# Hierarchical inference for genome-wide association studies

Claude Renaux, Laura Buzdugan, Markus Kalisch and Peter Bühlmann
Seminar for Statistics, ETH Zürich

May 3, 2019

**This vignette is based on the pre-print Renaux et al. (2018) and contains the part about the illustration of our R-package `hierinf`.**

## 1 Cite `hierinf`

If you use the `hierinf` package, please cite the paper Renaux, C., Buzdugan, L., Kalisch, M., and Bühlmann, P. (2018). Hierarchical inference for genome-wide association studies: a view on methodology with software. *arXiv preprint arXiv:1805.02988*.

## 2 Introduction

Hierarchical inference is a key technique for computationally and statistically efficient hypothesis testing and multiple testing adjustment. We consider inference in a multivariate model which quantifies effects after adjusting for all remaining single nucleotide polymorphism (SNP) covariates. The hierarchy enables in a fully data-driven way to infer significant groups or regions of SNPs at an adaptive resolution, by controlling the familywise error rate (FWER). We have recently proposed high-dimensional hierarchical inference for assigning statistical significance in terms of p-values for groups of SNPs being associated to a response variable: Buzdugan et al. (2016) considers this approach for human GWAS and Klasen et al. (2016) for GWAS with plants. The methodological and theoretical concepts have been worked out in Mandozzi and Bühlmann (2016a) and Mandozzi and Bühlmann (2016b).

The R package `hierinf` is an implementation of the hierarchical inference described in Renaux et al. (2018) and it is easy to use for GWAS. The package is a re-implementation of the R package `hierGWAS` (Buzdugan, 2017) and includes new features like straightforward parallelization, an additionally option for constructing a hierarchical tree based on spatially contiguous genomic positions, and the possibility of jointly analyzing multiple datasets.

## 3 Software

To summarize the method, one starts by clustering the data hierarchically. This means that the clusters can be represented by a tree. The main idea is to pursue testing top-down and successively moving downwards until the null-hypotheses cannot be rejected. The p-value of a given cluster is

calculated based on the multiple sample splitting approach and aggregation of those p-values as described in Renaux et al. (2018).

The work flow is straightforward and is composed in two function calls. We note that the package `hierinf` requires complete observations, i.e. no missing values in the data, because the testing procedure is based on all the SNPs which is in contrast to marginal tests. If missing values are present, they can be imputed prior to the analysis. This can be done in R using e.g. `mice` (van Buuren and Groothuis-Oudshoorn, 2011), `mi` (Shi et al., 2011), or `missForest` (Stekhoven and Bühlmann, 2012).

A small simulated toy example with two chromosomes is used to demonstrate the procedure. The toy example is taken from (Buzdugan, 2017) and was generated using PLINK where the SNPs were binned into different allele frequency ranges. The response is binary with 250 controls and 250 cases. Thus, there are $n = 500$ samples, the number of SNPs is $p = 1000$, and there are two additional control variables with column names "age" and "sex". The first 990 SNPs have no association with the response and the last 10 SNPs were simulated to have a population odds ratio of 2. The functions of the package `hierinf` require the input of the SNP data to be a `matrix` (or a list of matrices for multiple datasets). We use a `matrix` instead of a `data.frame` since this makes computation faster.

```
# load the package
library(hierinf)

# random number generator (for parallel computing)
RNGkind("L'Ecuyer-CMRG")

# We use a small build-in dataset for our toy example.
data(simGWAS)

# The genotype, phenotype and the control variables are saved in
# different objects.
sim.geno  <- simGWAS$x
sim.pheno <- simGWAS$y
sim.clvar <- simGWAS$clvar
```

The two following sections correspond to the two function calls in order to perform hierarchical testing. The third section gives some notes about running the code in parallel.

### 3.1 Software for clustering

The package `hierinf` offers two possibilities to build a hierarchical tree for corresponding hierarchical testing. The function `cluster_var` performs hierarchical clustering based on some dissimilarity matrix and is described first. The function `cluster_position` builds a tree based on recursive binary partitioning of consecutive positions of the SNPs. For a short description, see at the end of Section 2.3 in Renaux et al. (2018).

