The analysis of genome-wide association studies (GWAS) examining hundreds of thousands of single nucleotide polymorphism (SNP) markers requires strict control of the type I error (Dudbridge and Gusnanto, 2008; Manly et al., 2004; Pe’er et al., 2008). Many simple approaches to multiple testing correction such as the Bonferroni method fail to account for linkage disequilibrium (LD) among SNPs, which leads to an overly conservative P-value correction. The resulting loss of power matters increasingly because the number of genetic markers and the marker density both grow constantly (Howie et al., 2009).

Permutation-based corrections fully account for the correlation among SNPs caused by LD and therefore are considered the gold standard of multiple testing correction in GWAS. They provide the highest statistical power among the procedures controlling family wise type I error risk. On the other hand, they require a lot more computational effort than the simple Bonferroni adjustment. For example, running a large number of permutations (~100K) for large-scale marker sets using standard software such as PLINK (Purcell et al., 2007) can take up to several years of computing time (Gao et al., 2008; Han et al., 2009). Progress has been made by the introduction of accelerated permutation procedures (Browning, 2008; Kimmel and Shamir, 2006). The software PRESTO (Browning, 2008) allows to perform moderate numbers of permutations (1000 to 10,000) for large datasets within a day or more and thus already the calculation of adjusted P-values in the region of $10^{-3}$ to $10^{-4}$. Nevertheless, there has been an ongoing demand for faster methods to compute genome-wide adjusted P-values, which has motivated the development of various approximation algorithms over the last years.

A first alternative approach is based on the Bonferroni correction adjusting the testing threshold for $M$ markers being tested to $a' = a/M$. Cheverud (2001) suggested to replace the ‘Bonferroni $M$’ by an effective number of independent tests ($M_{\text{eff}}$), which is derived from eigenvalues of the marker’s correlation matrix. In this way, information about the correlation between SNPs is used and therefore results in a less conservative P-value adjustment than Bonferroni; that is, $M_{\text{eff}} < M$. Based on the initial idea, several authors proposed different ways of estimating $M_{\text{eff}}$ (Gao et al., 2008; Li and Ji, 2005; Moskvina and Schmidt, 2008; Nyholt, 2004). However, in general it still yields conservative estimates in comparison with the permutation test (Han et al., 2009).

Another alternative framework is based on the multivariate normal distribution (MVN), which is used as an approximation of the unknown distribution of the marker set. Lin (2005) and Seaman and Müller-Myhsok (2005) were the first to propose MVN-based methods for multiple testing adjustment in association studies, followed by Conneely and Boehnke (2007) who increased its efficiency by numerically computing the asymptotic MVNs (Genn, 1992), instead of deriving them by simulation. However, due to the numerical limitations of integrating high-dimensional MVNs, these approaches require a block-wise strategy in large marker sets, which does not consider correlations between disjoint marker blocks. To answer this problem, Han et al. (2009) proposed a resampling-based method called SLIDE, which uses a sliding window locally accounting for the inter-marker correlation. However, both accuracy and computational efficiency depend on the size of the window; that is, extending the window increases accuracy but at the same time results in a considerable loss of computational efficiency.
2 METHODS

2.1 General notation

2.1.1 Familywise error rate The approach to multiple testing correction used throughout this article relies on controlling the familywise error rate (FWER)—also known as the overall type 1 error rate—the probability of observing one or more false positives in a family of tests (Westfall and Young, 1993). In particular, we consider the common frequentists approach of controlling the FWER in the strong sense such that FWER ≤ α for some significance level α.

