# Vignette for the package *poolfstat* (version 3.1+)

Mathieu Gautier

2026-02-23

## Contents

This vignette describes how the R package *poolfstat* can be used to compute various F-, f- and D-statistics (estimation of $F_{ST}$, hierarchical $F$–statistics, Patterson's $F_2$, $F_3$, $F_3^\star$, $F_4$ and $D$ parameters[1]) in population genomics studies from allele count or Pool-Seq read count data. The package also includes functions to fit and construct admixture graphs to infer the demographic history of populations based on the estimated $f$-statistics along with their visualization. This document is intended as a hands-on tutorial, providing users with an overview of the package's functions with working examples analyzing the Pool-Seq and allele count simulated data sets described in section 1 and publicly available for download from the Zenodo repository[2]. Details and (numerous) references for underlying methods are available in Gautier *et al.* (2022).

The *poolfstat* package is currently available for most platforms (Linux, MS Windows and MacOSX) from the CRAN repository (http://cran.r-project.org/) and can be installed using a standard procedure. Once the package has been successfully installed on your system, it can be loaded by typing:

```
library(poolfstat)
```

# 1 Preamble: presentation of the working example data set

Genetic data were simulated using the coalescent simulator *msprime* (Kelleher *et al.* 2016) for a total of 150 diploid individuals belonging to 6 different populations (n=25 per population) historically related by the admixture graph shown in Figure 1. Each genome consisted of 20 independent chromosomes of $L = 100$ Mb assuming a scaled chromosome-wide recombination rate of $\rho = 4LN_e r = 4,000$[3]. Similarly, the scaled chromosome-wide mutation $\theta = 4LN_e\mu = 4,000$[4]. More specifically, the following *msprime* command was used:

```
mspms 300 20 -t 4000 -I 6 50 50 50 50 50 50 0 -es 0.0125 6 0.25 -ej 0.0125 6 2 -ej 0.0125
7 3 -ej 0.025 2 1 -ej 0.05 3 1 -ej 0.075 5 4 -ej 0.1 4 1 -r 4000 100000000 -p 8
```

The simulation output was further parsed to remove all variants with a Minor Allele Frequency (MAF)[5] less than 1% resulting in a total of 472,410 remaining SNPs (from 23,246 to 24,237 per chromosome) with positional information stored in the file *snp6p.snpinfo.gz*[2]. From the resulting genotyping data, allele counts for both the ancestral and derived alleles (taken as reference) were easily obtained by simple counting[6]. A Pool-Seq data set without sequencing error was subsequently simulated from the allele count data as described in Gautier *et al.* (2022) and implemented in the *sim.readcounts* function (see section 2.4.2). Briefly, for each SNP $i$ in each population $j$, a read count $r_{ij}$ for the reference allele we sampled from a Binomial distribution following $r_{ij} \sim Bin\left(\frac{y_{ij}}{n_j}; c_{ij}\right)$ where $y_{ij}$ is the derived allele count for SNP $i$ in population $j$, $n_j$ is the (haploid) sample size of population $j$ (here $n_j = 50$ for all $j$) and $c_{ij}$ is the total read coverage at SNP $i$ position[7]. Varying total read coverage across pools and SNPs was simulated by sampling the $c_{ij}$'s from a Poisson distribution with mean $\lambda = 30$, i.e., assuming 30X read coverage for the different pools[8]. Two files representative of "real-life" data format were finally created to store the simulated allele count and read count data:

- a file named *sim6p.genotreemix.gz*[2] containing allele count data for each SNP and population in the same format as the one used in the population program *Treemix* (Pickrell and Pritchard 2012)
- a file named *sim6p.poolseq30X.vcf.gz*[2] containing read count data for each SNP and population in a *vcf* format similar to the one obtained with the software *VarScan* (Koboldt *et al.* 2012)

[1] Following Patterson *et al.* (2012), we use $F$ to refer to the parameter and $f$ to the statistic estimated from the data

[2] See http://doi.org/10.5281/zenodo.4709728

[3] For instance, $\rho = 4,000$ if the recombination rate per-base and per-generation is $r = 10^{-8}$ (i.e., the cM per Mb ratio is equal to 1) in a population of constant diploid effective size $N_e = 10^3$

[4] For example, a nucleotide diversity of $\theta = 4,000$ is expected at mutation-drift equilibrium in a population of constant diploid effective size $N_e = 10^3$ if the per-base mutation rate is $\mu = 10^{-8}$

[5] MAF was estimated over all 300 haploid individuals

[6] Since the simulated data are haploid, this implicitly assumes Hardy-Weinberg equilibrium for the different populations

[7] Note that the read count for the alternate allele is simply $c_{ij} - r_{ij}$

[8] Such coverage is actually at the lower limit of what is usually recommended for Pool-Seq experiment

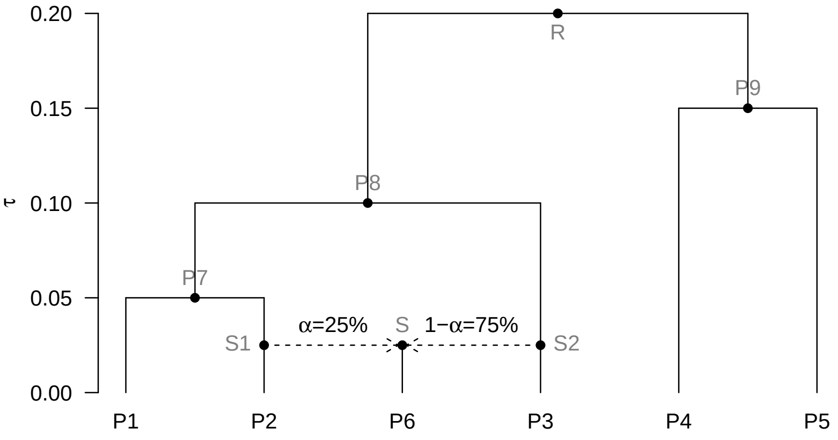

Figure 1: Simulated scenario relating the 6 populations of the working example. Names of the internal node populations for which no data is available are written in grey.

# 2 Reading and manipulating input data

The *poolfstat* package offers several tools for parsing allele count or Pool-Seq read count input data stored in various standard formats. It is important to note that **distinguishing standard allele count data**[9] **from Pool-Seq read count data is crucial** to rely on the appropriate *f*-statistics estimator (Hivert *et al.* 2018; Gautier *et al.* 2022). Therefore, in the *poolfstat* package, there are two different S4 object classes defined for storing data:

- the *countdata* S4 class for storing allele count data. The documentation page outlines the various elements (slots) of the class and is accessible through the *help* function (or *?* operator) as follows:

```
help(countdata)
```

- the *pooldata* S4 class for storing read count data (e.g., PoolSeq data). The documentation page outlines the various elements (slots) of the class and is accessible through the *help* function (or *?* operator) as follows:

```
help(pooldata)
```

These classes characterize the type and origin of the data, and are automatically detected by the *computeFST* (section 3.1.1), *compute.pairwiseFST* (section 3.2) and *compute.fstats* (section 4) functions, implementing different unbiased estimators, which ensure the appropriate estimation procedure is utilized.

## 2.1 Creating a countdata object for allele count data

A *countdata* object can be created from allele count data stored in two different input file formats:

- The input file format needed for the well-known *Treemix* program (Pickrell and Pritchard 2012) using the *genotreemix2countdata* function
- The input file format required by *BayPass* program (Gautier 2015) for allele count data using the function *genobaypass2countdata* function

[9] i.e., obtained from individual genotyping data

The following example demonstrates how to create a *countdata* object (here named *sim6p.allelecount*) from the *sim6p.genotreemix.gz* example file. This file contains the allele count data (in *Treemix* file format) for the simulated example data outlined in section 1:

```
sim6p.allelecount<-genotreemix2countdata(genotreemix.file = "sim6p.genotreemix.gz")
```

Information on marker position (chromosome or scaffold of origin of the markers and position on the chromosome) can be provided using the *snp.pos* argument in the *genotreemix2countdata* and *genobaypass2countdata* functions. For example, to incorporate the mapping information stored in the *snp6p.snpinfo.gz* example file (section 1), use the following commands[10]:

```
positions<-read.table("sim6p.snpinfo.gz",header=TRUE,row.names=1,stringsAsFactors = FALSE)
sim6p.allelecount<-genotreemix2countdata(genotreemix.file = "sim6p.genotreemix.gz",
                                         snp.pos=positions,verbose=FALSE)
sim6p.allelecount #display a summary of the object
```

```
 * * * Countdata Object * * *
 * Number of SNPs  =  472410
 * Number of Pops  =  6
 * Pop Names       :
 P1; P2; P3; P4; P5; P6
 * * * * * * * * * * * * * *
```

**Genomic ordering of SNP markers**

For operations that necessitate marker position information[a], markers are consistently presumed to be ordered in the genome by their relative position in the input files. If no map information is provided, a default position is assigned to all SNPs assuming they all map to the same chromosome. In such instances, some windows or blocks may extend over two consecutive contigs[b].

[a] e.g., block-jackknife estimation of standard errors (Appendix A.1); or estimation of multi-locus statistics on sliding windows throughout the genome (e.g., section 3.1.1)

[b] If the reference assembly used to order the markers is not too fragmented, the block-jackknife estimates of standard errors may only be slightly affected since the blocks are defined by a number of consecutive markers rather than a physical length

Additional arguments may permit filtering the data for low marker polymorphism levels (*min.maf*) or genotyping call rate (*min.indgeno.per.pop*). Further information is available in the documentation pages of the *genotreemix2countdata* and *genobaypass2countdata* functions accessible with the commands `?genotreemix2countdata` and `?genobaypasstreemix2countdata`.

## 2.2 Creating a pooldata object for Pool-Seq read count data

A **pooldata** object can be created from Pool-Seq read count data stored in any of the four formats listed below:

- *vcf* file generated by most of the SNP calling software commonly used to analyze Pool-Seq data including *VarScan* (Koboldt *et al.* 2012), *bcftools/SAMtools* (Li *et al.* 2009), *GATK* (McKenna *et al.* 2010) or *FreeBayes* (Garrison and Marth 2012) using the *vcf2pooldata* function[11]
- *rsync* files generated by the *PoPoolation* software (Kofler *et al.* 2011) using the *popsync2pooldata* function
- The two input files (pool read count and pool haploid sizes) required by the program *BayPass* (Gautier 2015) to analyze Pool-Seq data using the *genobaypass2pooldata* function

[10] If the number of markers in the *snp.pos* object does not match the allele count file or the matrix given is not in 2 columns, default values for marker positions are provided and a warning message is printed in the console

[11] The *vcf* file's format is determined automatically from the genotype format field that includes i) both an AD and RD fields for *VarScan vcf* files; or ii) only an AD field (with comma-separated read counts for the different allele) other than *VarScan vcf*. Parsing of *vcf* files has been substantially improved since *poolfstat* version *1.2* with computationally intensive text manipulation now implemented in *C++* routines inspired by those of the *vcfR* package (Knaus and Grünwald 2017)

- The input file required by the *SelEstim* program (Vitalis *et al.* 2014) to analyze Pool-Seq data using the *genoselestim2pooldata* function

The following example shows how to create a *pooldata* object for the simulated Pool-Seq data present in the *sim6p.poolseq30X.vcf.gz* vcf file (section 1).

```
sim6p.readcount30X <-vcf2pooldata(vcf.file="sim6p.poolseq30X.vcf.gz",poolsizes=rep(50,6),
                                  poolnames = paste0("Pool",1:6))
```

```
**********************
Reading Header lines
  *VarScan like format detected for allele count data:
  the AD and RD fields contain allele depth for the alternate and reference alleles, respectively
  (N.B., positions with more than one alternate allele will be ignored)
**********************
Parsing allele counts
472410  lines processed in 0 h  0 m  7 s : 472410 SNPs found
Data consists of 472410 SNPs for 6 Pools
```

```
sim6p.readcount30X  #display a summary of the resulting pooldata object
```

```
 * * * PoolData Object * * *
 * Number of SNPs   =  472410
 * Number of Pools  =  6
 * Pool Names       :
 Pool1; Pool2; Pool3; Pool4; Pool5; Pool6
 * * * * * * * * * * * * * *
```

As illustrated in the example below, as of version *3.1* of the package, pool names can be inferred directly from the *vcf* header when no pool sample names are supplied to *vcf2pooldata* (i.e., the poolnames argument is left at its default value, *NA*). Similarly, if no (haploid) pool sample sizes are provided (*poolsizes* left at its default *NA*), all pools are assigned a default (haploid) sample size of 100. These default values can be modified after the *pooldata* object is created by updating its *poolnames* and/or *poolsizes* slots:

```
sim6p.readcount30X <-vcf2pooldata(vcf.file="sim6p.poolseq30X.vcf.gz")
```

```
**********************
Reading Header lines
  *The following sample names for the 6 pools were retrieved from the vcf:
[1] "P1" "P2" "P3" "P4" "P5" "P6"
  *Warning: No pool sizes available from poolsizes argument
  => the (haploid) pool size were set to an arbitrary default value of 100 for the 6 pools included in the vcf.
  !!!!Don't forget to update the slot @poolsizes of the output pooldata object if needed!!!!
  *VarScan like format detected for allele count data:
  the AD and RD fields contain allele depth for the alternate and reference alleles, respectively
  (N.B., positions with more than one alternate allele will be ignored)
**********************
Parsing allele counts
472410  lines processed in 0 h  0 m  7 s : 472410 SNPs found
Data consists of 472410 SNPs for 6 Pools
```

```
sim6p.readcount30X@poolnames = paste0("Pool",1:6)
sim6p.readcount30X@poolsizes = rep(50,6)
sim6p.readcount30X
```

```
 * * * PoolData Object * * *
 * Number of SNPs   =  472410
 * Number of Pools  =  6
 * Pool Names       :
 Pool1; Pool2; Pool3; Pool4; Pool5; Pool6
 * * * * * * * * * * * * * *
```

Additional arguments may allow filtering the data according to the read coverage of the pool such as *min.maf*, *min.cov*, *max.cov*, *cov.qthres*, *min.rc*, *min.cov.per.pool* or *max.cov.per.pool*. Additional details are available in the documentation pages of each function, which can be accessed though the commands `?vcf2pooldata`, `?popsync2pooldata`, `?genobaypass2pooldata` and `?genoselestim2pooldata` respectively.

## 2.3 Manipulating *countdata* and *pooldata* objects

### 2.3.1 The *pooldata.subset* and *countdata.subset* functions

The *pooldata.subset* function (respectively *countdata.subset*) enable the retrieval of particular portions of the original *pooldata* (resp. *countdata*) object (e.g., some SNPs and/or population samples) according to various criteria (see `?pooldata.subset` or `?countdata.subset` for more details). In the example below, a *pooldata* object is generated from the *sim6p.readcount* object previously created, which only includes data for populations *P2*, *P3* and *P6* for SNP with a MAF>0.05:

```
sim6p.readcount30X.subset<-pooldata.subset(sim6p.readcount30X,pool.index=c(2,3,6),
                                          min.maf=0.05,verbose=FALSE)
sim6p.readcount30X.subset #display a summary of the resulting pooldata object
```

```
 * * * PoolData Object * * *
 * Number of SNPs   =  241280
 * Number of Pools  =  3
 * Pool Names       :
 Pool2; Pool3; Pool6
 * * * * * * * * * * * * * *
```

Note that indexes of the retained SNPs from the original data set can be obtained by setting the *return.snp.idx* argument to TRUE within the *pooldata.subset* or *countdata.subset* functions. If this is done, the rows of the matrix stored in the *snpinfo* slot of the *pooldata* or *countdata* output objects are named *"rs"snp.idx* (where *snp.idx* is the SNP index for the original object) making it easy to obtain the indexes of the selected SNPs as demonstrated below:

```
sim6p.readcount30X.subset<-pooldata.subset(sim6p.readcount30X,pool.index=c(2,3,6),
                                          min.maf=0.05,return.snp.idx=TRUE,verbose=FALSE)
selected.snps.idx <- as.numeric(sub("rs","",rownames(sim6p.readcount30X.subset@snp.info)))
head(selected.snps.idx)
```

```
[1] 1 3 4 5 6 8
```

For *pooldata* objects, the *pooldata2genobaypass*, *pooldata2selestim* and *pooldata2diyabc* functions can be used to generate input files for the aforementioned *BayPass* and *SelEstim* programs and *DIYABC* software (Collin *et al.* 2021). These functions also allow for the creation of sub-samples from the original data (see `?pooldata2genobaypass` and `?pooldata2selestim` for additional information).

### 2.3.2 Merging *pooldata* and/or *countdata* objects with *data.merge*

The *data.merge* function allows the combination of two *pooldata* and/or *countdata* objects into a single data object, while automatically checking structural consistency between inputs. Depending on the relationship between SNP identifiers and (pool or population) sample names, *data.merge* can either i) merge distinct samples with genetic information for the same set of SNPs; or ii) merge distinct SNP data sets available for the same samples. The function returns a *countdata* object when two *countdata* object are merged, and a *pooldata* object when two *pooldata* objects are merged.

It is also possible to merge a *countdata* object with a *pooldata* object. In this case the function always returns a *pooldata* object: allele counts from the *countdata* object are converted into pseudo read counts, and the corresponding haploid sample size is set to the value specified by the *fake.pool.size* argument (default = 1e6). Using a large value for this parameter ensures that each read count is effectively treated as originating from a distinct haploid individual, thereby mimicking Pool-Seq data and disabling Pool-Seq-specific bias corrections in downstream analyses. Several typical use cases of *data.merge* are illustrated in the examples provided in the documentation pages of the function `?data.merge`.

## 2.4 Generating *pooldata* objects

### 2.4.1 Creating a (pseudo-)pooled sample with *poolify*

The *poolify* function generates a single "poolified" sample from samples contained in a *pooldata* or *countdata* object. The output is always a *countdata* object containing the combined "poolified" sample, that can be used to simulate Pool-Seq data (e.g., using *sim.readcounts*, see 2.4.2) or merged with other samples (e.g., using *data.merge*, see 2.3.2).

The behavior of the function depends on the type of input data. If the input is a *countdata* object, the output consists of the sum of allele counts across the selected samples (specified via the *sample.index* argument). If the input is a *pooldata* object, a random-allele sampling strategy is applied: for each SNP, one read is randomly sampled per selected sample. This downsampling ensures that all reads in the resulting pooled sample originate from distinct sample chromosomes.

Several typical use cases of *poolify* are illustrated in the examples provided in the documentation pages of the function `?poolify`.

### 2.4.2 Simulating Pool-Seq read count data from a count data object with *sim.readcounts*

The *sim.readcounts* function allows simulation read count data in the form of a *pooldata* object (see section 2.2) from read count data (stored in *countdata* object, see section 2.1). It implements the simulation approach described in Gautier *et al.* (2024) that combines simulation procedures previously described in Hivert *et al.* (2018) and Gautier *et al.* (2022). Briefly, read coverages for each and every marker position and within each pool from a distribution specified by the *lambda.cov* vector specifying the expected coverages $\lambda_p$ of each pool $p$, and the *overdisp* scalar specifying the overdispersion $\alpha$ of marker coverages within each pool. In other words, the overdispersion of SNP coverages is expected to the same magnitude for all the pools but the expected coverages are allowed to vary across pools. If *overdisp=1* (default), coverages are assumed Poisson distributed with mean (and variance) equal to the value $\lambda_p$ specified in the *lambda.cov* vector. If *overdisp>1*, coverages follow a Negative Binomial distribution with mean equal to $\lambda_p$ and variance equal to $\alpha \times \lambda_p$. Finally, if *overdisp<1*, no variation in coverage is introduced and all coverages are equal to the value specified in the lambda vector although they may (slightly) vary in the output when *seq.eps* $> 0$ due to the removal of some error reads. Indeed, the *seq.eps* parameter control the rate $\epsilon_{seq}$ of sequencing errors, which are modeled following Gautier *et al.* (2022) i.e., the vector of $\boldsymbol{r}$ read counts for the four possible bases are sampled from a multinomial distribution:

$$\boldsymbol{r} \sim Mult\left(c; \left\{(1-\epsilon_{seq})f + \frac{1}{3}\epsilon_{seq}(1-f); \frac{1}{3}\epsilon_{seq}f + (1-\epsilon_{seq})(1-f); \frac{1}{3}\epsilon_{seq}; \frac{1}{3}\epsilon_{seq}\right\}\right)$$

where $c$ is the read coverage and $f$ the reference allele frequencies (obtained from the count data). When $\epsilon_{seq} > 0$, spurious SNPs may be generated even at monomorphic positions. The *genome.size* argument giving the size of the genome $G$ in bp allows providing a number of monomorphic positions which is equal to $G - n_{snp}$ where $n_{snp}$ is obtained from the input *countdata* object). The spurious SNPs are then simulated using the same error model i.e. sampling read counts $\boldsymbol{r}$ for the four possible bases (the reference allele count corresponding to the first element of the output vector) from:

$$\boldsymbol{r} \sim Mult\left(c; \left\{1-\epsilon_{seq}; \frac{1}{3}\epsilon_{seq}; \frac{1}{3}\epsilon_{seq}; \frac{1}{3}\epsilon_{seq}\right\}\right)$$

Only bi-allelic SNPs passing filtering conditions specified by *min.rc* (which controls the minimal read count for an allele to be deemed as true, i.e. if more than two alleles have $\geq$*min.rc* counts, the SNP is excluded because non-bi–allelic) and *maf.thr* (threshold on the major allele frequency computed over all read counts) are included in the output. Finally, experimental error *exp.eps* control the contribution of individual (assumed diploid) to the pools following the model described in Gautier *et al.* (2013), where the parameter *exp.eps* corresponds to the coefficient of variation of the individual contributions[12]. When *exp.eps=0* (default), all individuals are assumed to equally contribute (on average) to the pool sequences (i.e., there is no experimental error).