Hierarchical clustering is computationally expensive and prohibitive for large datasets. Thus, it makes sense to pre-define dis-joint sets of SNPs which can be clustered separately. One would typically assume that the second level of a cluster tree structure corresponds to the blocks given

by the chromosomes as illustrated in Figure 1. For the method based on binary partitioning of consecutive positions of SNPs, we recommend to pre-define the second level of the hierarchical tree as well. This allows to run the building of the hierarchical tree and the hierarchical testing for each block or in our case for each chromosome in parallel, which can be achieved by adding the two commented arguments in the function calls below. If one does not want to specify the second level of the tree, then the argument `block` in both function calls can be omitted.

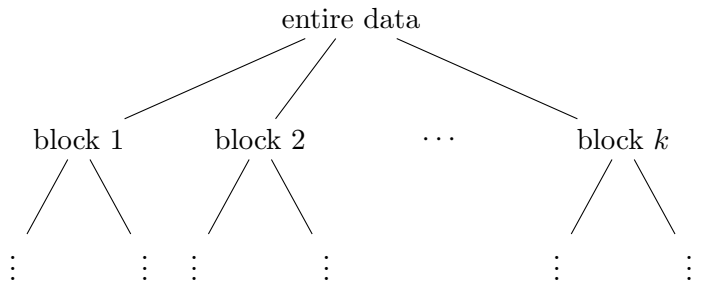


Figure 1: The top two levels of a hierarchical tree used to perform multiple testing. The user can optionally specify the second level of the tree with the advantage that one can easily run the code in parallel over different clusters in the second level, denoted by block 1, ..., block $k$. A natural choice is to choose the chromosomes as the second level of the hierarchical tree, which define a partition of the SNPs. If the second level is not specified, then the first split is estimated based on clustering the data, i.e. it is a binary split. The user can define the second level of the tree structure using the argument `block` in the functions `cluster_var` / `cluster_position`. The function `cluster_var` / `cluster_position` builds a separate binary hierarchical tree for each of the blocks.

In the toy example, we define the second level of the tree structure as follows. The first and second 500 SNPs of the SNP data `sim.geno` correspond to chromosome 1 and chromosome 2, respectively. The object `block` is a `data.frame` which contains two columns identifying the two blocks. The blocks are defined in the second column and the corresponding column names of the SNPs are stored in the first column. The argument `stringsAsFactors` of the function `data.frame` is set to `FALSE` because we want both columns to contain integers or strings.

```
# Define the second level of the tree structure.
block <- data.frame("colname" = paste0("SNP.", 1:1000),
                    "block" = rep(c("chrom 1", "chrom 2"), each = 500),
                    stringsAsFactors = FALSE)

# Cluster the SNPs
dendr <- cluster_var(x = sim.geno,
                     block = block)
                     # # the following arguments have to be specified
                     # # for parallel computation
                     # parallel = "multicore",
                     # ncpus = 2)
```

By default, the function `cluster_var` uses the agglomeration method average linkage and the dissimilarity matrix given by $1 - (\text{empirical correlation})^2$.

Alternatively, `cluster_position` builds a hierarchical tree using recursive binary partitioning of consecutive genomic positions of the SNPs. As for `cluster_var`, the function can be run in parallel if the argument block defines the second level of the hierarchical tree and the two commented arguments parallel and ncpus are added.

```
# Store the positions of the SNPs.
position <- data.frame("colnames" = paste0("SNP.", 1:1000),
                       "position" = seq(from = 1, to = 1000),
                       stringsAsFactors = FALSE)


# Build the hierarchical tree based on the position
# The argument block defines the second level of the tree structure.
dendr.pos <- cluster_position(position = position,
                              block = block)
                              # # the following arguments have to be
                              # # specified for parallel computation
                              # parallel = "multicore",
                              # ncpus = 2)
```

### 3.2 Software for hierarchical testing

The function `test_hierarchy` is executed after the function `cluster_var` or `cluster_position` since it requires the output of one of those two functions as an input (argument `dendr`).

The function `test_hierarchy` first randomly splits the data into two halves (with respect to the observations), by default `B = 50` times, and performs variable screening on the second half. Then, the function `test_hierarchy` uses those splits and corresponding selected variables to perform the hierarchical testing according to the tree defined by the output of one of the two functions `cluster_var` or `cluster_position`.

