0.0975 0.0850 0.0850 0.0950 0.1100 0.0975 0.1100 0.1075 0.1050
## [141] 0.1075 0.0975 0.1000 0.1050 0.1000 0.0975 0.0900 0.0800 0.0700 0.0575
```

The *getPhased* function returns the phased genotype matrix for the current population:

```
geno <- getPhased(popRun)
geno[1:6, 1:10]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    0    0    0    0    1    0    0    1     1
## [2,]    0    0    0    0    0    1    0    0    1     1
## [3,]    0    0    0    0    1    1    0    0    0     0
## [4,]    0    0    0    0    1    1    0    0    0     0
## [5,]    0    0    0    0    0    0    0    0    1     1
## [6,]    0    0    0    0    0    0    0    0    1     1
```

Alternatively, the *getGeno* function returns the unphased genotype matrix for the current population:

```
geno <- getGeno(popRun)
geno[1:6, 1:5]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    1    0    2
## [2,]    0    0    1    0    2
## [3,]    0    0    2    0    0
## [4,]    0    0    2    0    0
## [5,]    0    0    0    0    2
## [6,]    0    0    0    0    2
```

Finally, we can create a new subpopulation based on the current population by specifying the IDs to use:

```
ID <- getComponents(popRun)$ID
ID <- sample(ID, 50)
popRun2 <- getSubPop(popRun, ID)
```

Using the pedigree dropper

Finally, we can use a pedigree data frame to determine selection, with the data frame giving IDs for each individual as well as its sire and dam IDs. Such a data frame can be retrieved from a previous simulation run using the *getPedigree* function on the resulting population, or we can instead use a new data frame like so:

```
pedData[201:210, ]
```

```
##      ID Sire Dam
## 201 201  116 100
## 202 202  116 100
## 203 203   91 169
## 204 204   91 169
## 205 205  154 129
## 206 206  154 129
## 207 207   23 192
## 208 208   23 192
## 209 209   66 185
## 210 210   66 185
```

To use this “pedigree dropper”, we call *runSim* with the *pedigree* argument:

```
popRunPed <- runSim(pop, pedigree = pedData)
```

The pedigree dropper first sorts the pedigree into the implicit number of generations, then runs the simulation using selection according to the given data frame. (Note that if you're using pedigree data from a previous run, the number of generations reported by the pedigree dropper may be different from the number of iterations of the simulator that produced the pedigree.)

As usual, we can plot the resulting run:

```
plot(popRunPed)
```

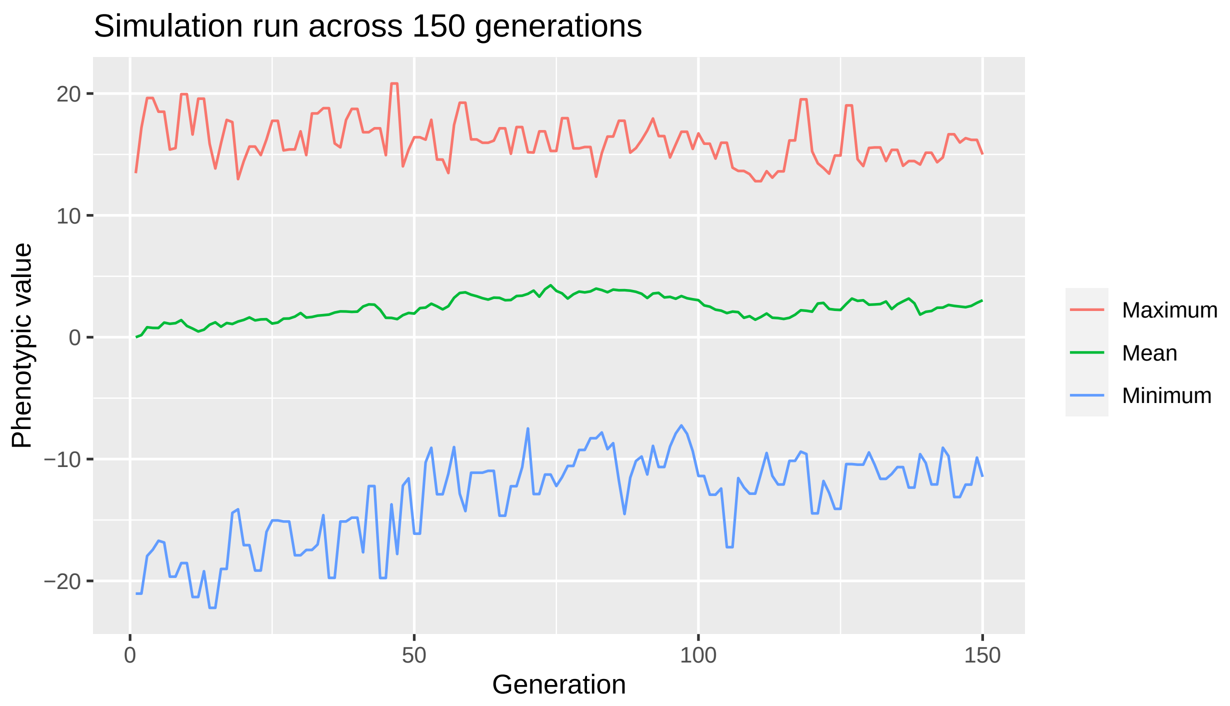

Figure 18: A graphical representation of a simulation run using a pedigree data frame.

Conclusion

The *epinetr* package is a flexible suite of functions designed to allow for the analysis of epistasis under a multitude of conditions, with complex interactions being a core component of the simulation. This vignette is intended as an overview of the package, with as much detail as possible included. That said, the help pages for each function will provide further detail.

References

1. VanRaden, P. M. Efficient methods to compute genomic predictions. *Journal of dairy science* **91**, 4414–4423 (2008).
2. Yang, J., Zeng, J., Goddard, M. E., Wray, N. R. & Visscher, P. M. Concepts, estimation and interpretation of SNP-based heritability. *Nature genetics* **49**, 1304–1310 (2017).
3. Clark, S. A. & Werf, J. van der. Genomic best linear unbiased prediction (gBLUP) for the estimation of genomic breeding values. in *Genome-wide association studies and genomic prediction* 321–330 (Springer, 2013).

4. Habier, D., Fernando, R. L. & Garrick, D. J. Genomic BLUP decoded: A look into the black box of genomic prediction. *Genetics* **194**, 597–607 (2013).

5. Barabási, A.-L. & Albert, R. Emergence of scaling in random networks. *science* **286**, 509–512 (1999).

6. Goldberg, D. E. & Deb, K. A comparative analysis of selection schemes used in genetic algorithms. in *Foundations of genetic algorithms* vol. 1 69–93 (Elsevier, 1991).