2.1.2 Statistical model For a marker with two alleles, a genotype is defined as the number of minor alleles, yielding a set of three possible genotypes Γ = {0, 1, 2}. Here, we do not treat missing genotype values for sake of simplicity, but the methods presented in this article can be readily extended to incorporate them as well.1 In a standard case–control association study involving N individuals—at each marker—each individual is genotyped with one of the three genotypes. For each marker, we can construct a 2 × 3 contingency table (Table 1). To detect a disease marker, we compare genotype frequencies between affected (cases) and non-affected (controls) individuals. A commonly applied test statistic is based on Armitage’s trend test (Armitage, 1955), which can be written as (Sasieni, 1997)

\[
T^2(\alpha) = \frac{N}{N-R}(N-R)^2 \left( \frac{\sum x_i}{N} - \frac{1}{2} \right)^2
\]

where \(N_i = i, i = 0, 1, 2\), denote the scores of the three genotypes, and \(n_i\) denote the genotype counts in the cases and the pooled sample, respectively (Table 1). Basically any test statistic suitable for analyzing case–control SNP data can be calculated from these counts. Let \(g \in \Gamma^N\), be a vector (or array) of length \(N\) that contains the genotype data of a marker for all individuals coded as 0, 1, 2 for the number of minor alleles. If \(g\) is stored in terms of an array \(g[1:N]\) in the computer memory, the entire genotype information is extracted by accessing each of the \(N\) cells of \(g\). Thus, calculating the test statistic for a marker basically consists of two steps:

1. determine genotype frequencies by accessing all \(N\) cells of the corresponding array \(g[1:N]\); and
2. conduct arithmetic operations to compute the test statistic (Equation (1)).

When using a permutation test, each permutation of the case–control status modifies the genotype counts in the contingency table so that both steps have to be done over and over again. Due to the fact that GWAS involve large populations of up to several thousand individuals, Step 1 is much more time consuming than Step 2 so that accelerating the permutation test means to improve the way of determining the genotype frequencies.

1 We interpret missing values as kind of a special genotype leading to an additional column in the contingency table. The PERMORY software handles missing values.

Table 1. Genotype distribution of a marker in a case–control study

<table>
<thead>
<tr>
<th>Genotype</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case</td>
<td>(n_0)</td>
<td>(n_1)</td>
<td>(n_2)</td>
<td>(R)</td>
</tr>
<tr>
<td>Control</td>
<td>(n_0)</td>
<td>(n_1)</td>
<td>(n_2)</td>
<td>(S)</td>
</tr>
<tr>
<td>Total</td>
<td>(n_0)</td>
<td>(n_1)</td>
<td>(n_2)</td>
<td>(N)</td>
</tr>
</tbody>
</table>

Fig. 1. Flow chart presenting the basic algorithm structure. The key optimization modules are placed in the center lane (blue). For each marker, one of the three methods is chosen at runtime to be used for permutation.

Altogether, numerous approximation methods have been proposed over the last few years steadily improving accuracy, but some concerns still remain. First of all, there is no agreement about a standard alternative method. Second, it is usually left to the user to set one or more method-specific parameters, especially since an optimal parameter choice frequently depends on the structure of the data at hand. Finally, alternative methods usually cannot cope with missing values and hence require additional imputation methods. In contrast permutation tests are not only the gold standard but also well known and easily applied independently from the underlying data. Thus, it is desirable to make permutation tests feasible for genome-wide application, which is done with the present article.

We propose an optimized permutation test algorithm called PERMORY, which in terms of runtime performance is comparable to the fastest available approximation methods. Since our algorithm does not involve any kind of approximation, it provides exact results to the fastest available approximation methods. Since our algorithm is extracted by accessing each of the \(N\) individuals—at each marker—each individual is genotyped with one of the three genotypes. For each marker, we can construct a 2 × 3 contingency table (Table 1). To detect a disease marker, we compare genotype frequencies between affected (cases) and non-affected (controls) individuals. A commonly applied test statistic is based on Armitage’s trend test (Armitage, 1955), which can be written as (Sasieni, 1997).
2.2 Permutation matrix

All algorithms in the present article require the permutations to be stored in advance. Formally, let \( P \in \{0,1\}^{k \times N} \) be the \( k \times N \) permutation matrix that encodes \( k \) permutations of the diseases status and \( i \in \{1,2, \ldots, N\} \) the index for an individual:

\[
P[k,i]= \begin{cases} 
1, & \text{i-th individual of k-th permutation is affected} \\
0, & \text{otherwise}
\end{cases}
\]

(2)

In our software implementation, large numbers of permutations are processed in blocks of 10K permutations, which costs a little extra time due to repeated file reading but has the added benefit that PERMORY usually does not need more than 1 GB of RAM tested successfully for 10^9 permutations using 3K cases and 3K controls.

