CLUMPP[1]: CLUster Matching and Permutation Program Version 1.1

Mattias Jakobsson[2]
Center for Computational Medicine and Biology
Department of Human Genetics
University of Michigan

Noah A. Rosenberg
Center for Computational Medicine and Biology
Department of Human Genetics
University of Michigan

April 30, 2007

The *CLUMPP* software is available at
`http://rosenberglab.bioinformatics.med.umich.edu/clumpp.html`[3]

[1] Jakobsson, M. & Rosenberg, N. A. (2007) *CLUMPP*: a cluster matching and permutation program for dealing with label switching and multimodality in analysis of population structure. *Bioinformatics* in press.

[2] Comments on *CLUMPP* can be sent to mjakob@umich.edu

[3] *CLUMPP* software and manual copyright © 2007 Mattias Jakobsson and Noah Rosenberg, University of Michigan. This software is distributed "as is" without warranty of any kind.

Contents

1 Introduction **2**

2 Algorithms **3**

3 Getting started **9**

3.1 Availability . . . 9

3.2 Installing *CLUMPP* . . . 9

3.3 Running *CLUMPP* . . . 9

4 Input files **11**

4.1 *paramfile* . . . 11

4.2 *indfile* . . . 11

4.3 *popfile* . . . 12

4.4 *permutationfile* . . . 13

5 Usage options **14**

5.1 Main parameters . . . 14

5.2 Additional parameters for the *Greedy* and *LargeKGreedy* algorithms . . . 15

5.3 Optional outputs . . . 16

5.4 Advanced options . . . 17

5.5 Command-line arguments . . . 17

6 Output files **19**

6.1 *outfile* . . . 19

6.2 *miscfile* . . . 19

6.3 *permuted_datafile* . . . 21

6.4 *every_permfile* . . . 21

6.5 *random_inputorderfile* . . . 22

7 Examples **23**

7.1 Using *CLUMPP* for small numbers of runs and clusters . . . 23

7.2 Using *CLUMPP* for large numbers of runs and clusters . . . 24

Acknowledgments **27**

References **28**

1 Introduction

A variety of population-genetic applications – such as association mapping, molecular ecological studies, and studies of human evolution – make use of the clustering of individual multilocus genotypes into populations. Many clustering algorithms have now been developed for employing population-genetic data to assign individuals – and fractions of individuals – to clusters (Pritchard *et al.*, 2000; Dawson & Belkhir, 2001; Anderson & Thompson, 2002; Corander *et al.*, 2003; Falush *et al.*, 2003; Chen *et al.*, 2006; Corander *et al.*, 2004; Corander & Marttinen, 2006; François *et al.*, 2006; Pella & Masuda, 2006). The result of a single cluster analysis is typically given as a matrix, where each individual is given a "membership coefficient" for each cluster – interpreted as a probability of membership, or as a fraction of the genome with membership in the cluster, depending on the setting – with membership coefficients summing to 1 across K clusters. The number of clusters is predefined by the user for some methods, and for others it is inferred.

Because clustering algorithms may incorporate stochastic simulation as part of the inference, independent analyses of the same data may result in several distinct outcomes, even though the same initial conditions were used. The main differences across replicates are of two types: "label switching" and "genuine multimodality." "Label switching" refers to a scenario in which different replicates obtain the same membership coefficient estimates, except with a different permutation of the cluster labels (Stephens, 2000; Jasra *et al.*, 2005). In unsupervised cluster analyses, because the meaning of each cluster label is not known in advance, a clustering algorithm may be equally likely to reach any of $K!$ permutations of the same collection of estimated membership coefficients.

It is also possible that replicate cluster analyses arrive at truly distinct solutions that are not equivalent up to permutation. This "genuine multimodality" may result from difficulties in searching the space of possible membership coefficients, or from real biological factors that cause multiple parts of this space to provide similarly appropriate explanations for the data.

Regardless of the source of differences in clustering outcomes, some method is needed for handling the results from replicate analyses. We develop three algorithms for finding optimal alignments of R replicate cluster analyses of the same data (Jakobsson & Rosenberg, 2007), which we have implemented in the computer program *CLUMPP*. Our program takes as input the estimated cluster membership coefficient matrices of multiple runs of a clustering program, for any number of clusters. It outputs these same matrices, permuted so that all replicates have as close a match as possible. *CLUMPP* also outputs a mean of the permuted matrices across replicates. The input file for *CLUMPP* is a file similar to the output from *structure* (Pritchard *et al.*, 2000; Falush *et al.*, 2003), and the output from *CLUMPP* can be used directly as input by the cluster visualization program *distruct* (Rosenberg, 2004).

2 Algorithms

We refer to the $C \times K$ matrix of membership coefficients for a single cluster analysis as the *Q-matrix*, with the C rows corresponding to individuals (or populations) and the K columns corresponding to clusters. *CLUMPP* attempts to maximize a measure of similarity of the Q-matrices of R replicates over all $(K!)^{R-1}$ possible alignments of the replicates.

Consider a pair of Q-matrices, Q_i and Q_j for runs i and j, where the value in the cth row and kth column of Q_i is the membership coefficient for individual c in cluster k as inferred in run i. Each matrix consists of nonnegative entries, and the sum of the entries in any row is 1. We define the pairwise similarity of matrices Q_i and Q_j as follows:

$$G(Q_i, Q_j) = 1 - \frac{\|Q_i - Q_j\|_F}{\sqrt{\|Q_i - W\|_F \|Q_j - W\|_F}}. \tag{1}$$

In this equation, W is a $C \times K$ matrix with all elements equal to $1/K$ and $\|\cdot\|_F$ is the Frobenius matrix norm (Golub & Van Loan, 1996)

$$\|A\|_F = \sqrt{\sum_{c=1}^{C} \sum_{k=1}^{K} a_{ck}^2}, \tag{2}$$

where C and K respectively denote the numbers of rows and columns of A, and a_{ck} is the value in row c and column k.

Using G to measure similarity, the optimal alignment of matrices Q_i and Q_j is defined as the permutation of the columns of Q_j that maximizes the similarity G over all permutations P in the set S_K of permutations of K clusters. The maximum value, or

$$\mathrm{SSC}(Q_i, Q_j) = \max_{P \in S_K} G(Q_i, P(Q_j)), \tag{3}$$

is the quantity named by Nordborg *et al.* (2005) the "symmetric similarity coefficient" (SSC) of the pair of runs (see also Rosenberg *et al.* (2002) for an earlier statistic). The SSC for two runs is bounded above by 1 — which it equals if the Q-matrices are identical up to a permutation of the clusters — and it decreases as the similarity of the Q-matrices decreases. The SSC statistic is generally expected to be positive if nontrivial clustering patterns are present in Q_i and Q_j, although it is possible for it to be negative.

For a collection of R replicates, the average pairwise similarity is defined as

$$H(Q_1, Q_2, \ldots, Q_R) = \frac{2}{R(R-1)} \sum_{i=1}^{R-1} \sum_{j>i}^{R} G(Q_i, Q_j). \tag{4}$$

To find the optimal alignment of R replicates, we search for the vector of permutations that maximizes this average pairwise similarity:

$$\mathrm{SSC}_R(Q_1, Q_2, \ldots, Q_R) = \max_{(P_1,P_2,\ldots,P_R)\in S_K^R} H(P_1(Q_1), P_2(Q_2), \ldots, P_R(Q_R)). \tag{5}$$

Without loss of generality, we take P_1 to be the identity permutation I, so that the clusters of runs $2, \ldots, R$ are permuted to align with the clusters of run 1. As S_K contains $K!$ permutations, with P_1 set to equal I, the maximum in eq. 5 is taken over $(K!)^{R-1}$ vectors.

We make use of three algorithms for attempting to find the optimal alignment of R replicates. In decreasing order of the extent of the search, and in increasing order of computational speed, these algorithms are termed *FullSearch*, *Greedy* and *LargeKGreedy*. These algorithms supersede earlier methods that we described in Nordborg *et al.* (2005).