As mentioned in Section 3.1, we can exploit the proposed hierarchical structure which assumes the chromosomes to form the second level of the tree structure as illustrated in Figure 1. This allows to run the testing in parallel for each block, which are the chromosomes in the toy example.

The following function call performs first the global null-hypothesis test for the group containing all the variables/SNPs and continues testing in the hierarchy of the two chromosomes and their children.

```
# Test the hierarchy using multi sample split
set.seed(1234)
result <- test_hierarchy(x = sim.geno,
                         y = sim.pheno,
                         clvar = sim.clvar,
                         # alternatively: dendr = dendr.pos
                         dendr = dendr,
```

```
                        family = "binomial")
                        # # the following arguments have to be
                        # # specified for parallel computation
                        # parallel = "multicore",
                        # ncpus = 2)
```

The function `test_hierarchy` allows to fit models with continuous or binary response, the latter being based on logistic regression. The argument `family` is set to `"binomial"` because the response variable in the toy example is binary.

The output looks as follows:

```
print(result, n.terms = 4)

##   block   p.value   significant.cluster
## 1 chrom 1 0.0372845 SNP.1, SNP.2, SNP.3, SNP.4, ... [496]
## 2 chrom 2 0.0253367 SNP.605, SNP.792, SNP.636, SNP.857, ... [21]
## 3 chrom 2 0.0015598 SNP.992
## 4 chrom 2 2.642e-05 SNP.991
## 5 chrom 2 0.0034648 SNP.1000
## 6 chrom 2 0.0152762 SNP.994
## 7 chrom 2 0.0005863 SNP.993
## 8 chrom 2 0.0117842 SNP.997
```

The output shows significant groups of SNPs or even single SNPs if there is sufficient strong signal in the data. The block names, the p-values, and the column names (of the SNP data) of the significant clusters are returned. There is no significant cluster in chromosome 1. That's the reason why the p-value and the column names of the significant cluster are `NA` in the first row of the output. Note that the large significant cluster in the second row of the output is shortened to better fit on screen. In our toy example, the last 8 column names are replaced by "`... [8]`". The maximum number of terms can be changed by the argument `n.terms` of the `print` function. One can evaluate the object `result` in the console and the default values of the `print` function are used. In this case, it would only display the first 5 terms.

The only difference in the R code when using a hierarchical tree based on binary recursive partitioning of the genomic positions of the SNPs (whose output is denoted as `dendr.pos`) is to specify the corresponding hierarchy: `test_hierarchy(..., dendr = dendr.pos, ...)`.

We can access part of the output by `result$res.hierarchy` which we use below to calculate the $R^2$ value of the second row of the output, i.e. `result$res.hierarchy[[2, "significant.cluster"]]`. Note that we need the double square brackets to access the column names stored in the column `significant.cluster` of the output since the last column is a list where each element contains a character vector of the column names. The two other columns containing the block names and the p-values can both be indexed using single square brackets as for any `data.frame`, e.g. `result$res.hierarchy[2, "p.value"]`.

```
(coln.cluster <- result$res.hierarchy[[2, "significant.cluster"]])

##  [1] "SNP.605" "SNP.792" "SNP.636" "SNP.857" "SNP.858" "SNP.911" "SNP.571"
```

```
##  [8] "SNP.998" "SNP.708" "SNP.867" "SNP.612" "SNP.932" "SNP.803" "SNP.920"
## [15] "SNP.643" "SNP.653" "SNP.778" "SNP.808" "SNP.720" "SNP.714" "SNP.732"
## [22] "SNP.854" "SNP.876" "SNP.727" "SNP.738"
```

The function `compute_r2` calculates the adjusted $R^2$ value or coefficient of determination of a cluster for a continuous response. The Nagelkerke's $R^2$ (Nagelkerke et al., 1991) is calculated for a binary response as e.g. in our toy example.

```
compute_r2(x = sim.geno, y = sim.pheno, clvar = sim.clvar,
           res.test.hierarchy = result, family = "binomial",
           colnames.cluster = coln.cluster)

## [1] 0.07316735
```

The function `compute_r2` is based on multi-sample splitting. The $R^2$ value is calculated per split based on the second half of observations and based on the intersection of the selected variables and the user-specified cluster. Then, the $R^2$ values are averaged over the different splits. If one does not specify the argument `colnames.cluster`, then the $R^2$ value of the whole dataset is calculated.