2.3 Dummy code and bit arithmetic

Different software packages internally use different data formats for keeping the genotype data. Often the data are stored in integer or Boolean typed arrays, which is not ideal with respect to memory consumption. Here, we propose the usage of binary dummy codes, one for each genotype except the common genotype. Formally, for some genotype array \( X \), let \( U \in \{0,1\}^{(t-1)\times N} \) be the dummy coded genotype matrix:

\[
U[k,i]= \begin{cases} 
1, & \gamma \in g[i] \\
0, & \text{else}
\end{cases}
\]

(3)

We omit the index for the common genotype because its frequency can be derived via the marginals of the contingency table (Table 1). Thus, the row index \( \gamma \) starts at 1. Since here \( \Gamma=\{0,1,2\} \), \( U \) consists of just two rows. Internally each row of \( U \) is stored as a bitwise, reducing the required memory significantly as compared with integer arrays. While some other programs already use economic ways of storing the genotype data, for example, by ‘packing’ several genotypes into each internal word of data, our approach additionally enables efficient bit arithmetics on the bits. The pooled genotype frequencies \( n_\gamma \) (Table 1) for some array \( y \) can now be derived as \( n_\gamma = \sum U[k,i] \) by simply counting bits, which is done a lot faster than summing up integer arrays. A little extra work is required, because we are not interested in the pooled frequencies \( n_\gamma \) but rather in the case frequencies \( f_\gamma \) that is, we must only count the ‘case-bits’ but not the ‘control-bits’. An efficient approach is to set the ‘control-bits’ to ‘0’ before the bit counting takes place. For this purpose every permutation array \( X[k,*] \) [Equation (2)] is applied as a bit mask using the logical AND conjunction, in this way blanking out the ‘control-bits’ in the dummy bitset. The corresponding pseudo-code can be outlined as follows:

Listing 1 Permutation using bit arithmetics

```c
for (k in 1:K) { // for each permutation
    for (y in 1:2) { // for each genotype
        r_y = BITCOUNT(U[y,*] AND P[k,*]);
    }
}
```

An example of the basic bit arithmetic algorithm is presented in Figure 2A.

2.4 Genotype indexing

Let \( d \in \{0,1\}^R \) be a vector encoding the disease status for \( R \) cases and \( S = N-R \) controls

\[
d(i)\in \{0,1\}, \text{i-th individual is affected}
\]

hence \( \sum_i d(i) = R \). As a matter of fact, any permutation \( d \) of \( d \) does not change the number of cases (i.e. \( \sum_i d(i)=R \)) so that \( n_0,n_1,n_2,R,S \) and \( N \) (Table 1) are all invariant with respect to permutations of \( d \). Furthermore, the \( s_i \) for \( i=0,1,2 \) can be derived as \( s_i = n_0 - n_1 - n_2 \) and \( n_0 = n_1 + n_2 \) (Table 1). Thus, for any permutation of the genotype data, determining just the genotype counts \( r_1 \) and \( r_2 \) is entirely sufficient for the construction of the corresponding contingency table. Browning (2008) was the first to use this property; instead of checking the disease status of every single individual, he basically considered only the heterozygous and least common homozygous genotypes and determined how many of them referred to affected individuals, thereby obtaining \( r_1 \) and \( r_2 \). Formally, he treated the index sets

\[
X^+_\gamma = \{ i : g(i) = \gamma \}
\]

(4)

\[
D = \{ i : d(i) = 1 \}
\]

(5)

That is, \( X^+_\gamma \) for \( \gamma = 1,2 \) contains all indices \( i \) of the data array \( y \) that belong to a specific genotype \( \gamma \) while \( D \) contains all \( i \) that are marked affected. The genotype frequencies in cases are then determined as cardinals of the pairwise conjunctions of these sets: \( r_1 = |X^+_1 \cap D|, r_2 = |X^+_2 \cap D| \). Listing 2 presents both the standard and the corresponding genotype indexing approach for determining the genotype counts of a genotype array. Note that here as well as in the following listings we just consider a single genotype array to simplify matters.

