# EPIHAPMPI manual

Dzianis Prakapenka

June 12, 2025

# Contents

# EPIHAPMPI

EPIHAPMPI is a genomic prediction software package for HPC clusters

# Installation

This guide shows how to compile and run the **reml** and **grm**, and other applications using the provided Makefile. GPU support is under active development and described in the final section.

---

## Prerequisites

1. **Modules / Environment**

```
# if using on a cluster, load Intel oneAPI components, e.g.:
module load intel-oneapi-mkl
module load intel-oneapi-mpi
module load intel-oneapi-compilers
# or source the environment variables from your
# installation. replace the path to setvars.sh
# if needed
source /opt/intel/oneapi/setvars.sh
```

2. **Make sure** `$MKLROOT` is defined:

```
echo "$MKLROOT"
/opt/intel/oneapi/mkl/2025.0
```

3. **MPI implementation** installed and in your `PATH`.

---

## Directory Layout

```
epihap.mpi/
|-- src/                    # source code
|   |-- reml.cpp
|   |-- options.cpp
|   |-- ...
|-- obj/                    # object files
|-- bin/                    # executables built here
|-- include/                # header files
|-- manual/                 # markdown documentation files
|-- scripts/                # helper scripts
|-- test/                   # example test folder
|-- tests/                  # unit tests
|-- example/                # example input files
|-- Makefile                # makefile for compilation
```

---

## Makefile Targets

- **`make reml`**
  Builds the CPU-only version (`bin/reml`) for variance component estimation, heritabilites, and GBLUP.
- **`make grm`**
  Builds the CPU-only version (`bin/grm.snp`) for second order snp genomic relationship matrices(GRMs).
- **`make grm.hap`**
  Builds the CPU-only version (`bin/grm.hap`) for haplotype additive genomic relationship matrix (GRM).
- **`make grm.3order`**
  Builds the CPU-only version (`bin/grm.snp.3order`) for third order genomic relationship matrices (GRMs).
- **`make rank`**
  Builds the CPU-only version (`bin/rank`) calcualtes the rank of given GRMs.
- **`make read.bin`**
  Builds the CPU-only version (`bin/read.bin`) displays a portion of the binary GRM.
- **`make manual`**
  Generates a pdf version of the manual from .md and manual/*.md files in this repository
- **`make clean`**
  Deletes object files and bianries

---

## Building

1. **Clean previous builds**

```
make clean
```

2. **Build CPU-only**

```
make all
# produces reml, grm.snp, grm.hap, and grm.3order executables
```

---

## Running

Invoke with MPI as usual:

```
# build the GRMs for the model you want to use
# replace 4 with the number of MPI tasks you have available
mpirun -np 4 ./bin/grm.snp [options]
mpirun -np 4 ./bin/grm.hap [options]
./bin/grm.snp.3order [options]

# run the prediction using GRMs
mpirun -np 4 ./bin/reml [options]
```

---

## Troubleshooting

- **Header not found**: Ensure MKL include path is in your compiler flags (the variable MKLROOT is set).
- **Library not found**: Ensure `$MKLROOT/lib` is in your linker flags.

---

## Upcoming GPU acceleration

**This section is in development.** GPU acceleration is provided via an override library that intercepts BLAS calls:

1. **Environment**

```
module load cuda            # sets $CUDA_HOME
export NGPUS=1              # GPUs per MPI rank
```

2. **Directory additions**

```
obj/gpu/                    # GPU object files, including dgemm.override.o
```

3. **Makefile changes**

- `make reml.gpu` target compiles sources into `obj/gpu/*.o`
  - Intercepts `dgemm_` and `cblas_dgemm` via `src/dgemm.override.cpp`
  - Links with `-lcublas` from `$CUDA_HOME/lib64`

4. **Building GPU version**

```
make reml.gpu
# produces bin/reml.gpu
```

5. **Running GPU version**

```
mpirun -np 4 ./bin/reml.gpu [options]
```

# Introduction

EPIHAPMPI is a parallel, distributed memory program designed for bi-allelic diploid species and is based on the quantitative genetics model (Da *et al.*, 2022; Wang *et al.*, 2014). This model incorporates multifactorial effects (SNP, haplotype, and epistasis) to investigate the contributions of global low-order and local high-order epistasis effects to the phenotypic variance and the accuracy of genomic prediction of quantitative traits. Users can select up to third-order epistatic effects, additive and dominance effects, and haplotype effects to analyze the undrlying genetic mechanisms. Analogous to EPIHAP (shared memory), this program is intended for large datasets that can not be computed on a regular computer or a single node; it is indeded to run on clusters and requires MPI. This analysis provides genomic best linear unbiased prediction (GBLUP) with associated reliability for individuals with and without phenotypic observations, including a computationally efficient method for large validation populations, and genomic restricted maximum estimation (GREML) of the variance and associated heritability using a combination of EM-REML and AI-REML iterative algorithms.

## Single SNP effects

EPIHAPMPI can construct genetic relationship matrices (GRMs) for additive (A), dominance (D) effects (Wang *et al.*, 2014).

## Epistasis

EPIHAPMPI can construct genetic relationship matrices (GRMs) for additive $\times$ additive (AA), additive $\times$ dominance (AD), dominance $\times$ dominance (DD), additive $\times$ additive $\times$ additive (AAA), additive $\times$ additive $\times$ dominance (AAD), additive $\times$ dominance $\times$ dominance (ADD), dominance $\times$ dominance $\times$ dominance (DDD) effects using the Approximate Genomic Epistasis Relationship Matrices (AGERM) method (CITE: Henderson, 1985; Jiang and Reif, 2020)(Da *et al.*, 2022).

## Haplotype

User defined haplotype blocks can incorporate structural and function information for genomic prediction and estimation, but this method requires large amounts of data preparation due to large number of candidate models. EPIHAPMPI, and the accompanying pipeline programs provide capability to analyze many large-scale haplotype models and automate the process. EPIHAPMPI implements a multi-allelic haplotype mixed model that treats each haplotype block as a 'locus' and each haplotype within the haplotype block as an 'allele' (Da *et al.*, 2022), based on the GVCHAP pipeline (Prakapenka *et al.*, 2020) and uses the GREML_CE method described in (Wang *et al.*, 2014).

## Combination

EPIHAPMPI can combine effects of single SNP, epistasis, and haplotype models to estimate variance components and the associated heritability.

# Input data

The input data required:

- Phenotype file
- Map file
- Genotype/Haplotype files

## Phenotype file format

Phenotype file consists of an id column, followed by the phenotype or trait columns. It can be separated with spaces, tabs, or commas. Missing value is set to -9999111 by default but can be specified with an option.

```
ID Sex Phenotype1 Phenotype2
1 1.23 10.11
2 4.56 7.89
3 12.13 14.15
```

or csv style

```
ID,Sex,Phenotype1,Phenotype2
1,1.23,10.11
2,4.56,7.89
3,12.13,14.15
```

## Map file

Map file consists of an id column, followed by chromosome number column, followed by position column. It can be separated with spaces, tabs, or commas.

```
ID Chr Position
1 1 1456
2 1 2487
3 2 893
```

or csv style

```
ID,Chr,Position
1,1,1456
2,1,2487
3,2,893
```

## Genotype and Haplotype files

These can be all SNPs in one file or separated by chromosomes.

## SNP files

The genotype file format consists of id column followed by SNP columns. SNPs are coded 0,1,2; any other number is treated as missing value

```
ID SNP1 SNP2 SNP3
1 1 0 1
2 0 1 2
3 2 1 5
```

## Haplotype files

The haplotype file format consists of id column followed by SNP allele columns. Each SNP has two columns for allel1 and allele2, they are coded 0 or 1. This file, along with generated hap_info files are used to produce the hap_geno files used for haplotype GRM.

```
ID SNP1_0 SNP1_1 SNP2_0 SNP2_1 SNP3_0 SNP3_1
1 1 0 1 1 0 1
2 0 1 0 0 1 1
3 1 1 1 0 1 1
```

### Generating hap_geno files

Hapgeno files are created by defining a "multi-allelic loci" or a blocks of phased SNPs using get.hap.geno.py script and helper scripts to generate the block definition files. To produce this file the user needs to define haplotype blocks.

### Defining haplotype blocks

The first step to define haplotype genotypes is to define haplotype blocks, where each block is treated as a 'locus' and each haplotype within the block as an 'allele'. Functional and structural genomics information can be used to define haplotype blocks (based on genes, ChIP-seq, etc). Haplotype blocks are considered independent in the model and can be defined as adjacent blocks or overlapping (sliding blocks). The sliding block method is more computationally intensive because there are many more blocks but may provide better resultsby capturing interactions that may have been split by adjacent blocking method.