Note that our approach can proceed analogously using alternative functions to measure similarity in place of G. Although it is undefined when one of the two matrices equals W, G is designed to have large negative values when one of the two runs reflects substantial population structure and the other has relatively little structure (that is, little difference from W). We can define a second similarity function G', which is guaranteed to lie in $[0, 1]$:

$$G'(Q_i, Q_j) = 1 - \frac{\|Q_i - Q_j\|_F}{\sqrt{2C}}. \tag{6}$$

The normalization constant $\sqrt{2C}$, which guarantees that G' lies in $[0, 1]$, arises from the definition of the Frobenius norm:

$$\|A - B\|_F = \sqrt{\sum_{c=1}^{C}\left(\sum_{k=1}^{K} a_{ck}^2 + b_{ck}^2 - 2a_{ck}b_{ck}\right)}.$$

If A and B have nonnegative entries and row sums of 1, then $-2a_{ck}b_{ck} \le 0$ and $\sum_{k=1}^{K} a_{ck}^2 = (\sum_{k=1}^{K} a_{ck})^2 - 2\sum_{k=1}^{K-1}\sum_{\ell>k}^{K} a_{ck}a_{c\ell} = 1 - 2\sum_{k=1}^{K-1}\sum_{\ell>k}^{K} a_{ck}a_{c\ell} \le 1$. Similarly, $\sum_{k=1}^{K} b_{ck}^2 \le 1$. It then follows that $\|A - B\|_F \le \sqrt{2C}$.

The quantities SSC$'$, H', and SSC'_R can then be defined by replacing G in eqs. 3, 4, and 5 with G'. We proceed to describe our algorithms using the G statistic to measure similarity; to instead use G', the approach is analogous with G', SSC$'$, H', and SSC'_R in place of G, SSC, H, and SSC_R.

FullSearch

The *FullSearch* algorithm computes H for each of the $(K!)^{R-1}$ alignments of the K clusters in R replicates. Considering all possible vectors of permutations $(I, P_2, P_3, \ldots, P_R)$, the algorithm

Table 1: Example of two ($R = 2$) Q-matrices (left), that is, two replicate cluster analyses, that have been permuted (right) in order to align the clusters (columns). The leftmost column indicates the population labels ($C = 95$), the following 3 columns indicate the membership coefficient of each population in 3 clusters ($K = 3$), and the last column indicates the number of individuals in each population.

1:	0.315	0.002	0.683	10	1:	0.315	0.002	0.683	10
2:	0.475	0.014	0.511	1	2:	0.475	0.014	0.511	1
3:	0.090	0.005	0.905	1	3:	0.090	0.005	0.905	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
94:	0.004	0.003	0.993	1	94:	0.004	0.003	0.993	1
95:	0.004	0.010	0.985	1	95:	0.004	0.010	0.985	1
1:	0.687	0.002	0.310	10	1:	0.310	0.002	0.687	10
2:	0.490	0.011	0.500	1	2:	0.500	0.011	0.490	1
3:	0.898	0.007	0.095	1	3:	0.095	0.007	0.898	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
94:	0.993	0.004	0.003	1	94:	0.003	0.004	0.993	1
95:	0.987	0.008	0.005	1	95:	0.005	0.008	0.987	1

computes $H(I(Q_1), P_2(Q_2), P_3(Q_3), \ldots, P_R(Q_R))$ and returns the vector of permutations that maximizes SSC_R. As the number of possible vectors grows quickly with K and R, however, even for moderate values of K and R, it is unrealistic to test every alignment. The *FullSearch* algorithm runs in time proportional to $T_{FullSearch} = (K!)^{R-1}[R(R-1)/2]KC$: the number of permutation vectors is $(K!)^{R-1}$, the number of computations of G in each evaluation of eq. 4 is $R(R-1)/2$, and the time required for each computation of G is proportional to KC. Although it proceeds slowly, unlike our other algorithms, the *FullSearch* algorithm is guaranteed to find the optimal alignment of clusters across multiple runs.

Greedy

The *Greedy* algorithm employed by *CLUMPP* proceeds as follows:

1. Choose one run, Q_1.

2. Choose a second run, Q_2, and fix the permutation, P_2, that maximizes $G(P_1(Q_1), P(Q_2))$ over all possible permutations P (where P_1 is the identity permutation).

3. Continue sequentially with each remaining run, Q_x, where $x = 3, ..., R$, and fix the permutation, P_x of Q_x, that maximizes the average similarity with the previously fixed $x-1$ runs, or

$$J(P_1(Q_1), P_2(Q_2), \ldots, P_{x-1}(Q_{x-1}), P(Q_x)) = \frac{1}{x-1} \sum_{\ell=1}^{x-1} G(P_\ell(Q_\ell), P(Q_x)), \qquad (7)$$

over all permutations P.

This algorithm runs in time proportional to $T_{Greedy} = (K!)[R(R-1)/2]KC$. The number of permutations tested for each run from 2 to R is $K!$. For run r (r ranging from 2 to R), the number of computations of G performed for each permutation is $r-1$. Thus, considering all runs from 2 to R, the total number of computations of G performed is $(K!)[R(R-1)]/2$. Each computation of G runs in time proportional to KC.

Because the order in which the runs are considered can affect the result, several different sequences of runs should be tested. *CLUMPP* offers three options for testing different sequences: test all possible sequences of runs, test a pre-defined number of random sequences, and test a specific set of user-defined sequences.

LargeKGreedy

When $K \gtrsim 15$, the number of permutations, $K!$, is very large, and it may not be possible to calculate G for all permutations of a particular pair of Q-matrices. Instead of testing every permutation as in the *Greedy* algorithm, the *LargeKGreedy* algorithm proceeds as follows:

1. Choose one run, Q_1.

2a. Choose a second run, Q_2. Compute G for all pairs of columns, one from Q_1 and one from Q_2. This computation is simply the value of G for two columns – hence no permutations of Q_2 are computed, unlike in step 2 for the *Greedy* algorithm.

2b. Pick the pair of columns Q_{1,y_1} and Q_{2,z_1} with highest G-value and fix these columns (Q_{1,y_1} refers to column y_1 of matrix Q_1). Then pick the pairs of columns Q_{1,y_2} and Q_{2,z_2} with the next highest G-value, ignoring all G-values of pairs containing either of the previously chosen columns Q_{1,y_1} and Q_{2,z_1}. Repeat this procedure until K pairs of columns, one each from Q_1 and Q_2, have been picked, and fix the permutation of Q_2 that matches up these pairs of columns, or $P_2(Q_2)$.

3a. Continue sequentially with each remaining run, Q_x, where $x = 3, ..., R$. For each y and z from 1 to K, compute the average similarity,

$$J(P_{1,y}(Q_1), \ldots, P_{x-1,y}(Q_{x-1}), Q_{x,z})) = \frac{1}{x-1} \sum_{\ell=1}^{x-1} G(P_{\ell,y}(Q_\ell), Q_{x,z})), \qquad (8)$$

where $P_{\ell,y}$ denotes column y of the permuted matrix $P_\ell(Q_\ell)$. This quantity is the similarity of column z of Q_x to column y of each of the previously fixed permutations, averaged across all runs previously considered. No permutations of Q_x are computed, unlike in step 3 for the *Greedy* algorithm.

3b. Pick the pair of columns y_1 of $P_1(Q_1), P_2(Q_2), \ldots, P_{x-1}(Q_{x-1})$ and z_1 of Q_x with highest average similarity in eq. 8. Then pick the columns y_2 and z_2 with the next highest similarity in eq. 8, ignoring similarity scores of pairs containing either of the previously chosen columns y_1 and z_1. Repeat this procedure until K pairs of columns, one for the matrices $P_1(Q_1), P_2(Q_2), \ldots, P_{x-1}(Q_{x-1})$ and one for Q_x, have been picked. Fix the permutation of Q_x that matches up these pairs of columns, or $P_x(Q_x)$.

A candidate for the vector of permutations of the R runs that maximizes H across all possible vectors has now been constructed. This algorithm runs in time proportional to $T_{LargeKGreedy} = [R(R-1)/2]K^2C$. The number of pairs of columns, one from the run under consideration and one from the previously fixed runs, is K^2. For run r (r ranging from 2 to R), the number of computations of G performed for each pair of columns is $r-1$. Considering all runs from 2 to R, the total number of computations of G performed is $(K!)[R(R-1)]/2$. Since G is computed only for columns rather than for whole matrices, the time of computation of G is proportional only to C rather than to KC, as in the other algorithms.