Listing 2 Counting genotype frequencies using (1) a standard approach and (2) the genotype indexing approach

```c
// 1. Standard approach
for (i in 1:N) { // for 'i from 1 to N'
    if (d[i] == 1) { // if affected
        if (g[i] == 1) r_1 += 1; //r_1 = r_1 + 1; //r_1 = r_1 + 1;
        if (g[i] == 2) r_2 += 1;
    }
}
// 2. Genotype indexing approach
for (i in 1:length(X_y)) { // i in 1:length(X_y)
    r_y += d[X_y[i]]; //r_y += d[X_y[i]];
}
```

A graphical example of the genotype indexing approach is given in Figure 2B.

2.5 Transposed permutation

In theory, the genotype indexing algorithm should compute in \( (|X_y|+|X_S|)/N=n_1+n_2/N \) the time than the standard approach, but in practice this is not the case due to compiler and low-level optimization. The standard approach allows for a much better optimization in this regard because accessing all \( N \) cells in the genotype array \( g[1:N] \) is implemented as a loop with a strictly monotonically increasing index. In contrast, the marker indices in the genotype indexing approach are nonmonotonic and only known at runtime. To answer this problem, we modify the inner permutation loop such that it no longer depends on the way the genotype frequencies are derived. First, consider the usual way of conducting all permutations in sequential order:

- (1) Outer loop: consider permutation \( k (k=1, \ldots, K) \).
- (2) Inner loop: derive the genotype frequencies for the permuted affection status.
- (3) Next Permutation.

Using our notation, the permutation matrix \( P \) is processed row by row. Second, consider processing \( P \) column-wise instead.

(1) Outer loop: consider individual \( i (i=1, \ldots, N) \).
2. Inner loop: count in how many of all permutations the individual is affected, that is, compute \( \sum_{i=1}^{K} P(k,i) \).

3. Next individual.

Basically, the entire set of permutations now is processed individual or, using our notation, \( P \) is transposed and then processed the usual way, which we therefore call "transposed permutation". As a result, the index of the sum in the inner loop has become monotonically increasing (i.e., \( k = 1, \ldots, K \)) and genotype indexing in combination with transposed permutation (GTP) indeed requires just about \((n_1 + n_2)/N\) the time than the standard approach and therefore is very effective for permutation of markers with low minor allele frequencies.

Listing 3 Transposed permutation and genotype indexing combined

```r
for (i in 1:n) { // outer loop
  for (k in 1:K) { // permutation (inner loop)
    R[k,i] = P[k,i];
  }
}
```

Note that we need to keep track of the resulting genotype frequencies separately for each permutation. For this purpose, we define \( R[k,i] \) as the \( k \times i \) matrix of genotype frequencies resulting from \( K \) permutations where \( R[k,1] \) and \( R[k,2] \) correspond to \( r_1 \) and \( r_2 \) of the \( k \)-th permutation, respectively. At the end of the procedure (Listing 3), row \( R[k,\cdot] \) can be used to construct the \( 2 \times 3 \) contingency table of the \( k \)-th permutation.

### 2.6 Reconstruction memoization

In computing, memoization\(^3\) is a (machine-independent) strategy to speed up computer programs by avoiding the repeated calculation of results for previously processed inputs. A memoized function remembers the results, and subsequent calls with remembered inputs return the remembered result rather than recalculating it. That is, memoization is a means of lowering a function's time cost in exchange for space cost. As a matter of principle, a function can only be memoized if calling the function has the exact same effect as replacing that function call with its return value.

Let \( \Delta(g,f) \) be the distance between two genotype arrays \( g \) and \( f \) defined as

\[
\Delta(g,f) = \sum_{i=1}^{n} |g[i] − f[i]|
\]

which is the total number of different positions between both arrays. First, consider two markers with identical genotype arrays \( g = f \), or \( \Delta(g,f) = 0 \). Since a permutation of the disease status does not influence the genotype data itself, the genotype frequencies \( r_1 \) and \( r_2 \) of both markers will be pairwise identical for any permutation. With all \( r_1 \) and \( r_2 \) resulting from \( K \) permutations of \( g \) being stored in \( R[k,\cdot] \), we can therefore omit all permutations for \( f \) and instead set \( R[f,\cdot] = R[g,\cdot] \), thus memoizing the case frequencies under all permutations for \( g \).