The following Python programs implement a number of methods for defining haplotype blocks.

- **block-by-snp.py**:
  - Purpose: Generate haplotype block information files by fixed number of SNPs. Blocks are defined consequtively and are adjacent to eachother.
  - Input: map file, number of SNPs per block
  - Output: hap_info files
- **block-by-snp-sliding.py**:
  - Purpose: Generate haplotype block information files by fixed number of SNPs. Blocks overlap so that the beginning of each block is a set number of SNPs apart.
  - Input: map file, number of SNPs per block
  - Output: hap_info files
- **block-by-kb.py**:
  - Purpose: Generate haplotype block information files by fixed distance in kb. Blocks are defined consequtively and are adjacent to eachother.
  - Input: map file, size of block
  - Output: hap_info files
- **block-by-kb-sliding.py**:
  - Purpose: Generate haplotype block information files by fixed distance in kb. Blocks overlap so that the beginning of each block is a set number of SNPs apart.
  - Input: map file, size of block

  - Output: hap_info files
- **block-by-pos.py**:
  - Purpose: Generate haplotype block information files with a list of begin and end positions to provide flexibility to use functional and structural information to define haplotype blocks.
  - Input: list of blocks in the format "chr:beginPos-endPos", map file
  - Output: hap_info files

The **hap__info** files generated by these programs are editable or can be created by the user for custom definitions of haplotype blocks. An example of a **hap__info** file for a chromosome with 4 SNP blocks:

```
blk1    0 3
blk2    4 7
blk3    8 9
```

### Producing haplotype genotype files from defined blocks

**get-hap-geno.py** can be used to generate the haplotype files from user provided input files (imputed SNP genotypes and input files for the positions of haplotype blocks of each chromosome: hap and hap_info files).

- **get-hap-geno.py**:
  - Purpose: generate haplotype genotype files using block definitions
  - Input: hap_info files, haplotype files, (coding for missing alleles)
  - Output: hap_geno files

The haplotypes files are provided by the user from popular software (Beagle, FINDHAP, etc). Each haplotype block is described by two columns in the haplotype file, where each haplotype is treated as an 'allele' and each haplotype block is treated as a 'multi-allelic locus'. One individual per row.

Example of a haplotype genotype input file:

```
ID hap_1_1 hap_1_1 hap_1_2 hap_1_2 hap_1_3 hap_1_3
1 1 1 1 1 1 1
2 1 2 1 2 2 2
3 3 1 1 1 3 2
```

# Running EPIHAPMPI

## Creating GRMs

There are 3 orders of Genomic Relationship Matrices that can be made using this software:

1. Single order, conventional GRM

- SNP Additive
- SNP Dominance
- Haplotype Additive

2. Second order SNP epistasis GRM

- (AxA) Additive by Additive
- (AxD) Additive by Dominance
- (DxD) Dominance by Dominance

3. Third order SNP epistasis GRM

- (AxAxA) Additive by Additive by Additive
- (AxAxD) Additive by Additive by Dominance
- (AxDxD) Additive by Dominance by Dominance
- (DxDxD) Dominance by Dominance by Dominance

## Running full models

Pick the effects you want to include in your model.

For additive and dominance snp effects only:

```
# replace 4 with the number of MPI tasks you have available

mpirun -np 4 ./bin/grm.snp -o mygrms chr1.dat chr2.dat chr3.dat -a -d

# or if all chromosomes in one file:
mpirun -np 4 ./bin/grm.snp -o mygrms all.dat -a -d

# Output:
#  mygrms.g.A
#  mygrms.g.D
```

For all effects up to second order excluding dominance:

```
# replace 4 with the number of MPI tasks you have available

mpirun -np 4 ./bin/grm.snp -o mygrms chr1.dat chr2.dat chr3.dat --a --aa
# or if all chromosomes in one file:
```

```
mpirun -np 4 ./bin/grm.snp -o mygrms all.dat --a --aa

# Output:
#  mygrms.g.A
#  mygrms.g.AA
```

Or for all effects up to second order:

```
# replace 4 with the number of MPI tasks you have available

mpirun -np 4 ./bin/grm.snp -o out/mygrms chr1.dat chr2.dat chr3.dat
# or if all chromosomes in one file:
mpirun -np 4 ./bin/grm.snp -o out/mygrms all.dat

# Output:
#  out/mygrms.g.A
#  out/mygrms.g.D
#  out/mygrms.g.AA
#  out/mygrms.g.AD
#  out/mygrms.g.DD
```

For haplotype effects:

```
# replace 4 with the number of MPI tasks you have available

mpirun -np 4 ./bin/grm.hap chr1 chr2 chr3 -o mygrms
# or if all chromosomes in one file:
mpirun -np 4 ./bin/grm.hap -o out/mygrms all.hap

# Output:
#  mygrms.g.AH
```

For all effects up to third order:

```
# uses already created first and second order GRMs under prefix "out/mygrms"
./bin/grm.snp.3order --threads 8 --buffer 128M out/mygrms

# Output:
#  out/mygrms.g.AAA
#  out/mygrms.g.AAD
#  out/mygrms.g.ADD
#  out/mygrms.g.DDD
```

For just AAA (AxAxA) effects:

```
# uses already created first and second order GRMs under prefix "out/mygrms"
# to create additive by additive by additive GRM
./bin/grm.snp.3order --threads 16 --aaa out/mygrms
# or all possible third order GRMs
./bin/grm.snp.3order --threads 16 out/mygrms

# Output:
#  out/mygrms.g.AAA
```

Then run the perdiction model using the effects you want to include by specifying the variance componentes. Residual variance component is always included and must be specified:

```
# run the prediction using GRMs
# for snp additive, snp additive by addive, and haplotype additive
# in the reml model
# replace 4 with the number of MPI tasks you have available
mpirun -np 4 ./bin/reml \
    --input out/mygrms \
    --pheno 3,pheotype.txt \
    --va 2.435 \
    --vaa 3.12 \
    --vah 1.349E-01 \
    --out a.d.ah.model
```

## grm.snp

### Purpose

The `grm.snp` program calculates Genomic Relationship Matrices (GRMs) from SNP (Single Nucleotide Polymorphism) data. It can compute various types of GRMs, including:

- Additive GRM (`.g.A`)
- Dominance GRM (`.g.D`)
- Second-order epistatic GRMs:
  - Additive x Additive (`.g.AA`)
  - Additive x Dominance (`.g.AD`)
  - Dominance x Dominance (`.g.DD`)

These matrices are typically used in genetic analyses, such as variance component estimation with REML. The program is parallelized using MPI and ScaLAPACK.

### Options

- **Positional Arguments:** `<chr_file1> [<chr_file2> ...]`
  - One or more input files containing SNP data. These files are expected to be per-chromosome. The format is typically a text file where the first line is a header (e.g., marker IDs) and subsequent lines are individuals, with genotypes coded numerically (e.g., 0, 1, 2). The first column of each data line is usually an individual ID.
- `-n <block_size>`:
  - Specifies the block size for ScaLAPACK operations. This can affect performance and memory distribution. Default: `2`.
- `-r <num_proc_rows>`:
  - Overrides the automatically determined number of processor rows in the BLACS (Basic Linear Algebra Communication Subprograms) grid.
- `-c <num_proc_cols>`:
  - Overrides the automatically determined number of processor columns in the BLACS grid.
- `-o <output_name_prefix>`:
  - Prefix for the output GRM files. The program will append extensions like `.g.A`, `.g.D`, `.g.AA`, etc., to this prefix. Default: `grm`.
  - For example, if `-o mygrm` is used, output files will be `mygrm.g.A`, `mygrm.g.D`, and so on.

### Example Usage

To calculate GRMs from SNP data spread across three chromosome files (`chr1.snps`, `chr2.snps`, `chr3.snps`), using 8 MPI processes, and saving the output files with the prefix `dairy_cattle`:

```
# run with 8 mpi tasks (ranks) and 2 threads per task/rank
export MKL_NUM_THREADS=2
mpirun -np 8 ./bin/grm.snp -o dairy_cattle chr1.snps chr2.snps chr3.snps
```

This command will produce files such as: * `dairy_cattle.g.A` * `dairy_cattle.g.D` * `dairy_cattle.g.AA` * `dairy_cattle.g.AD` * `dairy_cattle.g.DD`

Each file is a binary matrix storing the respective genomic relationships.

## grm.snp.3order

### Purpose

The `grm.snp.3order` program calculates third-order epistatic Genomic Relationship Matrices (GRMs). It takes previously computed second-order GRMs (like additive `.g.A`, additive x additive `.g.AA`, dominance x dominance `.g.DD`) as input and computes their Hadamard (element-wise) products to form third-order interaction matrices.