As is true for the *Greedy* algorithm, the order in which the runs are considered can affect the result. For the *LargeKGreedy* algorithm *CLUMPP* offers the same three options for selecting the input sequence of runs as it provides for the *Greedy* algorithm.

To get an idea of which algorithm to use, we have found it useful to compute the quantity $D = TCN$ for each algorithm, where T is a quantity proportional to the time required by an algorithm (as described above), C is the number of individuals, and N is the number of input sequences to be tested (for *FullSearch*, $N = 1$). If $D \lesssim 10^{13}$ for *FullSearch*, then this algorithm is fast enough and is preferred; otherwise the *Greedy* algorithm can be used. If $D \gtrsim 10^{13}$ for the *Greedy* algorithm, then this algorithm is probably also too slow. In that case, the *LargeKGreedy* algorithm should be used, as it can handle $K > 20$, $R > 100$, and $C > 1000$ in reasonable time. We recommend testing at least 100 input sequences for the *Greedy* and *LargeKGreedy* algorithms – and preferably many more – to find the approximately highest SSC_R value.

Table 2 shows all possible combinations of the options M and GREEDY_OPTION used by CLUMPP. Table 3 gives some guidelines of when to use the different algorithms and options. Generally, these choices are a balance between the extent of the search and the speed of the search.

Table 2: All possible combinations of the options M and GREEDY_OPTION.

M	**GREEDY_OPTION** 1	2	3
1	All possible permutations of the runs are tested. GREEDY_OPTION does not matter.		
2	The best permutation is constructed by greedily aligning runs. All possible input orders for the list of permutations are tested and the one that produces the highest H (or H') is taken.	The best permutation is constructed by greedily aligning runs. A number of input orders specified by REPEATS are tested and the one that produces the highest H (or H') is taken.	The best permutation is constructed by greedily aligning runs. A number of input orders specified by the permutations in the *permutationfile* are tested and the one that produces the highest H (or H') is taken.
3	The best permutation is constructed by greedily aligning clusters. All possible input orders for the list of permutations are tested and the one that produces the highest H (or H') is taken.	The best permutation is constructed by greedily aligning clusters. A number of input orders specified by REPEATS are tested and the one that produces the highest H (or H') is taken.	The best permutation is constructed by greedily aligning clusters. A number of input orders specified by the permutations in the *permutationfile* are tested and the one that produces the highest H (or H') is taken.

Table 3: Recommended use of algorithms and GREEDY_OPTION.

M	**GREEDY_OPTION** 1	2	3
1	This is the gold standard of searching, which one should strive to use, but it can often be too slow. When $D \lesssim 10^{13}$ for *FullSearch*, then this algorithm is fast enough and should be preferred.		
2	For this option, every input order is tested, which will cause rather long run-times. For some choices of K and R this option can be slower than the *FullSearch*. This option can be used to test accuracy of non-exhaustive searches.	This option has reasonable run-times for large R, but not if K becomes large. The user can specify the run-time by changing the number of random input orders to be tested (REPEATS).	This option has reasonable run-times for large R, but not if K becomes large. The user can specify the input orders to be tested in the *permutationfile*.
3	For this option, every input order is tested, which may cause rather long run-times. This option can by used to test accuracy of non-exhaustive searches.	This option has short run-times even for large values of R and K. The user can specify the number of random input orders to be tested (REPEATS).	This option has short run-times even for large values of R and K. The user can specify the input orders to be tested in the *permutationfile*.

3 Getting started

We distribute executables for *CLUMPP* to run under Linux, MacOS X and Windows. The program is written in C++, and if you would like to compile the source code on your own favorite platform please email `mjakob@umich.edu` and we will send you the source code.

3.1 Availability

Pre-compiled executables for Linux/Unix, MacOS X, and Windows are available at:
`http://rosenberglab.bioinformatics.med.umich.edu/CLUMPP.html`
When using *CLUMPP*, please cite:

Jakobsson, M. and Rosenberg, N. A. (2007). *CLUMPP*: a cluster matching and permutation program for dealing with label switching and multimodality in analysis of population structure. *Bioinformatics* in press.

The *structure* software is available at: `http://pritch.bsd.uchicago.edu`. The appropriate citations for *structure* are Pritchard *et al.* (2000) and Falush *et al.* (2003). The *distruct* software is available at `http://rosenberglab.bioinformatics.med.umich.edu/distruct.html`. The appropriate citation for *distruct* is Rosenberg (2004).

3.2 Installing *CLUMPP*

The executable comes in a gzipped tar file. In Unix/Linux (and in MacOS X, from a command prompt), extract the appropriate `.tar.gz` file by typing: `gunzip CLUMPP_Linux.xx.tar.gz`, and then typing: `tar -xvf CLUMPP_Linux.xx.tar`, where `xx` is the version number. This will create a new directory called `CLUMPP_Linux.xx`.

In Windows, extract the file `CLUMPP_Windows.xx.zip`. This will create a directory called `CLUMPP_Windows.xx`.

3.3 Running *CLUMPP*

In Unix (and in MacOS X, from a command prompt), the program is executed by typing `./CLUMPP` ***`paramfile`***, where *paramfile* is the name of the parameter file. If no *paramfile* is given after typing `./CLUMPP`, the program will search for a *paramfile* called "paramfile", and if this file is not found, *CLUMPP* will exit with an error message. The *paramfile* should be located in the same directory as *CLUMPP*. If the *paramfile* is located elsewhere, the path must be provided for it. If *CLUMPP* is executed from another directory, *CLUMPP* will search for the files it needs in the current directory, and if *paramfile*, *indfile* or *popfile*, and possibly

permutationfile are located elsewhere, the path for these files must be specified (make sure to not exceed the limit of 50 characters for paths and names specified in the *paramfile*).

In Windows, *CLUMPP* is run from a command prompt. In the command prompt (which can be accessed by going to the START menu, clicking on Run, and typing `cmd`), move to the directory where *CLUMPP* is located (by typing `cd c:Program Files\CLUMPP` on our machine). The program is then executed by typing `CLUMPP` ***`paramfile`***, where *paramfile* is the name of the parameter file. If no *paramfile* is specified after typing `CLUMPP`, *CLUMPP* will search for a *paramfile* called "paramfile", and if this file is not found, *CLUMPP* will exit with an error message. It is also possible to double-click on the *CLUMPP* icon to run the program; *CLUMPP* will then use the parameters in the file called "paramfile". All files that *CLUMPP* needs must be located in the same directory as the program, or the path to these files must be included in the *paramfile*.

4 Input files

CLUMPP reads the necessary parameters from a file (the *paramfile*). The program will also read in a file containing the Q-matrices (the *popfile* or the *indfile*). In some cases, an additional file of permutations of runs (the *permutationfile*) is required by *CLUMPP*.

CLUMPP can be used for any data that have population or individual membership coefficients and for any number of clusters. In particular, *CLUMPP* can use a file that is similar (but not quite identical) to the *structure* output file. The *indfile* and the *popfile* of *CLUMPP* can easily be obtained from *structure* output files by, for example, cutting and pasting. If another program is used to create the Q-matrices, these data need to be formatted to match the *CLUMPP indfile* or *popfile* format (see below).

4.1 *paramfile*

CLUMPP reads a *paramfile*, which follows (separated by a space) after typing *CLUMPP* on the command-line: `./CLUMPP paramfile` in UNIX, and `CLUMPP paramfile` in Windows. (If no *paramfile* is specified, *CLUMPP* will search for a file called "paramfile". *CLUMPP* will therefore run in Windows if the user double-clicks on the *CLUMPP*-icon and if a *paramfile* with the name "paramfile" resides in the same directory as the program.) This *paramfile* can have any name, but will hereafter be referred to as the *paramfile*. There are a number of parameters that must be defined in the *paramfile*, and there are also additional parameters that, in specific cases, must be defined (see Section 5). Some of the parameters of the *paramfile* can be overridden by command-line arguments (see Section 5.5).