### 3.3 Software for parallel computing

The function calls of `cluster_var`, `cluster_position`, and `test_hierarchy` above are evaluated in parallel since we set the arguments `parallel = "multicore"` and `ncpus = 2`. The argument `parallel` can be set to `"no"` for serial evaluation (default value), to `"multicore"` for parallel evaluation using forking, or to `"snow"` for parallel evaluation using a parallel socket cluster (PSOCKET); see below for more details. The argument `ncpus` corresponds to the number of cores to be used for parallel computing. We use the `parallel` package for our implementation which is already included in the base R installation (R Core Team, 2017).

The user has to select the "L'Ecuyer-CMRG" pseudo-random number generator and set a seed such that the parallel computing of `hierinf` is reproducible. This pseudo-random number generator can be selected by `RNGkind("L'Ecuyer-CMRG")` and has to be executed once for every new R session; see R code at the beginning of Section 3. This allows us to create multiple streams of pseudo-random numbers, one for each processor / computing node, using the `parallel` package; for more details see the vignette of the `parallel` package published by R Core Team (2017).

We recommend to set the argument `parallel = "multicore"` which will work on Unix/Mac (but not Windows) operation systems. The function is then evaluated in parallel using forking which is leaner on the memory usage. This is a neat feature for GWAS since e.g. a large SNP dataset does not have to be copied to the new environment of each of the processors. Note that this is only possible on a multicore machine and not on a cluster.

On all operation systems, it is possible to create a parallel socket cluster (PSOCKET) which corresponds to setting the argument `parallel = "snow"`. This means that the computing nodes or processors do not share the memory, i.e. an R session with an empty environment is initialized for each of the computing nodes or processors.

How many processors should one use? If the user specifies the second level of the tree, i.e. defines the `block` argument of the functions `cluster_var` / `cluster_position` and `test_hierarchy`, then the building of the hierarchical tree and the hierarchical testing can be easily performed in parallel

across the different blocks. Note that the package can make use of as many processors as there are blocks, say, 22 chromosomes. In addition, the multi sample splitting and screening step, which is performed inside the function `test_hierarchy`, can always be executed in parallel regardless if we defined blocks or not. It can make use of at most $B$ processors where $B$ is the number of sample splits.

# 4 Meta-analysis for several datasets

The naive (and conceptually wrong) approach would be to pool the different datasets and proceed as if it would be one homogeneous dataset, say, allowing for a different intercept per dataset. We advocate meta analysis and aggregating corresponding p-values; see (Renaux et al., 2018, Sec. 4) for more details.

**Fast computational methods for pooled GWAS.** There has been a considerable interest for fast algorithms for GWAS with very large sample size in the order of $10^5$; see Lippert et al. (2011); Zhou and Stephens (2014). Often though, such large sample size comes from pooling different studies or sub-populations. We argue in favor of meta analysis and aggregating corresponding p-values. Besides more statistical robustness against heterogeneity (arising from the different sub-populations), meta-analysis is also computationally very attractive: the computations can be trivially implemented in parallel for every sub-population and the p-value aggregation step comes essentially without any computational cost.

## 4.1 Software for aggregating p-values of multiple studies

It is very convenient to combine the information of multiple studies by aggregating p-values as described in (Renaux et al., 2018, Sec. 4). The package `hierinf` offers two methods for jointly estimating a single hierarchical tree for all datasets using either of the functions `cluster_var` or `cluster_position`; compare with Section 3.1. Testing is performed by the function `test_hierarchy` in a top-down manner given by the joint hierarchical tree. For a given cluster, p-values are calculated based on the intersection of the cluster and each dataset (corresponding to a study) and those p-values are then aggregated to obtain one p-value per cluster using either Tippett's rule or Stouffer's method as in (Renaux et al., 2018, Sec. 4); see argument `agg.method` of the function `test_hierarchy`. The difference and issues of the two methods for estimating a joint hierarchical tree are described in the following two paragraphs.

The function `cluster_var` estimates a hierarchical tree based on clustering the SNPs from all the studies. Problems arise if the studies do not measure the same SNPs and thus, some of the entries of the dissimilarity matrix cannot be calculated. By default, pairwise complete observations for each pair of SNPs are taken to construct the dissimilarity matrix. The issue affects the building of the hierarchical tree but the testing of a given cluster remains as described before.