The specific third-order GRMs it can generate are:

- Additive x Additive x Additive (`.g.AAA`), from `.g.A` and `.g.AA`.
- Additive x Dominance x Dominance (`.g.ADD`), from `.g.A` and `.g.DD`.
- Additive x Additive x Dominance (`.g.AAD`), from `.g.D` and `.g.AA`.
- Dominance x Dominance x Dominance (`.g.DDD`), from `.g.D` and `.g.DD`.

These matrices represent higher-order epistatic interactions and can be used in advanced genetic models. The program is optimized for large files using memory mapping or buffered I/O and can leverage multithreading.

### Options

- **Positional Argument:** `<base_name>`
  - This is the primary input. It's the base name from which input GRM file names are constructed and to which output GRM extensions are appended.
  - For example, if `<base_name>` is `myproject`, the program will look for input files like `myproject.g.A`, `myproject.g.AA`, `myproject.g.D`, `myproject.g.DD` (depending on the operation). Output files will be named like `myproject.g.AAA`.
- `-h, --help`:
  - Displays a help message summarizing usage and options, then exits.
- `-b, --buffer SIZE`:
  - Specifies the buffer size for I/O operations. This can be given with units like `M` (for Megabytes) or `G` (for Gigabytes), e.g., `256M` or `2G`.
  - The program uses this as a guideline, and the actual memory used will be roughly three times this size (for two input buffers and one output buffer).
  - Default: System-dependent, often based on L3 cache size and number of threads (e.g., 32M per L3 cache slice).
- `-t, --threads N`:
  - Sets the number of OpenMP threads to use for computation.
  - Default: Maximum number of threads available on the system.

#### Operation Filters:

If none of these options are specified, the program will attempt to compute all four third-order GRMs.

- `--aaa`:
  - Compute only the Additive x Additive x Additive (`.g.AAA`) matrix. Requires `<base_name>.g.A` and `<base_name>.g.AA` to exist.
- `--add`:

  - Compute only the Additive x Dominance x Dominance (`.g.ADD`) matrix. Requires `<base_name>.g.A` and `<base_name>.g.DD` to exist.
- `--aad`:
  - Compute only the Additive x Additive x Dominance (`.g.AAD`) matrix. Requires `<base_name>.g.D` and `<base_name>.g.AA` to exist.
- `--ddd`:
  - Compute only the Dominance x Dominance x Dominance (`.g.DDD`) matrix. Requires `<base_name>.g.D` and `<base_name>.g.DD` to exist.

### Example Usage

Suppose you have already computed GRMs for a project named `proj1` and have the following files: * `proj1.g.A` * `proj1.g.D` * `proj1.g.AA` * `proj1.g.DD`

To compute all third-order GRMs for `proj1` using 8 threads and a buffer of 512MB:

```
./bin/grm.snp.3order --threads 8 --buffer 512M proj1
```

This will produce: * `proj1.g.AAA` * `proj1.g.AAD` * `proj1.g.ADD` * `proj1.g.DDD`

To compute only the `.g.AAA` matrix:

```
./bin/grm.snp.3order --aaa proj1
```

This will produce `proj1.g.AAA`.

## grm.hap

### Purpose

The `grm.hap` program calculates a Genomic Relationship Matrix (GRM) from haplotype data. Specifically, it computes an Additive Haplotype GRM, which is stored with the extension `.g.AH`. This type of GRM can be useful for analyses considering haplotype effects rather than individual SNP effects. The program is parallelized using MPI and ScaLAPACK.

### Options

The command-line options for `grm.hap` are very similar to those for `grm.snp`.

- **Positional Arguments:** `<chr_file1> [<chr_file2> ...]`
  - One or more input files containing haplotype data. These files can be be per-chromosome or as one large file (often faster). The input format usually consists of phased haplotype pairs per individual. The first column is individual ID, followed by pairs of haplotype identifiers or sequences.
- `-n <block_size>`:
  - Specifies the block size for ScaLAPACK operations. This can impact performance and memory distribution. Default: `128`.
- `-r <num_proc_rows>`:
  - Overrides the automatically determined number of processor rows in the BLACS (Basic Linear Algebra Communication Subprograms) grid.
- `-c <num_proc_cols>`:
  - Overrides the automatically determined number of processor columns in the BLACS grid.
- `-o <output_name_prefix>`:
  - Prefix for the output GRM file. The program will append the extension `.g.AH` to this prefix. Default: `grm`.
  - For example, if `-o myhapgrm` is used, the output file will be `myhapgrm.g.AH`.

## Example Usage

To calculate an additive haplotype GRM from data in two files (`chr1.haps`, `chr2.haps`), using 4 MPI processes, and saving the output file with the prefix `wheat_haplotypes`:

```
mpirun -np 4 ./bin/grm.hap -o wheat_haplotypes chr1.haps chr2.haps
```

This command will produce the binary matrix file: * `wheat_haplotypes.g.AH`

# reml

## Purpose

The `reml` program (and its GPU-accelerated version `reml.gpu`) performs Restricted Maximum Likelihood (REML) analysis to estimate variance components for various genetic models. It reads genomic relationship matrices (GRMs) and phenotype data to partition the phenotypic variance into components attributed to different genetic (e.g., additive, dominance, epistatic, haplotype) and residual effects.

## Options

The program uses a common option parser. Options can be specified with long names (e.g., `--input`) or short names (e.g., `-i`).

- `--input <load_name>`: Base name for input GRM files (e.g., if `load_name` is `myproject`, it will look for `myproject.g.A`, `myproject.g.D`, etc.). Default: `grm`.
- `--pheno <col,file>`: Specifies the phenotype file and the column to be used as the trait. Format: `<column_number>,<filepath>` (e.g., `2,phenotypes.txt`).
- `--missing <value>`: Defines the numerical value used to represent missing phenotypes in the phenotype file. Default: `-9999111.0`.
- `--iterations <num>`: Maximum number of iterations for the REML algorithm. Default: `1`. (Note: Default of 1 might be for testing; real analyses usually require more).
- `--tol <value>`: Convergence tolerance for changes in variance components. The algorithm stops if changes are below this threshold. Default: `1.0e-8`.
- `--htol <value>`: Convergence tolerance for heritability. If negative, this check is skipped. Default: `1.0e-6`.
- `--ai_start <iteration_num>`: Specifies the iteration number at which to switch to the Average Information (AI) algorithm, which can be faster but less stable initially. If negative, AI is not used or used based on internal defaults. Default: `3`.
- `--out/-o <save_name>`: Prefix for output files generated by the program (e.g., iteration logs, results). Default: `out`.

**Initial Variance Component Estimates:** The following options allow providing starting values for variance components. If not specified, the program might use defaults (often 0 or small values) or estimate them.

- `--va   <value>`: Initial additive genetic variance (e.g., for GRM ending in `.g.A`).
- `--vd   <value>`: Initial dominance genetic variance (e.g., for `.g.D`).
- `--ve   <value>`: Initial residual variance [always required].
- `--vah  <value>`: Initial haplotype additive variance (e.g., for `.g.AH`).
- `--vaa  <value>`: Initial additive x additive epistatic variance (e.g., for `.g.AA`).
- `--vad  <value>`: Initial additive x dominance epistatic variance (e.g., for `.g.AD`).
- `--vdd  <value>`: Initial dominance x dominance epistatic variance (e.g., for `.g.DD`).
- `--vaaa <value>`: Initial additive x additive x additive epistatic variance (e.g., for `.g.AAA`).
- `--vaad <value>`: Initial additive x additive x dominance epistatic variance (e.g., for `.g.AAD`).
- `--vadd <value>`: Initial additive x dominance x dominance epistatic variance (e.g., for `.g.ADD`).
- `--vddd <value>`: Initial dominance x dominance x dominance epistatic variance (e.g., for `.g.DDD`).

**MPI/BLACS Grid Options:** * `-r <num_proc_rows>`: Overrides the automatically calculated number of processor rows in the BLACS grid. * `-c <num_proc_cols>`: Overrides the automatically calculated number of processor columns in the BLACS grid. * `--blocks <size>`: Sets the ScaLAPACK block size for matrix operations. This affects performance and memory distribution. Default: `128`.

## Example Usage

To run `reml` for a project named `cattle_study` with phenotype data in `cattle_pheno.txt` (using the 3rd column), estimate additive and dominance variance, and save outputs with the prefix `run01`:

```
mpirun -np 4 ./bin/reml --input cattle_study --pheno 3,cattle_pheno.txt --va 0.5 --vd 0.2 --ve 0.3 --ou
```

For GPU acceleration:

```
mpirun -np 4 ./bin/reml.gpu --input cattle_study --pheno 3,cattle_pheno.txt --va 0.5 --vd 0.2 --ve 0.3
```

Note: The exact GRM files used (e.g., `cattle_study.g.A`, `cattle_study.g.D`) depend on the variance components specified (e.g. `--va` implies use of `.g.A`).