4.2 *indfile*

The cluster analyses may have been conducted for individuals or for populations. There are some small differences in the format of the *CLUMPP* input data from individuals and populations. If DATATYPE = 0, *CLUMPP* expects a file, the *indfile*, containing R sets of *individual Q-matrices*. The *indfile* is specified in the *paramfile* by INDFILE. If C is the number of predefined individuals, K is the number of predefined clusters and R is the number of predefined runs, *CLUMPP* expects an *indfile* with $C \times R$ rows and $K + 5$ columns per row. The file should be organized in such a way that the results from the first run (this is an arbitrary choice) are followed (below the results of the first run) by the results from the second run and so on. Blank lines and extra space are tolerated. Table 4 shows an example when $C = 95$, $K = 3$ and $R = 2$. In Table 4, each row represents the membership coefficients for an individual in a specific run. Row 1 represents the membership coefficients for individual 1 in run 1; row $C + 1$ (not counting blank lines) represents the membership coefficients for individual 1 in run 2. Columns 1, 3, 4, and 5 are ignored by *CLUMPP*. Column 2 is an

integer that identifies the individual. The following K columns (column 6 to column $K+5$) are the membership coefficients for clusters 1, 2, ..., K (real numbers in [0,1]). The numbers in the K columns for each individual should ideally sum to 1. *CLUMPP* will normalize these numbers by their sum, and if the sum deviates too much from 1 (± 0.02), *CLUMPP* will produce a warning message. This warning can be automatically overridden by setting OVERRIDE_WARNINGS to 1. Note that the order of individuals must be the same in every run; *CLUMPP* exits with an error otherwise. The format of the *indfile* is the same as the format of the input file (INFILE_INDIVQ) of *distruct*.

Table 4: Example of an *indfile* when $C = 95$, $K = 3$ and $R = 2$.

```
 1  indnr  (x)  popnr  :  0.315  0.002  0.683
 2  indnr  (x)  popnr  :  0.475  0.014  0.511
 3  indnr  (x)  popnr  :  0.090  0.005  0.905
 ⋮      ⋮    ⋮      ⋮  ⋮    ⋮      ⋮      ⋮
94  indnr  (x)  popnr  :  0.004  0.003  0.993
95  indnr  (x)  popnr  :  0.004  0.010  0.985

 1  indnr  (x)  popnr  :  0.687  0.002  0.310
 2  indnr  (x)  popnr  :  0.490  0.011  0.500
 3  indnr  (x)  popnr  :  0.898  0.007  0.095
 ⋮      ⋮    ⋮      ⋮  ⋮    ⋮      ⋮      ⋮
94  indnr  (x)  popnr  :  0.993  0.004  0.003
95  indnr  (x)  popnr  :  0.987  0.008  0.005
```

4.3 *popfile*

If DATATYPE = 1, *CLUMPP* expects a file, the *popfile*, which is specified in the *paramfile* by POPFILE. The *popfile* should be organized in such a way that the results from the first run (this is an arbitrary choice) are followed (below the results of the first run) by the results from the second run and so on. The Q-matrices of the populations will be referred to as the *population Q-matrices*. The *popfile* should contain $C \times R$ rows and $K+2$ columns per row. Blank lines and extra space are tolerated in the *popfile*. Table 5 shows an example when $C = 95$, $K = 3$ and $R = 2$. In Table 5, each row represents the membership coefficients for a population in a specific run. Row 1 represents the membership coefficients for population 1 in run 1; row $C+1$ (not counting blank lines) represents the membership coefficients for population 1 in run 2. The first column is an integer that identifies the population, and it is followed by a colon. The following K columns are the membership coefficients for clusters 1, 2, ..., K (real numbers in [0,1]). The numbers in the K columns for each population should ideally sum to 1. *CLUMPP* will normalize these numbers by their sum, and if the sum deviates too much from 1 (± 0.02), *CLUMPP* will produce a warning message. This warning can be automatically overridden by setting OVERRIDE_WARNINGS to 1. The final column gives

the number of individuals of each population. Note that the order of populations must be the same in every run. The format of the *popfile* is the same as the format of the input file (INFILE_POPQ) of *distruct*.

Table 5: Example of a *popfile* when $C = 95$, $K = 3$ and $R = 2$.

1:	0.315	0.002	0.683	10
2:	0.475	0.014	0.511	1
3:	0.090	0.005	0.905	1
⋮	⋮	⋮	⋮	⋮
94:	0.004	0.003	0.993	1
95:	0.004	0.010	0.985	1
1:	0.687	0.002	0.310	10
2:	0.490	0.011	0.500	1
3:	0.898	0.007	0.095	1
⋮	⋮	⋮	⋮	⋮
94:	0.993	0.004	0.003	1
95:	0.987	0.008	0.005	1

4.4 *permutationfile*

The *permutationfile* contains pre-defined orders of runs. For the *Greedy* and the *LargeKGreedy* algorithms, the input order of runs can have some effect on the value of H (or H'), and the input order may therefore have a small effect on the resulting alignment of the columns of the runs. Each line in the *permutationfile* must be a permutation of the integers 1, 2, ..., R, indicating the order of the runs in the *indfile* or *popfile* that *CLUMPP* will use. The integer in the first position indicates which run to input first, the integer in the second position indicates which run to input second, and so on. Table 6 shows an example of a *permutationfile* in which $R = 10$ and 5 different input orders will be tested by *CLUMPP*. Note that the parameter REPEATS must be set to match the number of lines (excluding blank lines) in the *permutationfile*. The *permutationfile* is expected by *CLUMPP* if the *Greedy* or the *LargeKGreedy* algorithm is chosen (M = 2 or M = 3) at the same time as GREEDY_OPTION = 3.

Table 6: Example of a *permutationfile* for 5 different user-defined input orders of runs when $R = 9$. The second row indicates that the runs in the *indfile* or *popfile* will be used starting with the 2nd run, then the 4th run, and so on.

1	2	3	4	5	6	7	8	9
2	4	6	8	1	3	5	7	9
5	4	3	2	1	9	8	7	6
1	9	2	8	3	7	4	6	5
9	8	7	6	5	4	3	2	1

5 Usage options

CLUMPP reads the parameters from the file that follows the program name. This file is known as the *paramfile*, but can be given any name. Each parameter in the *paramfile* is printed in capital letters followed by one or more blank spaces. The spaces are then followed by the particular parameter value. Everything on a line after the symbol "#" is ignored; this symbol is used for making comments about the parameters in the *paramfile*, such as what type of value *CLUMPP* expects. "K 3 # Number of clusters" is an example that informs *CLUMPP* to expect three clusters in the *indfile* or *popfile*. The parameter values used for a run of *CLUMPP* are printed to the screen and to a "miscellaneous" output file (the *miscfile*) that is specified in the *paramfile*. The program is insensitive to the order of the parameters in the *paramfile*. Some options, such as the *Greedy* algorithm (M = 2), require additional parameters to be specified. If an extra parameter is not expected by the program, it will ignore the parameter regardless of whether or not the parameter is specified in the *paramfile*. Several of the parameters can also be set by command-line arguments (see Section 5.5).

5.1 Main parameters

The main parameters concern the input datafile (*indfile* or *popfile*), the output files (*outfile* and *miscfile*), and the choice of algorithm.

DATATYPE (int) Type of datafile to be used. If DATATYPE = 0, *CLUMPP* expects to read the individual Q-matrices from an *indfile* (see Section 4.2), and if DATATYPE = 1, *CLUMPP* expects to read the population Q-matrices from a *popfile*.

INDFILE (string) The name of the *indfile* (maximum length of 50 characters). This file contains the individual Q-matrices for all runs (see Section 4.2 for more information on formatting the *indfile*). The *indfile* is required if DATATYPE = 0.

POPFILE (string) The name of the *popfile* (maximum length of 50 characters). This file contains the population Q-matrices for all runs (see Section 4.3 for more information on formatting the *popfile*). The *popfile* is required if DATATYPE = 1.

OUTFILE (string) The name of the *outfile* (maximum length of 50 characters). This file contains the mean Q-matrix over all runs after the "optimal" permutation has been identified (see Section 6.1 for more information on the *outfile*). The *outfile* is always required.