The function `cluster_position` estimates a hierarchical tree based on the genomic positions of the SNPs from all the studies. The problems mentioned above do not show up here since SNPs, may be different ones for various datasets, can still be uniquely assigned to genomic regions.

The only differences in the function calls are that the arguments `x`, `y`, and `clvar` are now each a list of matrices instead of just a single matrix. Note that the order of the list elements of the arguments `x`, `y`, and `clvar` matter, i.e. the user has to stick to the order that the first element

of the three lists corresponds to the first dataset, the second element to the second datasets, and so on. One would replace the corresponding element of the list containing the control covariates (argument `clvar`) by `NULL` if some dataset has no control covariates. If none of the datasets have control covariates, then one can simply omit the argument. Note that the argument `block` defines the second level of the tree which is assumed to be the same for all datasets or studies. The argument `block` has to be a `data.frame` which contains all the column names (of all the datasets or studies) and their assignment to the blocks. The aggregation method can be chosen using the argument `agg.method` of the function `test_hierarchy`, i.e. it can be set to either `"Tippett"` or `"Stouffer"`. The default aggregation method is Tippett's rule.

The example below demonstrates the functions `cluster_var` and `test_hierarchy` for two datasets / studies measuring the same SNPs.

```
# The datasets need to be stored in different elements of a list.
# Note that the order has to be the same for all the lists.
# As a simple example, we artificially split the observations of the
# toy dataset in two parts, i.e. two datasets.
set.seed(89)
ind1 <- sample(1:500, 250)
ind2 <- setdiff(1:500, ind1)
sim.geno.2dat  <- list(sim.geno[ind1, ],
                       sim.geno[ind2, ])
sim.clvar.2dat <- list(sim.clvar[ind1, ],
                       sim.clvar[ind2, ])
sim.pheno.2dat <- list(sim.pheno[ind1],
                       sim.pheno[ind2])

# Cluster the SNPs
dendr <- cluster_var(x = sim.geno.2dat,
                     block = block)
                     # # the following arguments have to be specified
                     # # for parallel computation
                     # parallel = "multicore",
                     # ncpus = 2)

# Test the hierarchy using multi sample split
set.seed(1234)
result <- test_hierarchy(x = sim.geno.2dat,
                         y = sim.pheno.2dat,
                         clvar = sim.clvar.2dat,
                         dendr = dendr,
                         family = "binomial")

## Warning in test_only_hierarchy(x = res$x, y = res$y, dendr = dendr, res.multisplit
= res.multisplit, :  There occurred some warnings while testing the hierarchy.  See
attribute 'warningMsgs' of the corresponding list element of the return object for
more details.
```

```
## Warning in test_hierarchy(x = sim.geno.2dat, y = sim.pheno.2dat, clvar = sim.clvar.2dat,
:  There occurred some warnings while multi splitting and / or testing the hierarchy.
See attribute 'warningMsgs' of the corresponding list element of the return object
for more details.
```

```
                        # # the following arguments have to be
                        # # specified for parallel computation
                        # parallel = "multicore",
                        # ncpus = 2)
```

The above R code can be evaluated in parallel if one adds the two commented arguments parallel and ncpus; compare with Section 3.3 for more details about the software for parallel computing.

The output shows three significant groups of SNPs and one single SNP.

```
print(result, n.terms = 4)
```

```
##   block   p.value   significant.cluster
## 1 chrom 1 NA        NA
## 2 chrom 2 0.0046333 SNP.991
## 3 chrom 2 0.0414371 SNP.994
## 4 chrom 2 0.0056558 SNP.993
## 5 chrom 2 0.0008686 SNP.997
```

The significance of a cluster is based on the information of both datasets. For a given cluster, the p-values of each dataset were aggregated using Tippett's rule as in (Renaux et al., 2018, Sec. 4) or Tippett (1931). Those aggregated p-values are displayed in the output above. We cannot judge which dataset (or both or combined) inherits a strong signal such that a cluster is shown significant but that is not the goal. The goal is to combine the information of multiple studies.

The crucial point is that the testing procedure goes top-down through a single jointly estimated tree for all the studies and only continues if at least one child is significant (based on the aggregated p-values of the multiple datasets) of a given cluster. The algorithm determines where to stop and naturally we get one output for all the studies. A possible single jointly estimated tree of the above R code is illustrated in Figure 2. In our example, both datasets measure the same SNPs. If that would not be the case, then intersection of the cluster and each dataset is taken before calculating a p-value per dataset / study and then aggregating those.