The examples below shows the simulation of a Pool-Seq data set from the *sim6p.allelecount* object containing allele count data for different simulation designs with increasing complexity:

- with varying expected coverages for the different pools (from 50X to 100X) which result in estimated $F_{ST}$ (see 3) similar to those obtained with the original allele count data set:

```
sim<-sim.readcounts(sim6p.allelecount,lambda.cov = seq(50,100,10))
```

```
Start simulation of Read counts for the 472410 polymorphic SNPs included in the count data object
Simulation ended: 470708 SNPs retained
Simulation finished: 470708 retained in the output pooldata object
```

```
computeFST(sim,nsnp.per.bjack.block = 1000,verbose=FALSE)$Fst
```

```
    Estimate   bjack mean   bjack s.e.      CI95inf      CI95sup
0.1355991675 0.1357707210 0.0008061717 0.1341906245 0.1373508175
```

- with varying expected coverages for the different pools (from 50X to 100X) and increased overdispersion of the coverages which also result in estimated $F_{ST}$ similar to those obtained with the original allele count data set:

```
sim<-sim.readcounts(sim6p.allelecount,lambda.cov = seq(50,100,10),overdisp = 2)
```

```
Start simulation of Read counts for the 472410 polymorphic SNPs included in the count data object
Simulation ended: 470423 SNPs retained
Simulation finished: 470423 retained in the output pooldata object
```

```
computeFST(sim,nsnp.per.bjack.block = 1000,verbose=FALSE)$Fst
```

```
    Estimate   bjack mean   bjack s.e.      CI95inf      CI95sup
0.1355383256 0.1357310926 0.0008071495 0.1341490796 0.1373131055
```

- with varying expected coverages for the different pools (from 50X to 100X) and an experimental error (i.e., unequal contribution of individuals to their pool sequences) of 50%, which leads to a slight inflation of the estimates:

```
sim<-sim.readcounts(sim6p.allelecount,lambda.cov = seq(50,100,10),exp.eps = 0.5)
```

```
Start simulation of Read counts for the 472410 polymorphic SNPs included in the count data object
Simulation ended: 469521 SNPs retained
Simulation finished: 469521 retained in the output pooldata object
```

```
computeFST(sim,nsnp.per.bjack.block = 1000,verbose=FALSE)$Fst
```

```
    Estimate   bjack mean   bjack s.e.      CI95inf      CI95sup
0.1399991031 0.1401787893 0.0008044759 0.1386020165 0.1417555620
```

- with varying expected coverages for the different pools (from 50X to 100X) and a sequencing error rate of 0.1% with a genome size set to 2 Gb (see section 1) to account for spurious SNPs generated by errors (at monomorphic positions). Note that running time are significantly increased (ca. linearly with the genome size). In addition, the default SNP filtering options then leads to several tens of millions of (spurious) SNPs with very low polymorphism levels. As a result the estimated $F_{ST}$ is almost null:

```
sim<-sim.readcounts(sim6p.allelecount,lambda.cov = seq(50,100,10),
                                           seq.eps = 1e-3,genome.size = 2e9)
```

[12] For example, with 10 individuals, *exp.eps=0.5* correspond to a situation where the 5 most contributing individuals contribute $> 2$ times reads than the others when $\lambda = 100$.

```
Start simulation of Read counts for the 472410 polymorphic SNPs included in the count data object
Simulation ended: 470377 SNPs retained
Start simulation of spurious SNPs for 1999529623 monomorphic positions
Simulation of spurious SNPs ended: 59863011 SNPs retained
Simulation finished: 60333388 retained in the output pooldata object
```

```
computeFST(sim,nsnp.per.bjack.block = 1000,verbose=FALSE)$Fst
```

```
    Estimate   bjack mean   bjack s.e.      CI95inf      CI95sup
8.567755e-04 7.870962e-04 6.875827e-05 6.523300e-04 9.218624e-04
```

```
rm(sim)
```

- same as above but with increasing *maf.thr* to 1% to limit the number of these spurious SNPs (this could also have been done with *pooldata.subset* function applied to the previous output object), which allows removing most of the bias introduced by spurious SNPs:

```
sim<-sim.readcounts(sim6p.allelecount,lambda.cov = seq(50,100,10),
                                         seq.eps = 1e-3,genome.size = 2e9,maf.thr = 0.01)
```

```
Start simulation of Read counts for the 472410 polymorphic SNPs included in the count data object
Simulation ended: 449342 SNPs retained
Start simulation of spurious SNPs for 1999550658 monomorphic positions
Simulation of spurious SNPs ended: 124295 SNPs retained
Simulation finished: 573637 retained in the output pooldata object
```

```
computeFST(sim,nsnp.per.bjack.block = 1000,verbose=FALSE)$Fst
```

```
    Estimate   bjack mean   bjack s.e.      CI95inf      CI95sup
0.1316023007 0.1317320065 0.0007843292 0.1301947214 0.1332692917
```

## 2.5 First exploratory analyses of the overall genetic structure

### 2.5.1 Principal Component Analysis using *randomallele.pca*:

The *randomallele.pca* implements a Principal Component Analysis on allele count data (stored in *countdata* objects, see section 2.1) or Pool-Seq read count data (stored in *pooldata* objects, see section 2.2) to provide an overall visualization of the genetic structuring of (population) samples. To account for possible unequal sample size and read coverages (for pool-seq data), the PCA is performed (via singular-value decomposition) on a *npop* (or *npools*) x *nsnp* matrix of a single randomly sampled allele (or read for pooldata object) for each SNP and for each population[13]. This procedure was inspired by Skoglund and Jakobsson (2011) and is similar to that implemented in the *PCA_MDS* module of the software *ANGSD* (Korneliussen *et al.* 2014).

Figure 2 illustrates PCA of the allele and read count simulated data sets using *randomallele.pca*. Note that the resulting objects contains loadings for the different PCA allowing to draw custom plots:

```
tmp=cbind(matrix(rep(c(1,3),each=2),4,2),matrix(rep(c(1,4),each=2),4,1),
          matrix(rep(c(2,4),each=2),4,1),matrix(rep(c(2,5),each=2),4,2))
layout(tmp)
#PCA on the count data (the object)
sim6p.allelecount.pca=randomallele.pca(sim6p.allelecount,col=1:6,pch=16,main="Allele Count data")
#PCA on the read count data (the object)
sim6p.readcount30X.pca=randomallele.pca(sim6p.readcount30X,col=1:6,pch=16,main="Read Count data")
#plotting PC1 and PC3; PC1 and PC4; and PC1 and PC5
#(using the sim6p.readcount30X.pca information to avoid
#rerunning randomallele.pca with plot.pcs=c(1,3)....)
i=1
for(j in 3:5){
```

[13] The resulting information loss has generally (very) little impact on the resulting representation as the number of SNPs is usually very high

```
plot(sim6p.readcount30X.pca$pop.loadings[,i],sim6p.readcount30X.pca$pop.loadings[,j],
     xlab=paste0("PC",i," (",round(sim6p.readcount30X.pca$perc.var[i],2),"%)"),
     ylab=paste0("PC",j," (",round(sim6p.readcount30X.pca$perc.var[j],2),"%)"),
     col=1:6,pch=16,main="Read Count data")
    text(sim6p.readcount30X.pca$pop.loadings[,i],
         sim6p.readcount30X.pca$pop.loadings[,j],sim6p.readcount30X@poolnames)
    abline(h=0,lty=2,col="grey") ; abline(v=0,lty=2,col="grey")
}
```

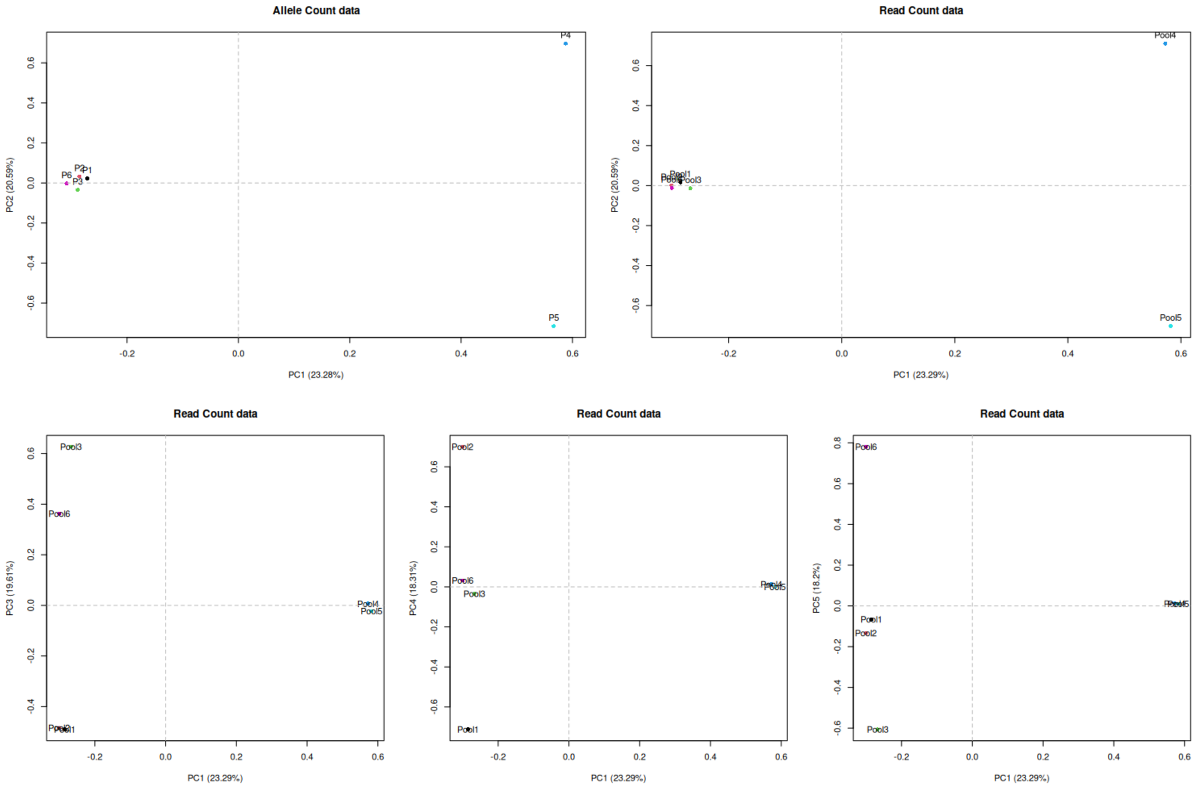

Figure 2: Principal Component Analyses (using random allele sampling) for the allele count and 30X Pool-Seq data sets

As expected, very similar representations are obtained with the allele or read count data. Also, as expected from the simulation scenario (Figure 1), the first axis separates the two outgroup populations $P4$ and $P5$ from the other populations, the second axis separates $P4$ and $P5$; etc.

# 3 Estimating $F_{ST}$

The $F_{ST}$ parameter is commonly used to quantify the level of structuring of genetic diversity among populations (see e.g. Hivert *et al.* 2018 and references therein). It may be defined as:

$$F_{ST} \equiv \frac{Q_1 - Q_2}{1 - Q_2}$$

where $Q_1$ is the Identity In State (IIS) probability for genes sampled within populations (or pools), and $Q_2$ is the IIS probability for genes sampled between populations (or pools).

The *computeFST* and *compute.pairwiseFST* (for all pairs of populations) functions implement two distinct $F_{ST}$ estimators relying on:

- a decomposition of the total variance of allele or read count frequencies in an analysis-of-variance framework (Weir and Cockerham 1984) which is the default procedure of the functions (as specified by the argument *method="Anova"*). The implemented estimators are derived in Weir (1996) (eq. 5.2) (see also Akey *et al.* 2002) for allele count data (i.e., *countdata* objects, see 2.1); and in Hivert *et al.* (2018) (eq. 9) for (Pool-Seq) read count data (i.e., *pooldata* objects, see 2.2).
- unbiased estimators $\widehat{Q_1}$ and $\widehat{Q_2}$ of the IIS probabilities $Q_1$ and $Q_2$ (as specified by the *method="Identity"* argument). For allele count data (i.e., *countdata* objects, see 2.1) this estimator actually correspond to the one used by Karlsson *et al.* (2007). For Pool-Seq read count data (i.e., *pooldata* objects, see 2.2), equations A39 and A43 in Hivert *et al.* (2018) Supplementary Materials describe the estimators for $\widehat{Q_1}$ of the $\widehat{Q_2}$ respectively. By default, when using *method=Identity*, the overall $\widehat{Q_1}$ and pairwise $\widehat{Q_2}$ are computed as simple averages of all population-specific $\widehat{Q_1}$ and pairwise population $\widehat{Q_2}$, respectively. For completion, when setting *weightpid=TRUE*, an alternative weighting scheme is performed, as described in eqs. A46 and A47 of Hivert *et al.* (2018) for PoolSeq data, and Rousset (2007) for allele count data.

Note that multi-locus estimates (i.e., genome-wide estimates or sliding windows estimates) are derived as the sum of locus-specific numerators over the sum of locus-specific denominators of the different quantities (see, e.g., Hivert *et al.* 2018 and references therein).

## 3.1 Estimating genome-wide $F_{ST}$ across all the populations

### 3.1.1 The *computeFST* function

The *computeFST* function automatically uses the appropriate estimator given the input object class (either allele count for *countdata* objects or Pool-Seq read count data for *pooldata* objects). For example with the simulated example data, we obtain the following estimates of $F_{ST}$ with:

- allele count data:

```
sim6p.allelecount.fst<-computeFST(sim6p.allelecount)
```

```
Computation of Fst (Hivert et al., 2018)
Computing SNP-specific Q1 and Q2 (Anova estimator)
```

```
sim6p.allelecount.fst$Fst      #genome-wide Fst over all populations
```

```
  Estimate bjack mean bjack s.e.     CI95inf     CI95sup
  0.132319          NA          NA          NA          NA
```

- Pool-Seq read count data:

```
sim6p.readcount30X.fst<-computeFST(sim6p.readcount30X,verbose=FALSE)
sim6p.readcount30X.fst$Fst      #genome-wide Fst over all populations
```

```
  Estimate bjack mean bjack s.e.     CI95inf     CI95sup
 0.1324199          NA          NA          NA          NA
```

Note that the *computeFST* function defaults to using the *Anova* method, but this can be modified using the *method* argument (as explained in section 3).

### 3.1.2 Block-Jackknife estimation of $\hat{F}_{ST}$ standard-error and confidence intervals:

Standard error of the $F_{ST}$ estimates can be estimated using a block-jackknife sampling approach (see Appendix A.1). To specify the number of consecutive SNPs defining a block, use the argument *nsnp.per.bjack.block* (the default value is *nsnp.per.bjack.block=0*, which means that no block-jackknife is carried out). The resulting genome-wide $F_{ST}$ estimated as the mean over block-jackknife samples, the block-jackknife standard error and the 95% confidence interval (i.e., block-jackknife mean ±1.96 s.e.) are then given in the *Fst* element of the output list. An example is illustrated below for:

- allele count data:

```
sim6p.allelecount.fst<-computeFST(sim6p.allelecount,nsnp.per.bjack.block = 1000)
```

```
Computation of Fst (Hivert et al., 2018)
Computing SNP-specific Q1 and Q2 (Anova estimator)
Starting Block-Jackknife sampling
462 Jackknife blocks identified with 462000 SNPs (out of 472410 ).
 SNPs map to 20 different chrom/scaffolds
Average (min-max) Block Sizes: 4.232 ( 3.515 - 4.975 ) Mb
```

```
sim6p.allelecount.fst$Fst
```

```
    Estimate   bjack mean   bjack s.e.      CI95inf      CI95sup
0.1323189779 0.1324888937 0.0007603234 0.1309986599 0.1339791276
```

Note that the block-jackknife mean may slightly differ from the default genome-wide estimate of $F_{ST}$ as it is only computed from the SNPs eligible for block-jackknife (see Appendix A.1).

- Pool-Seq read count data:

```
sim6p.readcount30X.fst<-computeFST(sim6p.readcount30X,nsnp.per.bjack.block = 1000,verbose=FALSE)
sim6p.readcount30X.fst$Fst
```

```
    Estimate   bjack mean   bjack s.e.      CI95inf      CI95sup
0.1324199011 0.1326089168 0.0007620463 0.1311153061 0.1341025275
```

### 3.1.3 Computing multi-locus $F_{ST}$ to scan the genome over sliding-windows of SNPs

The *sliding.window.size* argument allows computing multi-locus $F_{ST}$ for sliding windows over the different chromosomes (or scaffolds/contigs), e.g., to carry out genome-scans for adaptive differentiation. Each sliding window includes a number of consecutive SNPs specified by the *sliding.window.size* argument. This is illustrated below for the Pool-Seq read count example data (similar results would be obtained with allele count data):

```
sim6p.readcount30X.fst<-computeFST(sim6p.readcount30X,sliding.window.size=50)
```

```
Computation of Fst (Hivert et al., 2018)
Computing SNP-specific Q1 and Q2 (Anova estimator)
Start sliding-window scan
20 chromosomes scanned (with more than 50 SNPs)

Average (min-max) Window Sizes 207.4 ( 90.1 - 453.2 ) kb
```

```
plot(sim6p.readcount30X.fst$sliding.windows.fvalues$CumMidPos/1e6,
     sim6p.readcount30X.fst$sliding.windows.fvalues$MultiLocusFst,
     xlab="Cumulated Position (in Mb)",ylab="Muli-locus Fst",
     col=as.numeric(sim6p.readcount30X.fst$sliding.windows.fvalues$Chr),pch=16)
abline(h=sim6p.readcount30X.fst$Fst,lty=2)
```

No discernible signal of adaptive differentiation, such as an excessively differentiated tower of windows, is apparent. This was expected, as the data set was simulated under neutrality (see Figure 3).

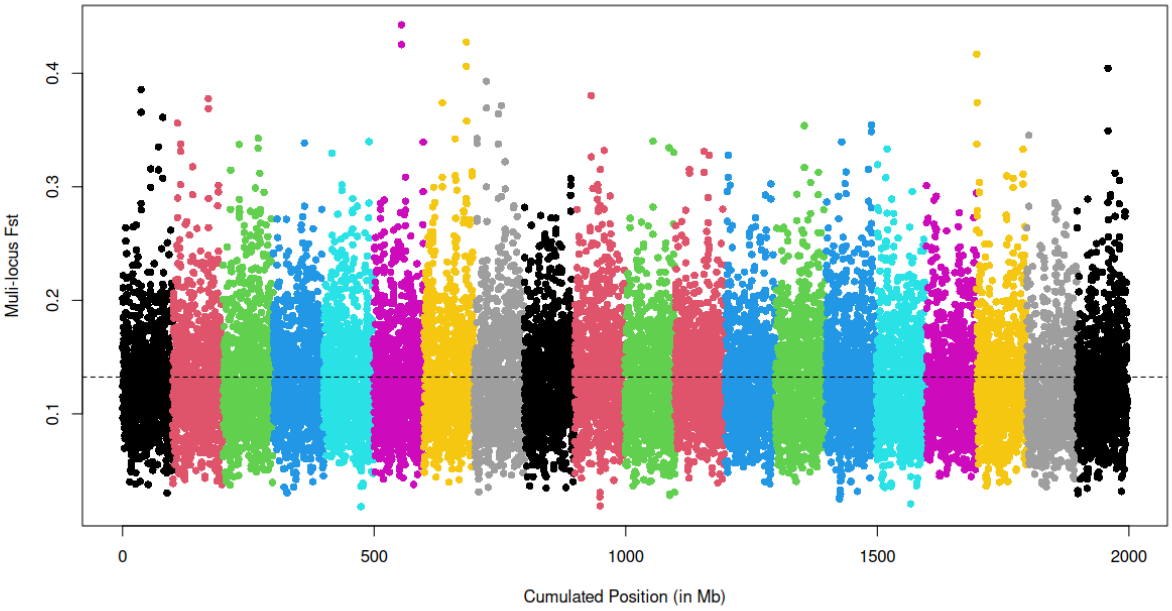

Figure 3: Manhattan plot of the multi-locus $F_{ST}$ computed over sliding-windows of 50 SNPs on the Pool-Seq example data. The dashed line indicates the estimated overall genome-wide $F_{ST}$. The 20 simulated chromosomes are represented by alternate colors.

## 3.2 Estimating and visualizing pairwise-population $F_{ST}$

**Pairwise $F_{ST}$: methods and computational tips**

Computation of the *npop* × *npop* matrix of pairwise-population $F_{ST}$ with the *compute.pairwiseFST* function can be computationally demanding (in terms of time and memory usage), particularly when the number of population samples in the *pooldata* or *countdata* input objects exceeds 50. In this case, one may rather use the *compute.fstats* function (section 4.1.1), which is significantly faster and more memory-efficient. The pairwise $F_{ST}$ matrix, acquired through *compute.fstats* and saved in the *pairwise.fst* slot within the *fstats* output object, can be visualized using *heatmap* or other conventional clustering techniques (see section 4.1.1). Note that the $F_{ST}$ estimation approach in *compute.fstats* corresponds to the "Identity" method described above (section 3), and the function does not provide SNP-specific estimates in the output (as opposed to *compute.pairwiseFST* when *output.snp.values=TRUE*, see below).

### 3.2.1 The *compute.pairwiseFST* and the *heatmap* functions

The *compute.pairwiseFST* function enables estimation of genome-wide $F_{ST}$ for each of the $\frac{npop(npop-1)}{2}$ population pairs from data stored in a *countdata* or *pooldata* object. As for the *computeFST* function (section 3), the *compute.pairwiseFST* function automatically uses the appropriate estimation procedure for the type of input data (either allele count for *countdata* objects or Pool-Seq read count data for *pooldata* objects). The function returns an S4 object of class *pairwisefst* class. The documentation page providing detailed information about the elements (slots) of this class can be accessed with the *?* operator or the following command:

```
help(pairwisefst)
```

The pairwise-population $F_{ST}$ may then be visualized using the generic *heatmap* function directly applied on the obtained *pairwisefst* object as illustrated below for Pool-Seq example results (similar results are obtained with the allele count data):

```
sim6p.pairwisefst<-compute.pairwiseFST(sim6p.readcount30X,verbose=FALSE)
```

```
Overall Analysis Time: 0 h 0 m 2 s
```

```
heatmap(sim6p.pairwisefst)
```

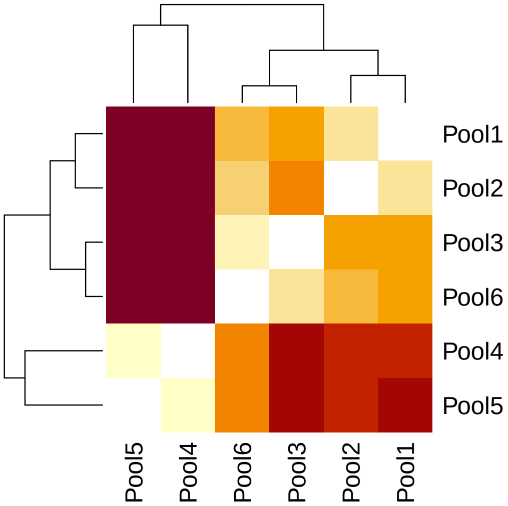

Figure 4: Heatmap representing the pairwise-population $F_{ST}$ matrix of the six populations of the 30X Pool-Seq example data set

The resulting heat map (Figure 4) is consistent with the simulated scenario (Figure 1). Note that the population *P3* is the closest to the admixed population *P6* (leading to their early clustering in the binary tree representation) as expected from the high contribution of the *P3* ancestor ($1-\alpha = 75\%$) to the admixed ancestor of *P6* and the short timing of admixture ($\tau = \frac{t}{2N_e} = 0.0125$).