# Calculations

This section details the mathematical calculations and algorithms used to create the Genomic Relationship Matrices (GRMs) from SNP and haplotype data. These GRMs quantify the genetic similarity between individuals. Then the iterative calculations performed in the REML (Restricted Maximum Likelihood) analysis are described.

## Matrix Dimensions

- $Nm$ = number of non-missing phenotypic records (these are used as the training population)
- $n$ = number of genotyped individuals in the population
- $mq$ = number of SNPs
- $y = N \times c$ (non-missing phenotypic records, $c = 1$ if no fixed effects or covariables), assumed to follow normal distribution
- $\mathbf{Z} = N \times n$ (incidence matrix)
- $\mathbf{X} = n \times c$ (model matrix)
- $\mathbf{G}_i = n \times n$ (genomic-relationship matrix for this variance)

## GRM Calculations

### SNP-Based Genomic Relationship Matrices

SNP-based GRMs are derived from single nucleotide polymorphism (SNP) markers. Let $n$ be the number of individuals and $m$ be the number of SNPs. The genotype for individual $i$ at SNP $j$ is denoted $X_{ij}$, coded as 0, 1, or 2, representing the count of a specific reference allele. Let $p_j$ be the frequency of this reference allele at SNP $j$.

#### Additive GRM ($\mathbf{G}_A$)

The additive GRM captures variance due to average effects of alleles. The elements $w_{A,ij}$ of an intermediate matrix $\mathbf{W}_A$ (size $n \times m$) are defined as:

$$w_{A,ij} = -(X_{ij} - 2p_j)$$

This results in the following codings for $w_{A,ij}$ based on genotype $X_{ij}$: - If $X_{ij} = 0$: $w_{A,ij} = 2p_j$ - If $X_{ij} = 1$: $w_{A,ij} = 2p_j - 1$ - If $X_{ij} = 2$: $w_{A,ij} = 2p_j - 2$

The additive GRM $G_A$ (an $n \times n$ matrix) is then:

$$\mathbf{G}_A = \mathbf{W}_A \mathbf{W}_A^T$$

An element $(\mathbf{G}_A)_{ik}$ is the sum of products of these coded variables across all SNPs:

$$(\mathbf{G}_A)_{ik} = \sum_{j=1}^{m} w_{A,ij} w_{A,kj} = \sum_{j=1}^{m} (-(X_{ij} - 2p_j))(-(X_{kj} - 2p_j)) = \sum_{j=1}^{m} (X_{ij} - 2p_j)(X_{kj} - 2p_j)$$

**Dominance GRM ($\mathbf{G}_D$)**

The dominance GRM accounts for variance due to interactions between two alleles at the same locus (dominance deviations). The elements $w_{D,ij}$ of an intermediate matrix $\mathbf{W}_D$ (size $n \times m$) are defined based on genotype $X_{ij}$ and allele frequency $p_j$ (frequency of the allele whose count is $X_{ij}$): - If $X_{ij} = 0$ (homozygous for the non-reference allele): $w_{D,ij} = -2p_j^2$ - If $X_{ij} = 1$ (heterozygous): $w_{D,ij} = 2p_j(1 - p_j)$ - If $X_{ij} = 2$ (homozygous for the reference allele): $w_{D,ij} = -2(1 - p_j)^2$

The dominance GRM $\mathbf{G}_D$ (an $n \times n$ matrix) is then:

$$\mathbf{G}_D = \mathbf{W}_D \mathbf{W}_D^T$$

An element $(\mathbf{G}_D)_{ik}$ is $\sum_{j=1}^{m} w_{D,ij} w_{D,kj}$.

**Second-Order SNP GRMs (Epistasis GRMs)**

These matrices capture epistatic interactions involving additive and dominance effects. They are calculated using element-wise (Hadamard) products of the $\mathbf{G}_A$ and $\mathbf{G}_D$ matrices. Let $\otimes$ denote the Hadamard product.

- **Additive by Additive ($\mathbf{G}_{AA}$):**

$$\mathbf{G}_{AA} = \mathbf{G}_A \otimes \mathbf{G}_A \quad ((\mathbf{G}_{AA})_{ik} = (\mathbf{G}_A)_{ik} \times (\mathbf{G}_A)_{ik})$$

- **Dominance by Dominance ($\mathbf{G}_{DD}$):**

$$\mathbf{G}_{DD} = \mathbf{G}_D \otimes \mathbf{G}_D \quad ((\mathbf{G}_{DD})_{ik} = (\mathbf{G}_D)_{ik} \times (\mathbf{G}_D)_{ik})$$

- **Additive by Dominance ($\mathbf{G}_{AD}$):**

$$\mathbf{G}_{AD} = \mathbf{G}_A \otimes \mathbf{G}_D \quad ((\mathbf{G}_{AD})_{ik} = (\mathbf{G}_A)_{ik} \times (\mathbf{G}_D)_{ik})$$

**Third-Order SNP GRMs (Epistasis GRMs)**

- **Additive by Additive by Additive ($\mathbf{G}_{AAA}$):**

$$\mathbf{G}_{AAA} = \mathbf{G}_{AA} \otimes \mathbf{G}_A \quad ((\mathbf{G}_{AAA})_{ik} = (\mathbf{G}_{AA})_{ik} \times (\mathbf{G}_A)_{ik})$$

- **Additive by Additive by Dominance ($\mathbf{G}_{AAD}$):**

$$\mathbf{G}_{AAD} = \mathbf{G}_{AA} \otimes \mathbf{G}_D \quad ((\mathbf{G}_{AAD})_{ik} = (\mathbf{G}_{AA})_{ik} \times (\mathbf{G}_D)_{ik})$$

- **Additive by Dominance by Dominance ($\mathbf{G}_{ADD}$):**

$$\mathbf{G}_{ADD} = \mathbf{G}_{DD} \otimes \mathbf{G}_A \quad ((\mathbf{G}_{ADD})_{ik} = (\mathbf{G}_{DD})_{ik} \times (\mathbf{G}_A)_{ik})$$

- **Additive by Dominance by Dominance ($\mathbf{G}_{DDD}$):**

$$\mathbf{G}_{DDD} = \mathbf{G}_{DD} \otimes \mathbf{G}_D \quad ((\mathbf{G}_{DDD})_{ik} = (\mathbf{G}_{DD})_{ik} \times (\mathbf{G}_D)_{ik})$$

## Haplotype-based Additive Genomic Relationship Matrix

This GRM utilizes phased haplotype information, where unique haplotype sequences within defined genomic blocks are treated as distinct alleles. For more information refer to the (Da *et al.*, 2022).

### Input Data

The primary input for this calculation is a "hap_geno" file which can be generated as described in Haplotype Files section using "scripts/get.hap.geno.parallel.py"[TODO:LINK]. For each individual in the population, and for each defined haplotype block in the genome, the file specifies a pair of unique integer identifiers for the haplotype sequences within the block. These integers are unique labels assigned to distinct haplotype sequences that occur within that specific block across the population.

### Methodology

The $\mathbf{G}_{AH}$ matrix is constructed by processing the genome block by block. Let $n$ be the number of individuals.

Then for each haplotype block $b$ in the genome:

**Calculate block frequencies** All unique haplotype identifiers within block $b$ across all $n$ individuals are identified. For each unique haplotype identifier $h$ in block $b$, let $count_{bh}$ be its total count considering both haplotype copies from each of the $n$ individuals. The frequency of haplotype $h$ in block $b$, denoted $p_{bh}$, is then calculated as:

$$p_{bh} = \frac{count_{bh}}{2n}$$

**Reference-Minor Coding for Each Block** The most frequent haplotype identifier in block $b$ is designated as the reference (or major) haplotype for that block. This reference haplotype is denoted as $M_b$ and corresponds to `hap_max_key` in the code.