Second, assume \( g \neq f \) is equal to \( f \) except for one single genotype in some individual \( x \) (i.e. \( g(x) \neq f(x) \)) and \( \Delta(g,f) = 1 \). For each permutation in which the \( x \)-th individual is affected, for some \( g \neq f \) there are six possible distinct pairs of genotypes and for each pair \( R[\cdot,\cdot] \) can be constructed from \( R[g,\cdot] \) as shown in Table 2. Thus, in the second scenario, the case frequencies \( r_1 \) and \( r_2 \) for \( f \) can be almost completely memoized, requiring only one or two additional operations (Table 2) per permutation. For each permutation in which the \( x \)-th individual is not affected, hence \( x \notin D \) (Equation (5)), the \( r_1 \) and \( r_2 \) do not change at all so that simply \( R[g,\cdot] = R[f,\cdot] \).

\(^3\)While memoization might be confused with memorization (because of the shared cognate), memoization has a specialized meaning in computing.
been processed, the range can be thought of as a sliding tail, keeping the data information of the last \( c \) markers. It is important to note that in order to benefit from the REM method using the sliding tail, the markers must be sorted in accordance with their genomic locations.

In reality, \( c \) is chosen to be small in comparison with the number of permutations \( K \), which makes the determination of \( l \) relatively cheap. In our algorithm we set the default value to \( c = 100 \), which is basically adequate for any data situation. A smaller value (e.g. \( c = 25 \)) might improve the performance when using a small number of permutations (like 1000) or with marker sets that show low inter-marker correlation. Likewise, a higher value (e.g. \( c = 200 \)) favors a lot of permutations (>100 000) and a very high marker density. The amount of improvement in either direction, however, is marginal at best, mainly because the most correlated markers are expected to be located very close to each other. In other words, a tail of size \( c = 100 \) covers the most correlated markers most of the time so that in practice tuning this value is probably of no use. On the other hand, it might make sense to tune that option in huge simulation studies where the computation might take several weeks.

2.6.2 Method choice

Both GIT and REM are most applicable for specific kinds of marker data. While the GIT method performs well with low minor allele frequencies, the REM method benefits from highly correlated markers. The computational performance of both methods hence varies with the particular marker at hand. In contrast, the performance of the bit arithmetic method solely depends on the number of the individuals and hence is constant over all markers. During computation, for each marker, we estimate the performance of the methods based on the characteristics of the marker at hand (i.e. minor allele frequency, similarity to neighbor markers) and accordingly choose the fastest method to perform all permutations for that marker (Fig. 1).

3 RESULTS

We compare our method with existing permutation-based software, namely PRESTO 1.0.1 (Browning, 2008) and PLINK 1.06 (Purcell et al., 2007) as well as alternative approaches, for which we select simpleM (Gao et al., 2008) representing the methods using \( M \)-bit and SLIDE 1.0.4 (Han et al., 2009) representing the MVN framework, respectively. To the best of our knowledge, these two algorithms both represent the fastest and most accurate methods of their class. We do not treat the RAT software by Kimmel and Shamir (2006), which is based on importance sampling, because it was designed as a special application to adjust a single, preferably highly significant \( P \)-value, whereas here we are interested in simultaneously adjusting a wide range of \( P \)-values.

3.1 Accuracy

We initially present an accuracy evaluation, which serves both as a recap of accuracy results for the alternative methods and as a proof of concept for the permutation-based algorithms. We follow (Han et al., 2009) using a similar type of presentation and the same basic dataset, which is the chromosome 22 data (5563 SNPs) of the Type 2 diabetes (T2D) study (1928 cases + 2934 controls) as part of the Welcome Trust Case Control Consortium Phase I study (WTCCC, 2007). We randomly shuffle case-control status and compute \( P \)-values until we obtain a dataset with uncorrected \( P \)-values in the range of \( 10^{-7} \) to \( 10^{-5} \) (x-axis in Fig. 3). This forms

\[ P \]

\[ \text{We downloaded the corresponding dataset (example1.slide.gz) from their web site http://slide.cs.ucla.edu.} \]

\[ \text{The analysis includes all non-polymorphic SNPs.} \]

\[ 3 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]

\[ 0 \]
(*)For simpleM the default PCA cutoff of 0.995 is used.