### 3.2.2 Block-Jackknife estimation of $\hat{F}_{ST}$ standard-error and visualisation of confidence intervals

As with the *computeFST* function, standard-error of the pairwise-population $F_{ST}$ estimates may be estimated using a block-jackknife sampling approach (see Appendix A.1) by specifying the number of consecutive SNPs forming each block with the argument *nsnp.per.bjack.block* (by default *nsnp.per.bjack.block=0*, i.e., no block-jackknife is carried out). The resulting estimated standard-errors may directly be used to derive confidence intervals (see above) that can also be plotted with the *plot_fstats* function (or directly using the *plot* command that calls *plot_fstats* for *pairwisefst* objects). This is illustrated below with the allele count example data (similar results are obtained with the Pool-Seq read count data) and applying a MAF filtering of SNPs on each pairwise comparison (for the sake of illustration) :

```
sim6p.pairwisefst<-compute.pairwiseFST(sim6p.allelecount,min.maf=0.01,
                                       nsnp.per.bjack.block = 1000,verbose=FALSE)
```

```
Overall Analysis Time: 0 h 0 m 3 s
```

```
#Estimated pairwise Fst are stored in the slot values:
#5 first estimated pairwise
head(sim6p.pairwisefst@values)
```

```
      Fst Estimate Fst bjack mean Fst bjack s.e. Q2 Estimate Q2 bjack mean Q2 bjack s.e.   Nsnp
P1;P2   0.04946710     0.04931981   0.0007439500   0.7369276     0.7121075  0.0006681293 301102
```

```
P1;P3   0.09478083      0.09497510   0.0011284444   0.7509383      0.7260094  0.0006276102 333075
P1;P4   0.17816331      0.17846116   0.0015637673   0.7438803      0.7221825  0.0006692343 351210
P1;P5   0.17716282      0.17708538   0.0016019588   0.7434524      0.7246795  0.0006850680 350882
P1;P6   0.07039455      0.07065709   0.0009229865   0.7572024      0.7312169  0.0006507648 337522
P2;P3   0.09543866      0.09543830   0.0011134127   0.7523652      0.7243749  0.0006443638 336438
```

```
plot(sim6p.pairwisefst)
```

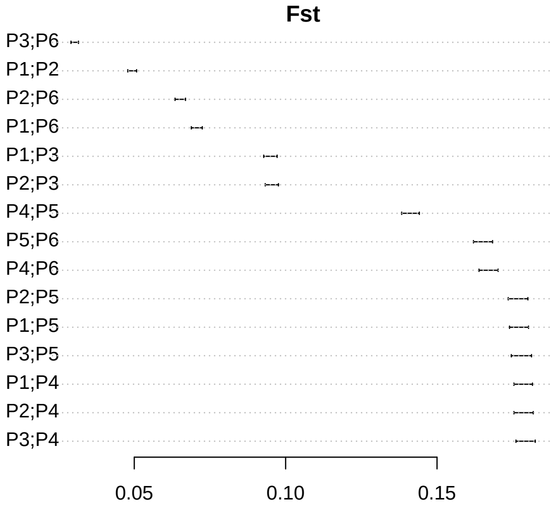

Figure 5: Estimated pairwise-population $F_{ST}$ with their 95% confidence intervals for allele count example data set

The resulting estimated pairwise-population $F_{ST}$ displayed in Figure 5 are consistent with the simulated scenario (Figure 1). The lowest level of differentiation is observed for the *P3* and *P6* population pair as expected from the high contribution of the *P3* ancestor ($1 - \alpha = 75\%$) to the admixed ancestor of *P6* and the short timing of admixture ($\tau = \frac{t}{2N_e} = 0.0125$).

### 3.3 Hierarchical $F$–statistics

Computation of hierarchical $F$–statistics were introduced in version 2.3.0 of the package as described in Gautier *et al.* (2024). Given a user-defined grouping of population samples according to a one-level hierarchy, the model aims at partitioning the total genetic differentiation, denoted $F_{ST}^{(h)}$ (to distinguish it from the standard $F_{ST}$ described above, which does not involving any population grouping) into a within-group ($F_{SG}$) and between-group ($F_{GT}$) component. More specifically, considering the three IIS probabilities of pairs of genes sampled within the same population ($Q_1$), or each in a different pair of populations belonging to the same group ($Q_2$) or belonging to two different groups ($Q_3$), these hierarchical $F$–statistics are defined respectively as:

- $F_{SG} \equiv \frac{Q_1 - Q_2}{1 - Q_2} = \frac{(1-Q_2)-(1-Q_1)}{1-Q_2}$ which following Nei (1973) may also be interpreted as the relative excess of genetic diversity (non IIS probability of pairs of genes) attributable to the within-group structuring
- $F_{GT} \equiv \frac{Q_2 - Q_3}{1 - Q_3} = \frac{(1-Q_3)-(1-Q_2)}{1-Q_3}$ which may similarly be interpreted as the relative excess of genetic diversity attributable to the between-group structuring (relative to the whole population)
- $F_{ST}^{(h)} \equiv \frac{Q_1 - Q_3}{1 - Q_3} = \frac{(1-Q_3)-(1-Q_1)}{1-Q_3}$

Note that the three hierarchical $F$–statistics are related by the formula: $(1 - F_{ST}^{(h)}) = (1 - F_{SG})(1 - F_{GT})$

Unbiased estimators of hierarchical $F$–statistics have been implemented in the *computeFST* function for both allele count data (stored in *countdata* objects, see section 2.1) and Pool-Seq read count data (stored in *pooldata* objects, see section 2.2), where the appropriate estimator is automatically detected given the input object class (either allele count for *countdata* objects or Pool-Seq read count data for *pooldata* objects). The population grouping must be specified using the *struct* argument with a vector of length equal to the number of population samples containing the group of origin of each. Note that by default *struct=NULL*, i.e., the standard $F_{ST}$ is computed. In addition, as for standard $F_{ST}$ (see section 3.1.1), hierarchical $F$–statistics estimators consist of method-of-moments estimators developed within either an *Anova* framework (default option *method="Anova"*) or based on unbiased estimators of IIS probability (option *method="Identity"*[14]. For example with the simulated example data, and defining two groups consisting of i) populations *P1*, *P2*, *P3* and *P6* (named *"GRP_A"* below) populations; and ii) the two outgroup populations *P4* and *P5* (named *"GRP_B"* below) ; we obtain the following estimates for the hierarchical $F$–statistics:

- Pool-Seq read count data:

```
sim6p.readcount30X.fst<-computeFST(sim6p.readcount30X,
                            struct=c("GRP_A","GRP_A","GRP_A","GRP_B","GRP_B","GRP_A"),
                            nsnp.per.bjack.block = 1000)
```

```
Computation of hierarchical Fst (Gautier et al., 2024)
2 groups of pop samples declared in struct object:
GRP_A GRP_B
    1     2
Computing SNP-specific Q1, Q2 and Q3 (Anova estimator)
Starting Block-Jackknife sampling
462 Jackknife blocks identified with 462000 SNPs (out of 472410 ).
 SNPs map to 20 different chrom/scaffolds
Average (min-max) Block Sizes: 4.232 ( 3.515 - 4.975 ) Mb
```

```
sim6p.readcount30X.fst$Fsg      #genome-wide Fsg (within-group differentation)
```

```
   Estimate  bjack mean  bjack s.e.     CI95inf     CI95sup
0.087329745 0.087459585 0.000586475 0.086310094 0.088609077
```

```
sim6p.readcount30X.fst$Fgt      #genome-wide Fgt (between-group differentation)
```

```
   Estimate  bjack mean  bjack s.e.     CI95inf     CI95sup
0.088748182 0.088877100 0.001176005 0.086572130 0.091182071
```

```
sim6p.readcount30X.fst$Fst      #genome-wide hFst (within-group differentation)
```

```
   Estimate  bjack mean  bjack s.e.     CI95inf     CI95sup
0.168404146 0.168639834 0.001115557 0.166453342 0.170826325
```

- allele count data:

```
sim6p.allelecount.fst<-computeFST(sim6p.allelecount,
                           struct=c("GRP_A","GRP_A","GRP_A","GRP_B","GRP_B","GRP_A"),
                           nsnp.per.bjack.block = 1000,verbose=FALSE)
sim6p.allelecount.fst$Fsg      #genome-wide Fsg (within-group differentation)
```

```
    Estimate    bjack mean    bjack s.e.      CI95inf      CI95sup
0.0872526862 0.0873775127 0.0005814018 0.0862379652 0.0885170602
```

```
sim6p.allelecount.fst$Fgt      #genome-wide Fgt (between-group differentation)
```

```
   Estimate  bjack mean  bjack s.e.     CI95inf     CI95sup
0.088742957 0.088839709 0.001149783 0.086586134 0.091093283
```

[14] As for standard $F_{ST}$, when *method="Identity"*, the overall $\widehat{Q_1}$, $\widehat{Q_2}$ and $\widehat{Q_3}$ are computed by default as unweighted averages of the corresponding and underlying population-specific $Q_1$ or pairwise populations $Q_2$. Weighted averages can be computed by setting *weightpid=TRUE*

```
sim6p.allelecount.fst$Fst       #genome-wide hFst (within-group differentation)
```

```
   Estimate   bjack mean   bjack s.e.      CI95inf      CI95sup
0.168252582 0.168454629 0.001104235 0.166290328 0.170618930
```

Finally, as described above for the standard $F_{ST}$, the *sliding.window.size* argument allows computing multi-locus hierarchical $F$–statistics (i.e., $F_{SG}$, $F_{GT}$ and $F_{ST}^{(h)}$) for sliding windows over the different chromosomes (or scaffolds/contigs), e.g., to carry out genome-scans for adaptive differentiation (either within or across groups of populations). Each sliding window includes a number of consecutive SNPs specified by the *sliding.window.size* argument.

## 4 Estimating and visualizing $f$-statistics ($f_2$, $F_{ST}$, $f_3$, $f_3^\star$, $f_4$ and D)

The $f_2$, $f_3$ and $f_4$ statistics were introduced in a seminal paper by Reich and co-workers (Reich *et al.* 2009) retracing the history of Indian human population and forms the core components of a general framework for demographic history inference detailed in Patterson *et al.* (2012; see also Lipson *et al.* 2013; Peter 2016; Lipson 2020). These statistics measure (expected) covariance in allele frequencies among sets of two ($F_2$), three ($F_3$) or four ($F_4$) populations and are formally defined as follows (denoting $p_i$ the SNP reference allele frequency in population $i$):

- $\mathrm{F}_2(A;B) \quad \equiv \mathbb{E}\left[(p_A - p_B)^2\right]$
- $\mathrm{F}_3(A;B,C) \quad \equiv \mathbb{E}\left[(p_A - p_B)(p_A - p_C)\right] \quad \equiv \frac{1}{2}\left(\mathrm{F}_2(A;B) + \mathrm{F}_2(A;C) - \mathrm{F}_2(B;C)\right)$
- $\mathrm{F}_4(A,B;C,D) \quad \equiv \mathbb{E}\left[(p_A - p_B)(p_C - p_D)\right] \quad \equiv \frac{1}{2}\left(\mathrm{F}_2(A;D) + \mathrm{F}_2(B;C) - \mathrm{F}_2(A;C) - \mathrm{F}_2(B;D)\right)$

The definitions of the F parameters are not depending on the reference allele choice since $\left((1-p_A)-(1-p_B)\right)^2 = (p_B - p_A)^2 = (p_A - p_B)^2$. As a consequence, $F_2$ and all the other $F$ parameters may also be defined in terms of IIS within and between pairs of population as $\mathrm{F}_2(A;B) = Q_1 - Q_2$ (see section 3) which allows deriving unbiased estimators for both Pool-Seq read count and standard allele count data (Gautier *et al.* 2022).

**Combinatorics and linear dependence of *f-statistics***

With $I$ populations, there are $\binom{I}{2} = \frac{1}{2}I(I-1)$ possible $F_2$ (i.e., 15 with $I = 6$ populations); $3\binom{I}{3} = \frac{1}{2}I(I-1)(I-2)$ possible $F_3$ (i.e., 60 with $I = 6$ populations); and $3\binom{I}{4} = \frac{1}{8}I(I-1)(I-2)(I-3)$ possible $F_4$ (i.e., 45 with $I = 6$ populations). However, due to their underlying linear dependency (see the above definitions), these $\frac{1}{8}I(I-1)(I^2-I+2)$ F-statistics form a vector space of dimension $\frac{1}{2}I(I-1)$ the basis of which may be specified by the set of all the $\binom{I}{2}$ possible $F_2$ statistics or, given a reference population $i$ (randomly chosen among the I ones) the set of all the $I-1$ $F_2$ statistics of the form $F_2(i;j)$ (with $j \neq i$) and the $\binom{I-1}{2}$ $F_3$ statistics of the form $F_3(i;j,k)$ (with $j \neq i$; $k \neq i$ and $j \neq k$) (Patterson *et al.* 2012; Lipson 2020). The resulting basis is informative about population history and may be used to fit admixture graph (see section 5).

Moreover, although $f_2$ statistics are difficult to interpret or to compare across pairs of populations (see the notice below), formal tests of population admixture (“three-populations” test) and tests of treeness of population quadruplets (“four-populations” test) can directly be performed using $f_3$ (see e.g., section 4.1.2) and $f_4$ (see e.g., section 4.1.3) statistics respectively (Patterson *et al.* 2012). Indeed, if $f_3(A;B,C) < 0$, we can conclude that population A originates from a population that is admixed between two source populations related to populations B and C respectively (although the signal may vanish if population A has drifted too much drifted since admixture or the admixture rates are too close to 0 or 1). Conversely, if $f_4(A,B;C,D) = 0$, the populations A, B, C and D are related by a bifurcating tree with the unrooted topology (A,B;C,D) although some may be admixed (if the paths connecting the (A,B) and (C,D) population pairs are not overlapping, see section 4.1.3 for an example). Finally, under certain circumstances, proportions of ancestry that contributed to a given admixed population can be estimated with ratios of $f_4$ statistics (see e.g., 4.3) for set of carefully related populations (Patterson *et al.* 2012).

The parameters $F_2$, $F_3$ and $F_4$ are not scaled with respect to the distribution of marker information content (i.e., heterozygosities). As a consequence, their resulting estimates may strongly depend on the chosen set of genetic markers (Patterson *et al.* 2012). The well-known $F_{ST}$ parameter and the two parameters $\mathrm{F}_3^\star$ and $D$ introduced by Patterson *et al.* (2012) correspond to scaled versions of $F_2$, $F_3$ and $F_4$ expected to be less sensitive to the SNP ascertainment and thus more comparable across data sets. As shown in Gautier *et al.* (2022), these can be defined in terms of IIS probabilities as:

- $\mathrm{F}_{\mathrm{ST}}(A;B) \equiv \dfrac{\mathrm{F}_2(A;B)}{1-Q_2^{A,B}} = \dfrac{Q_1^A + Q_1^B - 2Q_2^{A,B}}{2\left(1-Q_2^{A,B}\right)}$
- $\mathrm{F}_3^\star(A;B,C) \equiv \dfrac{\mathrm{F}_3(A;B,C)}{1-Q_1^A} = \dfrac{Q_1^A + Q_2^{B,C} - Q_2^{A,B} - Q_2^{A,C}}{2\left(1-Q_1^A\right)}$
- $\mathrm{D}(A,B;C,D) \equiv \dfrac{\mathrm{F}_4(A,B;C,D)}{\left(1-Q_2^{A,B}\right)\left(1-Q_2^{C,D}\right)} = \dfrac{Q_2^{A,C} + Q_2^{B,D} - Q_2^{A,D} - Q_2^{B,C}}{2\left(1-Q_2^{A,B}\right)\left(1-Q_2^{C,D}\right)}$

Three-population and Four-population tests naturally extend to $F_3^\star$ and $D$ statistics respectively. An advantage of the latter over $F_4$ is that $D$ is constrained to the $[-1,1]$ interval and may thus be interpreted as the magnitude of deviation to treeness of the tested quadruplet (Patterson *et al.* 2012).

**On the $D$-statistics sign convention**

The $D$-statistics ($D_{dur}$) was originally introduced as a site-pattern–based approach (*a.k.a.* the *ABBA-BABA* framework) to detect deviations from a strictly bifurcating population history caused by gene flow Durand *et al.* (2011). As detailed in Appendix A.2, the covariance-based ($D_{pat}$, Patterson *et al.* 2012) or IIS-based ($D_{pf}$, Gautier *et al.* 2022) estimators, which can be viewed as extensions to population samples using allele frequency, are algebraically equivalent to the original definition. An **important difference**, however, lies in the sign convention, since $D_{pf} = D_{pat} = -D_{dur}$. In other word, the $D$-statistics estimated in *poolfstat* is the opposite of that defined in Green *et al.* (2010) and Durand *et al.* (2011).

This sign change has direct consequences for interpretation. Under the Durand *et al.* (2011) convention, a positive value $D(P_1, P_2; P_3, P_O) > 0$ (where $P_O$ is an outgroup species) indicates an excess of *ABBA* sites and is therefore interpreted as evidence for gene flow between $P_3$ and $P_2$ (assuming no gene flow betweed $P_O$ and $P_1$). In contrast, under the Patterson *et al.* (2012) and Gautier *et al.* (2022) convention implemented in *poolfstat*, a positive value indicates gene flow between $P_3$ and $P_1$ (assuming no gene flow betweed $P_O$ and $P_2$).

## 4.1 The *compute.fstats* function and *fstats* objects

The *compute.fstats* function implements unbiased estimators of the parameters $F_2$ (and $F_{ST}$), $F_3$ (and $F_3^\star$), $F_4$ and $D$ defined above for allele count data (stored in *countdata* objects, see section 2.1) or Pool-Seq read count data (stored in *pooldata* objects, see section 2.2) as described in Gautier *et al.* (2022)[15]. The function also allows estimating within-population heterozygosities (defined as $1 - Q_1$) which is needed to scale branch lengths of admixture graphs in drift units (see section 5) or for rooting neighbor-joining trees of unadmixed populations (see section 6.1); and absolute divergence (defined as $1 - Q_2$) between all pairs of populations.

[15] Although not defined in the same way, estimator for allele count data are strictly equivalent to those by Patterson *et al.* (2012)

**Heterozygosities and absolute divergence estimated with *compute.fstats***

Heterozygosities and pairwise-population absolute divergences are estimated using SNPs ascertained in the original *pooldata* (or *countdata*) input object. These SNPs have typically undergone multiple filtering criteria, such as discarding genomic positions that are monomorphic in all samples. In addition, other more complex ascertainment scheme may also have been applied when selecting polymorphic SNPs, especially for allele counts derived from SNP genotyping assays. Thus, these heterozygosity and absolute divergence estimates are expected to be (highly) upwardly biased estimates of within-population nucleotide diversity (often referred to as $\pi_w$ or $\theta$) and between-population nucleotide absolute divergence (often referred to as $\pi_b$ or $d_{XY}$) respectively (e.g., Cruickshank and Hahn 2014). Yet when using data from whole genome sequencing, the estimates of heterozygosity, absolute divergence, and $f_2$ are expected to be related[a] to $\pi_w$, $d_{XY}$ and $d_a$ respectively.

[a] up to a proportional constant determined by the number of callable monomorphic sites and, particularly for Pool-Seq data, to biases introduced by rare variants or sequencing errors (e.g., Ferretti *et al.* 2013)

Block-jackknife estimates of standard errors of the different estimators (needed for "Three-populations" and "Four-populations" tests, see below; and to fit admixture graph, see section 5) and their covariance (needed to fit admixture graph, see section 5) may also be performed.

As for the *computeFST* (section 3.1.1) and *compute.pairwiseFST* (section 3.2), the *compute.fstats* functions automatically detects which estimator to implement according to the class of the input object (either *countdata* or *pooldata*). The function estimates by default all the within population heterozygosities, the $F_2$ (and its scaled version $F_{ST}$[16]), $F_3$ (and its scaled version $F_3^\star$) $F_4$ statistics (and the scaled version $F_{ST}$ )[17]. Computation of $D$ statistics (i.e., scaled $F_4$) is not carried out by default (as specified with the *computeDstat* argument set to *FALSE* by default) since this may add some non negligible additional computation time for data with a large number of populations due to the extra computation of the $F_4$ scaling factor[18] although in the example below (with only 6 populations) the difference in running time is negligible.

The results are then stored in an object of class *fstats* whose elements (slots) are detailed in the documentation page accessible with the following command (or the *?* operator):

```
help(fstats)
```

The underlying f-statistics may then be easily accessible or visualized with the *plot_fstats* function (or directly using the *plot* command that calls *plot_fstats* for *fstats* objects) as illustrated below for the allele count (*sim6p.allelecount*) and Pool-Seq read count (*sim6p.readcount30X*) example data[19].

```
##Estimation of f-statistics on count data
sim6p.allelecount.fstats<-compute.fstats(sim6p.allelecount,nsnp.per.bjack.block = 1000,
                                         computeDstat = TRUE)
```

```
Block-Jackknife specification
462 Jackknife blocks identified with 462000 SNPs (out of 472410 ).
 SNPs map to 20 different chrom/scaffolds
Average (min-max) Block Sizes: 4.232 ( 3.515 - 4.975 ) Mb
Estimating Q1
Estimating Q2
Estimating within-population heterozygosities
Estimating F2, pairwise Fst (method=Identity), and pairwise divergence (1-Q2)
Estimating F3 and F3* (n= 60 configurations)
Estimating F4 and Dstat (n= 45 configurations)
```

[16] The estimator is then actually exactly the same as the one implemented in the *compute.pairwiseFST* or *computeFST* functions when the argument *method="Identity"*

[17] The *compute.fstats* function is optimized in such a way that the computational cost for the estimation of pairwise $F_{ST}$, $F_3$, $F_3^\star$ and $F_4$ from the $F_2$-statistics and $Q_2$ estimates is negligible

[18] For instance on a tested allele count real data set consisting of 640,000 SNPs genotyped on 24 populations, *compute.fstats* ran in 3 m 13 s (4 m 45 s if *nsnp.per.bjack.block = 5000*) with *computeDstat=TRUE* and only 2 s (3 s) if *computeDstat=FALSE*

[19] Note that estimates of the different statistics are highly similar between the allele count and the Pool-Seq read count data.

```
  Step 1/2: Estimating Dstat denominators
  Step 2/2: Estimating F4 and D- statistics
Estimating Qmat, the error covariance matrix ( 75 x 75 )

Overall Analysis Time: 0 h 0 m 2 s
```

```
sim6p.allelecount.fstats
```

```
 * * * fstats Object * * *
Example of useful visualization functions are plot.fstats
```

```
##Estimation of f-statistics on Pool-Seq data (without computation of Dstat)
sim6p.readcount30X.fstats<-compute.fstats(sim6p.readcount30X,nsnp.per.bjack.block = 1000,
                                          verbose=FALSE)
##Estimation of f-statistics on Pool-Seq data (with computation of Dstat)
sim6p.readcount30X.fstats<-compute.fstats(sim6p.readcount30X,nsnp.per.bjack.block = 1000,
                                         computeDstat = TRUE,verbose=FALSE)
```

#### 4.1.1 $f_2$, $F_{ST}$ and pairwise absolute divergence estimates (*f2.values*, *fst.values* and *divergence* slots of the *fstat* object)

```
# count data (3 first f2)
head(sim6p.allelecount.fstats@f2.values,3)
```

```
         Estimate  bjack mean   bjack s.e.
P1,P2 0.008294424 0.008274108 0.0001298376
P1,P3 0.016643720 0.016680443 0.0002108642
P1,P4 0.033924154 0.033971079 0.0003456714
```

```
# 30X Pool-Seq data (3 first f2)
head(sim6p.readcount30X.fstats@f2.values,3)
```

```
              Estimate  bjack mean   bjack s.e.
Pool1,Pool2 0.00829429 0.008275487 0.0001351787
Pool1,Pool3 0.01665818 0.016696323 0.0002159221
Pool1,Pool4 0.03398383 0.034045968 0.0003525136
```

```
# count data (3 first Fst)
head(sim6p.allelecount.fstats@fst.values,3)
```

```
        Estimate bjack mean   bjack s.e.
P1,P2 0.04946710 0.04936227 0.0007453263
P1,P3 0.09478083 0.09499692 0.0010947422
P1,P4 0.17816331 0.17843286 0.0015418662
```

```
# 30X Pool-Seq data (3 first Fst)
head(sim6p.readcount30X.fstats@fst.values,3)
```

```
              Estimate bjack mean  bjack s.e.
Pool1,Pool2 0.04947320 0.04937253 0.000774369
Pool1,Pool3 0.09487048 0.09508626 0.001119595
Pool1,Pool4 0.17839940 0.17872795 0.001572758
```

```
# count data (3 first pairwise genetic divergence)
head(sim6p.allelecount.fstats@divergence,3)
```

```
       Estimate bjack mean   bjack s.e.
P1,P2 0.1676756  0.1676201 0.0005895541
P1,P3 0.1756022  0.1755893 0.0005931841
P1,P4 0.1904104  0.1903858 0.0006251644
```

```
# 30X Pool-Seq data (3 first pairwise genetic divergence)
head(sim6p.readcount30X.fstats@divergence,3)
```

```
              Estimate bjack mean   bjack s.e.
Pool1,Pool2 0.1676522  0.1676132 0.0005941263
Pool1,Pool3 0.1755887  0.1755913 0.0006022428
Pool1,Pool4 0.1904930  0.1904905 0.0006272189
```

As mentioned above, notice that the pairwise $F_{ST}$ estimates are the same as those obtained with the *compute.pairwiseFST* function (see section 3.2) run with *method=Identity* (which are here actually equal with estimates obtained by the default *Anova* method because there is no variation in the total allele counts for all SNPs).