MISCFILE (string) The name of *miscfile* (maximum length of 50 characters). This file

contains parameter settings for the current run of *CLUMPP*, the highest value of H (or H'). Note that for the *FullSearch* algorithm the highest H-value is guaranteed to equal SSC_R. This file also contains the corresponding permutation of the Q-matrices for each run (see Section 6.2 for more information on the *miscfile*). The *miscfile* is always required.

K (int) Number of clusters.

C (int) Number of populations.

R (int) Number of Q-matrices, or runs, to be aligned.

M (int) Algorithm to be used for aligning the runs. Valid choices are 1, 2 or 3: 1 to use the *FullSearch* algorithm, 2 to use the *Greedy* algorithm and 3 to use the *LargeKGreedy* algorithm (see Section 2 for more information about the algorithms).

W (boolean) If the user has data from populations that contain more than one individual (indicated by the last column in the *popfile*), *CLUMPP* offers the option W of computing H (or H') weighted by the number of individuals in each population. Choices for W are 1 to weight the alignment procedure by the number of individuals in each population (indicated in last column in *popfile*), and 0 to not weight, that is, to give each population equal weight regardless of the number of individuals in the population. This option is only meaningful if DATATYPE = 1 and the data are from populations. If DATATYPE = 0, this option is automatically set to 0.

S (int) Pairwise matrix similarity statistic to be used. Valid choices are 1 or 2: 1 to use the statistic G and 2 to use the statistic G' (see Section 2 for more information about the G and G' statistics).

5.2 Additional parameters for the *Greedy* and *LargeKGreedy* algorithms

If the *Greedy* or *LargeKGreedy* algorithm is chosen (M = 2 or M = 3), the user needs to specify the additional parameter GREEDY_OPTION.

GREEDY_OPTION (int) Input order of runs to be tested. Valid choices are 1, 2 or 3: 1 to test all possible input orders of runs (note that this option increases the run-time substantially unless R is small), 2 to test a specified number of random input orders of runs, and

3 to use a pre-specified input order of runs (see Section 2 for more information on the input order of runs). This option is required if M = 2 or M = 3.

REPEATS (int) The number of input orders of runs to be tested. If GREEDY_OPTION = 2, the parameter REPEATS specifies the number of random input orders of runs that will be tested. If GREEDY_OPTION = 3, REPEATS specifies the number of permutations of runs (= lines) in the *permutationfile*. If GREEDY_OPTION = 1, REPEATS is not expected (and will be ignored even if it is defined; see Section 2 for more information on the input order of runs). This option is required if M = 2 or M = 3 at the same time as GREEDY_OPTION = 2 or GREEDY_OPTION = 3.

PERMUTATIONFILE (string) The name of the *permutationfile* (maximum length of 50 characters) that contains the permutations of the input order of runs to be tested. Note that REPEATS must match the number of input orders of runs to be tested. This option is required if M = 2 or M = 3 together with GREEDY_OPTION = 3.

5.3 Optional outputs

The optional outputs allow the user to print additional results produced by *CLUMPP*.

PRINT_PERMUTED_DATA (int) Print each Q-matrix of the datafile that has been used (either an *indfile* or a *popfile*) with the columns permuted according to the best alignment. Valid choices are 0, 1 or 2: 0 to suppress printing, 1 to print the permuted Q-matrices to one file, and 2 to print each permuted Q-matrix into a separate file (see Section 6.3 for more information on the *permuted_datafile*). This option is always required.

PERMUTED_DATAFILE (string) The name of the *permuted_datafile* (maximum length of 50 characters) where each permuted Q-matrix will be printed. If PRINT_PERMUTED_DATA = 2, a file for each permuted Q-matrix will be created. These files will have an extension of consecutive numbers, "*permuted_datafile*_X", where X ranges from 1 to R. This file is required if PRINT_PERMUTED_DATA = 1 or PRINT_PERMUTED_DATA = 2.

PRINT_EVERY_PERM (boolean) 1 to print every tested permutation of columns of the Q-matrices and the corresponding H-value, 0 to not print this information. Note that turning this option on may result in a very large *every_permfile* (see Section 6.4 for more information on *every_permfile*). This option is always required.

EVERY_PERMFILE (string) The name of the *every_permfile* (maximum length of 50 char-

acters) to print every tested permutation of the columns in the Q-matrices. This file is required if PRINT_EVERY_PERM = 1.

PRINT_RANDOM_INPUTORDER (boolean) 1 to print all random input orders of runs generated by *CLUMPP* if GREEDY_ORDER = 2, 0 to not print this information (see Section 6.5 for more information on *random_inputorderfile*). This option is required if GREEDY_ORDER = 2.

RANDOM_INPUTORDERFILE (string) The name of the *random_inputorderfile* (maximum length of 50 characters) in which to print all random input orders of runs generated by *CLUMPP* if GREEDY_ORDER = 2. This option is required if GREEDY_ORDER = 2 and PRINT_RANDOM_INPUTORDER = 1.

5.4 Advanced options

OVERRIDE_WARNINGS (boolean) 1 to override non-critical warnings, 0 to print non-critical warnings to the screen. These non-critical warnings may require input from the user. This option is always required.

ORDER_BY_RUN (integer) Permute the clusters according to the cluster order of a specific run. Set this parameter to a number r from 1 to R to order the clusters by run r from the INDFILE or POPFILE; set to 0 to not specify a run. When M=1, setting ORDER_BY_RUN to 0 will result in the clusters being ordered by run 1, and when M=2 or 3, the clusters will be ordered by the first run in the first input sequence tested. Setting ORDER_BY_RUN to a nonzero value allows the user to determine how the clusters of the Q-matrices will be ordered after they have been aligned using *CLUMPP*. By reordering the aligned clusters of the runs by the input from one specific run, this option is useful for assessing the consistency of the results of the *Greedy* or *LargeKGreedy* algorithms across different input sequences of runs.

5.5 Command-line arguments

The command-line flags give the user the option to enter information from the command-line. All command-line arguments will overwrite values specified in the *paramfile*. The command-line flag in question is followed by a space and then the parameter-value. All command-line flags and arguments are given after the name of the *paramfile*. The command-line arguments can be given in any order. For example to change the number of clusters to 4, the appropriate command, in Unix, would be:

```
./CLUMPP paramfile -k 4
```

and the appropriate command in Windows would be:
`CLUMPP paramfile -k 4`

-i (INDFILE) Read a different *indfile* from the one specified in *paramfile*.

-p (POPFILE) Read a different *popfile* from the one specified in *paramfile*.

-o (OUTFILE) Print to a different *outfile* from the one specified in *paramfile*.

-j (MISCFILE) Print to a different *miscfile* from the one specified in *paramfile*.

-k (K) Change the number of clusters.

-c (C) Change the number of populations.

-r (R) Change the number of runs.

-m (M) Change the choice of algorithm (1 = *FullSearch*, 2 = *Greedy*, 3 = *LargeKGreedy*).

-w (W) Change the procedure for weighting by the number of individuals (1 = weight by number of individuals, 0 = weight each line equally). If DATATYPE = 0, this option is automatically set to 0.

-s (S) Change the choice of pairwise matrix similarity statistic (1 for G and 2 for G').

6 Output files

CLUMPP will always create two output files, *outfile* and *miscfile*. If the user desires, *CLUMPP* can also print additional output files, *every_permfile*, *random_inputorderfile*, and a *permuted_datafile* (or several such files).

6.1 *outfile*

If DATATYPE = 0, the *outfile* will contain one Q-matrix with $K+5$ columns and C rows. The first five columns are the same as for one of the runs in the *indfile* (these columns are identical for all runs). The second column is the individual identifier (from the *indfile*). Columns 6 to $K+5$ are the mean individual Q-matrix. This individual Q-matrix of *outfile* is computed as the mean over all individual Q-matrices after the columns have been aligned according the permutation with the greatest H-value (that is, the *ck* element of the output Q-matrix is the mean of the *ck* entries of the permuted Q-matrices, appropriately aligned).