For every unique haplotype identifier $k'$ that is present in block $b$ but is *not* the reference haplotype $M_b$ (i.e., $k' \neq M_b$):

- Let $p_{bk'}$ be the frequency of this non-reference haplotype $k'$ in block $b$
- For each individual $i$, determine $X_{ibk'}$, which is the count of haplotype $k'$ that individual $i$ possesses in block $b$. This count can be 0, 1, or 2.
- $w_{ibk'}$, is then created for individual $i$ corresponding to this specific non-reference haplotype $k'$ in block $b$. The formula for this variable is:

  $$w_{ibk'} = -(X_{ibk'} - 2p_{bk'})$$

  This formula expands to the following values based on the count $X_{ibk'}$:
  - If $X_{ibk'} = 0$: $w_{ibk'} = 2p_{bk'}$
  - If $X_{ibk'} = 1$: $w_{ibk'} = 2p_{bk'} - 1$
  - If $X_{ibk'} = 2$: $w_{ibk'} = 2p_{bk'} - 2$
- If an individual's block allele information for block $b$ is 0 it is treated as missing, the corresponding $w_{ibk'}$ stay 0, meaning they do not contribute to relationship calculations for that specific block allele.
  - If individual $i$ has zero copies of $k'$ in block $b$ ($X_{ibk'} = 0$): $w_{ibk'} = 2p_{bk'}$
  - If individual $i$ has one copy of $k'$ in block $b$ ($X_{ibk'} = 1$): $w_{ibk'} = 2p_{bk'} - 1$
  - If individual $i$ has two copies of $k'$ in block $b$ ($X_{ibk'} = 2$): $w_{ibk'} = 2p_{bk'} - 2$

**Assembly of the Global ($\mathbf{W}_{AH}$)** The coded variables $w_{ibk'}$ are assembled into a global predictor matrix $\mathbf{W}_{AH}$. This matrix has $n$ rows (one for each individual). The columns of $\mathbf{W}_{AH}$ are formed by concatenating the $w_{ibk'}$ vectors for all non-reference haplotypes $k'$ from all haplotype blocks $b$. If there are $B$ blocks in total, and block $b$ contains $u_b$ unique haplotype identifiers (thus $u_b - 1$ non-reference haplotypes), the total number of columns in $\mathbf{W}_{AH}$ will be $\sum_{b=1}^{B}(u_b - 1)$.

**Calculation of the Haplotype GRM ($\mathbf{G}_{AH}$)** The $n \times n$ matrix $\mathbf{G}_{AH}$ is calculated as the product of $\mathbf{W}_{AH}$ and its transpose:

$$\mathbf{G}_{AH} = \mathbf{W}_{AH}\mathbf{W}_{AH}^T$$

Each element $(\mathbf{G}_{AH})_{il}$ in this matrix, representing the relationship between individual $i$ and individual $l$, is computed as:

$$(\mathbf{G}_{AH})_{il} = \sum_{b=1}^{B} \sum_{k' \neq M_b} w_{ibk'} w_{lbk'}$$

Substituting the expression for $w_{ibk'}$:

$$(\mathbf{G}_{AH})_{il} = \sum_{b=1}^{B} \sum_{k' \neq M_b} (-(X_{ibk'} - 2p_{bk'}))(-(X_{lbk'} - 2p_{bk'}))$$

$$(\mathbf{G}_{AH})_{il} = \sum_{b=1}^{B} \sum_{k' \neq M_b} (X_{ibk'} - 2p_{bk'})(X_{lbk'} - 2p_{bk'})$$

## Normalization

The raw $\mathbf{G}_f$ matrix for all of the above GRMs undergoes a normalization step before being saved: The matrix $\mathbf{G}_f$ is then divided by the mean of its diagonal elements:

$$\mathbf{G}_{f,norm} = \frac{\mathbf{G}_f}{tr(\mathbf{G}_f)/N}$$

This normalization ensures that the average of the diagonal elements of the final $\mathbf{G}_{f,norm}$ matrix is 1 and suitable for variance component estimation.

## Algorithmic and Implementation Choices

The computation of these GRMs involves handling potentially a large number of SNPs or unique haplotypes per block and many blocks across the genome. Several key strategies for efficiency and scalability with large genomic datasets:

- **Parallel Processing (MPI)**: The Message Passing Interface (MPI) is used to distribute the computational workload across multiple processors or compute nodes. This allows for parallel execution of data reading, matrix construction, and calculations.
- **ScaLAPACK Library**: For high-performance distributed linear algebra operations, the software utilizes ScaLAPACK (Scalable Linear Algebra PACKage) routines, often via the Intel MKL implementation.
- Each MPI rank builds its own chunk of $\mathbf{W}_i$ matrices (only reads and assembles the corresponding columns for the set of SNPs or haplotype blocks assigned to it). The local chunks are then aggregated into a global block 2d block-cyclic distributed matrix for subsequent calculations so that no single process ever holds all $n \times N$ or $n \times n$ resulting matrix. As a result, both memory and compute scale well with the number of mpi tasks (ranks).
- **Iterative Accumulation over Genomic Segments**: GRMs are typically constructed by processing the genome in segments (e.g., per chromosome or large blocks of markers). Intermediate $\mathbf{W}_{segment}\mathbf{W}^T_{segment}$ matrices are computed for each segment and summed up: $\mathbf{G} = \sum_{segment} \mathbf{W}_{segment}\mathbf{W}^T_{segment}$. This approach manages memory effectively and allows for parallel processing.
- **Hadamard Products for Interaction GRMs**: Second-order interaction GRMs ($\mathbf{G}_{AA}, \mathbf{G}_{AD}, \mathbf{G}_{DD}$) are computed using element-wise (Hadamard) products of the first-order GRMs using local matrix chunks in parallel. This is computationally much more efficient than explicitly forming large interaction $\mathbf{W}$ matrices, takes very little memory and can be done on a regular computer using all of the available cores for parallelization.
- **Missing Data Handling**: Genotypes or haplotypes marked as missing (e.g., coded as 0 for haplotypes or a value other than 0, 1, 2 for SNPs in the raw input) are typically assigned a score of 0 in the intermediate $\mathbf{W}$ matrices. This means they do not contribute to the relationship scores for those specific alleles or SNPs, effectively being an imputation to the mean before centering.

# REML

Iterative calculations using REML (Restricted Maximum Likelihood) CE (Conditional Expectation) method to estimate variance components and calculate heritabilities.

## Initialization

Before the iterative process begins, several matrices and variables are initialized:

1. **Variance Components ($\sigma^2$)**: Initial estimates for variance components are provided or set to default values. These include genetic variances (e.g., $\sigma^2_{a,d,aa,ah,...}$) and the residual variance ($\sigma^2_e$).
2. **Genomic Relationship Matrix (G)**: This matrix quantifies the genetic relatedness between individuals. It's computed from marker and haplotype data (GRMs).
3. **Phenotype Vector (y)**: A vector containing the phenotypic observations for the trait of interest.
4. **Design Matrix for Fixed Effects (X)**: This matrix relates phenotypic observations to fixed effects (e.g., herd, year, sex, season, etc). It includes a column of ones for the overall mean.
5. **Incidence Matrix for Random Effects (Z)**: This matrix relates phenotypic observations to the random genetic effects of individuals. If phenotypes are directly on individuals in the G matrix (no missing data), Z can be an identity matrix or a submatrix of an identity matrix if not all individuals in G have phenotypes. If there are multiple random effects, there will be a corresponding Z_i for each G_i.
6. **Phenotypic Variance-Covariance Matrix (V)**: This is constructed based on the initial variance components:

   $$\mathbf{V} = (\sum_i \sigma_i^2 * \mathbf{Z}_i * \mathbf{G}_i * \mathbf{Z}_i^T) + \sigma_e^2 * \mathbf{I}$$

   where:

   - $\sigma_i^2$ is the variance for the i-th random effect.
   - $\sigma_e^2$ is the residual variance.
   - $\mathbf{Z}_i$ is the incidence matrix for the i-th random effect.
   - $\mathbf{G}_i$ is the Genomic Relationship Matrix for the i-th random effect.
   - $\mathbf{I}$ is the identity matrix.
7. **P Matrix**: This matrix is central to REML equations and is derived from V and X:

   $$\mathbf{P} = \mathbf{V}^{-1} - \mathbf{V}^{-1}\mathbf{X}(\mathbf{X}^T\mathbf{V}^{-1}\mathbf{X})^{-}\mathbf{X}^T\mathbf{V}^{-1}$$

   This matrix projects the data onto the space orthogonal to the fixed effects.

## Expectation-Maximization EM-REML Iteration

The EM-REML algorithm is an iterative approach to estimate variance components. Each iteration consists of an Expectation (E) step and a Maximization (M) step.