if its adjusted windows size of PRESTO and PLINK, each with 1 M permutations, SLIDE with a which are compared with the P

possible permutations) of the presented methods.6 As expected, all region in order to cover the relative sampling error, caused by the adjusted and reference Figure 3, we depict the relative error of the methods as ratios between P

P

permutation-based methods are colored gray because they are expected to stay within the confidence region of the permutation reference P

P

constitutes the relative adjustment error as the ratio of the adjusted P-value and the corresponding reference P-value derived by a permutation test using 100 M permutations. The shaded area represents the 95% confidence region that covers the relative sampling error regarding 1 M permutations; that is, each point within that area is considered an accurate adjustment. All permutation-based methods are colored gray because they are expected to stay within the confidence region of the permutation reference P-values. (*For simpleM the default PCA cutoff of 0.995 is used.)

our basic dataset. Next we calculate adjusted P-values for this dataset using a permutation test with 100 million permutations, which results in adjusted P-values ranging from 0.00057 to 0.066. These P-values constitute the reference adjusted P-values for this dataset, which are compared with the P-values produced by PERMORY, PRESTO and PLINK, each with 1 M permutations, SLIDE with a windows size of w=100 and 1 M samplings, and simpleM using its default settings, respectively. We assume a method is accurate if its adjusted P-values are close to the reference P-values. In Figure 3, we depict the relative error of the methods as ratios between adjusted and reference P-values. We also construct a confidence region in order to cover the relative sampling error, caused by the Monte Carlo approach (in contrast to exhaustive enumeration of all possible permutations) of the presented methods.5 As expected, all permutation-based methods stay within the confidence region of the reference. The SLIDE curve is consistent with the results presented in Han et al. (2009) as (for this dataset) it is nearly as accurate as the permutation test. Instead of simpleM, Han et al. (2009) in their work used a similar program called Keffective, for which simpleM might be considered an improved version (Gao et al., 2010). In our simulation, simpleM performs slightly better than Keffective did in the work of Han et al. (2009). However, it still exhibits a trend of increasingly conservative adjustment for less significant P-values.

which is a consequence of the plain Bonferroni-like correction using just a single threshold (M_{null}) for all markers.

With the exception of simpleM, which is not based on sampling, the precision of the other methods in Figure 3 can be controlled by the number of the applied permutations. Table 3 illustrates how the confidence region would change in Figure 3 if we had applied a different number of permutations. Particularly, the curve of the simpleM method would be covered by the confidence region of the simulation was based on <10K permutations, at least for the range of the depicted P-values. This also shows that for increasingly smaller adjusted (or true) P-values, a decent number of permutations is required in order to achieve a high precision.

3.2 Runtime performance

All computations were done on a 64 bit AMD 2.4GHz CPU running Debian GNU/Linux v5.0. First, the runtime performance is presented for real data from the Welcome Trust Case Control Consortium Phase II (WTCCC2) study. The data consists of 5667 individuals genotyped on the 1.2M Illumina chip. Particularly, we use the data from all 1 115 428 SNPs of the 22 autosomal chromosomes. We create a case-control dataset by randomly dividing the 5667 individuals into 2833 cases and 2834 controls and then compute corrected P-values using 1K, 10K, 100K and 1M permutations, respectively (Table 4). Note that the actual P-values have no impact on the performance results and are thus omitted. Since simpleM is not permutation based, its result is placed in the footnote of the table. The PLINK7 software was not primarily constructed for permutation testing and therefore yields an impractical runtime result for this data set. Using 10K permutations, PRESTO still would take about 31 h to analyze all 1.1M SNPs, while PERMORY finishes in just 3 h, which is comparable to both approximative methods SLIDE and simpleM (Table 4). Overall the runtimes of the permutation-based methods increases linearly with the number of permutations, and so does the precision. Basically, a precision of ε requires 1/ε permutations (Kimmel and Shamir, 2006).