Table 7: Example of an *outfile* for population data (DATATYPE = 1), when $C = 95$, $K = 3$ and $R = 9$.

```
  1:  0.3103  0.0022  0.6874  10
  2:  0.4898  0.0095  0.5008  1
  3:  0.0837  0.0060  0.9103  1
  ⋮     ⋮       ⋮       ⋮     ⋮
 94:  0.0032  0.0039  0.9929  1
 95:  0.0049  0.0092  0.9856  1
```

If DATATYPE = 1, the *outfile* will contain one Q-matrix with $K+2$ columns and C rows. The first column is the population identifier (from the *popfile*), the following K columns are the Q-matrix, and the last column indicates the number of individuals in each population (from the *popfile*). This population Q-matrix of *outfile* is computed as the mean over all Q-matrices after the columns have been aligned according to the permutation with the greatest H-value. Table 7 shows an example of an *outfile* of population data when $C = 95$, $K = 3$ and $R = 10$. The program *distruct* can read the *outfile* as an input file.

6.2 *miscfile*

The *miscfile* contains the parameters that are used for a particular run of *CLUMPP*. The program will also print the largest H- or H'-value (equal to SSC_R or SSC'_R for the *FullSearch* algorithm) found and the corresponding permutation of columns for every run. Figure 1 shows an example of a *miscfile*.

```
Using the parameter settings in the file:
''paramfile''

Parameter settings
----------------------- Main parameters -------------------------
DATATYPE = 1
INDFILE =
POPFILE = arabid.popfile
OUTFILE = arabid.outfile
MISCFILE = arabid.miscfile
K = 3
C = 95
R = 9
M = 1
W = 1
S = 2
- Additional options for the Greedy and LargeKGreedy algorithms -
GREEDY_OPTION =
REPEATS =
PERMUTATIONFILE =
----------------------- Optional outputs ------------------------
PRINT_PERMUTED_DATA = 1
PERMUTED_DATAFILE = arabid.perm_datafile
PRINT_EVERY_PERM = 0
EVERY_PERMFILE =
PRINT_RANDOM_INPUTORDER =
RANDOM_INPUTORDER =
----------------------- Advanced options ------------------------
OVERRIDE_WARNINGS = 0
ORDER_BY_RUN = 1


In total, 1679616 configurations of runs and clusters will be tested.

Results
------------------------------
The highest value of H' is: 0.98591755835543

The list of permutations of the clusters that produces that
H' value is (runs are listed sequentially on separate rows)

1 2 3
3 2 1
2 3 1
1 2 3
1 3 2
3 1 2
2 3 1
2 3 1
1 2 3

The pairwise G' values for each pair of runs where the clusters
of each run are permuted according to the list of permutations above

1.0000 0.9697 0.9728 0.9806 0.9751 0.9706 0.9573 0.9731 0.9642
0.9697 1.0000 0.9818 0.9635 0.9749 0.9718 0.9640 0.9648 0.9823
0.9728 0.9818 1.0000 0.9669 0.9771 0.9752 0.9615 0.9664 0.9782
0.9806 0.9635 0.9669 1.0000 0.9729 0.9700 0.9530 0.9745 0.9602
0.9751 0.9749 0.9771 0.9729 1.0000 0.9801 0.9613 0.9738 0.9732
0.9706 0.9718 0.9752 0.9700 0.9801 1.0000 0.9564 0.9651 0.9718
0.9573 0.9640 0.9615 0.9530 0.9613 0.9564 1.0000 0.9559 0.9636
0.9731 0.9648 0.9664 0.9745 0.9738 0.9651 0.9559 1.0000 0.9640
0.9642 0.9823 0.9782 0.9602 0.9732 0.9718 0.9636 0.9640 1.0000
```

Figure 1: Example of a *miscfile*.

6.3 *permuted_datafile*

If PRINT_PERMUTED_DATA = 1, the *permuted_datafile* will contain the Q-matrices from the datafile (*indfile* or *popfile*), with the only difference being that the columns of each Q-matrix are permuted according to the permutation with the largest H-value (Table 8). The order of the Q-matrices is the same as in the datafile. The *permuted_datafile* has the same format as the datafile that was read in by *CLUMPP*, that is, either the *indfile* or the *popfile*.

If PRINT_PERMUTED_DATA = 2, then the permuted Q-matrices will be printed to individual files. The files will be named by adding an underscore and consecutive numbers to the filename defined by parameter EVERY_PERMFILE. These files have the same format as the *outfile* (see Section 6.1) and can be used as input files by the program *distruct*.

Table 8: Example of a *permuted_datafile* for population data (DATATYPE = 1), when $C = 95$, $K = 3$ and $R = 2$.

```
 1:  0.315  0.002  0.683  10
 2:  0.475  0.014  0.511  1
 3:  0.090  0.005  0.905  1
  ⋮      ⋮      ⋮      ⋮   ⋮
94:  0.004  0.003  0.993  1
95:  0.004  0.010  0.985  1

 1:  0.310  0.002  0.687  10
 2:  0.500  0.011  0.490  1
 3:  0.095  0.007  0.898  1
  ⋮      ⋮      ⋮      ⋮   ⋮
94:  0.003  0.004  0.993  1
95:  0.005  0.008  0.987  1
```

6.4 *every_permfile*

Every permutation of the columns of the Q-matrices that is considered by *CLUMPP* can be written to *every_permfile*. The value of H for a particular permutation will be written directly above the permutation. This file can easily become very large for certain parameter settings. The number of permutations (REPEATS) that will be considered is printed on the screen when *CLUMPP* starts to run. Note that if more than 1000 permutations are about to be written, a warning will appear and the user will be given an option to change his/her mind. This warning can be automatically overridden by setting OVERRIDE_WARNINGS to 1. Table 9 shows an example of an *every_permfile*.

Table 9: Example of an *every_permfile* when using the *FullSearch* algorithm ($K = 3$ and $R = 9$) and using the H statistic.

```
-0.322805329280457
1  2  3
1  2  3
1  2  3
1  2  3
1  2  3
1  2  3
1  2  3
1  2  3
1  2  3

-0.245668492940854
1  2  3
1  3  2
1  2  3
1  2  3
1  2  3
1  2  3
1  2  3
1  2  3
1  2  3
   ⋮
0.969019617115145
1  2  3
3  2  1
2  3  1
1  2  3
1  3  2
3  1  2
2  3  1
2  3  1
1  2  3
   ⋮
```

6.5 *random_inputorderfile*

CLUMPP will produce random input orders of runs if the *Greedy* or*LargeKGreedy* algorithms are chosen (M = 2 or M = 3) at the same time as GREEDY_OPTION = 2. Every random order of runs generated by *CLUMPP* can be printed to *random_inputorderfile*. Each line in the *random_inputorderfile* is a permutation of the integers 1, 2, ..., R, indicating the order of the runs in the *indfile* or *popfile* that *CLUMPP* used. The integer in the first position indicates which run was used first, the integer in the second position indicates which run was used second, and so on. The number of input orders will match the parameter REPEATS. It can sometimes be useful to re-run *CLUMPP* with the exact same input orders of runs. This can be done by using the *random_inputorderfile* as the *permutationfile* (and by setting GREEDY_OPTION to 3).

7 Examples

Supplied with the *CLUMPP* software and this manual is a folder containing examples. Below follows a quick run-through of two examples. The first example contains 104 individual *Arabidopsis thaliana* individuals from 95 populations ($C = 95$). The populations have been assigned to 3 clusters in 9 repeated runs of *structure* ($K = 3$ and $R = 9$). The second example is taken from Rosenberg *et al.* (2001), where 600 chickens from 20 breeds were assigned to 19 clusters using *structure*. There are 100 independent *structure* runs in this example.

7.1 Using *CLUMPP* for small numbers of runs and clusters

In the CLUMPP-folder, there are four files, *paramfile*, *arabid.popfile* and *arabid.permutationfile* and *arabid.outfile* (copies of these files can also be found in the folder `example/arabid/`). The *arabid.popfile* contains the results for 95 populations (one population containing 10 individuals and 94 populations containing 1 individual) when the number of clusters is set to 3. Nine independent *structure* runs were used to assign populations to the 3 clusters. These data are taken from Nordborg *et al.* (2005); some changes (9 runs instead of 10 runs, and population 1 contains 10 individuals instead of 1 individual) have been made to the original data in order to make the data more instructional as an example. The *paramfile* is pre-set to use the *FullSearch* algorithm. If we run *CLUMPP* with these initial settings (in Unix: `./CLUMPP arabid.paramfile`; in Windows: `CLUMPP arabid.paramfile`) we will find, after a little while (∼40 seconds on a 2.4 Ghz desktop running Linux), that the highest value of H (= SSC_R) equals 0.969019617... and that the permutation that corresponds to this value is:

1 2 3
3 2 1
2 3 1
1 2 3
1 3 2
3 1 2
2 3 1
2 3 1
1 2 3

This means that, keeping the first run unaltered, the second run will be permuted so that the 1st and 3rd columns switch places and the 2nd column stays in the same place, and so on.