The P matrix is recalculated in each iteration using the updated variance components from the previous iteration:

$$\mathbf{V}_{(t)} = \sum_i (\sigma^2_{i,(t)} * \mathbf{Z}_i\mathbf{G}_i\mathbf{Z}_i^T) + \sigma^2_{e,(t)} * \mathbf{I}$$

$$\mathbf{P}_{(t)} = \mathbf{V}^{-1}_{(t)} - \mathbf{V}^{-1}_{(t)}\mathbf{X}(\mathbf{X}^T\mathbf{V}^{-1}_{(t)}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{V}^{-1}_{(t)}$$

The update equations for the variance components ($\sigma^2$) are as follows:

For a genetic variance component $\sigma_j^2$ (associated with $\mathbf{S}_j = \mathbf{Z}_j\mathbf{G}_j\mathbf{Z}_j^T$):

$$\sigma^2_{j,(t+1)} = (\sigma^2_{j,(t)} * y^T \mathbf{P}_{(t)} \mathbf{S}_j \mathbf{P}_{(t)} y)/tr(\mathbf{P}_{(t)} \mathbf{S}_j)$$

For the residual variance component $\sigma^2_e$ (associated $\mathbf{S}_e = \mathbf{I}$):

$$\sigma^2_{e,(t+1)} = (\sigma^2_{e,(t)} * y^T \mathbf{P}_{(t)} \mathbf{P}_{(t)} y)/tr(\mathbf{P}_{(t)})$$

Where: * '$\sigma^2_{k,(t+1)}$' is the new estimate for the variance component `k` at iteration `t+1`. * '$\sigma^2_{k,(t)}$' is the estimate from the previous iteration `t`. * '$y$' is the vector of phenotypes. * '$\mathbf{P}_t$' is the $\mathbf{P}$ matrix calculated using variance components from iteration `t`. * '$\mathbf{S}_j = \mathbf{Z}_j \mathbf{G}_j \mathbf{Z}_j^T$' is the matrix for the j-th genetic component. * '$\mathbf{I}$' is the Identity matrix.

The process repeats until convergence criteria are met: heritability or variance threshold.

## Average Information AI-REML Iteration

The AI-REML algorithm often converges faster than EM-REML. It uses an approximation of the Hessian of the REML log-likelihood, known as the Average Information (AI) matrix.

The update rule for the vector of variance components $\theta = [\sigma^2_1, \sigma^2_2, ..., \sigma^2_k]^T$ at iteration '$t$' is:

$$\theta_{(t+1)} = \theta_{(t)} + \Delta\theta_{(t)}$$

Where $\Delta\theta_{(t)}$ is the solution to the system:

$$AI_{(t)} * \Delta\theta_{(t)} = s_{(t)}$$

Where:

- '$\theta_{(t)}$' is the vector of current variance component estimates.
- '$AI_{(t)}$' is the Average Information matrix evaluated at $\theta_{(t)}$.
- '$s_{(t)}$' is the vector of first derivatives of the REML log-likelihood (score vector) with respect to each variance component, evaluated at $\theta_{(t)}$.

**1. Score Vector (s)**: The $k^{th}$ element of the score vector, $s_k$, corresponding to variance component $\sigma^2_k$ (associated with $\mathbf{S}_k$, where $\mathbf{S}_k = \mathbf{Z}_k \mathbf{G}_k \mathbf{Z}_k^T$ for a genetic component or $\mathbf{S}_e = \mathbf{I}$ for residual), is calculated as:

$$s_k = \frac{1}{2} * (y^T \mathbf{P}_{(t)} \mathbf{S}_k \mathbf{P}_{(t)} y - tr(\mathbf{P}_{(t)} \mathbf{S}_k))$$

In `reml.cpp`, this corresponds to `delta_ai[current_name] = (dtmp - trace)/2.0;` where `dtmp` is the quadratic form '$y^T \mathbf{PSP} y$' and `trace` is '$tr(\mathbf{PS})$' for the specific component.

**2. Average Information (AI) Matrix**: The element $AI_{(i,j)}$ corresponding to variance components $\sigma^2_i$ and $\sigma^2_j$ (with $\mathbf{S}_i$ and $\mathbf{S}_j$ respectively), is given by:

$$AI_{ij} = \frac{1}{2} * tr(\mathbf{P}_{(t)} \mathbf{S}_i \mathbf{P}_{(t)} \mathbf{S}_j)$$

(Note: We calculate AI elements as '$\frac{1}{2} * y^T \mathbf{P} \mathbf{S}_i \mathbf{P} \mathbf{S}_j \mathbf{P} y$'. This is known as the *Observed Information* matrix. For simplicity in this manual, we refer to it as AI,.)

**3. Update Step**: First, the P matrix is formed using current variance estimates:

$$\mathbf{V}_{(t)} = \sum (\sigma^2_{i,(t)} * \mathbf{Z}_i \mathbf{G}_i \mathbf{Z}_i^T) + \sigma^2_{e,(t)} * \mathbf{I}$$

$$\mathbf{P}_{(t)} = \mathbf{V}_{(t)}^{-1} - \mathbf{V}_{(t)}^{-1}\mathbf{X}(\mathbf{X}^T\mathbf{V}_{(t)}^{-1}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{V}_{(t)}^{-1}$$

Then, the system $AI_{(t)} * \Delta\theta_{(t)} = s_{(t)}$ is solved for $\Delta\theta_{(t)}$. In the code, `dppsv` is used to solve this, storing the result $\Delta\theta_{(t)}$ in `dltmp`.

The new variance components are then proposed as:

$$\theta_{(t+1),proposed} = \theta_{(t)} + \Delta\theta_{(t)}$$

The code includes safeguards to the minimum eigenvalue of the AI matrix for positive definiteness and step quality) to ensure that the proposed update is sensible (e.g., variance estimates remain non-negative). If the AI step is not deemed appropriate (e.g., if the AI matrix is not positive definite or the update leads to out-of-bounds estimates), the algorithm might revert to an EM-REML step for that iteration.

If the AI step is accepted, $\theta_{(t+1)} = \theta_{(t+1),proposed}$. These new estimates then become the current estimates for the next iteration.

## Convergence Criteria

The iterative process (either EM-REML or AI-REML) continues until one or more convergence criteria are met, or the maximum number of iterations is reached. The common criteria used are:

1. **Change in Variance Estimates**: The algorithm monitors the absolute difference in variance estimates between successive iterations. Let $\sigma^2_{k,(t)}$ be the estimate of the $k^{th}$ variance component at iteration '$t$'. The change is

   $$\Delta\sigma^2_k = |\sigma^2_{k,(t+1)} - \sigma^2_{k,(t)}|$$

   If the maximum of these changes ($max(\Delta\sigma^2_k)$ for all $k$) falls below a predefined tolerance, the algorithm will stop.

2. **Change in Heritabilities**: Similarly, the change in heritability estimates can be used as a convergence criterion. Heritability ($h^2$) for a component '$j$' is typically calculated as $h^2_j = \sigma^2_j/\sigma^2_{Total}$, where $\sigma^2_{Total}$ is the sum of all variance components (excluding residual variance for some definitions, or including it for others. The change is $\Delta h^2_k = |h^2_{k,(t+1)} - h^2_{k,(t)}|$. If the maximum of these changes ($max(\Delta h^2_k)$) falls below a user defined heritability tolerance, the algorithm will stop.

3. **Maximum Number of Iterations**: A hard limit on the number of iterations is set by the user. If convergence is not achieved within this limit, the algorithm stops, and the current estimates are reported.

The **REML** iterations will terminate if: * The maximum absolute difference between variance estimates from the current and previous iteration is less than or equal to the user-defined variance tolerance. * OR, the maximum absolute difference between heritability estimates from the current and previous iteration is less than or equal to the user-defined heritability tolerance. * OR, the number of iterations reaches user defined max iterations.

# Pipeline scripts

## Purpose

Helper programs to prepare data for analysis and assist in analyzing the results.

**Note:** The actual option names and usage may differ. Please consult the script's `--help` output for precise information.

## block-by-snp-with-map.py

### Purpose

make x snp block definition file using map file as input

### Options

To view the specific command-line options and their descriptions for `block-by-snp-with-map.py`, please execute the script with the `--help` argument:

```
python scripts/block-by-snp-with-map.py --help
```

The script accepts the following command-line options:

- `-m/--map MAP`: path to map file [required] (Default: map.txt)
- `--snp SNP`: block size in snp [required] (Default: 4)
- `-o/--output OUTPUT`: output folder (Default: hap_info)
- `-V/--verbose`: verbose output (Default: False)

### Example Usage

Below is an example command showing how `block-by-snp-with-map.py` could be used:

```
python scripts/block-by-snp-with-map.py --map mymap.txt \
        --snp 5 \
        --output my_hap_blocks \
        --verbose
```

This command tells the script to use `mymap.txt` as the map file, create blocks of 5 SNPs, save the output to a folder named `my_hap_blocks`, and provide verbose output during the process.

**Note:** The actual option names and usage may differ. Please consult the script's `--help` output for precise information.