**Symmetry and ordering of population pairs in $F_2$ and $F_{ST}$**

By construction $F_2(A;B) = F_2(B,A)$ (and $F_{ST}(A;B) = F_{ST}(B,A)$). If $i_P$ is the index of population $P$ in the *popnames* or *poolnames* slots of the *countdata* or *pooldata* objects (i.e., the column order in the corresponding allele or read count data matrices) used to obtain the *fstats* object, the $F_2(A,B)$ (resp. $F_{ST}(B,A)$) configurations reported in the slot *f2.values* (resp. *fst.values*) satisfy $i_A < i_B$.

When *compute.fstats* is run with options *output.pairwise.fst* and *output.pairwise.div* set to *TRUE* (default behavior), the object contains the pairwise-population $F_{ST}$ and absolute divergence estimates organized in a matrix format (*pairwise.fst* and *pairwise.div* slots respectively) . These pairwise matrice can then easily be visualized in the form of heatmaps as illustrated in Figure 6 obtained with utilities from the package *ComplexHeatmap*[20]:

```
require(ComplexHeatmap)
div.hm <- Heatmap(sim6p.readcount30X.fstats@pairwise.div,
                  cluster_rows =TRUE,cluster_columns=TRUE,name="Divergence",
                  clustering_method_rows = "average",clustering_method_columns = "average",
                  clustering_distance_rows = function(m) as.dist(m),
                  clustering_distance_columns = function(m) as.dist(m),
                  show_heatmap_legend=FALSE,column_title = "Divergence (1-Q2)")
fst.hm <- Heatmap(sim6p.readcount30X.fstats@pairwise.fst,
                  cluster_rows =TRUE,cluster_columns=TRUE,name="values",
                  clustering_method_rows = "average",clustering_method_columns = "average",
                  clustering_distance_rows = function(m) as.dist(m),
                  clustering_distance_columns = function(m) as.dist(m),
                  column_title = "Fst=(Q1-Q2)/(1 - Q2)")
div.hm+fst.hm
```

### 4.1.2 $f_3$ and $f_3^\star$ estimates (*f3.values* and *f3star.values* slots of the *fstat* object) and 3-Population tests:

```
# count data (3 first f3)
head(sim6p.allelecount.fstats@f3.values,3)
```

```
             Estimate  bjack mean   bjack s.e.  Z-score
P1;P2,P3 0.004053338 0.004048392 0.0001176972 34.39668
P1;P2,P4 0.004089298 0.004087474 0.0001326605 30.81155
P1;P2,P5 0.004155216 0.004149524 0.0001341405 30.93416
```

```
# 30X Pool-Seq data (3 first f3)
head(sim6p.readcount30X.fstats@f3.values,3)
```

```
                     Estimate  bjack mean   bjack s.e.  Z-score
Pool1;Pool2,Pool3 0.004057570 0.004053794 0.0001247034 32.50750
Pool1;Pool2,Pool4 0.004114142 0.004114670 0.0001365234 30.13893
Pool1;Pool2,Pool5 0.004157049 0.004154896 0.0001411765 29.43051
```

[20] https://jokergoo.github.io/ComplexHeatmap-reference/

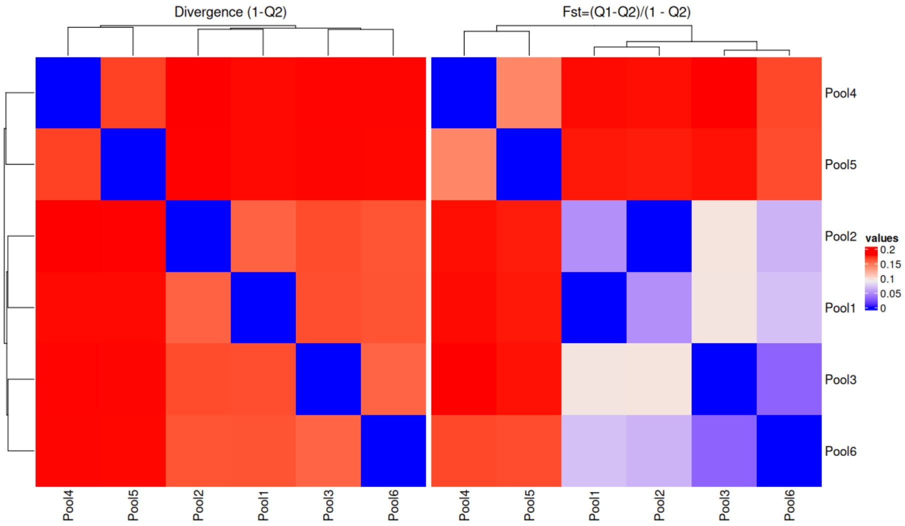

Figure 6: Heatmaps of the matrices of estimated pairwise-population absolute divergence (left) and $F_{ST}$ (right) for the 30X Pool-Seq data set.

```
# count data (3 first f3*)
head(sim6p.allelecount.fstats@f3star.values,3)
```

```
           Estimate bjack mean   bjack s.e.  Z-score
P1;P2,P3 0.02552286 0.02549608 0.0007467871 34.14102
P1;P2,P4 0.02574929 0.02574221 0.0008480235 30.35554
P1;P2,P5 0.02616436 0.02613299 0.0008581050 30.45430
```

```
# 30X Pool-Seq data (3 first f3*)
head(sim6p.readcount30X.fstats@f3star.values,3)
```

```
                    Estimate bjack mean   bjack s.e.  Z-score
Pool1;Pool2,Pool3 0.02555404 0.02553297 0.0007888059 32.36915
Pool1;Pool2,Pool4 0.02591032 0.02591640 0.0008722134 29.71337
Pool1;Pool2,Pool5 0.02618054 0.02616977 0.0009009973 29.04533
```

**Symmetry and ordering of the $F_3$ statistics**

By construction $F_3(A;B,C) = F_3(A;C,B)$ (and $F_3^{\star}(A;B,C) = F_3^{\star}(A;C,B)$). If $i_P$ is the index of population $P$ in the *popnames* or *poolnames* slots of the *countdata* or *pooldata* objects (i.e., the column order in the corresponding allele or read count data matrices) used to obtain the *fstats* object, the $F_3(A;B,C)$ (and $F_3^{\star}(A;B,C)$) configurations reported in the slot $f3.values$ satisfy $i_B < i_C$.

As shown in the above example, activation of block-jackknife estimation of standard errors (i.e., argument *nsnp.per.bjack.block*>0) results in the computation of Z-scores (i.e., ratio of the block-jackknife estimated mean and standard-error) which quantifies the deviation of the estimated $f_3$-statistics from 0 (in units of s.e.). This gives a simple decision criterion for three-population tests of admixture (i.e., negative $f_3$ or negative $f_3^{\star}$). For instance a Z-score $< -1.65$ provides evidence for admixture (i.e., significantly negative $f_3$) at the 95% significance threshold:

```
# count data (F3-based 3-pop test)
tst.sel<-sim6p.allelecount.fstats@f3.values$`Z-score`< -1.65
sim6p.allelecount.fstats@f3.values[tst.sel,]
```

```
              Estimate    bjack mean    bjack s.e.   Z-score
P6;P2,P3 -0.0002671143 -0.0002623528 8.638581e-05 -3.036989
```

```
# 30X Pool-Seq data (F3-based 3-pop test)
tst.sel<-sim6p.readcount30X.fstats@f3.values$`Z-score`< -1.65
sim6p.readcount30X.fstats@f3.values[tst.sel,]
```

```
                       Estimate    bjack mean    bjack s.e.   Z-score
Pool6;Pool2,Pool3 -0.0002417509 -0.0002304611 8.851744e-05 -2.603567
```

```
# count data (F3*-based 3-pop test)
tst.sel<-sim6p.allelecount.fstats@f3star.values$`Z-score`< -1.65
sim6p.allelecount.fstats@f3star.values[tst.sel,]
```

```
             Estimate   bjack mean   bjack s.e.   Z-score
P6;P2,P3 -0.001631657 -0.001603205 0.0005274779 -3.039379
```

```
# 30X Pool-Seq data (F3*-based 3-pop test)
tst.sel<-sim6p.readcount30X.fstats@f3star.values$`Z-score`< -1.65
sim6p.readcount30X.fstats@f3star.values[tst.sel,]
```

```
                      Estimate   bjack mean   bjack s.e.   Z-score
Pool6;Pool2,Pool3 -0.001475866 -0.001407435 0.0005402774 -2.605024
```

In agreement with the simulated scenario (Figure 1), both the allele count and Pool-Seq read count data support an admixed origin for population *P6* with ancestral sources related to *P2* and *P3*.

### 4.1.3 $f_4$ and $D$ estimates (*f4.values* and *Dstat.values* slots of the *fstat* object) and 4-Population tests:

```
# count data (3 first f4)
head(sim6p.allelecount.fstats@f4.values,3)
```

```
                Estimate   bjack mean   bjack s.e.   Z-score
P1,P2;P3,P4 3.595944e-05 3.908225e-05 1.154193e-04 0.3386111
P1,P2;P3,P5 1.018776e-04 1.011325e-04 1.197917e-04 0.8442359
P1,P2;P3,P6 5.594310e-04 5.622762e-04 6.138658e-05 9.1595941
```

```
# 30X Pool-Seq data (3 first f4)
head(sim6p.readcount30X.fstats@f4.values,3)
```

```
                            Estimate   bjack mean   bjack s.e.   Z-score
Pool1,Pool2;Pool3,Pool4 5.657166e-05 6.087576e-05 1.194004e-04 0.5098456
Pool1,Pool2;Pool3,Pool5 9.947887e-05 1.011014e-04 1.253135e-04 0.8067878
Pool1,Pool2;Pool3,Pool6 5.653400e-04 5.621051e-04 6.409453e-05 8.7699380
```

```
# count data (3 first D)
head(sim6p.allelecount.fstats@Dstat.values,3)
```

```
                Estimate   bjack mean  bjack s.e.   Z-score
P1,P2;P3,P4 0.0006651386 0.0007230774 0.002135688 0.3385688
P1,P2;P3,P5 0.0018826436 0.0018705234 0.002216171 0.8440338
P1,P2;P3,P6 0.0107508025 0.0108152267 0.001182842 9.1434253
```

```
# 30X Pool-Seq data (3 first D)
head(sim6p.readcount30X.fstats@Dstat.values,3)
```

```
                           Estimate  bjack mean  bjack s.e.   Z-score
Pool1,Pool2;Pool3,Pool4 0.001045743 0.001125524 0.002208046 0.5097373
Pool1,Pool2;Pool3,Pool5 0.001837431 0.001868847 0.002316956 0.8065961
Pool1,Pool2;Pool3,Pool6 0.010853673 0.010799873 0.001235136 8.7438717
```

**Symmetry and ordering of the $F_4$ statistics**

When comparing two pairs of populations $(A, B)$ and $(C, D)$, the $f_4$ statistics for the 8 quadruplets (A,B;C,D); (B,A;C,D); (A,B;D,C); (B,A;D,C); (C,D;A,B); (C,D;B,A); (D,C;A,B) and (D,C;B,A) have the same absolute value by definition of the $F_4$ parameter:

$$\begin{aligned} F_4(A,B;C,D) &= F_4(B,A;D,C) = F_4(C,D;A,B) = F_4(D,C;A,B) \\ -F_4(A,B;C,D) &= F_4(B,A;C,D) = F_4(C,D;B,A) = F_4(D,C;B,A) = F_4(A,B;D,C) \end{aligned}$$

and similarly

$$\begin{aligned} D(A,B;C,D) &= D(B,A;D,C) = D(C,D;A,B) = D(D,C;A,B) \\ -D(A,B;C,D) &= D(B,A;C,D) = D(C,D;B,A) = D(D,C;B,A) = D(A,B;D,C) \end{aligned}$$

If $i_P$ is the index of population $P$ in the *popnames* or *poolnames* slots of the *countdata* or *pooldata* objects (i.e., the column order in the corresponding allele or read count data matrices) used to obtain the *fstats* object, the $F_4(A,B;C,D)$ (and $D(A,B;C,D)$) configurations reported in the slot $f4.values$ (and $Dstat.values$) satisfy $i_A < i_B$; $i_A < i_C$ and $i_C < i_D$.

As for $f_3$ and $f_3^\star$ (section 4.1.2), activating block-jackknife estimation of standard errors (i.e., the argument *nsnp.per.bjack.block*>0) results in the computation of Z-scores (i.e., ratio of the block-jackknife estimated mean and standard-error) which quantifies the deviation of the estimated $f_4$-statistics from 0 (in units of s.e.). This gives a simple decision criterion for four-population tests of treeness (i.e., non null $F_4$ or $D$). For instance a Z-score lower than 1.96 in absolute value provides no evidence against the null-hypothesis of treeness for the tested population configuration at the 95% significance threshold:

```
# count data
tst.sel<-abs(sim6p.allelecount.fstats@f4.values$`Z-score`)<1.96
sim6p.allelecount.fstats@f4.values[tst.sel,]
```

```
                  Estimate    bjack mean    bjack s.e.    Z-score
P1,P2;P3,P4   3.595944e-05  3.908225e-05 1.154193e-04  0.3386111
P1,P2;P3,P5   1.018776e-04  1.011325e-04 1.197917e-04  0.8442359
P1,P2;P4,P5   6.591816e-05  6.205022e-05 9.313675e-05  0.6662270
P1,P3;P4,P5   1.309202e-05  1.464589e-05 1.268061e-04  0.1154983
P1,P6;P4,P5   4.031413e-05  4.458874e-05 1.058348e-04  0.4213050
P2,P3;P4,P5 -5.282615e-05 -4.740433e-05 1.302898e-04 -0.3638376
P2,P6;P4,P5 -2.560403e-05 -1.746147e-05 1.085089e-04 -0.1609221
P3,P6;P4,P5   2.722212e-05  2.994286e-05 7.650662e-05  0.3913760
```

```
# 30X Pool-Seq data
tst.sel<-abs(sim6p.readcount30X.fstats@f4.values$`Z-score`)<1.96
as.data.frame(sim6p.readcount30X.fstats@f4.values)[tst.sel,]
```

```
                              Estimate    bjack mean    bjack s.e.     Z-score
Pool1,Pool2;Pool3,Pool4   5.657166e-05  6.087576e-05 1.194004e-04  0.50984558
Pool1,Pool2;Pool3,Pool5   9.947887e-05  1.011014e-04 1.253135e-04  0.80678780
Pool1,Pool2;Pool4,Pool5   4.290721e-05  4.022566e-05 9.502690e-05  0.42330817
Pool1,Pool3;Pool4,Pool5   3.769747e-07 -3.302134e-06 1.277902e-04 -0.02584028
Pool1,Pool6;Pool4,Pool5   4.073843e-05  4.734750e-05 1.053345e-04  0.44949661
Pool2,Pool3;Pool4,Pool5 -4.253023e-05 -4.352780e-05 1.336022e-04 -0.32580156
Pool2,Pool6;Pool4,Pool5 -2.168779e-06  7.121841e-06 1.102031e-04  0.06462467
Pool3,Pool6;Pool4,Pool5   4.036145e-05  5.064964e-05 8.000836e-05  0.63305432
```

In other words, both the allele count and Pool-Seq read count data only provide no evidence against the null hypothesis of treeness at the 95% threshold for quadruplets involving *i)* non-admixed populations (*P1*, *P2*, *P3*, *P4* and *P5*) for configurations consistent with the simulated scenario; and *ii)* the admixed population *P6* for configurations of the form *(P6,X;P4,P5)* where *P4* and *P5* are the two outgroup populations and X=*P1*, *P2* or *P3*. This is actually expected since for these latter quadruplets, the path connecting *P4* and *P5* is not overlapping with either of the paths connecting *P6* to *P1*, *P2* or *P3* in the simulated graph (Figure 1).

### 4.1.4 Population heterozygostity estimates (*heterozygosities* slot of the *fstat* object)

```
# count data  (3 first populations)
head(sim6p.allelecount.fstats@heterozygosities,3)
```

```
    Estimate bjack mean   bjack s.e.
P1 0.1588121  0.1587849 0.0005972556
P2 0.1599502  0.1599070 0.0006057583
P3 0.1591048  0.1590328 0.0005900926
```

```
# 30X Pool-Seq data (3 first populations)
head(sim6p.readcount30X.fstats@heterozygosities,3)
```

```
       Estimate bjack mean   bjack s.e.
Pool1 0.1587839  0.1587670 0.0006012812
Pool2 0.1599319  0.1599083 0.0006096762
Pool3 0.1590771  0.1590230 0.0006024449
```

## 4.2 The *plot_fstats* function for visualization of heterozygosities, $f_2$ (and pairwise $F_{ST}$ and absolute divergence), $f_3$ (and $f_3^\star$) and $f_4$ (and $D$) estimates and their confidence intervals

The *plot_fstats* function (that may be called directly using *plot* on *fstats* objects) allows plotting all or only some (using the *pop.sel*, *pop.f3.target* or *value.range* arguments) of the estimated $f_2$, $f_{ST}$, $f_3$, $f_3^\star$, $f_4$ or $D$ statistics from a *fstats* object. In addition, for $f_3$, $f_3^\star$, $f_4$ and $D$ statistics, the *highlight.signif* argument allows highlighting in red significant (as defined with the *ci.perc* argument) three-population or four-population tests. Some example plots are shown in Figures 8, 9 and 10 which were generated with the following codes:

### 4.2.1 Example of heterozygosities plot (Figure 7)

```
layout(matrix(1:2,1,2,byrow=T))
plot(sim6p.allelecount.fstats,stat.name="heterozygosities",main="Heterozygosities (Allele Count)")
plot(sim6p.readcount30X.fstats,stat.name="heterozygosities",main="Heterozygosities (30X Pool-Seq)")
```

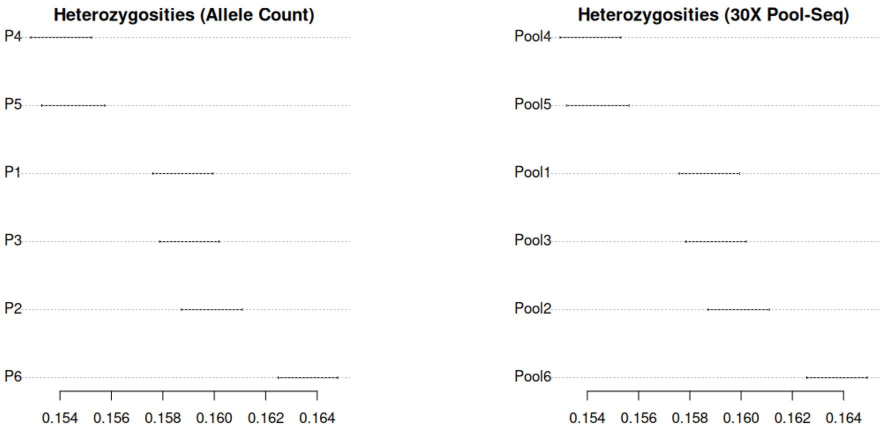

Figure 7: Estimated within-population heterozygosities with their 95% confidence intervals for the allele count and 30X Pool-Seq data sets

As expected, the admixed population *P6* shows the highest heterozygosity.