If we change the algorithm to the *Greedy* algorithm (M = 2) and run this algorithm for 10,000 random input orders of runs (GREEDY_OPTION = 2, REPEATS = 10000 in

the file *paramfile*), we find that the exact same permutation has the highest H-value, but the program finishes in ~4 seconds (on the same machine as above). The *arabid.outfile* can be used with the program *distruct* to visualize the results (Fig. 2). To make it easiest to visualize each run independently with *distruct*, set PRINT_PERMUTED_DATA = 2, and the permuted Q-matrices for each run will be printed to the files *arabid.perm_datafile_1* to *arabid.perm_datafile_9* (unless the PERMUTED_DATAFILE option has been changed). If we change the algorithm to the *LargeKGreedy* algorithm (M = 3) and run this algorithm for 10,000 random input orders of runs (GREEDY_OPTION = 2, REPEATS = 10000), we find that the exact same permutation as before has the highest H-value, and the algorithm finishes in ~ 1 second. By changing the options in the *arabid.paramfile*, the user can now explore different settings and output options.

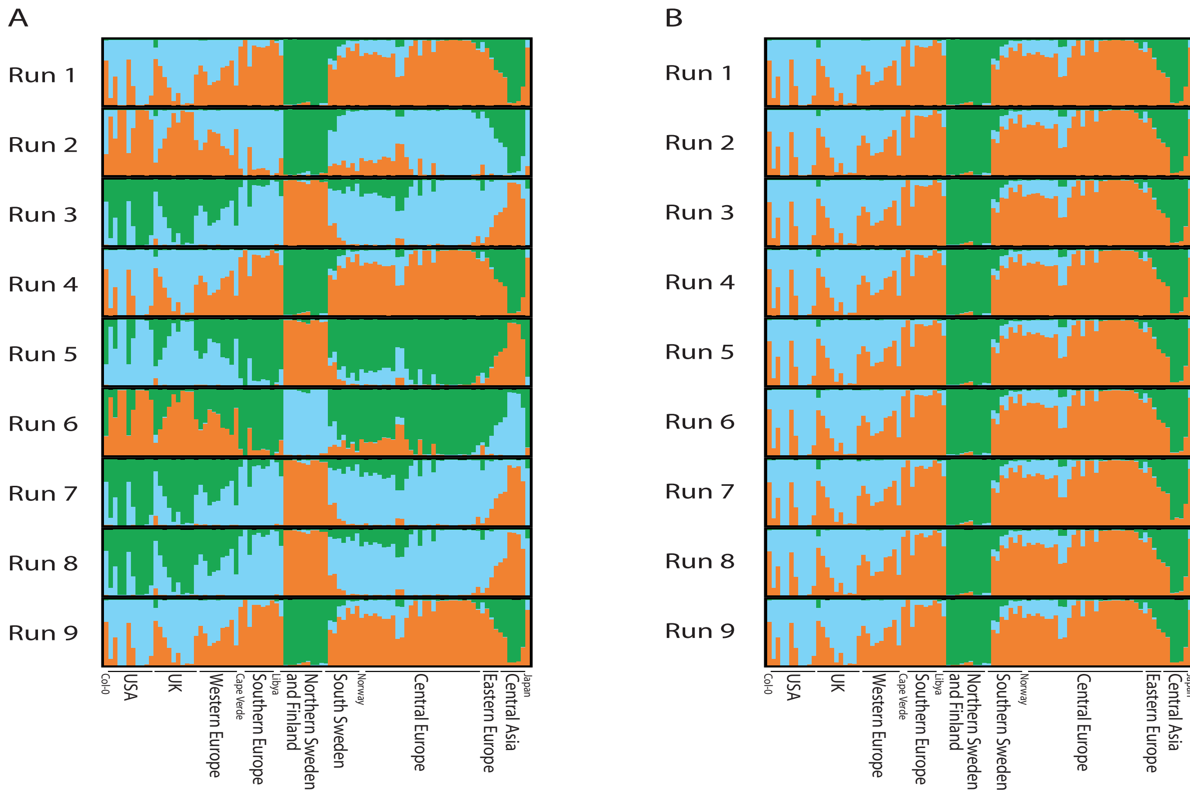

Figure 2: The membership coefficients (Q-matrices) for 95 *Arabidopsis thaliana* individuals, visualized using *distruct*. (A) Membership coefficients from nine independent runs of *structure*. (B) The clusters of each run have been permuted, using *CLUMPP*, to match the configuration in Run 1 (the membership coefficients are the same as in A).

7.2 Using *CLUMPP* for large numbers of runs and clusters

In the folder `example/chicken/` there are two files, *chicken.paramfile* and *chicken.indfile*. The *chicken.indfile* contains the results for 600 chickens when the number of clusters is set

to 19. One hundred independent *structure* runs were used to assign the chickens to the 19 clusters. These data are taken from Rosenberg *et al.* (2001). Figure 3A displays the cluster membership estimates from 25 of these 100 replicates.

The *chicken.paramfile* is pre-set to use the *LargeKGreedy* algorithm and to test 100 random orders of inputting the 100 runs. If we run *CLUMPP* with these initial settings (in Unix: `./CLUMPP chicken.paramfile`; in Windows: `CLUMPP chicken.paramfile`) we find, after a little while (∼5 minutes and 50 seconds on a 2.4 Ghz desktop running Linux), the highest value of H (of 100 random input orders of runs) is likely to be in the range of 0.51 to 0.54 (and the highest value of H' is likely to be between 0.69 and 0.70). To (possibly) get a greater H- or H'-value, we need to increase the number of random input orders (REPEATS) in the *chicken.paramfile*. Running the *LargeKGreedy* algorithm for 30,000 random input orders (options PRINT_EVERY_PERM = 1 and ORDER_BY_RUN = 1), the highest H equals 0.5546. The highest-scoring input sequences tend to produce quite similar alignments of the replicates. Excluding the input sequence that produced the highest H-value, the next 10 highest-scoring input sequences all produced H-values of at least 0.5441, and had on average 2.0% differences compared to the input sequence that led to the highest H. In other words, considering the permuted position of a randomly chosen cluster (among 19) in a randomly chosen run (among 100) based on the output of *CLUMPP* using a randomly chosen input sequence (among the 10 highest-scoring sequences, excluding the one with the highest H-value), the cluster had a 98.0% chance of being aligned in the same way that it was aligned when using the highest-scoring of all input sequences. The permuted membership coefficients of the 25 runs in Figure 3A for the input order among the 30,000 that leads to the highest H are shown in Figure 3B.

When we consider the *LargeKGreedy* algorithm with 10,000 fixed input sequences (chosen randomly among the 30,000 described above), for each of the 10,000 sequences, the alignment of the replicates obtained by *CLUMPP* is identical regardless of which of two statistics — H or H' — is used. The same input sequence that produces the highest H-value ($H = 0.5508$) produces the highest H'-value ($H' = 0.7099$). Excluding the input sequence that produced the highest values of H and H', the next 10 highest-scoring input sequences — the same sequences for both statistics — all produced H-values of at least 0.5392 and H'-values of at least 0.7022, and had on average 7.0% differences compared to the input sequence that led to the highest H and H'. These results suggest that although matrices can be constructed so that the two statistics can lead to different alignments, in practice, their properties are extremely similar. The larger number of differences for the highest-scoring input sequences from among 10,000 sequences compared to the the highest-scoring among the 30,000 sequences described above highlights the importance of employing a large number of input sequences whenever possible.