## block-by-snp.py

### Purpose

generate x snp block definition file

### Options

To get the precise command-line options and their descriptions, you should run the script with the `--help` argument:

```
python scripts/block-by-snp.py --help
```

The script accepts the following command-line options:

- `-i/--input INPUT [INPUT ...]`: path(s) to hap chr file(s) [required]
- `--snp SNP`: block size in snp (Default: 4)
- `-o/--output OUTPUT`: output folder (Default: hap_info)
- `--nosort`: Disable sorting of input files by number in filename. (Sorting is enabled by default)
- `--noheader`: Set if there is no header in the input files. (Default: False, assumes header is present)
- `-V/--verbose`: verbose output (Default: False)

### Example Usage

A command to run `block-by-snp.py` might look like this:

```
python scripts/block-by-snp.py --input chr1.hap chr2.hap \
        --snp 10 \
        --output custom_name \
        --noheader \
        --verbose
```

This command tells the script to:

- Process `chr1.hap` and `chr2.hap` as input files.
- Define blocks of 10 SNPs.
- Save the output to a folder named `custom_blocks`.
- Assume the input files do not have a header row.
- Provide verbose output during processing.

**Note:** The actual option names and usage may differ. Please consult the script's `--help` output for precise information.

## block-by-sliding-snp.py

### Purpose

generate block definition files based number of SNPs per, the start of each block slides by user defined number of SNPs

### Options

To get the precise command-line options and their descriptions, you should run the script with the `--help` argument:

```
python scripts/block-by-sliding-snp.py --help
```

The script accepts the following command-line options:

- `-s/--size SIZE`: size of each block in number of SNPs (Default: 2)
- `-t/--step STEP`: number of SNPs between block starting points (Default: 1)
- `-m/--map MAP`: path to map file [required]. Format: 'SNPID Chr Pos' with header.
- `--noheader`: use if map file has no header (Default: False)
- `-o/--output OUTPUT`: output folder name (Default: hap_info)
- `-V/--verbose`: verbose output (Default: False)

### Example Usage

A command to run `block-by-sliding-snp.py` might look like this:

```
python scripts/block-by-sliding-snp.py --map mygenome.map \
                --size 50 \
                --step 2 \
                --output sliding_snp_blocks_50_25 \
                --verbose
```

This command instructs the script to: - Use `mygenome.map` as the input map file (SNPID, Chromosome, Position). - Define blocks that each span 50 SNPs. - Slide the starting point of each subsequent block by 2 SNPs. - Save the output to a folder named `sliding_snp_blocks_50_2`. - Provide verbose output during the process.

**Note:** The actual option names and usage may differ. Please consult the script's `--help` output for precise information.

## block-by-kb.py

### Purpose

generate block definition files based on a list of positions for each block

### Options

To get an accurate list of options and their descriptions, please run the script with the `--help` flag:

```
python scripts/block-by-kb.py --help
```

The script accepts the following command-line options.

- `-m/--map MAP`: path to map file [required]. Format: 'SNPID Chr Pos' with header.
- `-s/--size SIZE`: size of each block in kbp (Default: 500)
- `-o/--output OUTPUT`: output file name (Default: hap_info)
- `-V/--verbose`: verbose output (Default: False)

### Example Usage

An example command to run `block-by-kb.py`:

```
python scripts/block-by-kb.py --map mygenome.map --size 250 --output kb_blocks_250kbp --verbose
```

This command instructs the script to:

- Use `mygenome.map` as the input map file (SNPID, Chromosome, Position).
- Define blocks that each span approximately 250 kilobase pairs (kbp).
- Save the output to a folder named `kb_blocks_250kbp`.
- Provide verbose output during the process.

**Note:** The actual option names and usage may differ. Please consult the script's `--help` output for precise information.

## block-by-sliding-kb.py

### Purpose

generate block definition files based number of kbs, the minimum distance per block the start of each block slides by user defined number of snps

### Options

For a detailed and accurate list of command-line options and their usage, run the script with the `--help` flag:

```
python scripts/block-by-sliding-kb.py --help
```

The script accepts the following command-line options. Arguments marked with an asterisk (*) are inherited from a common utility parser.

- `-m/--map MAP` (*): path to map file [required]. Format: 'SNPID Chr Pos' with header.
- `-s/--size SIZE`: minimum size of each block in kbs (Default: 100)
- `-t/--step STEP`: number of snps between block starting points (Default: 1)
- `--noheader`: use if map file has no header (Default: False)
- `-o/--output OUTPUT` (*): output file name (Default: hap_info)
- `-V/--verbose` (*): verbose output (Default: False)

### Example Usage

Here's an example of how `block-by-sliding-kb.py` might be run:

```
python scripts/block-by-sliding-kb.py \
    --map mygenome.map \
    --size 150 \
    --step 10 \
    --output sliding_kb_blocks_150k_10s \
    --noheader \
    --verbose
```

This command instructs the script to:

- Use `mygenome.map` as the input map file (SNPID, Chromosome, Position).
- Define blocks that span a minimum of 150 kilobases (kbs).
- Slide the starting point of each subsequent block by 10 SNPs.
- Assume the input map file does not have a header row.
- Save the output to a folder named `sliding_kb_blocks_150k_10s`.
- Provide verbose output during the process.

**Note:** The actual option names and usage may differ. Please consult the script's `--help` output for precise information.

## block-by-pos.py

### Purpose

generate block definition files based on a list of positions for each block

### Options

To view the specific command-line options and their descriptions for `block-by-pos.py`, please execute the script with the `--help` argument:

```
python scripts/block-by-pos.py --help
```

The script accepts the following command-line options:

- `-p/--pos POS`: path to block position file [required]. Format: 'chr:begin_pos:end_pos' without header.
- `-m/--map MAP`: path to map file [required]. Format: 'SNPID Chr Pos' with header.
- `-o/--output OUTPUT`: output file name (Default: hap_info)
- `-V/--verbose`: verbose output (Default: False)

### Example Usage

An example command to run `block-by-pos.py`:

```
python scripts/block-by-pos.py \
    --pos myblocks.txt \
    --map mygenome.map \
    --output position_defined_blocks \
    --verbose
```

This command instructs the script to:

- Use `myblocks.txt` as the input file defining block positions (format: 'chr:begin_pos:end_pos').
- Use `mygenome.map` as the input map file (SNPID, Chromosome, Position) to identify SNPs within those positions.
- Save the output to a folder named `position_defined_blocks`.
- Provide verbose output during the process.

**Note:** The actual option names and usage may differ. Please consult the script's `--help` output for precise information.

## create.kfold.pheno.py

### Purpose

create k files with N/k random rows set to missing value

### Options

To view the specific command-line options and their descriptions for `create.kfold.pheno.py`, please execute the script with the `--help` argument:

```
python scripts/create-kfold-pheno.py --help
```

The script accepts the following command-line options:

- `pheno`: Path to the input phenotype/data file.
- `-n/--column-name NAME`: Name of the column to process (used if file has a header).
- `-c/--column N`: 0-based index of the column to process for creating validation sets. (Default: 2, i.e., the 3rd column).
- `-k/--kfolds K`: Number of k-folds to generate. (Default: 10)
- `-p/--percent P`: Percent of the population to set as missing for validation in each fold. (Default: 10)
- `-m/--missing-value VAL`: Symbol to use for missing values. (Default: -9999111)
- `--noheader`: Set if the input file has no header row. (Default: False, assumes header is present)
- `-V/--verbose`: Verbose output. (Default: False)

### Example Usage

Below is an example command showing how `create.kfold.pheno.py` could be used:

```
python scripts/create-kfold-pheno.py \
    my_phenotypes.txt \
    --column-name age \
    -k 5 \
    -p 20 \
    -m NA \
    --verbose
```

This command instructs the script to:

- Process the input file `my_phenotypes.txt`.
- Target the column named `age` for creating validation sets (assumes a header is present).
- Generate 5 k-fold files.
- In each file, set 20% of the individuals in the target column to the missing value `NA`.

The output files will be named by inserting the fold number before the last extension, e.g., `my_phenotypes.1.txt`, `my_phenotypes.2.txt`, . . . , `my_phenotypes.5.txt`.

## get.corr.graph.py

### Purpose

The `get.corr.graph.py` script calculates correlations of predicted GBLUP to phenotype of k-fold cross-validation runs and optionally generates a graph across the runs.