To determine the effect of different marker densities on the relative performance of our algorithm, we secondly create simulated datasets for three standard SNP chip sizes: 500K, 1M and 2.4M, the latter mimicking marker sets today already being routinely used in genetic meta-analysis studies. The 500K marker set consists of SNPs from the Illumina Human660W-Quad, the 1M of SNPs from the Illumina Human1M-Duo and the 2.4M of SNPs

The sampling error of the reference permutation using 100 M permutations is negligibly small and therefore ignored.

---

**Table 3. Confidence intervals of the relative sampling error for the most extreme P-values of Figure 3, depending on the number of permutations**

<table>
<thead>
<tr>
<th>P-value</th>
<th>Number of permutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unadjusted</td>
<td>Adjusted</td>
</tr>
<tr>
<td>1.63e-07</td>
<td>0.00057</td>
</tr>
<tr>
<td>2.63e-05</td>
<td>0.06610</td>
</tr>
</tbody>
</table>

The corresponding confidence intervals for 10^6 permutations as indicated in Figure 3 are (0.92, 1.08) and (0.99, 1.01), respectively.

---

5The sampling error of the reference permutation using 100 M permutations is negligibly small and therefore ignored.

6Command line options: plink --noweb --model-trend --mperm ...

7Command line options: java -Xmx2G -jar presto.jar missing = ? test = t ...
## Table 4. Runtime of analyzing 1.1 million SNPs over 5667 individuals of WTCCC2 data

<table>
<thead>
<tr>
<th>#Permutations</th>
<th>PERMORY</th>
<th>SLIDE(^a)</th>
<th>PRESTO</th>
<th>PLINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>0.5 h</td>
<td>2.9 h</td>
<td>3.2 h</td>
<td>7.7 day</td>
</tr>
<tr>
<td>10K</td>
<td>3 h</td>
<td>5 h</td>
<td>31 h</td>
<td>77 day(^b)</td>
</tr>
<tr>
<td>100K</td>
<td>1.2 day</td>
<td>1.0 day</td>
<td>12.8 day</td>
<td>2.1 year(^b)</td>
</tr>
<tr>
<td>1M</td>
<td>12 day</td>
<td>8 day</td>
<td>128 day</td>
<td>21 years(^b)</td>
</tr>
</tbody>
</table>

\(^a\)Using window size of \(w = 100\).
\(^b\)Extrapolated value.

The runtime of simpleM using default principal component analysis (PCA) cutoff of 88 and 215 days, respectively. Extrapolated runtimes of PLINK (omitted in the figure) are 43, 93, and 423 days, respectively.

---

### 3.3 Software implementation

PERMORY is written in the C++ language and makes extensive use of the Boost C++ Library (www.boost.org). For future releases, multithreading support is planned to increase the efficiency of the software on multi-core CPUs. The name PERMORY was coined by the words permutation and memory, emphasizing the use of memoization in the algorithm.

### 4 DISCUSSION

Multiple testing adjustment is important for genetic data analysis but it has been computationally challenging to use the gold standard method, permutation tests. One can think of two general approaches to this problem: either accelerate the permutation procedure or take an efficient approach to compute approximation and improve its accuracy. In recent years, research primarily has focused on the latter approach. We employed the former and have developed a permutation algorithm optimized for use with genetic data. Our algorithm not only presents a notable improvement over existing permutation algorithms but even can compete with the fastest alternative methods. We showed that our algorithm is also the fastest alternative methods. We showed that our algorithm is also particularly effective for high-density marker sets. Extrapolated runtimes of PLINK (omitted in the figure) are 43, 88 and 215 days, respectively.

---

3Command line: hapgen -h haplotype-data.haps -l legend-file.leg -r mapfile.map -Ne 11418 -optimtest -n 5000 5000.

6This is in accordance with the authors of SLIDE who suggest a window size of \(w = 100\) and \(w = 1000\) in scenarios of collecting a set of 1 million and 10 million SNPs, respectively.