### 4.2.2 Example of $f_2$ statistics plot (Figure 8)

```
layout(matrix(1:2,2,1,byrow=T))
plot(sim6p.allelecount.fstats,main="F2 (Allele Count)")
plot(sim6p.readcount30X.fstats,main="F2 (30X Pool-Seq)")
```

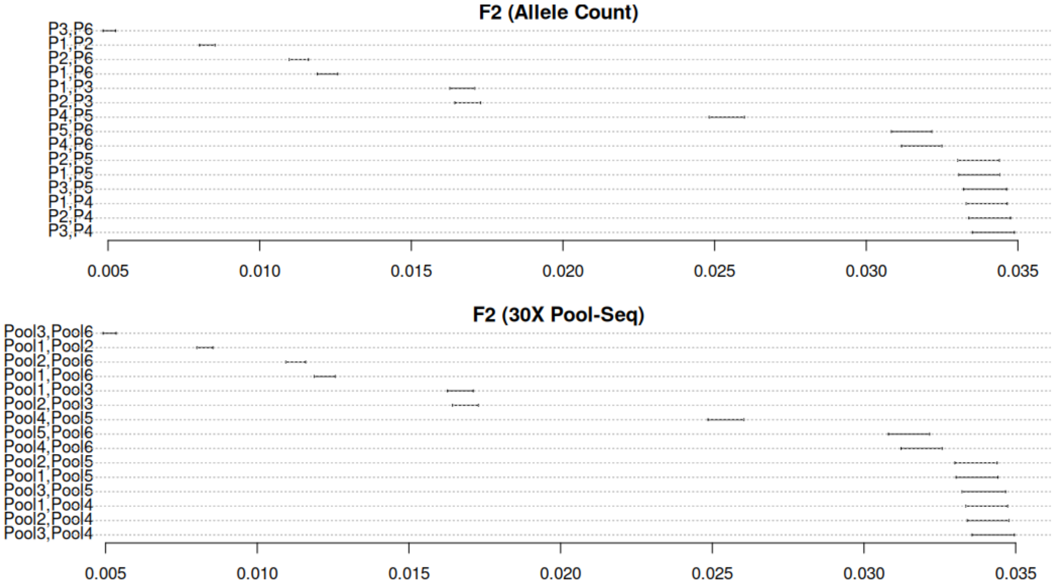

Figure 8: Estimated $f_2$ statistics with their 95% confidence intervals for the allele count and 30X Pool-Seq data sets

Similar plots may be obtained for the pairwise $F_{ST}$ (i.e., scaled $f_2$) by specifying *stat.name="Fst"* (see also the *compute.pairwiseFST* functions described in section 3.2) or absolute divergence (i.e., denominator of $F_{ST}$) by specifying *stat.name="divergence"*.

### 4.2.3 Example of $f_3$ statistics plot (Figure 9)

```
layout(matrix(1:4,2,2,byrow=T))
plot(sim6p.allelecount.fstats,stat.name="F3",main="F3 (Allele Count)")
plot(sim6p.readcount30X.fstats,stat.name="F3",main="F3 (30X Pool-Seq)")
plot(sim6p.readcount30X.fstats,stat.name="F3",pop.f3.target=c("Pool6","Pool1"),
     main="30X Pool-Seq (only F3 with P6 or P1 as target pops)")
plot(sim6p.readcount30X.fstats,stat.name="F3",value.range=c(NA,5e-3),
     main="30X Pool-Seq (only F3 < 5e-3)")
```

Similar plots may be obtained for the scaled $f_3$ (i.e., $f_3^{\star}$) by specifying *stat.name="F3star"*.

### 4.2.4 Example of $f_4$ and $D$ statistics plot (Figure 10)

```
layout(matrix(1:6,3,2,byrow=T))
plot(sim6p.allelecount.fstats,stat.name="Dstat",main="D (Allele Count)")
plot(sim6p.readcount30X.fstats,stat.name="Dstat",main="D (30X Pool-Seq)")
plot(sim6p.allelecount.fstats,stat.name="F4",main="F4 (Allele Count)")
plot(sim6p.readcount30X.fstats,stat.name="F4",main="F4 (30X Pool-Seq)")
plot(sim6p.readcount30X.fstats,stat.name="F4",pop.sel=c("Pool1","Pool2"),
     main="30X Pool-Seq (only F4 with both P1 and P2)")
plot(sim6p.readcount30X.fstats,stat.name="F4",value.range=c(-2e-3,2e-3),
     main="30X Pool-Seq (only -2e-3 < F4 < 2e-3)")
```

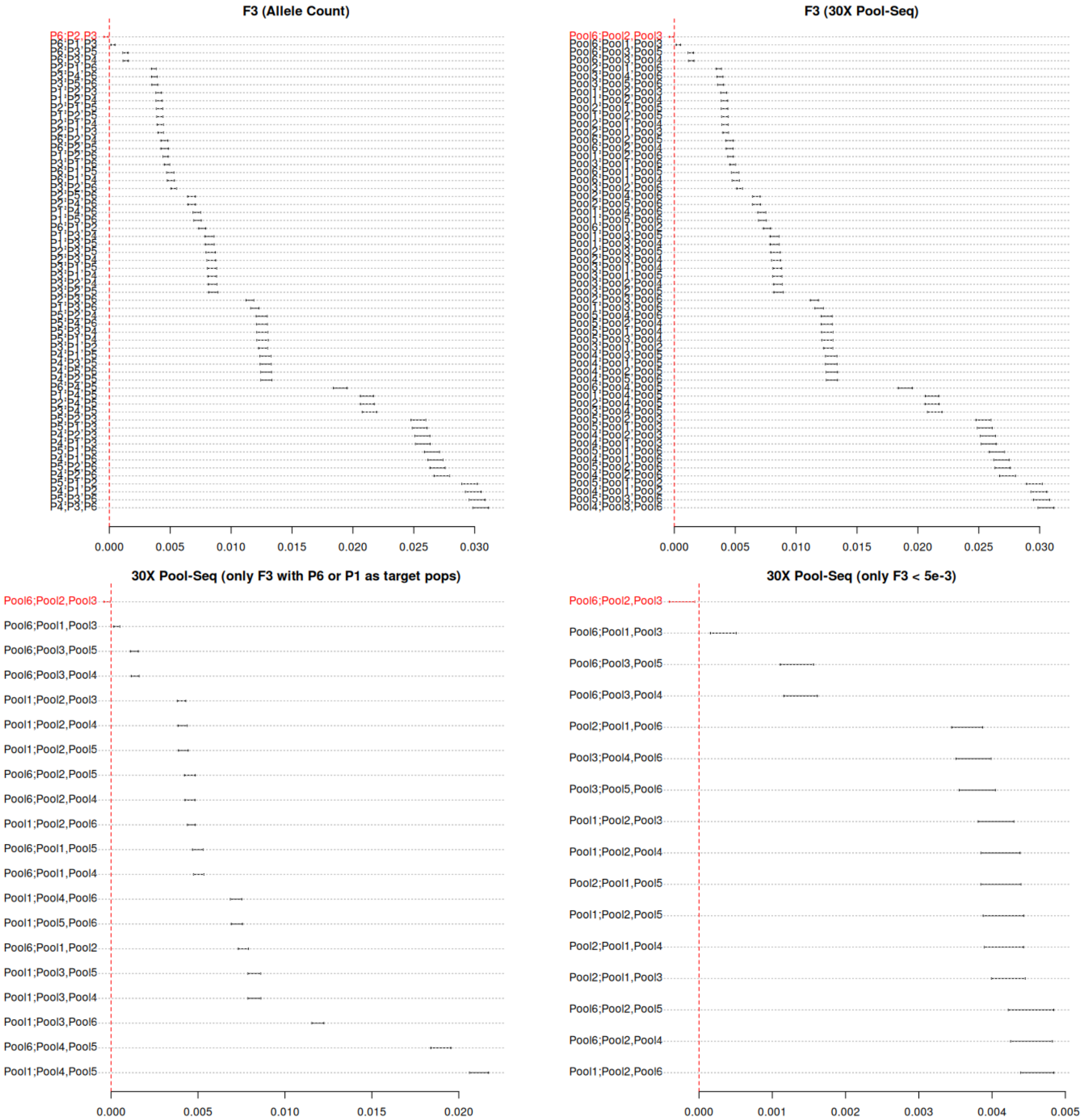

Figure 9: Estimated $f_3$ statistics with their 95% confidence intervals for the allele count and 30X Pool-Seq data sets

## 4.3 Estimating admixture proportions with $f_4$-ratios

Given an admixture graph (assumed to be correct), ratios of $f_4$-statistics (Patterson *et al.* 2012) may provide estimates of the relative contributions of the ancestral sources of a (two-way) admixed populations (*P6* in our example) if outgroups (*P1* and *P4* or *P5* in our example) for the two source population proxies (e.g., *P2* and *P3* in our example) have been sampled. For instance, the proportion $\alpha$ of $P2$-related ancestry in population $P6$ (Figure 1) is equal to:

$$\alpha = \frac{F_4\,(P1, P4; P3, P6)}{F_4\,(P1, P4; P3, P2)} = \frac{F_4\,(P1, P5; P3, P6)}{F_4\,(P1, P5; P3, P2)}$$

The *compute.f4ratio* function implements $f_4$-ratio based estimators of admixture proportion from an *fstats* object. It requires to specify both the numerator (*num.quadruplet* argument) and the denominator

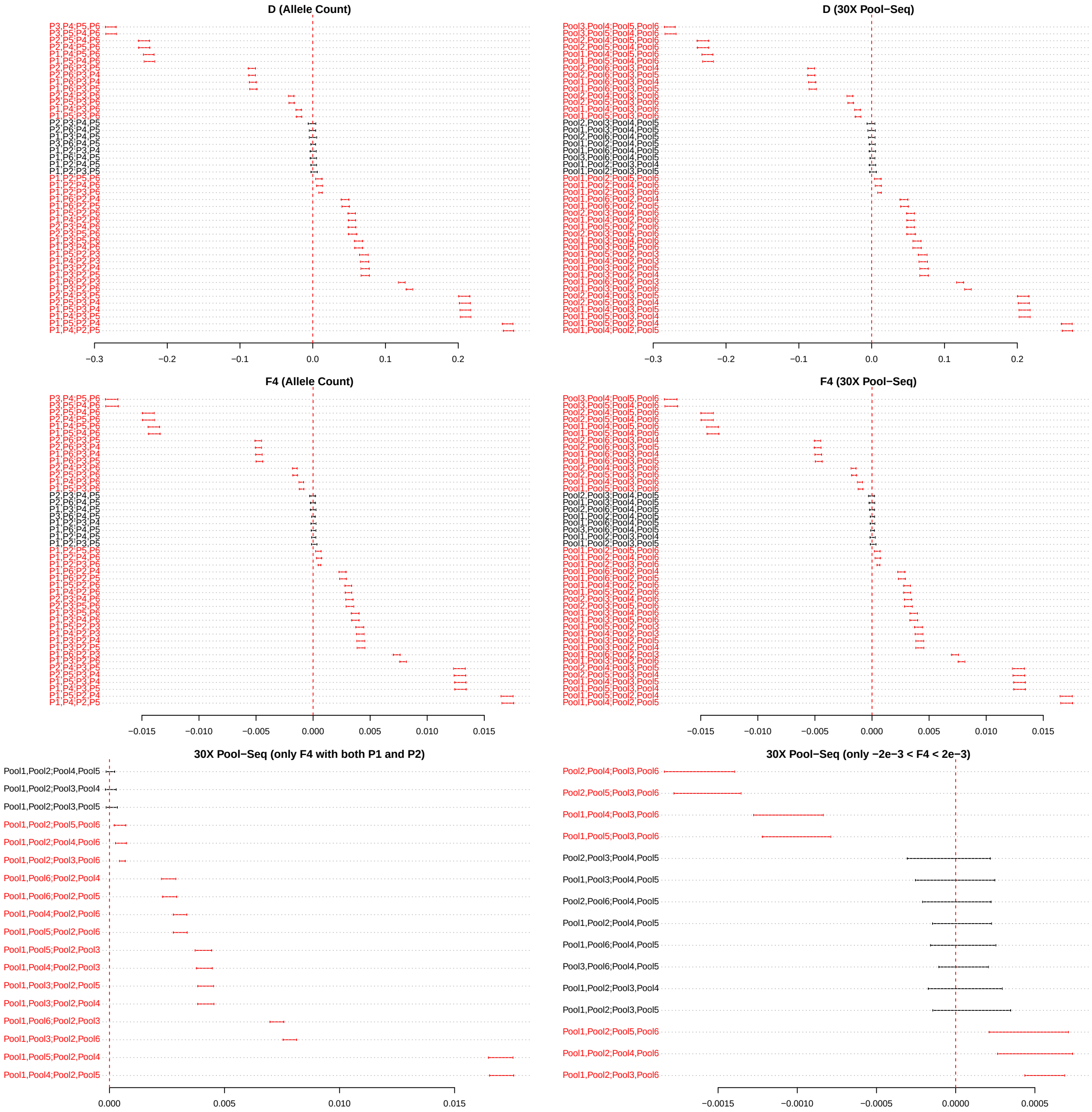

Figure 10: Estimated $f_4$ and $D$ statistics with their 95% confidence intervals for the allele count and 30X Pool-Seq data sets

(*den.quadruplet*) $F_4$ quadruplets. The following examples illustrates how to use the function to the admixture proportion $\alpha$ ($\alpha_{\text{simulated}} = 0.25$) from the P2-related source that contributed to the P6 ancestral population (Figure 1). Note that standard errors and 95% CI interval (i.e., block-jackknife mean ±1.96 s.e.) of the estimated admixture proportions are automatically computed if the input *fstats* object was obtained by running the *compute.fstats* function with *return.F2.blockjackknife.samples = TRUE* (and *nsnp.per.bjack.block>0*) to activate block-jackknife estimation of s.e. as showed below (if not, only the $\alpha$ estimate is provided in the output object):

```
# count data (two possible estimates)
sim6p.allelecount.fstats<-compute.fstats(sim6p.allelecount,nsnp.per.bjack.block = 1000,
                                         verbose=FALSE,return.F2.blockjackknife.samples = TRUE)
compute.f4ratio(sim6p.allelecount.fstats,num.quadruplet = "P1,P4;P3,P6",
                                         den.quadruplet="P1,P4;P3,P2")
```

```
  Estimate bjack mean bjack s.e.    CI95inf    CI95sup 
0.24972091 0.25119795 0.02164778 0.20876830 0.29362759 
```

```
compute.f4ratio(sim6p.allelecount.fstats,num.quadruplet = "P1,P5;P3,P6",
                                        den.quadruplet="P1,P5;P3,P2")
```

```
  Estimate bjack mean bjack s.e.    CI95inf    CI95sup 
0.24629232 0.24679424 0.02133432 0.20497898 0.28860950 
```

```
# 30X Pool-Seq data (two possible estimates)
sim6p.readcount30X.fstats<-compute.fstats(sim6p.readcount30X,nsnp.per.bjack.block = 1000,
                                   verbose=FALSE,return.F2.blockjackknife.samples = TRUE)
compute.f4ratio(sim6p.readcount30X.fstats,num.quadruplet = "Pool1,Pool4;Pool3,Pool6",
                                        den.quadruplet="Pool1,Pool4;Pool3,Pool2")
```

```
  Estimate bjack mean bjack s.e.    CI95inf    CI95sup 
0.25284998 0.25569842 0.02328972 0.21005057 0.30134627 
```

```
compute.f4ratio(sim6p.readcount30X.fstats,num.quadruplet = "Pool1,Pool5;Pool3,Pool6",
                                        den.quadruplet="Pool1,Pool5;Pool3,Pool2")
```

```
  Estimate bjack mean bjack s.e.    CI95inf    CI95sup 
0.24559004 0.24601949 0.02288319 0.20116844 0.29087053 
```

Note that the simulated value ($\alpha = 0.25$) is within the confidence interval and actually less than 0.25 standard error higher than the $F_4$-ratio estimate.

### 4.4 Genome-wide scan of $f-$statistics and related statistics using *sliding.window.fstat*

The *sliding.window.fstat* function allows the computation of multi-locus estimates of any user-defined $f$-statistic (including within-population heterozygosity) or ratios of $f$-statistics over sliding windows along chromosomes (or scaffolds/contigs). This extends the functionality of the previously described *computeFST* function (Section 3.1.1) by enabling genome-wide, window-based scan analyses.

Sliding windows can be defined either by a fixed number of consecutive SNPs or by physical distance in base pairs, using the *window.def* argument (set to *"nsnp"* or *"bp"*, respectively) together with *sliding.window.size* (specifying the number of SNPs or window size in bp).

The multi-locus statistic to be computed is specified using: i) *num.stat*, which gives the name of the statistic (one of *"het"*, *"div"*, *"F2"*, *"Fst"*, *"F3"*, *"F3star"*, *"F4"*, *"Dstat"*, *"Fh"*, *"Fd"*, or *"FdM"*); and ii) *num.pop.idx*, which specifies the populations (or pools) involved, either by name or by index in the input *countdata* or *pooldata* object. Optionally, ratios of statistics can be computed by providing *den.stat* and *den.pop.idx* to define the denominator statistic and its population configuration.

More precisely, the implemented statistics (for both numerator and denominator) include:

- the classical $f$-statistics (*"F2"*, *"Fst"*, *"F3"*, *"F3star"*, *"F4"*, and *"Dstat"*);
- observed within-population heterozygosity (*"het"*, defined as $1-Q_1$) and pairwise population divergence (*"div"*, defined as $1-Q_2$) computed over the ascertained SNPs
- the three introgression-focused statistics $F_h$ (*"Fh"*), $F_d$ (*"Fd"*), and $F_{dM}$ (*"FdM"*) proposed by Martin *et al.* (2015) and Malinsky *et al.* (2021), and detailed in Appendix A.2. These statistics are specifically designed to identify localized genomic regions affected by introgression. It should however be noted that the $F_h$, $F_d$, and $F_{dM}$ statistics can be sensitive to the window size, and may become unstable when the number of informative SNPs is too low. When defining windows in *bp*, it is therefore recommended to filter out windows containing too few SNPs (e.g., fewer than 100).

As an example, we investigate the distribution of introgressed genomic regions in the simulated population *P6*. According to the demographic history shown in Figure 3, *P6* is closely related to *P3* but has recently experienced gene flow affecting approximately 25% of its genome from the more distantly related population *P2*. Populations *P4* (or *P5*) serve as outgroups. Using the simulated Pool-Seq read count data (similar results are obtained with allele count data), we compute below the $D$ and $F_{dM}$ statistics for the $(P3, P6; P2, P4)$ population configuration in 1 Mb sliding windows across the genome:

```
sim6p.readcount30X.dstat<-sliding.windows.fstat(sim6p.readcount30X,
                                  num.pop.idx = c("Pool3","Pool6","Pool2","Pool4"),num.stat = "Dstat",
                                  window.def = "bp",sliding.window.size = 1e6)
```

```
Computing SNP-specific values
For the num.pop.idx combination
Defining Windows
4000 (overlapping) windows identified
  Average (min-max) window sizes (in kb): 1000 ( 1000 - 1000 )
  Average (min-max) nb. of SNPs per window: 235.6 ( 86 - 337 )
Computing window statistics
```

```
head(sim6p.readcount30X.dstat)
```

```
  Chr   Start     End  MidPos CumMidPos nsnp       Value
1   1    1282 1001282  501282    501282  217  0.11814864
2   1  501282 1501282 1001282   1001282  190  0.05269763
3   1 1001282 2001282 1501282   1501282  213  0.07394905
4   1 1501282 2501282 2001282   2001282  222  0.07552336
5   1 2001282 3001282 2501282   2501282  264 -0.05024077
6   1 2501282 3501282 3001282   3001282  280 -0.02336046
```

```
sim6p.readcount30X.fdM <-sliding.windows.fstat(sim6p.readcount30X,
                                  num.pop.idx = c("Pool3","Pool6","Pool2","Pool4"),num.stat = "FdM",
                                  window.def = "bp",sliding.window.size = 1e6,verbose=F)
```

We then generate the corresponding Manhattan plots (Figure 11) using the following code:

```
layout(matrix(c(rep(c(1,2),3),3,3),2,4))
plot(sim6p.readcount30X.dstat$CumMidPos/1e6,sim6p.readcount30X.dstat$Value,
     xlab="Cumulated Position (in Mb)",ylab="Muli-locus D(P3,P6;P2,P4)",
     col=as.numeric(sim6p.readcount30X.dstat$Chr),pch=16)
plot(sim6p.readcount30X.fdM$CumMidPos/1e6,sim6p.readcount30X.fdM$Value,
     xlab="Cumulated Position (in Mb)",ylab="Muli-locus FdM(P3,P6;P2,P4)",
     col=as.numeric(sim6p.readcount30X.fdM$Chr),pch=16)
plot(sim6p.readcount30X.dstat$Value,sim6p.readcount30X.fdM$Value,
     xlab="Muli-locus D(P3,P6;P2,P4)",ylab="Muli-locus FdM(P3,P6;P2,P4)",
     col=as.numeric(sim6p.readcount30X.fdM$Chr),pch=16)
abline(a=0,b=-1,lty=3)
```

As expected from the simulated history, there is an enrichment of windows with negative $D$-statistics, reflecting gene flow between *P2* and *P6* (that produces a negative contribution to the computed $D(P3, P6; P2, P4)$ statistic):

```
summary(sim6p.readcount30X.dstat$Value)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max. 
-0.48301 -0.07689 -0.02736 -0.02865  0.02292  0.32462 
```

The $F_{dM}$ statistic lead to similar observations, but with the sign reverted (see Appendix A.2), resulting in an excess of windows with positive values:

```
summary(sim6p.readcount30X.fdM$Value)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max. 
-0.58787 -0.05876  0.07094  0.06945  0.19531  0.86045 
```

Yet, no discernible signal of adaptive introgression from *P2* is apparent, such as a localized excess of windows

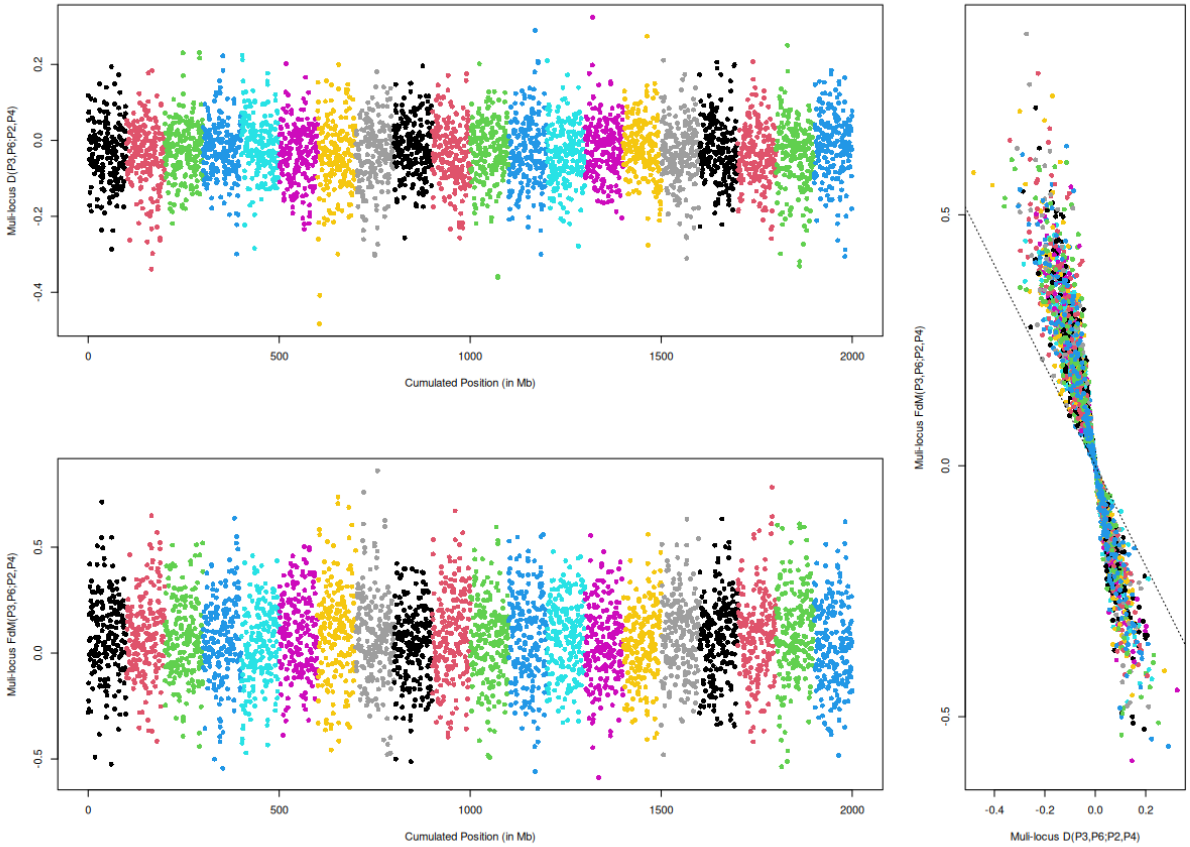

Figure 11: Manhattan plot of the multi-locus $D$- and $F_{dM}$ statistics for the (P3,P6;P2,P4) quadruplet configuration computed over sliding-windows of 1 Mb on the Pool-Seq example data. The 20 simulated chromosomes are represented by alternate colors.

with negative $D$-statistics (or positive $F_{dM}$ values). This is expected, as the data were simulated under neutrality.