### Options

To view the specific command-line options and their descriptions for `get.corr.graph.py`, please execute the script with the `--help` argument:

```
python scripts/get.corr.graph.py --help
```

Options:

- `-h, --help`: show this help message and exit
- `--gblup GBLUP [GBLUP ...]`
  - List of GBLUP file(s) (space-separated).
- `--pheno PHENO`
  - Path to the phenotype file.
- `--phenotype_col PHENOTYPE_COL`
  - Phenotype column name in the phenotype file.
- `--missing_value MISSING_VALUE`
  - Value denoting missing phenotypes.
- `--n_cores N_CORES`
  - Number of parallel processes to use (default: system's CPU count).
- `--gblup_cols GBLUP_COLS [GBLUP_COLS ...]`
  - The GBLUP columns to plot (space-separated). Use GBLUP_A, GBLUP_G, etc. or 'all' to plot all columns.
- `--save_plot SAVE_PLOT`
  - File to save the plot. If not specified, the plot will be shown.
- `--plot_set {T,V}`
  - Choose which set to plot: 'T' for training, 'V' for validation.
- `--save_csv SAVE_CSV`

– File to save the correlation results as a CSV.

### Example Usage

Below is a command showing how `get.corr.graph.py` could be used:

```
python scripts/get.corr.graph.py \
    --gblup ./run_*/out/proj1_gblup.csv \
    --pheno path_to_phentype/traits.txt \
    --missing_value -9999 \
    --pheno_col 3
```

Or to save a plot of the correlations:

```
python scripts/get.corr.graph.py \
    --gblup k1_gblup.csv k2_gblup.csv k3_gblup.csv \
    --pheno path_to_phentype/traits.txt \
    --pheno_col 3 \
    --gblup_cols GBLUP_A GBLUP_AA GBLUP_AH GBLUP_G \
    --save_plot correlation_heatmap.png
```

**Note:** The actual option names and usage may differ. Please consult the script's `--help` output for precise information.

## count-snps.py

### Purpose

counts the number of SNPs in each chromosome file

### Options

To view the specific command-line options and their descriptions for `count-snps.py`, please execute the script with the `--help` argument:

```
python scripts/count-snps.py --help
```

The script accepts the following command-line options:

- `-i/--input INPUT [INPUT ...]`: path to input files [required]
- `-o/--output OUTPUT`: output file name (Default: count-snps.log)
- `--prefix PREFIX`: prefix in front of output lines (default: geno_snp)
- `--nosort`: Disable sorting of input files by number in filename. (Sorting is enabled by default)

### Example Usage

Below is an example command showing how `count-snps.py` could be used:

```
python scripts/count-snps.py \
    --input chr1.dat chr2.dat chr10.dat \
    --prefix snp_geno \
    --output snp_counts.log
```

This command instructs the script to:

- Process the input files `chr1.dat`, `chr2.dat`, and `chr10.dat`.
- Count the number of snps in each file. The script determines this by reading the first line, splitting it by whitespace, and calculating `(number of columns - 1) / 2`.

- Print the counts to standard output in the format `geno_snp <count> <filename>`.
- Input files will be sorted by default based on numbers in their filenames before processing (e.g., `chr1.dat`, `chr2.dat`, `chr10.dat`). Use `--nosort` to process in the order they are provided on the command line.

**Note:** The actual option names and usage may differ. Please consult the script's `--help` output for precise information.

## count-haps.py

### Purpose

counts the number of haplotypes in each chromosome file

### Options

To view the specific command-line options and their descriptions for `count-haps.py`, please execute the script with the `--help` argument:

```
python scripts/count-haps.py --help
```

The script accepts the following command-line options:

- `-i/--input INPUT [INPUT ...]`: path to input files [required]
- `-o/--output OUTPUT`: output file name (Default: count-snps.log)
- `--nosort`: Disable sorting of input files by number in filename. (Sorting is enabled by default)

### Example Usage

Below is an example command showing how `count-haps.py` could be used:

```
python scripts/count-haps.py \
    --input chr1.haps chr2.haps chr10.haps \
    --output haplotype_counts.log
```

This command instructs the script to:

- Process the input files `chr1.haps`, `chr2.haps`, and `chr10.haps`.
- Count the number of haplotypes in each file. The script determines this by reading the first line, splitting it by whitespace, and calculating `(number of columns - 1) / 2`.
- Print the counts to standard output in the format `geno_hap <count> <filename>`.
- Input files will be sorted by default based on numbers in their filenames before processing (e.g., `chr1.haps`, `chr2.haps`, `chr10.haps`). Use `--nosort` to process in the order they are provided on the command line.

**Note:** The actual option names and usage may differ. Please consult the script's `--help` output for precise information.

## read.bin

### Purpose

The `read.bin` program is a utility tool designed to inspect binary matrix files that are generated by other programs in this suite (e.g., `grm.snp`, `grm.hap`). These binary files typically store a header with matrix dimensions (rows and columns) followed by the matrix elements as `double` precision floating-point numbers in row-major order.

This tool allows you to print the top-left N x N corner of such a matrix to the standard output for quick verification or inspection.

### Options

- **Positional Argument:** `<matrix_file>`
  - The path to the binary matrix file you want to inspect (e.g., `mygrm.g.A`, `output.bin`). This is a required argument.
- `-n N` (where `N` is an integer):
  - Specifies the number of rows and columns to display from the top-left corner of the matrix.
  - Default: If not specified, `N` defaults to `10`. So, it will print a 10x10 submatrix. If the matrix dimensions are smaller than N, it will print the entire matrix or the largest possible square submatrix.
- `-h, --help`:
  - Displays a help message summarizing usage and options, then exits.

### Example Usage

To print the top-left 5x5 corner of a binary matrix file named `project_alpha.g.AD`:

```
./bin/read.bin -n 5 project_alpha.g.AD
```

To print the default 10x10 corner of a matrix file `final_grm.bin`:

```
./bin/read.bin final_grm.bin
```

The output will be formatted numbers representing the matrix elements, printed to the console.

## rank

### Purpose

The `rank` program is used to calculate the numerical rank of one or more distributed symmetric matrices, typically Genomic Relationship Matrices (GRMs), stored in binary format. It reads GRM files (e.g., `.g.A`, `.g.D`, `.g.AA`) associated with a given base name, computes their eigenvalues using ScaLAPACK, and then counts how many eigenvalues are significantly greater than zero to determine the rank.

This is useful for diagnosing multicollinearity issues or understanding the effective dimensionality of the genomic information captured in a GRM.

### Options

The `rank` program uses the same common option parser as `reml`. However, many of the options applicable to `reml` (like those for phenotype or variance component values) are not directly used by `rank` for its core computation, though they will be parsed if provided.

#### Primary Options:

- `--input <load_name>` / `-i <load_name>` or **Positional Argument** `<load_name>`:
  - Specifies the base name for the GRM files. The program will iterate through a predefined list of known GRM extensions (e.g., `.g.A`, `.g.D`, `.g.AH`, `.g.AA`, `.g.AD`, `.g.DD`, `.g.AAA`, etc.) and attempt to load and calculate the rank for each file found under `<load_name><extension>`.
  - If `--input` is not used, the first positional argument is taken as `<load_name>`.
  - Default for `<load_name>` if not specified: `grm`.

#### ScaLAPACK/BLACS Options:

- `--blocks <size>` / `-n <size>`:
  - Sets the ScaLAPACK block size for matrix operations, which can affect performance and memory distribution during eigenvalue computation. Default: `128` (but the program might adjust this if the matrix dimension is smaller).

- `-r <num_proc_rows>`:
  - Overrides the automatically calculated number of processor rows in the BLACS grid.
- `-c <num_proc_cols>`:
  - Overrides the automatically calculated number of processor columns in the BLACS grid.

## Example Usage

To calculate the rank of all standard GRM files associated with the base name `dairy_study` using 4 MPI processes:

```
mpirun -np 4 ./bin/rank dairy_study
```

The program will output the base name, followed by a list of GRM types found (e.g., "a", "d", "aa") along with their calculated rank and the dimension of the matrix.

Example Output Snippet:

```
dairy_study
GRM, RANK, IND
a, 1000, 1000
d, 998, 1000
aa, 950, 1000
...
```

This indicates that for `dairy_study.g.A` (dimension 1000x1000), the rank is 1000, etc.

# References

Da,Y. *et al.* (2022) Multifactorial methods integrating haplotype and epistasis effects for genomic estimation and prediction of quantitative traits. *Frontiers in Genetics*, **Volume 13 - 2022**.

Prakapenka,D. *et al.* (2020) GVCHAP: A computing pipeline for genomic prediction and variance component estimation using haplotypes and SNP markers. *Frontiers in Genetics*, **Volume 11 - 2020**.

Wang,C. *et al.* (2014) GVCBLUP: A computer package for genomic prediction and variance component estimation of additive and dominance effects. *BMC Bioinformatics*, **15**, 270.