---

Fig. 4. Runtimes for differently sized marker sets. We consider three different marker sets consisting of 500K, 1M and 2.4M SNPs, respectively. For each marker set, we simulate a dataset of 3K cases + 3K controls and measure the runtimes required to compute adjusted \(P\)-values. In contrast to all other algorithms, the runtime of PERMORY does not increase in proportion to the number of SNPs, demonstrating the fact that the optimization techniques applied in PERMORY are particularly effective for high-density marker sets. Extrapolated runtimes of PLINK (omitted in the figure) are 43, 88 and 215 days, respectively.

---

from HapMap phase 2 (International HapMap Consortium, 2007). The data are simulated with the software HAPGEN proposed by Spencer et al. (2009), which mimics LD patterns in human populations based on existing phased haplotype data. As suggested on the HapMap web site, we use phased data releases of the CEU population from HapMap (http://hapmap.ncbi.nlm.nih.gov/downloads/phasing/2007-08_red2phas/), along with the recommended recombination rate files (http://mathgen.stats.ox.ac.uk/wtccc-software/recombination_rates/). We create case–control samples comprising 3K cases + 3K controls for each chip size (http://hapmap.ncbi.nlm.nih.gov/downloads/phasing/2007-08_red2phas/), along with the CEU population from HapMap (http://hapmap.ncbi.nlm.nih.gov/downloads/phasing/2007-08_red2phas/) for scenario of collecting a set of 1 million and 10 million SNPs, respectively.

---

\(\text{GWAS is likely to increase as well. Since a permutation test shuffles the phenotype status, an increasing number of individuals might adversely affect the performance of permutation test procedures. For this reason, we also perform runtime tests for a large sample comprising 10K cases + 10K controls but overall the relative performances between the methods is not much different as compared with Figure 4 (results not shown). It is worth noting in this context that the running time for permutation testing basically can be considered independent of sample size if the null distribution for large samples is estimated by only a subset of the samples as proposed by Dudbridge (2006). Another basic approach to enhancing permutation tests consists of parallelization using more than one CPU. If \(p\) permutations are desired and \(n\) CPUs are available, we only need to perform \(p/n\) permutations on each CPU and combine the resulting \(P\)-values. In this case the total runtime is cut down linearly in the number of CPUs.}

---

\(\text{DISCUSSION}

Multiple testing adjustment is important for genetic data analysis but it has been computationally challenging to use the gold standard method, permutation tests. One can think of two general approaches to this problem: either accelerate the permutation procedure or take an efficient approach to compute approximation and improve its accuracy. In recent years, research primarily has focused on the latter approach. We employed the former and have developed a permutation algorithm optimized for use with genetic data. Our algorithm not only presents a notable improvement over existing permutation test implementations but even can compete with the fastest alternative methods. We showed that our algorithm is also particularly effective for high-density marker sets. Extrapolated runtimes of PLINK (omitted in the figure) are 43, 88 and 215 days, respectively. PERMORY hence relieves the computational burden of permutation testing on a different scale.
genome-wide scale, for faster or more accurate determination of genome-wide \( P \)-values, respectively. It also extends application in research, for example, enabling more extensive simulation studies using permutation.

In the present article, we have covered genotypic trend tests for bi-allelic markers and binary traits. The PERMOMY software at this point (version 0.4.0) also supports allelic tests and we plan to integrate support for multi-allelic markers in a future release. Extending the algorithm to the analysis of data with quantitative phenotypes and multiple measured phenotypes should be possible but adds a level of complexity to both the algorithm and the software implementation and therefore is subject of further research.

ACKNOWLEDGEMENTS

We thank Thomas Gebhardt for his support with the MaRC Linux Cluster. We further thank Daniel F. Schwarz, Brandon Greene and the anonymous reviewers for their valuable suggestions and comments.

**Funding:** Von Behring Röntgen Stiftung (grant number 57–0048); German Research Foundation (grant number SCH542/9–5).

**Conflict of Interest:** none declared.

**REFERENCES**


Conneely,K.N. and Boehnke,M. (2007) So many correlated tests, so little time! rapid implementation and therefore is subject of further research.