## References


Buzdugan, L. (2017). *hierGWAS: Asessing statistical significance in predictive GWA studies*. R package version 1.6.0.

Buzdugan, L., Kalisch, M., Navarro, A., Schunk, D., Fehr, E., and Bühlmann, P. (2016). Assessing statistical significance in multivariable genome wide association analysis. *Bioinformatics*, 32:1990–2000.

Klasen, J., Barbez, E., Meier, L., Meinshausen, N., Bühlmann, P., Koornneef, M., Busch, W., and Schneeberger, K. (2016). A multi-marker association method for genome-wide association studies

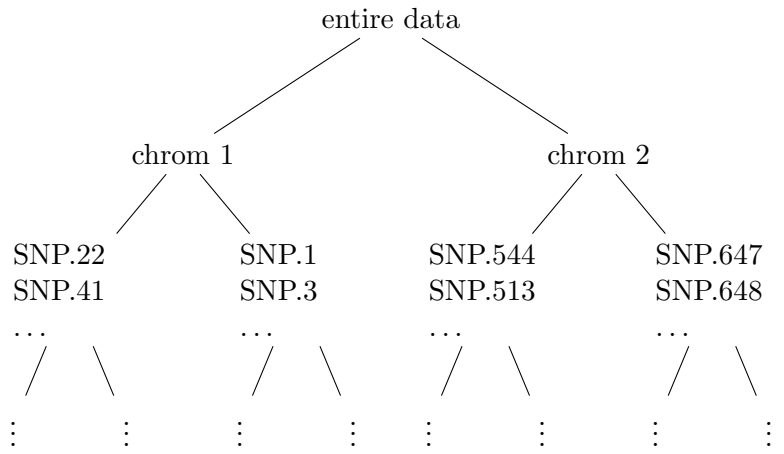


Figure 2: Illustration of a possible single jointly estimated tree for multiple studies based on clustering the SNPs. The second level of the hierarchical tree is defined by chromosome 1 and 2 (defined by the argument `block` of the functions `cluster_var` / `cluster_position`). The function `cluster_var` / `cluster_position` builds a separate hierarchical tree for each of the chromosomes.

without the need for population structure correction. *Nature Communications*, 7:Article number 13299 (doi:10.1038/ncomms13299).

Lippert, C., Listgarten, J., Liu, Y., Kadie, C. M., Davidson, R. I., and Heckerman, D. (2011). Fast linear mixed models for genome-wide association studies. *Nature Methods*, 8:833.

Mandozzi, J. and Bühlmann, P. (2016a). Hierarchical testing in the high-dimensional setting with correlated variables. *Journal of the American Statistical Association*, 111:331–343.

Mandozzi, J. and Bühlmann, P. (2016b). A sequential rejection testing method for high-dimensional regression with correlated variables. *International Journal of Biostatistics*, 12:79–95.

Nagelkerke, N. J. et al. (1991). A note on a general definition of the coefficient of determination. *Biometrika*, 78:691–692.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Renaux, C., Buzdugan, L., Kalisch, M., and Bühlmann, P. (2018). Hierarchical inference for genome-wide association studies: a view on methodology with software. *arXiv preprint arXiv:1805.02988*.

Shi, G., Boerwinkle, E., Morrison, A. C., Gu, C. C., Chakravarti, A., and Rao, D. (2011). Mining gold dust under the genome wide significance level: a two-stage approach to analysis of GWAS. *Genetic epidemiology*, 35:111–118.

Stekhoven, D. J. and Bühlmann, P. (2012). Missforest – non-parametric missing value imputation for mixed-type data. *Bioinformatics*, 28(1):112–118.

Tippett, L. H. C. (1931). *Methods of statistics*. Williams Norgate, London, 1st edition.

van Buuren, S. and Groothuis-Oudshoorn, K. (2011). mice: Multivariate imputation by chained equations in R. *Journal of Statistical Software, Articles*, 45:1–67.

Zhou, X. and Stephens, M. (2014). Efficient multivariate linear mixed model algorithms for genome-wide association studies. *Nature Methods*, 11:407–409.