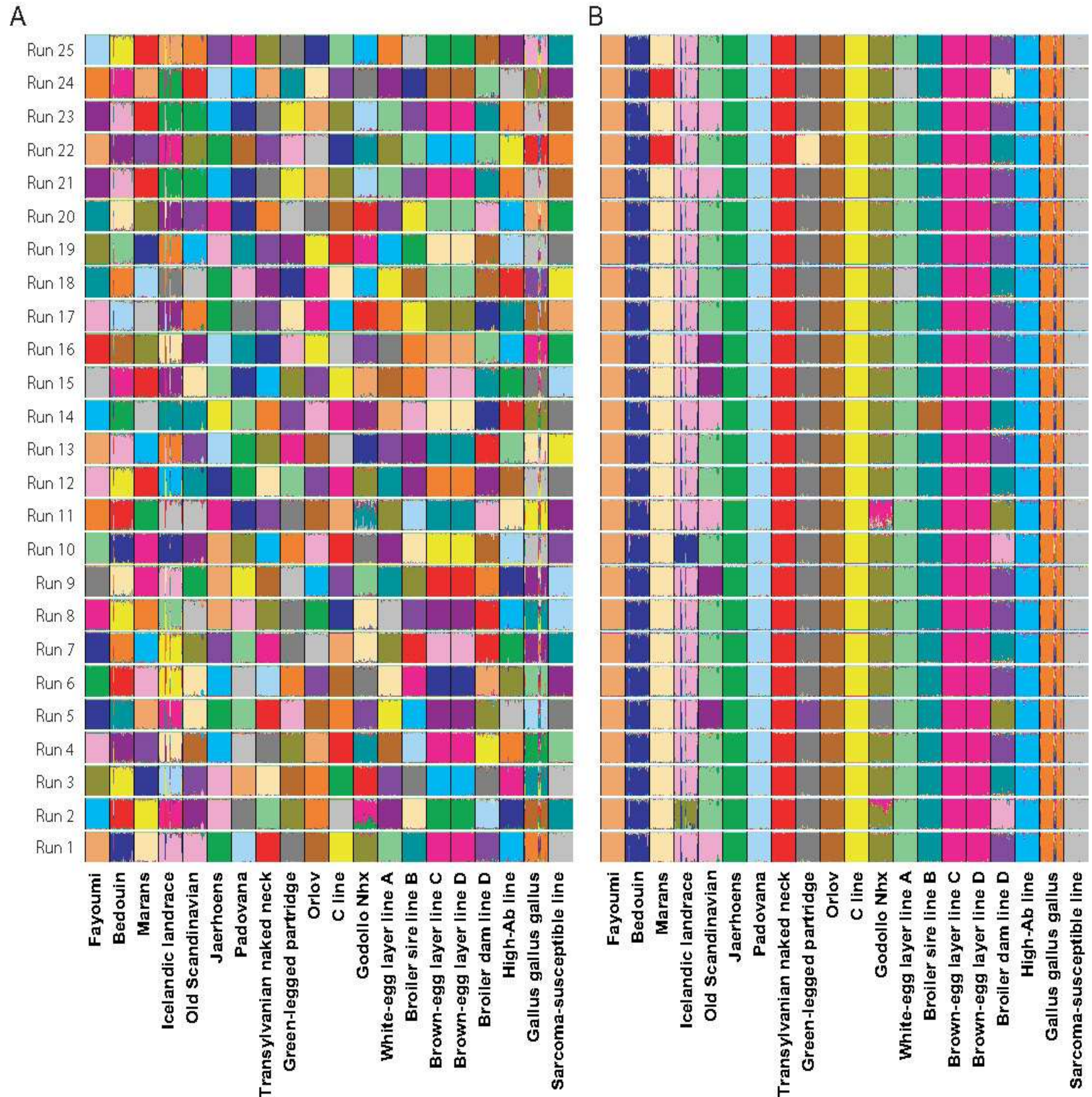

Figure 3: The membership coefficients (Q-matrices) for 600 chickens, pre-*CLUMPP* and post-*CLUMPP*. The matrices are visualized using *DISTRUCT*. (A) Membership coefficients from the first 25 runs of *STRUCTURE* from a total of 100 runs performed by Rosenberg *et al.* (2001). Each individual is represented by a vertical line partitioned into 19 colored segments corresponding to its membership coefficients in 19 clusters. Each color represents a different cluster, and black lines separate the individuals of different breeds. (B) The same membership coefficients as in A, permuted using *CLUMPP* so that cluster labels match across runs.

Acknowledgments

We thank Olivier François and Sijia Wang for testing the beta version of the software.

References

Anderson, E. C. and Thompson, E. A. 2002. A model-based method for identifying species hybrids using multilocus genetic data, *Genetics* **160**, 1217–1229.

Chen, C., Forbes, F., and François, O. 2006. FASTRUCT: model-based clustering made faster, *Mol. Ecol. Notes* **6**, 980–983.

Corander, J. and Marttinen, P. 2006. Bayesian identification of admixture events using multilocus molecular markers, *Mol Ecol* **15**, 2833–2843.

Corander, J., Waldmann, P., Marttinen, P., and Sillanpää, M. J. 2004. BAPS 2: enhanced possibilities for the analysis of genetic population structure, *Bioinformatics* **20**, 2363–2369.

Corander, J., Waldmann, P., and Sillanpää, M. J. 2003. Bayesian analysis of genetic differentiation between populations, *Genetics* **163**, 367–374.

Dawson, K. J. and Belkhir, K. 2001. A Bayesian approach to the identification of panmictic populations and the assignment of individuals, *Genet. Res.* **78**, 59–77.

Falush, D., Stephens, M., and Pritchard, J. K. 2003. Inference of population structure using multilocus genotype data: Linked loci and correlated allele frequencies, *Genetics* **164**, 1567–1587.

François, O., Ancelet, S., and Guillot, G. 2006. Bayesian clustering using hidden Markov random fields in spatial population genetics, *Genetics* **174**, 805–816.

Golub, G. H. and Van Loan, C. F. 1996. "Matrix Computations", Johns Hopkins University Press, Baltimore, 3rd edition.

Jakobsson, M. and Rosenberg, N. A. 2007. *CLUMPP*: a cluster matching and permutation program for dealing with label switching and multimodality in analysis of population structure, *Bioinformatics* in press.

Jasra, A., Holmes, C. C., and Stephens, D. A. 2005. Markov chain Monte Carlo methods and the label switching problem in Bayesian mixture modeling, *Statistical Science* **20**, 50–67.

Nordborg, M., Hu, T. T., Ishino, Y., Jhaveri, J., Toomajian, C., Zheng, H., Bakker, E., Calabrese, P., Gladstone, J., Goyal, R., Jakobsson, M., Kim, S., Morozov, Y., Padhukasahasram, B., Plagnol, V., Rosenberg, N. A., Shah, C., Wall, J. D., Wang, J., Zhao, K., Kalbfleisch, T., Schulz, V., Kreitman, M., and Bergelson, J. 2005. The pattern of polymorphism in *Arabidopsis thaliana*, *PLoS Biol.* **3**, 1289–1299.

Pella, J. and Masuda, M. 2006. The Gibbs and split-merge sampler for population mixture analysis from genetic data with incomplete baselines, *Can. J. Fish. Aquat. Sci.* **63**, 576–596.

Pritchard, J. K., Stephens, M., and Donnelly, P. 2000. Inference of population structure using multilocus genotype data, *Genetics* **155**, 945–959.

Rosenberg, N. A. 2004. *Distruct*: a program for the graphical display of population structure, *Mol. Ecol. Notes* **4**, 137–138.

Rosenberg, N. A., Burke, T., Elo, K., Feldman, M. W., Freidlin, P. J., Groenen, M. A. M., Hillel, J., Mäki-Tanila, A., Tixier-Boichard, M., Vignal, A., Wimmers, K., and Weigend, S. 2001. Empirical evaluation of genetic clustering methods using multilocus genotypes from 20 chicken breeds, *Genetics* **159**, 699–713.

Rosenberg, N. A., Pritchard, J. K., Weber, J. L., Cann, H. M., Kidd, K. K., Zhivotovsky, L. A., and Feldman, M. W. 2002. Genetic structure of human populations, *Science* **298**, 2381–2385.

Stephens, M. 2000. Dealing with label switching in mixture models, *J. R. Stat. Soc. B* **62**, 795–809.