Finally, to illustrate the use of ratio of statistics with *sliding.window.fstat*, we show below that the $F_h$ statistics is equivalent to a ratio of $F_4$ and $F_3$ statistics, i.e. $F_h\left(P_3, P_6; P_2, P_4\right) = -\frac{F_4(P_3,P_6;P_2,P_4)}{F_3(P_2;P_3,P_4)}$ (Appendix A.2):

```
sim6p.readcount30X.fh     <-sliding.windows.fstat(sim6p.readcount30X,
                                  num.pop.idx = c("Pool3","Pool6","Pool2","Pool4"),num.stat = "Fh",
                                  window.def = "bp",sliding.window.size = 1e6,verbose=F)
sim6p.readcount30X.fh_alt <-sliding.windows.fstat(sim6p.readcount30X,
                                  num.pop.idx = c("Pool3","Pool6","Pool2","Pool4"),num.stat = "F4",
                                  den.pop.idx = c("Pool2","Pool3","Pool4"),den.stat = "F3",
                                  window.def = "bp",sliding.window.size = 1e6,verbose=F)
cor(sim6p.readcount30X.fh$Value,sim6p.readcount30X.fh_alt$Value)
```

```
[1] -1
```

```
sum((sim6p.readcount30X.fh$Value)-(-1*sim6p.readcount30X.fh_alt$Value))
```

```
[1] 0
```

# 5 Using f-statistics to estimate parameters of admixture graphs

The f-statistics can be used to estimate the parameters (branch lengths and/or admixture proportions) of trees or admixture graphs (i.e., trees including admixture edges) that summarize the demographic history of the surveyed populations. The approach implemented in *poolfstat* to fit admixture graphs from f-statistics is directly inspired (and actually highly similar) to the one used in the *qpGraph* software originally described by Patterson *et al.* (2012; see also Lipson 2020). The core functions used for admixture graph fitting consist of:

- the *generate.graph.params* function to define graph parameters specifying the candidate graph to fit and the underlying f-statistics provided as an *fstats* object (section 4)
- the *fit.graph* function to estimate graph parameters using an optimization algorithm
- the *compare.fitted.fstats* function to assess model fit by comparing estimated and fitted f-statistics (Patterson *et al.* 2012; Lipson 2020)
- the *add.leave* function to evaluate all the possible admixture graphs (or trees) resulting from the addition of a new leave to an existing graph (connected with either non-admixed or admixed edges).

## 5.1 Creating a *graph.params* object with the *generate.graph.params* function

### 5.1.1 Specifying the structure of the admixture graph in a *graph.params* object

Admixture graph specification including the structure of the graph (i.e., topology consisting of edges and, if any admixture proportions) are defined in an object of class *graph.params* detailed in the documentation page accessible with the following command (or the *?* operator):

```
help(graph.params)
```

The *graph.params* objects may be constructed with the *generate.graph.params* function from a user-defined (character) matrix specifying the structure of the admixture graph (or tree if no admixture edges are included) and consisting of three columns defining for each edge (whether admixed or not) i) the child node; ii) the parent node; iii) the admixture proportion (blank for non-admixed edges). As a result, in the input matrix, each admixture event is specified by two rows for the two admixture edges corresponding to the same admixed child node and two different source nodes as the parent node (i.e., source populations). Their third column elements contain the two underlying admixture proportions coded as *a* and *(1-a)* (the parentheses are mandatory and an error message is printed if absent) where *a* is the name of the admixture proportion (names of admixture proportions should not include space). The example below shows the construction of a *graph.params* object specifying the admixture graph for the scenario used to simulate the example data (Figure 1):

```
sim.graph<-rbind(c("P1","P7",""),c("P2","s1",""),c("P3","s2",""),c("P6","S",""),
                c("S","s1","a"),c("S","s2","(1-a)"),c("s2","P8",""),c("s1","P7",""),
                c("P4","P9",""),c("P5","P9",""),c("P7","P8",""),
                c("P8","R",""),c("P9","R",""))
sim.graph.params<-generate.graph.params(sim.graph)
sim.graph.params
```

```
 * * * graph.params Object * * *
Example of useful functions are:
  plot() to visualize the graph (interface for grViz() from the DiagrammeR package)
  fit.graph() to estimate graph parameter values
 * * * * * * * * * * * * * * * *
```

The root is automatically identified by the *generate.graph.params* function (as the node only present in the parent node column) and several checks are made within the function. It is however recommended to check the graph by plotting using the *plot* function. The following code shows how to plot the graph stored in the example *sim.graph.params* object that was generated above (Figure 12):

```
plot(sim.graph.params)
```

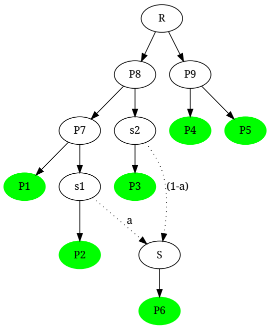

Figure 12: Plot of the admixture graph specifying the simulated scenario

### Rendering and exporting graph plots

The *plot* function applied to *graph.params* objects internally calls the *grViz* function from the *DiagrammeR* package (Iannone 2020) which actually generates an object of class *htmlwidget* that will print itself into HTML in a browser. Also, if run within the *Rstudio* IDE, the graph will be plotted in the View pane[a]. Hence, graph plots may not be easily exported from R into a pdf (or other) device although the following trick may be useful to that end:

```
require(webshot,htmlwidgets,imager)
tmp<-plot(sim.graph.params)           #plot the graph
saveWidget(tmp,"tmp.html") ; webshot("tmp.html","tmp.png")
load.image("tmp.png") %>% autocrop()  %>% plot(axes=F)
```

To insert a graph plot into an external document or edit it (e.g., to change node or edge colors, node names, etc.), one may also directly rely on its *dot* coded definition[b] stored in the *dot.graph* slot of the *graph.params* (or *fitted.graph*, see section 5.2) object outside R. A graph *dot* file may also be generated by specifying an output file name prefix with the *outfileprefix* argument of the *generate.graph.params* function[c].

[a] Regularly clearing the View items using the broom icon is recommended

[b] From the open source graph visualization software *graphviz* (https://graphviz.org/). For instance, *dot* files can be converted into *png* file using the command *dot -Tpng inputgraph.dot* in a Linux terminal. Several online user-friendly implementations also allow very convenient manipulation of *dot* files from a web-browser, see e.g., https://dreampuf.github.io/GraphvizOnline

[c] Alternatively, one may also use the command *writeLines(x@dot.graph,con=outfile)* where *x* is the *graph.params* object and *outfile* is the desired name of the dot output file (e.g., *"out.dot"*)

The *graph.params* object produced by the *generate.graph.params* function includes a symbolic representation of the graph incidence matrix (slot *graph.matrix*) which consists of a $n_l$ leaves by $n_e$ edges matrix containing the edges weight for the paths from each leave to the root[21]. Examples for the *sim.graph.params* object that specifies the simulation scenario are given below:

```
#names of the edges (automatically given)
sim.graph.params@edges.names
```

```
 [1] "P7<->P1" "s1<->P2" "s2<->P3" "S<->P6"  "P8<->s2" "P7<->s1" "P9<->P4" "P9<->P5" "P8<->P7" "R<->P8"  "R<->P9"
```

```
#names of the admixture proportions (automatically given)
sim.graph.params@adm.params.names
```

```
[1] "a"
```

```
#graph incidence matrix
sim.graph.params@graph.matrix
```

[21] In the symbolic representation, the names for the graph edges and admixture proportions correspond to those stored in the *edges.names* and *adm.params.names* slots of the *graph.params* object respectively

```
   P7<->P1 s1<->P2 s2<->P3 S<->P6 P8<->s2 P7<->s1 P9<->P4 P9<->P5 P8<->P7 R<->P8 R<->P9
P1 "1"     "0"     "0"     "0"    "0"     "0"     "0"     "0"     "1"     "1"    "0"   
P2 "0"     "1"     "0"     "0"    "0"     "1"     "0"     "0"     "1"     "1"    "0"   
P3 "0"     "0"     "1"     "0"    "1"     "0"     "0"     "0"     "0"     "1"    "0"   
P6 "0"     "0"     "0"     "1"    "1-a"   "a"     "0"     "0"     "a"     "1"    "0"   
P4 "0"     "0"     "0"     "0"    "0"     "0"     "1"     "0"     "0"     "0"    "1"   
P5 "0"     "0"     "0"     "0"    "0"     "0"     "0"     "1"     "0"     "0"    "1"   
```

**Identifiability of admixture edges**

As mentioned by Pickrell and Pritchard (2012), Patterson *et al.* (2012) and Lipson (2020), the three branch lengths surrounding an admixture event (e.g., edges s1 ↔ S, s2 ↔ S and S ↔ P6 connecting *s1* to *S*; *s2* to *S*; and *S* to *P6* respectively for the admixture event in Figure 12) are not identifiable and can only be estimated jointly in a single compound parameter (e.g., $\zeta = a^2 \times e_{\text{s1}\leftrightarrow\text{S}} + (1-a)^2 \times e_{\text{s2}\leftrightarrow\text{S}} + e_{\text{S}\leftrightarrow\text{P6}}$ in Figure 12) unless samples from the source populations (*s1*, *s2* and/or *S*) or samples from different populations deriving from the same admixed source are available (Lipson 2020). Following Patterson *et al.* (2012) and Lipson (2020), this identifiability issue is solved by nullifying the length of admixture edges (i.e., setting $e_{\text{s1}\leftrightarrow\text{S}} = e_{\text{s2}\leftrightarrow\text{S}} = 0$ in the above example) which may lead to some overestimation of the divergence (branch length) of the admixed population (here *P6*) from its direct admixed ancestor (here S) if the two source populations (here *s1* and *s2*) have experienced strong divergence since their separation from the graph connecting the other populations (which is not the case in the simulated example in which both $e_{\text{s1}\leftrightarrow\text{S}}$ and $e_{\text{s2->S}}$ are equal to 0). This differs from the choice made by Pickrell and Pritchard (2012) in the *Treemix* model consisting of setting $e_{\text{S}\leftrightarrow\text{P6}} = 0$ and $e_{\text{s2}\leftrightarrow\text{S}} = 0$ (if $a > 0.5$) or $e_{\text{s1}\leftrightarrow\text{S}} = 0$ (if $a < 0.5$).

The graph incidence matrix plays a pivotal role for graph fitting since the *fit.graph* function (see section 5.1.2) use it to build the model equations. A (simplified) symbolic representation of these model equations together with expression for the parameters $F_2$, $F_3$ and $F_4$ can also be generated from a *graph.params* object with the *graph.params2symbolic.fstats* function (see Appendix A.3).

### 5.1.2 Creating a *graph.params* object with f-statistics estimates for admixture graph fitting

The f-statistics estimates ($f_2$ and $f_3$) need to be included within the *graph.params* object to allow further fitting of the admixture graph (i.e., estimation of its edge lengths and, if any, admixture proportions) with the *fit.graph* function (see section 5.2). This can be done by providing the *generate.graph.params* function with an *fstats* object including all the $f_2$ and $f_3$ statistics involving the graph leaves (which is usually the case) and block-jackknife estimates of their standard errors and error covariance matrix (i.e., the *compute.fstats* function used to generate the *fstats* object must have been run with *nsnp.per.bjack.block*>0), otherwise an error message is returned[22]. If $n_l$ the number of leaves of the admixture graph and $A$ a "reference" population among the $n_l$ ones[23], the *generate.graph.params* function selects the estimates (block-jackknife means) for the $n_l - 1$ $f_2$ statistics of the form $f_2(A;B)$ (with $B$ a leave population other than $A$) and the $\binom{n_l-1}{2}$ $f_3$ statistics of the form $F_3(A;B,C)$ (with $B$ and $C$ two leave populations other than $A$) which form the basis of the f-statistics vector (see section 4). The basis f-statistics are stored in the *f2.target* and *f3.target* slots of the resulting *graph.params* object (with their corresponding names available in the *f2.target.pops* and *f3.target.pops* slots respectively). In addition, as required to further fit the admixture graph (see section 5.2), the $\frac{n_l(n_l-1)}{2}$ by $\frac{n_l(n_l-1)}{2}$ covariance matrix of the basis f-statistics is stored in the *f.Qmat* slot of the *graph.params* object. Optionally, if available in the *fstats* object, estimates of the leaves heterozygoties (needed to scale fitted branch lengths in drift units, see 5.2.1) are stored in the *Het* slot of the *graph.params* object. The following code shows how to generate a *graph.params* object for the example (allele count) data:

```
sim.graph.params<-generate.graph.params(sim.graph,fstats = sim6p.allelecount.fstats)
```

[22] Conversely, the *fstats* object may include f-statistics involving populations other than the graph leaves, the *generate.graph.params* function selecting only the f-statistics relevant for the fitting of the input graph

[23] by default, $A$ is the first population in the vector of leaves but, although of limited interest, another reference population may be specified with the *popref* argument

```
Total Number of Parameters: 11 (10 edges lengths + 1 adm. coeff.)
Total Number of Statistics: 15 (5 F2 and 10 F3)
```

As shown above, the functions returns the number of parameters $n_{\text{par}} = 2n_l + 2n_a - 3$ of the admixture graph where $n_a$ and $n_l$ are the number of admixture events $n_a$ and the number of leaves $n_l$, respectively (Lipson 2020). Note that the plotting properties of the *graph.params* object remain the same whether *fstats* information is included or not (i.e., the *plot* function may be used as above to generate the representation displayed in Figure 12).

## 5.2 Fitting a graph with the *fit.graph* function

The *fit.graph* functions provides estimate of the parameters (i.e., edge length and admixture proportions) of an admixture graph stored in a *graph.params* object as detailed in Gautier *et al.* (2022) and directly inspired by Patterson *et al.* (2012). Briefly, let $\widehat{\mathbf{f}}$ represent the vector of length $\frac{n_l(n_l-1)}{2}$ (where $n_l$ is the number of graph leaves) of the estimated $f_2$ and $F_3$ basis f-statistics[24] and $\mathbf{g}(\mathbf{e};\mathbf{a}) = \mathbf{X}(\mathbf{a}) \times \mathbf{e}$ the vector of these expected basis f-statistics values given the vector of graph edges lengths $\mathbf{e}$ and the incidence matrix $\mathbf{X}(\mathbf{a})$ that depends on the structure of the graph and the admixture rates $\mathbf{a}$[25]. Let further $\mathbf{Q}$ represent the $\frac{n_l(n_l-1)}{2}$ by $\frac{n_l(n_l-1)}{2}$ covariance matrix of the basis F-statistics estimates estimated by block-jackknife and stored in the slot *f.Qmat* of the input *graph.params* object (see 5.1.2)[26]. The function attempts to find the graph parameter values ($\hat{\mathbf{e}}$ and $\hat{\mathbf{a}}$) that minimize a cost (score of the model) defined as $S(\mathbf{e};\mathbf{a}) = \left(\widehat{\mathbf{f}} - \mathbf{g}(\mathbf{e};\mathbf{a})\right)' \mathbf{Q}^{-1} \left(\widehat{\mathbf{f}} - \mathbf{g}(\mathbf{e};\mathbf{a})\right)$. As mentioned by Patterson *et al.* (2012), given admixture rates $\mathbf{a}$, $S(\mathbf{e};\mathbf{a})$ is actually quadratic in the edge lengths $\mathbf{e}$ allow the *fit.graph* function to rely on the Lawson-Hanson non-negative linear least squares algorithm implemented in the *nnls* function (*nnls* package) to estimate the vector $\hat{\mathbf{e}}$ that minimizes $S(\mathbf{e};\mathbf{a})$ (subject to the constraint of positive edge lengths). Full minimization of $S(\mathbf{e};\mathbf{a})$ is thus reduced to the identification of the admixture rates $\mathbf{a}$ which is performed using the L-BFGS-B method[27] implemented in the *optim* function (*stats* package). The *eps.admix.prop* argument allows specifying the lower and upper bound of the admixture rates to *eps.admix.prop* and *1-eps.admix.prop* respectively. In addition, assuming $\widehat{\mathbf{f}} \sim \mathrm{N}\left(g(\hat{\mathbf{e}};\hat{\mathbf{a}}), \mathbf{Q}\right)$[28], $S(\hat{\mathbf{e}};\hat{\mathbf{a}}) = -2log(L) - K$ where $L$ is the likelihood of the fitted graph and $K = n\log(2\pi) + \log(|\mathbf{Q}|)$ allowing to almost directly derive a $BIC$ (Bayesian Information Criterion) of the fitted graph from the optimized score $S(\hat{\mathbf{e}};\hat{\mathbf{a}})$[29]. The BIC may be used for comparison of different admixture graphs (see section 5.3) providing they were all fitted based on the same vector of f-statistics (i.e., they include the same set of populations). Indeed, when comparing two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ with $BIC$ equal to $BIC_1$ and $BIC_2$, $\Delta_{12} = BIC_2 - BIC_1 \simeq 2\log\left(BF_{12}\right)$ where $BF_{12}$ is the Bayes Factor associated to $\mathcal{G}_1$ and $\mathcal{G}_2$ graph comparison (eq. 9, Kass and Raftery 1995). The (slightly) modified Jeffreys' rule proposed by Kass and Raftery (1995) might further be used to evaluate to which extent the data support $\mathcal{G}_1$ or $\mathcal{G}_2$ with e.g., $\Delta_{12} > 6$ (respectively $\Delta_{12} > 10$) providing "strong" (respectively "very strong") evidence in favor of $G_1$[30].

The *fit.graph* function returns an object of class *fitted.graph* detailed in the documentation page accessible with the following command (or the *?* operator):

```
help(fitted.graph)
```

The following code shows how to fit the example graph stored in the *graph.params* object *sim.graph.params* generated above and some features of the resulting *fitted.graph* object:

[24] stored in the *f2.target* and *f3.target* slots of the input *graph.params* object, see section 5.1.2

[25] If there is no admixture in the graph, $\mathbf{X}(\mathbf{a})$ only contains only 0 or 1

[26] The argument *Q.lambda* of the *fit.graph* function may be used to add a small constant (e.g., $10^{-4}$) to all to the diagonal elements of $\mathbf{Q}$ (i.e., the variance of the basis f-statistics estimates) as proposed by Patterson *et al.* (2012; see also Lipson 2020) to avoid numerical problems. Note that *Qmat.diag.adjust* is always set 0 in the final estimate of the score $S$ or the $BIC$

[27] Limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm with box constraints

[28] i.e., the observed vector of f-statistics is multivariate normal distributed around the expected $g(e;a)$ vector specified by the admixture graph and the covariance structure empirically estimated

[29] $\mathrm{BIC} = S + n_{\text{par}}\log\left(\frac{1}{2}n_l(n_l-1)\right) - \frac{1}{2}n_l(n_l-1)\log(2\pi) - \log(|\ \mathbf{Q}\ |)$

[30] The two thresholds of 6 and 10 on $\Delta_{\text{BIC}}$ corresponds to thresholds of 13 and 21 deciban (units of $10\log_{10}$ scale) on BF. The original Jeffreys' rule considered $BF_{12}$ thresholds of 10, 15 and 20 decibans (corresponding to $\Delta_{\text{BIC}}$ thresholds of 4.6, 6.9 and 9.2) as "strong", "very strong" and "decisive" evidence in favor of model 1.

```
sim.fittedgraph<-fit.graph(sim.graph.params)
```

```
Starting estimation of admixture rates (LBFGS score optimisation)
    Initial score= 316.0469
Estimation ended in 0 m  0 s
    Final Score: 1.004106
    BIC: 276.6086
```

```
#Estimated edge lengths
sim.fittedgraph@edges.length
```

```
    P7<->P1     s1<->P2     s2<->P3      S<->P6     P8<->s2     P7<->s1     P9<->P4     P9<->P5     P8<->P7      R<->P8      R<->P9
0.004079551 0.001970516 0.002148837 0.002111818 0.006334944 0.002228311 0.012858490 0.012529775 0.004160468 0.006445943 0.006445943
```

```
#Estimated admixture proportion
sim.fittedgraph@admix.prop
```

```
[1] 0.2478947
```

```
#Final Score
sim.fittedgraph@score
```

```
[1] 1.004106
```

```
#BIC
sim.fittedgraph@bic
```

```
[1] 276.6086
```

The *fitted.graph* object also stores the output results from the *optim* optimization function in the slot *optim.results*. For instance, in the example below, convergence was reached without any issue (*convergence=0*)[31]:

```
#optim function results (list)
sim.fittedgraph@optim.results
```

```
$par
[1] 0.2478947

$value
[1] 1.004106

$counts
function gradient 
       8        8 

$convergence
[1] 0

$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

As for *graph.params* objects (see section 5.1.1), the *plot* function may be directly applied to *fitted.graph* objects to plot the admixture graph with estimated parameter values[32]. The *admix.fact* and *edge.fact* argument of the *fit.graph* function allow to apply a multiplying factor to the printed branch lengths and admixture proportions (by default *admix.fact*=100, i.e., admixture proportions are printed in %; and *edge.fact*=1000,

[31] In case of convergence problem (i.e., *convergence* not equal to 0), a message detailing execution error in the optimization algorithm is stored in the *optim.results* component named *message*. For more details, see the documentation for the *optim* function with the command *?optim* providing the package *optim* is loaded

[32] the graph is also coded in *dot* format with a character vector stored in the *dot.graph* slot of the resulting *fitted.graph* object. As for *generate.graph.params*, a *dot* file may also be printed out by specifying an output file name prefix with the *outfileprefix* argument of the *fit.graph* function (or using the command *writeLines(x@dot.graph,con=outfile)* where *x* is the *fitted.graph* object and *outfile* is the desired named of the dot output file). See 5.1.1 for more details on how to externally export and customize *dot* files

i.e. edge lengths are printed in ‰). Figure 13 shows the fitted example graph and was generated with the following command:

```
plot(sim.fittedgraph)
```

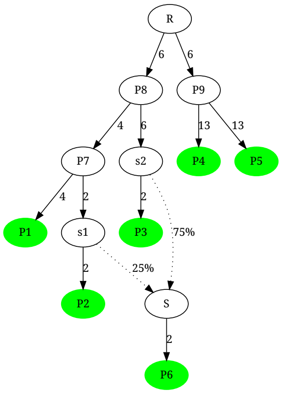

Figure 13: Plot of the admixture graph specifying the simulated scenario with fitted parameter values (x1000 for edge lengths)

**Root position convention**

The two edges from the root node of the graph are not identifiable and only their joint lengths can be estimated. The root position is arbitrarily set in the mid position (i.e., the two root edges have the same length by construction as shown in Figure 13).

### 5.2.1 Scaling of branch lengths in drift units

The estimated edge lengths are in the same scale as the other f-statistics which limits their interpretation since they strongly depend on the SNP ascertainment process (see section 4). Lipson *et al.* (2013) showed however that the lengths may be approximately scaled in genetic drift units (i.e., in units of $\tau = \frac{t}{2N_e}$ where $t$ is a number of generations and $N_e$ is the diploid effective population along the branch) using estimates of overall marker heterozygosities within (i.e., $1 - Q_1$) or across (i.e., $1 - Q_2$) population (Gautier *et al.* 2022). Briefly, for a given edge P ↔ C relating a child node $C$ to its parent node $P$ with an (unscaled) estimated branch length $\hat{e}_{\mathrm{P}\leftrightarrow\mathrm{C}}$, $\hat{\tau}_{\mathrm{P}\leftrightarrow\mathrm{C}} = 2\frac{\hat{e}_{\mathrm{P}\leftrightarrow\mathrm{C}}}{\widehat{h_P}}$ where $\hat{\tau}_{\mathrm{P}\leftrightarrow\mathrm{C}}$ is the estimated branch length scaled in drift units and $\widehat{h_P}$ is the estimated heterozygosity in the (parent) node $P$. The parent node heterozygosities can be estimated from leaves to root by using the property $h_P = h_C + 2e_{\mathrm{P}\leftrightarrow\mathrm{C}} = 0$ that relate the child $C$ and parent $P$ node heterozygosities ($h_C$ and $h_P$ respectively) and $e_{\mathrm{P}\leftrightarrow\mathrm{C}}$ (Lipson *et al.* 2013). When the *drift.scaling* argument is set to *TRUE*[33], the *fit.graph* function returns the edge lengths scaled in drift units in a slot named *edges.length.scaled* together with the estimated node heterozygosities (*nodes.het* slot) as shown below with the example data:

```
sim.fittedgraph.scaled<-fit.graph(sim.graph.params,drift.scaling = TRUE,verbose=FALSE)
```

Note that the obtained results are the same as above with no drift scaling since the latter is a post-processing step independent of admixture graph parameters estimation

```
#Estimated edge lengths
sim.fittedgraph.scaled@edges.length.scaled
```

```
   P7<->P1    s1<->P2    s2<->P3     S<->P6    P8<->s2    P7<->s1    P9<->P4    P9<->P5    P8<->P7     R<->P8     R<->P9
```

[33] providing the input *graph.params* object includes estimates of within-population heterozygosities (see section 5.1.2)

```
0.04964054 0.02405296 0.02631275 0.02516072 0.07378969 0.02711439 0.14321071 0.13954967 0.04846131 0.07700936 0.07700936
```

When plotting the resulting *fitted.graph* objects, branch lengths are displayed in drift scaled units as shown in the Figure 14 below obtained with the following command:

```
plot(sim.fittedgraph.scaled)
```

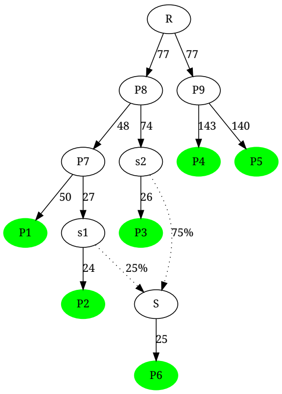

Figure 14: Plot of the admixture graph specifying the simulated scenario with fitted edge lengths scaled in drift units (x1,000)

The estimated branch lengths are close to the simulated ones (Figure 1) as represented below (see Gautier *et al.* 2022 for a more in-depth exploration of the accuracy of the estimates):

```
             P7<->P1    s1<->P2    s2<->P3     S<->P6    P8<->s2    P7<->s1   P9<->P4   P9<->P5    P8<->P7 R<->P8+R<->P9
Estimated 0.04964054 0.02405296 0.02631275 0.02516072 0.07378969 0.02711439 0.1432107 0.1395497 0.04846131     0.1540187
Simulated 0.05000000 0.02500000 0.02500000 0.02500000 0.07500000 0.02500000 0.1500000 0.1500000 0.05000000     0.1500000
```

### 5.2.2 Estimating 95% confidence intervals of the estimated parameters values

Calling the *fit.graph* function with the *compute.ci* argument set to *TRUE* allows deriving (rough) 95% confidence intervals for the admixture graph parameter estimates. The procedure considers each parameter in turn (the other parameters being set to their estimated values) and consists of interpreting the score difference $S_\nu - S^\star$ (where $S^\star$ is the fitted graph score associated with estimated parameter value $\nu^\star$ and $S_\nu$ is the score associated with a parameter value $\nu \neq \nu^\star$) as a likelihood-ratio test statistics following a $\chi^2$ distribution with one degree of freedom (see Gautier *et al.* 2022 for details). The lower and upper boundaries $\nu_{\min}$ and $\nu_{\max}$ of the 95% CI (such $S_\nu - S^\star < 3.84$ for all $\nu_{\min} < \nu < \nu_{\max}$) are then simply computed using a bisection method (with a $10^{-4}$ precision threshold).

```
sim.fittedgraph.with.ci<-fit.graph(sim.graph.params,compute.ci=TRUE,
                                   drift.scaling = TRUE,verbose = FALSE)
#95% CI for the admixture proportion
sim.fittedgraph.with.ci@admix.prop.ci
```

```
   95% Inf.  95% Sup.
a 0.2354879 0.2604726
```

```
#95% CI for edge length (including drift scaled as drift.scaling=TRUE)
sim.fittedgraph.with.ci@edges.length.ci
```

```
           95% Inf.    95% Sup. 95% Inf. (drift scaled) 95% Sup. (drift scaled)
P7<->P1 0.003888322 0.004322695              0.04731364              0.05259916
s1<->P2 0.001785780 0.002153260              0.02179800              0.02628362
s2<->P3 0.001947384 0.002392453              0.02384593              0.02929586
S<->P6  0.001913835 0.002294537              0.02280190              0.02733768
P8<->s2 0.006037994 0.006638187              0.07033080              0.07732187
```

```
P7<->s1 0.002019406 0.002471907              0.02457242              0.03007851
P9<->P4 0.012406433 0.013340492              0.13817596              0.14857899
P9<->P5 0.012040331 0.013011938              0.13409851              0.14491974
P8<->P7 0.003835432 0.004464375              0.04467527              0.05200122
R<->P8  0.006219328 0.006686937              0.07430200              0.07988849
R<->P9  0.006219328 0.006686937              0.07430200              0.07988849
```

The simulated values are all within the estimated 95% CI of the estimated value except for the long $e_{\text{P9}\leftrightarrow\text{P4}}$ and $e_{\text{P9}\leftrightarrow\text{P5}}$ branches that are slightly higher than the upper boundary. Note also that if the 95% CI for the admixture proportion $a$ contains the simulated value of 0.25, it is narrower than the one that may be derived from the block-jackknife estimate of the $F_4$-ratio standard error (see section 4.3 and Gautier *et al.* 2022 for a more thorough evaluation of the estimated 95% CI).

### 5.2.3 Assessing the fit of the graph with the *compare.fitted.fstats* function

As outlined by Patterson *et al.* (2012) and Lipson (2020), a straightforward but highly informative approach to assess the fit of the graph is to compare the f-statistics derived from the fitted admixture graph parameters to the estimated ones, i.e., to evaluate to which extent the fitted F-statistics lie within the confidence intervals of the estimated ones. This may be summarized by computing a Z-score of the residuals for each f-statistics as $Z = \frac{f-g}{\sigma_g^2}$ where $f$ and $g$ stand for the fitted and estimated values respectively and $\sigma_g^2$ is the standard error of $g$. This information is available for the basis f-statistics in the *fitted.outstats* slot of the *fitted.graph* object generated by the *fit.graph* functions, as illustrated below:

```
#Fitted basis F-stats
sim.fittedgraph.scaled@fitted.outstats
```

```
         Stat. value Fitted Value     Z-score
P1,P2    0.008274108  0.008278377  0.03287705
P1,P3    0.016680443  0.016723801  0.20561875
P1,P4    0.033971079  0.033990395  0.05587988
P1,P5    0.033727224  0.033661681 -0.18890298
P1,P6    0.012237561  0.012265163  0.16197671
P1;P2,P3 0.004048392  0.004079551  0.26473869
P1;P2,P4 0.004087474  0.004079551 -0.05972586
P1;P2,P5 0.004149524  0.004079551 -0.52164304
P1;P2,P6 0.004610668  0.004631937  0.19288765
P1;P3,P4 0.008230291  0.008240019  0.05210482
P1;P3,P5 0.008244937  0.008240019 -0.02672913
P1;P3,P6 0.011937638  0.011973206  0.20637613
P1;P4,P5 0.021148713  0.021131906 -0.06164584
P1;P4,P6 0.007189623  0.007208661  0.11673856
P1;P5,P6 0.007234212  0.007208661 -0.16250691
```

However, the fit should be evaluated for all the f-statistics (not only those forming the f-statistics vector-space basis used to fit the admixture graph) with the *compare.fitted.fstats* function. This may in turn provide insights into those populations (or graph edges) leading to poor fit (Lipson 2020). As shown below, the function requires the original *fstats* object (that may contain f-statistics for additional populations not represented in the admixture graph) and the *fitted.graph* object. It then produces a matrix with information on all the fitted stats and the *n.worst.stats* (by default *n.worst.stats*=5) f-statistics, i.e. with the lowest absolute Z-score, are directly printed in the console:

```
sim.fitted.fstats.comp<-compare.fitted.fstats(sim6p.allelecount.fstats,sim.fittedgraph)
```

```
5 Worst fit for:
               Estimated       Fitted   Z-score
P1,P2;P3,P5 1.011325e-04 0.0000000000 -0.8442359
P1,P2;P5,P6 4.611437e-04 0.0005523863  0.7459512
P1,P2;P4,P5 6.205022e-05 0.0000000000 -0.6662270
P2;P1,P5    4.124584e-03 0.0041988263  0.5401737
```

```
P1;P2,P5    4.149524e-03 0.0040795507 -0.5216430
```

```
#Information on the last five fitted F-statistics
tail(sim.fitted.fstats.comp)
```

```
                Estimated        Fitted    Z-score
P2,P6;P3,P4 -4.787097e-03 -4.764545e-03  0.16513038
P2,P6;P3,P5 -4.804558e-03 -4.764545e-03  0.27257790
P2,P6;P4,P5 -1.746147e-05 -3.469447e-18  0.16092207
P3,P4;P5,P6 -1.765179e-02 -1.765643e-02 -0.01671976
P3,P5;P4,P6 -1.762185e-02 -1.765643e-02 -0.12418834
P3,P6;P4,P5  2.994286e-05  3.469447e-18 -0.39137603
```

As shown above, no outlying fitted f-statistics (e.g., with $|Z| > 2$) is observed on the example providing strong support for the fitted admixture graph.

## 5.3 Adding a new leaf to an existing graph

The *add.leaf* function allows to perform iterative calls to the *fit.graph* function in order to evaluate all possible connections of a given leaf (population) to an existing graph with non-admixed and/or admixed edges. Three input arguments are required:

- a graph specified within a *graph.params* object (obtained with the *generate.graph.params* function, see section 5.1.1) or a *fitted.graph* object (obtained with the *fit.graph*, *add.leaf* or *graph.builder* functions, see sections 5.2 and 6.2 below allowing more convenient exploration of the admixture graph space via recursive calls)
- the name of the leaf to add (*leaf.to.add* argument)
- an *fstats* object (see 4) containing *a minima* estimates of all the $f_2$ and $f_3$ statistics (and their standard errors) involving the leaves of the input graph and the leaf to add

By default the function tests all the possible positions of the candidate leaf (*leaf.to.add*) to the graph with non-admixed (including a new rooting with the candidate leaf as an outgroup) or admixed edges. If $n_e$ is the number of non-admixed edges in the original graph, the number of tested graphs equals $n_e + 1$ for non-admixed candidate edges[34] and $\frac{1}{2}n_e(n_e - 1)$ for admixed candidate edges[35]. Optional arguments may allow disabling the evaluation of non-admixed (by setting the *only.test.non.admixed.edges* argument to *TRUE*) or admixed (by setting the *only.test.admixed.edges* argument to *TRUE*) candidate edges.

The object returned by the *add.leaf* function is a list consisting of:

- an element named *n.graphs* corresponding to the number of tested graphs
- an element named *fitted.graphs.list* which consists of a list of *fitted.graph* objects (indexed from 1 to n.graphs and in the same order as the list "graphs") containing the *fit.graph* function results for each candidate graph
- an element named *best.fitted.graph* which is the *fitted.graph* object associated to the candidate graph with minimal *BIC* (see 5.2) among all the *n.graphs* graphs within *fitted.graphs.list*.
- an element named *bic* which is a vector consisting of the *n.graphs BIC* (indexed from 1 to *n.graphs* and in the same order as the *fitted.graphs.list* list).

Use of the *add.leaf* function is illustrated below on the example data by evaluating the connection of the *P6* population on the graph (actually tree) connecting the five other populations (*P1*, *P2*, *P3*, *P4* and *P5*) specified according to the simulated topology in a *graph.params* object (5.1.1) named *sim5p.tree.params* and plotted in Figure 15:

```
sim5p.tree<-sim.graph<-rbind(c("P1","P7",""),c("P2","P7",""),c("P3","P8",""),
                             c("P7","P8",""),c("P4","P9",""),c("P5","P9",""),
                             c("P8","R",""),c("P9","R",""))
```

[34] The newly added node is named "N-"*leaf.to.add*

[35] The three added nodes are named "S-"*leaf.to.add*, "S1-"*leaf.to.add* and "S2-"*leaf.to.add* and the admixture proportions are named with a letter (A to Z depending on the number of admixed nodes already present in the graph)

```
sim5p.tree.params<-generate.graph.params(sim5p.tree)
#Note: fstats object is not necessary at this stage
plot(sim5p.tree.params)
```

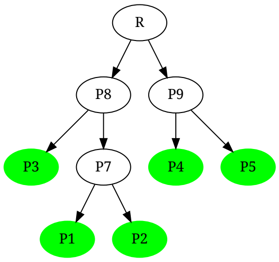

Figure 15: Plot of the five-population graph on which to add the *P*6 population with the *add.leaf* function

All the possible positions of the *P6* population on the *sim5p.tree* graph (here using the f-statistics estimated on the simulated allele count data, see section 4), are tested as follows:

```
add.P6<-add.leaf(sim5p.tree.params,leaf.to.add="P6",
                 fstats=sim6p.allelecount.fstats,
                 verbose=FALSE,drift.scaling=TRUE)
```

Note that the *verbose* option set to *FALSE* allows disabling the printing of the progress and timing of each analysis (which may be useful in practice) and the *drift.scaling* option set to *TRUE* allows passing it to each *fit.graph* call to obtained estimates of branch lengths in drift units (see section 5.2).

The graph with the lowest *BIC* among all the tested graphs may then be plotted by calling the *plot* function on the corresponding *fitted.graph* object stored in the *best.fitted.graph* element of the *add.P6* output list with the following command that generates Figure 16:

```
plot(add.P6$best.fitted.graph)
```

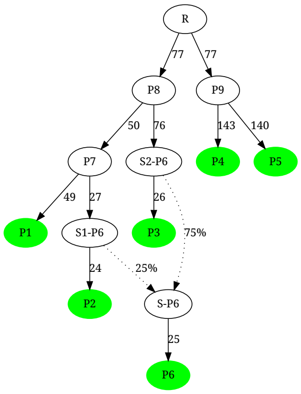

Figure 16: Plot of the graph with the lowest BIC among all the possible graphs connecting *P*6 to the five-population tree tested by the *add.leaf*. The fitted edge lengths are in drift units (x1000) since *drift.scaling* argument was set to *TRUE* when calling *add.leaf*.

The best fitting graph based on *BIC* criterion (stored in the *best.fitted.graph* slot of the output list) is in perfect agreement with the simulated scenario (Figure 1). It actually corresponds to the one directly fitted using the simulated scenario which was represented in Figure 14 above (only the names of the nodes involved in the admixture events differ since automatically given by the *add.leaf* function). In addition, comparisons of the *BIC* of the different graphs provide strong support in favor of this best fitting graph, the second lowest *BIC* being more than *86* units larger i.e., far above the threshold of 10 for very strong evidence (see section 5.2):

```
#D_BIC w.r.t. best fitted BIC
D_BIC=add.P6$bic-add.P6$best.fitted.graph@bic
#5 First lowest DeltaBIC (the first value of zero corresponding to the best fitted graph)
head(sort(D_BIC))
```

```
[1]   0.00000  86.31675  86.31675 279.48967 284.90577 284.90577
```

# 6 Building admixture graph from scratch

Lipson *et al.* (2013) proposed a two-step approach (implemented in the *MixMapper* software[36]) to build admixture graph when prior knowledge about history and relationships of investigated population is limited (which is usually the case). It consists of first building a scaffold tree of unadmixed populations and then adding the remaining populations successively on the graph. Such a supervised approach nevertheless requires to carefully assess at each step the graph fit and possibly try different ordering in the inclusion of populations (or removal of some populations). The *poolfstat* package provides functions to help building scaffold trees that may further be used as input tree for the *add.leaf* function previously described above (5.3) to implement the Lipson *et al.* (2013) two-step approach.

## 6.1 Building scaffold trees of unadmixed populations

In the absence of admixture, the $f_2$ statistics among all pairs of populations are expected to be additive along the paths of the (binary) tree summarizing the history of the populations (Lipson *et al.* 2013). As a result, the (unrooted) tree topology and branch lengths connecting unadmixed populations may be inferred with a neighbor-joining algorithm to derive a scaffold tree for further admixture graph construction. Based on the estimated f-statistics stored in a *fstats* object, the functions described below allows to *i)* identify candidate sets of unadmixed populations among the genotyped ones (*find.tree.popset* function); *ii)* infer a neighbor-joining scaffold tree from a candidate set of unadmixed populations (*rooted.njtree.builder* function); and *iii)* to infer root position based on the consistency of within population heterozygosities between the two resulting partitions of rooted trees (see p1799 in Lipson *et al.* 2013).

### 6.1.1 The *find.tree.popset* function to identify sets of candidate scaffold populations

The *find.tree.popset* function selects maximal sets of unadmixed populations from an *fstats* object[37]. The procedure involves *i)* discarding all the populations showing a significantly negative $f_3$ at a significance theshold specified with the *f3.zcore.threshold* argument (equal to $-1.65$ by default, i.e., 95% significance threshold, see section 4.1.2); and *ii)* keeping only sets of populations with all possible quadruplets passing the $f_4$-based test of treeness i.e., with an absolute $f_4$ Z-score lower than a threshold specified with the *f4.zcore.threshold* argument (equal 2 by default, i.e., 95% significance threshold, see section 4.1.3). The latter step is implemented via a greedy algorithm (that may be run in parallel by specifying a number of threads with the *nthreads* argument) consisting of trying to extend the size of the population sets from all sets of four populations after adding candidate populations one at a time. If the number of populations is large, this algorithm may take some times. Note that increasing (respectively decreasing) *f3.zcore.threshold* toward value closer to 0 may allow decreasing (respectively increasing) the number of initial candidate populations to be tested for inclusion in a set. Similarly, increasing (respectively decreasing) *f4.zcore.threshold* may allow increasing (respectively decreasing) the size of the sets. When applied to the example allele count and read count data, a single set of 5 unadmixed populations (*P1*, *P2*, *P3*, *P4* and *P5*) is retrieved as expected from the simulated scenario (Figure 1):

[36] http://cb.csail.mit.edu/cb/mixmapper/

[37] providing it was Z-scores were estimated for $f_3$ and $f_4$ statistics, i.e., that block-jackknife estimates of s.e. were carried out, see section 4

```
# count data
scaf.pops<-find.tree.popset(sim6p.allelecount.fstats,verbose=FALSE)
scaf.pops$pop.sets
```

```
        [,1] [,2] [,3] [,4] [,5]
PopSet1 "P1" "P2" "P3" "P4" "P5"
```

```
# 30X Pool-Seq data
scaf.pops<-find.tree.popset(sim6p.readcount30X.fstats,verbose=FALSE)
scaf.pops$pop.sets
```

```
        [,1]    [,2]    [,3]    [,4]    [,5]   
PopSet1 "Pool1" "Pool2" "Pool3" "Pool4" "Pool5"
```

As previously mentioned (section 4), for a given set consisting of n populations, a total of $3\binom{n}{4} = \frac{1}{8}n(n-1)(n-2)(n-3)$ quadruplets can be formed. In other words, a given set of four populations A, B, C and D is actually represented by only three quadruplets representative of the three possible unrooted tree topologies[38] i) (A,B;C,D); ii) (A,C;B,D); and iii) (A,D;B,C). Among these, only a single quadruplet is expected to pass the treeness test (i.e., if the correct unrooted tree topology is (A,C;B,D), then the absolute value of the Z-scores associated to f4(A,B;C,D) and f4(A,D;B,C) or their equivalent will be high). For each of identified sets of presumably unadmixed populations, the list of the $\binom{n}{4}$ quadruplets passing the treeness test is given in the *passing.quadruplets* element of the output list as illustrated below:

```
# list of the 15 quadruplets passing the treeness test for the identified set
scaf.pops$passing.quaduplets
```

```
        [,1]                      [,2]                      [,3]                     
PopSet1 "Pool1,Pool3;Pool4,Pool5" "Pool2,Pool3;Pool4,Pool5" "Pool1,Pool2;Pool4,Pool5"
        [,4]                      [,5]                     
PopSet1 "Pool1,Pool2;Pool3,Pool4" "Pool1,Pool2;Pool3,Pool5"
```

In addition, for each of the identified sets, the range of variation of the passing quadruplets is given in the *Z_f4.range* element of the output list:

```
scaf.pops$Z_f4.range
```

```
        Min. |Zscore| Max. |Zscore|
PopSet1    0.02584028     0.8067878
```

When several sets are identified, this information may be useful to prioritize the sets of unadmixed populations (e.g., via a minimax criterion consisting of choosing the set of populations that has the lowest maximal absolute Z-score for its underlying quadruplets that pass the treeness test).

### 6.1.2 The *rooted.njtree.builder* to building (and root) a tree of candidate scaffold populations

The *rooted.njtree.builder* allows first building a Neighbor-Joining[39] of a set of presumably unadmixed populations (as obtained e.g., from the *find.tree.popset* functions) given as a vector of population names in the *pop.sel* argument based on the matrix of their pairwise $f_2$ stored in the provided *fstats* object (*fstats* argument). The resulting (unrooted) tree is then rooted by relying on the property that root $R$ heterozygosity $h_R = 1 - Q_2^{A,B}$ estimated from all the possible pairs of populations $A$ and $B$ that satisfies the property of being only connected through R in the tree (i.e., $A$ and $B$ each belong to one of the two tree partitions defined by the $R$) should be consistent (Lipson *et al.* 2013). In other words, the most likely rooted tree among the $(2n_l - 3)$ possible ones should be the one displaying the narrower range of variation of the $h_R$ estimates. Note that the root position is always placed in the mid-position of the candidate branch.

The object returned by the *rooted.njtree.builder* function is a list consisting of:

[38] for each of these quadruplets, seven other equivalent combinations formed by permuting populations within each pair can be defined as mentioned in the notice p24

[39] relying on the *nj* function from the popular package *ape* (Paradis *et al.* 2004)

- an element named *n.rooted.trees* corresponding to the number of possible rooted binary trees that were evaluated
- an element named *fitted.rooted.trees.list* which consists of a list of *fitted.graph* objects (indexed from 1 to *n.rooted.trees*).
- an element named *best.rooted.tree* which corresponds to the *fitted.graph* object associated with the most likely rooted tree (among all the *fitted.rooted.trees.list* ones) identified as the one displaying the minimal standard deviation over the $h_R$ estimates
- an element named *root.het.est.var* consisting of a matrix of *n.rooted.trees* rows and 4 columns with i) the average estimated root heterozygosity $h_R$ across all the pairs of leaves that are relevant for estimation (see above); ii) the size of the range of variation; iii) the standard deviation of the $h_R$ estimates, and iv) the number of population pairs relevant for estimation
- if *n.edges*>3, an element named *nj.tree.eval* that gives for each evaluated rooted tree, the five $f-$statistics configuration displaying the worst fit, i.e., with the five highest absolute Z-score for the predicted value (obtained by internally calling the *compare.fitted.fstats* function). Note that these are independent of the rooting (so cannot be used to infer the root position).

Use of the *rooted.njtree.builder* function is illustrated below to build the scaffold tree (using Pool-Seq read count data) based on the set of unadmixed populations (identified with *find.tree.popset*) and plotted with the *plot* function applied to the *fitted.graph* object of the resulting list (*best.rooted.tree* element) to generate Figure 17:

```
scaf.tree<-rooted.njtree.builder(fstats=sim6p.readcount30X.fstats,
                                 pop.sel=scaf.pops$pop.sets[1,],plot.nj=FALSE)
```

```
Score of the NJ tree: 0.6884542
Construction of all the 7 possible rooted tree from the NJ tree
(stored as graph in the rooted.graph object of the output list)
```

```
plot(scaf.tree$best.rooted.tree)
```

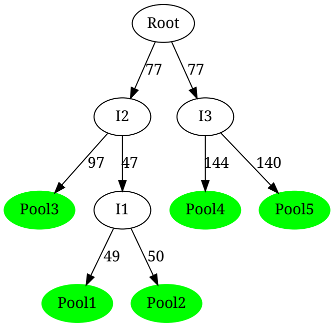

Figure 17: Plot of the rooted scaffold tree of unadmixed populations inferred by *rooted.njtree.builder*. The fitted edge lengths are in a drift scale (x1000).

The fit of the Neighbor-Joining tree can be checked by inspecting the *nj.tree.eval* element:

```
scaf.tree$nj.tree.eval
```

```
                            Estimated       Fitted    Z-score
Pool1,Pool2;Pool3,Pool5 1.011014e-04 0.000000e+00 -0.8067878
Pool1,Pool2;Pool3,Pool4 6.087576e-05 0.000000e+00 -0.5098456
Pool2;Pool1,Pool5       4.120591e-03 4.181198e-03  0.4330143
Pool1;Pool2,Pool5       4.154896e-03 4.094289e-03 -0.4293005
Pool1,Pool2;Pool4,Pool5 4.022566e-05 3.469447e-18 -0.4233082
```

The following range of variation of the $h_R$ estimates were obtained for the different possible root position:

```
scaf.tree$root.het.est.var
```

```
           Mean        Range           sd ncomps
Tree1 0.1814754 0.0234940763 0.0115371318      4
Tree2 0.1810025 0.0228772762 0.0113327886      4
Tree3 0.1857808 0.0155159278 0.0076153034      6
Tree4 0.1907099 0.0007923822 0.0002825614      6
Tree5 0.1880369 0.0113894828 0.0055518382      4
Tree6 0.1878868 0.0111334494 0.0054505348      4
Tree7 0.1833542 0.0152408946 0.0085429545      4
```

The "best" inferred rooted scaffold tree (i.e., the fourth one with both lowest range of $h_R$ variation) is consistent with the simulated scenario. It may further be used as a reference graph to construct the complete admixture graph after adding *Pool6* population using the *add.leaf* function (see 5.3). The obtained graph plotted in Figure 18 with the commands below is very close to the one previously inferred with allele count data (Figure 16) and very strongly supported by the data:

```
add.pool6<-add.leaf(scaf.tree$best.rooted.tree,leaf.to.add="Pool6",
                    fstats=sim6p.readcount30X.fstats,verbose=FALSE,drift.scaling=TRUE)
plot(add.pool6$best.fitted.graph)
```

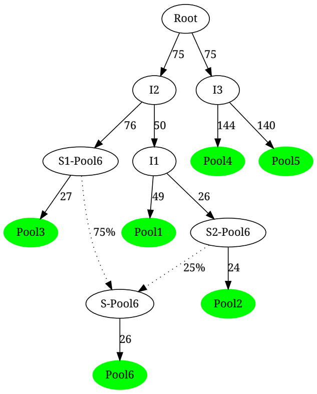

Figure 18: Plot of the graph with the lowest BIC among all the possible graphs connecting *Pool*6 to the scaffold tree of unadmixed population tested by the *add.leaf*. The fitted edge lengths are in drift units (x1000) since *drift.scaling* argument was set to *TRUE* when calling *add.leaf*.

```
#D_BIC w.r.t. best fitted BIC
D_BIC=add.pool6$bic-add.pool6$best.fitted.graph@bic
#5 First lowest DeltaBIC (the first value of zero corresponding to the best fitted graph)
head(sort(D_BIC))
```

```
[1]   0.00000  78.75611  78.75611 254.27229 259.68839 259.68839
```

**Caveats of rooted Neighbor-Joining tree**

In practice, the *rooted.njtree.builder* function should be used with caution since both the Neighbor-Joining tree construction and the heterozygosity-based rooting of the tree may be sensitive to long-branch attraction (most particularly if some highly diverged populations are included). The inferred topology may even violate treeness test for some of the quadruplets (see e.g., the empirical example detailed in the Supplementary Vignette V2 by Gautier *et al.* 2022).

## 6.2 Extending an initial tree (or graph) with the *graph.builder* function

The *graph.builder* function implements an heuristic to carry out a larger exploration of the space of possible graphs (but usually still not exhaustive) obtained from the joint addition of several populations (leaves) given as an input vector (*leaves.to.add* argument) to an existing graph (as generated using the *rooted.njtree.builder* function described above or included in a *graph.params* or *fitted.graph* object) or a list of graphs. The algorithm consists of adding the leaves in the order of the input vector to each of the graphs stored in a heap via successive calls to the *add.leaf* function (section 5.3). More precisely, the heap first consists of the initial input graph (or list of graph) and at each iteration, the function *add.leaf* is used to evaluate all the possible connections of each candidate leaf (with non-admixed or admixed edges) to all the graphs of the heap. For each of the latter, the newly fitted graphs displaying a *BIC* less than *heap.dbic* units (set to 6 by default) away from the best fitting graph (i.e., the one with the lowest *BIC*) included in the new heap. Once all the graphs have been evaluated, if the heap contains more than *max.heap.size* (set to 25 graphs by default) graphs, only the *max.heap.size* graphs with the lowest BIC are kept in the heap. Finally, after testing the latest candidate leaf, only the graphs with a *BIC* less than *heap.dbic* units away from the graph with lowest *BIC* in the heap are retained in the final list of graphs.

The object returned by the function is a list consisting of:

- an element named *n.graphs* corresponding to the final number of graphs
- an element named *fitted.graphs.list* which consists of a list of *fitted.graph* objects (indexed from 1 to *n.graphs*) containing the *fit.graph* function results for each graph
- an element named *best.fitted.graph* which is the *fitted.graph* object associated to the graph with the lowest *BIC* among all the *n.graphs* graphs included in *fitted.graphs.list.*
- an element named *bic* which is a vector containing the *BIC* of the *n.graphs BIC* (indexed from 1 to *n.graphs* and in the same order as *fitted.graphs.list*).

Use of the *graph.builder* function is illustrated below on the PoolSeq example data by starting from an initial rooted tree constructed with the *rooted.njtree.builder* for the three populations *Pool1*, *Pool3*, *Pool4* and *Pool5*. This tree is extended by successively adding the two remaining populations *Pool2* and *Pool6*:

```
#build an initial 3 population trees with "Pool1","Pool3","Pool4" and "Pool5"
init.tree<-rooted.njtree.builder(fstats=sim6p.readcount30X.fstats,
                                 pop.sel=c("Pool1","Pool3","Pool4","Pool5"),plot.nj=FALSE)
```

```
Score of the NJ tree: 0.0006790273
Construction of all the 5 possible rooted tree from the NJ tree
(stored as graph in the rooted.graph object of the output list)
```

```
#adding the three remaining pops
final.graphs<-graph.builder(x=init.tree$best.rooted.tree,leaves.to.add=c("Pool2","Pool6"),
                             fstats=sim6p.readcount30X.fstats)
```

```
####################
 Adding:  Pool2
21 graphs evaluated in 0 h 0 m 1 s
6 graphs stored in the heap
####################
 Adding:  Pool6
261 graphs evaluated in 0 h 0 m 4 s
7 graphs stored in the heap

Final Number of graphs: 7
(min. BIC= 275.8408 )

Overall Analysis Time: 0 h 0 m 5 s ( 282 graphs evaluated)
```

```
#D_BIC w.r.t. to the "true" graph as identified previously (object add.pool6$best.fitted.graph)
D_BIC=final.graphs$bic-add.pool6$best.fitted.graph@bic
#5 First lowest DeltaBIC (the first value of zero corresponding to the best fitted graph)
```

```
head(sort(D_BIC))
```

```
[1] 0.000000 4.667688 4.923348 4.923348 4.923348 5.416100
```

Among the 7 final graphs, the one with the lowest *BIC* is exactly the same as the one plotted in Figure 18 and corresponds to the simulated scenario. It should however be noticed that other alternative graphs are also identified with a good support. Moreover, starting with other population trees (e.g., a three population tree consisting of *Pool1*, *Pool2* and *Pool5*) could result in several graphs with the same support (i.e., $\Delta_{BIC} = 0$) but with a different positioning of the root (not shown). In practice, it may be important to start with scaffold trees that are as large as possible and representative of the structuring of diversity of the represented populations (i.e., not too unbalanced with respect to the leaves to be added). Some prior knowledge about the relationships of some of the populations may also be helpful to that respect. As examplified in the Supplementary Vignette V2 of Gautier *et al.* (2022), it is also highly recommended to test different orders of inclusion (possibly all) of the leaves (as specified in the vector *leaves.to.add*).

# 7 Exporting data for the other packages:

## 7.1 Creating files for the *qpGraph* software (with *graph.params2qpGraphFiles*):

The *graph.params2qpGraphFiles* function allows creating the files required by the *qpGraph* software (Patterson *et al.* 2012) from a *graph.params* object that includes estimates of f-statistics (see section 5.1.2). If *f* is the prefix character specified with the *outfileprefix* argument of the function (by default *f=out*), these are:

- a file named "*f*.graph" that specifies the graph in *qpGraph* format
- a file named "*f*.fstats" with estimates of F-statistics (and their covariance) included in the input *graph.params* object
- a parameter file named "*f*.parqpGraph" to run *qpGraph* (this file may be edited by hand if other options are needed).

The *qpGraph* software (*v7365* and above to allow f-statistics estimates to be provided as input) may then be run on a terminal using the following options:

```
qpGraph -p f.parqpGraph -g f.graph -o out.ggg -d out.dot.input
```

The "*f*.fstats" f-statistics file must be in the same directory or its PATH should be explicitly specified by editing the "*f*.parqpGraph" parameter file. The following example runs *qpGraph* (providing appropriate install of the software) on the *sim.graph.params* object generated in 5.2 (see Figure 13 representing the fitted graph obtained with the *fit.graph* function):

```
graph.params2qpGraphFiles(sim.graph.params,outfileprefix = "sim.graph",verbose = FALSE)
#running qpGraph (assumed to be installed locally) outside R
system("qpGraph -p sim.graph.parqpGraph -g sim.graph.graph -o sim.graph.g -d sim.graph.dot")
#plotting the dot file generated by qpGraph with grViz (as done internally by poolfstat)
require(DiagrammeR)
grViz("sim.graph.dot")
```

Comparison of Figures 19 and 13 shows that the same results are obtained with the two fitting methods (note that edge lengths are not scaled in drift units on the two figures).

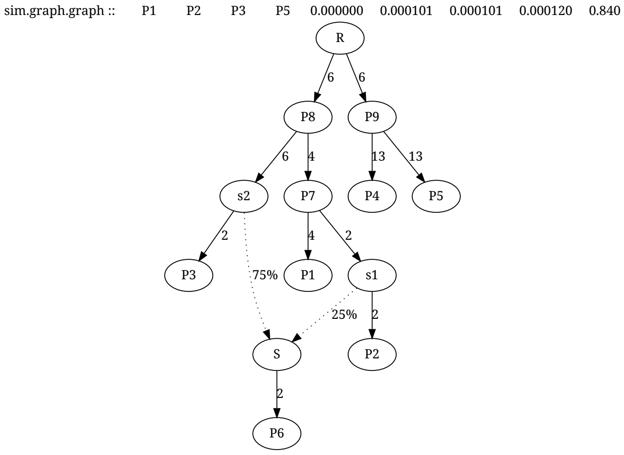
sim.graph.graph :: P1 P2 P3 P5 0.000000 0.000101 0.000101 0.000120 0.840
R
6
6
P8
P9
6
4
13
13
s2
P7
P4
P5
2
4
2
P3
75%
P1
s1
25%
2
S
P2
2
P6

Figure 19: Fitting results obtained by qpGraph on the same data as the one used to generate Figure 13.

## 7.2 Interfacing with the *admixtools2* R package using *compute.fstats*

***igraph* compatibility issue in *admixtools***

Versions of the *admixtools* package up to **2.0.10** are incompatible with recent versions of the *igraph* package, causing some functions to fail. A workaround is to install version **2.1.4** of *igraph*, for example using:

```
remotes::install_version("igraph",version = "2.1.4",
                                 repos = "https://cran.r-project.org")
```

and to load *igraph* before *admixtools* (see https://github.com/uqrmaie1/admixtools/issues/94).

Setting option *return.F2.blockjackknife.samples* to *TRUE* when running *compute.fstats* (section 4.1) allows to include an array with estimates of each pairwise-population $f_2$ for each block-jackknife blocks over the genome (i.e., of dimension $npop \times npop \times nblocks$) in the slot named *F2.bjack.samples* of the output *fstats* objects. This array can be directly imported into the R package *admixtools*[40] that contains highly valuable utilities implementing refined algorithms to build admixture graphs (Maier *et al.* 2023), along with functions to estimate *f-statistics*:

```
#computing fstats and outputting F2 block-jacknife block estimates
sim6p.readcount30X.fstats<-compute.fstats(sim6p.readcount30X,nsnp.per.bjack.block = 1000,
                                          return.F2.blockjackknife.samples = TRUE,verbose=FALSE)
#Example showing how functions from admixtools (here to compute F4)
#can simply be used with the F2 block-jacknife block estimates (slot F2.bjack.samples)
#obtained from poolfstat
require(admixtools)
f4.admixtools=admixtools::f4(sim6p.readcount30X.fstats@F2.bjack.samples)
head(f4.admixtools) #admixtools based on F2 unbiased estimates for Pool-Seq data obtained with poolfstat
```

```
# A tibble: 6 x 8
  pop1  pop2  pop3  pop4        est       se     z         p
  <chr> <chr> <chr> <chr>     <dbl>    <dbl> <dbl>     <dbl>
1 Pool1 Pool2 Pool3 Pool4 0.0000609 0.000119  0.510 6.10e-   1
2 Pool1 Pool3 Pool2 Pool4 0.00419   0.000179 23.3   1.76e-120
3 Pool1 Pool4 Pool2 Pool3 0.00413   0.000175 23.5   2.22e-122
4 Pool1 Pool2 Pool3 Pool5 0.000101  0.000125  0.807 4.20e-   1
```

[40] https://uqrmaie1.github.io/admixtools/articles/admixtools.html

```
5 Pool1 Pool3 Pool2 Pool5 0.00418   0.000175 23.9   5.42e-126
6 Pool1 Pool5 Pool2 Pool3 0.00408   0.000183 22.3   2.16e-110
```

```r
head(sim6p.readcount30X.fstats@f4.values)
```

```
                            Estimate   bjack mean   bjack s.e.   Z-score
Pool1,Pool2;Pool3,Pool4 5.657166e-05 6.087576e-05 1.194004e-04 0.5098456
Pool1,Pool2;Pool3,Pool5 9.947887e-05 1.011014e-04 1.253135e-04 0.8067878
Pool1,Pool2;Pool3,Pool6 5.653400e-04 5.621051e-04 6.409453e-05 8.7699380
Pool1,Pool2;Pool4,Pool5 4.290721e-05 4.022566e-05 9.502690e-05 0.4233082
Pool1,Pool2;Pool4,Pool6 5.087683e-04 5.012293e-04 1.208465e-04 4.1476513
Pool1,Pool2;Pool5,Pool6 4.658611e-04 4.610037e-04 1.275648e-04 3.6138791
```

As expected and illustrated above for $f_4$, the results are exactly the same (although the output matrices are not ordered in the same way) for estimates of *f-statistics*. More interestingly, the *poolfstat* object can further be used to fit or find graphs with *admixtools* functions *admixtools::qpGraph* and *admixtools::find_graphs* respectively as illustrated in the code below used to draw Figure 20. Note that the graph fitted with *admixtools::qpGraph* when specifying the simulated scenario is exactly the same as the similar one fitted with *poolfstat::fit.graph* function (compare Figures 20A and 13 respectively).

```r
library(igraph)
library(admixtools)
require(gridExtra)
#Specifying the simulation graph in admixtools2 format
#(matrix with each row specifying an edge
#i.e. the first two columns of poolfstat graph.params@graph object in reverse order)
adm.sim.graph=rbind(c("R","P9"),c("R","P8"),c("P9","Pool4"),c("P9","Pool5"),
                    c("P8","P7"),c("P8","S2"),c("S2","Pool3"),
                    c("P7","Pool1"),c("P7","S1"),c("S1","Pool2"),
                    c("S1","S"),c("S2","S"),c("S","Pool6"))
#fitting the graph with admixtools::qpgraph function
qpg_results = admixtools::qpgraph(sim6p.readcount30X.fstats@F2.bjack.samples, adm.sim.graph)
#find graph specifying one admixture event and pop P5 as an outgroup
fg_results=admixtools::find_graphs(sim6p.readcount30X.fstats@F2.bjack.samples,numadmix = 1,
                                   outpop = "Pool5",verbose=FALSE)
#plotting the two graphs
g1<-plot_graph(qpg_results$edges,title = "A) qpgraph results")
g2<-plot_graph(fg_results$edges[[which.min(fg_results$score)]],
               title = "B) find_graphs")
grid.arrange(g1,g2,ncol=2)
```

The graph with the lowest score obtained from scratch (and specifying a single admixture events) by using *admixtools::find_graphs* is similar to the true simulated one except for the positioning of the root (which is not identifiable). The root's position was actually influenced by specifying *P5* as an outgroup. The score of the top graph, which outperformed the second-place graphs by over one order of magnitude, is only slightly higher than the score achieved using the true simulated topology:

```r
head(sort(fg_results$score))
```

```
[1]   1.125752  78.821158  78.821207  97.571894 252.436877 259.327032
```

```r
qpg_results$score
```

```
[1] 1.124248
```

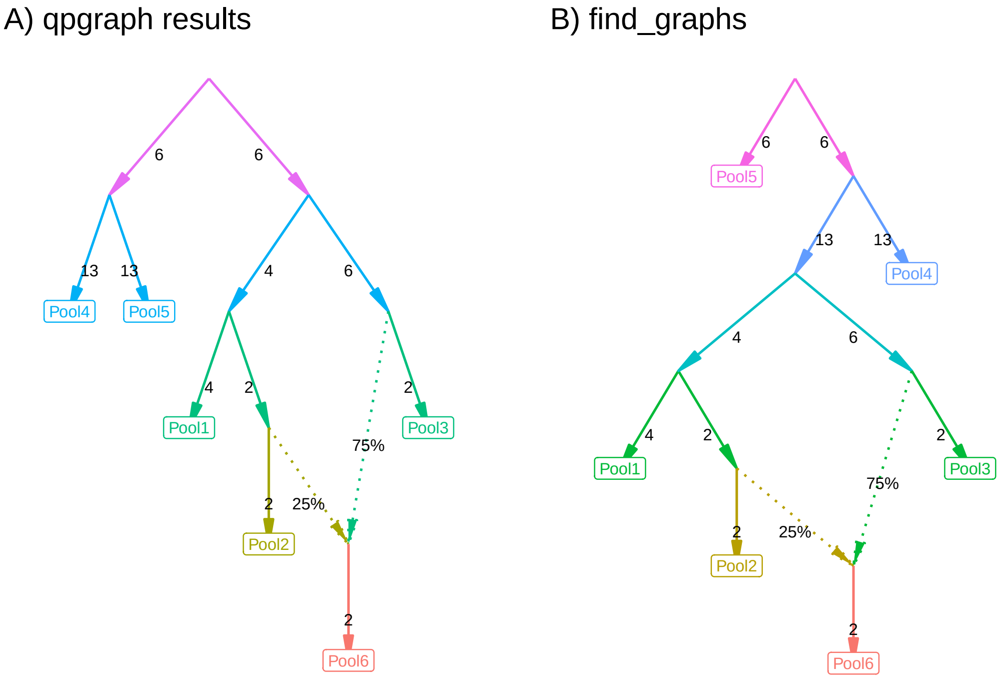

Figure 20: Results obtained when analyzing the *F2.bjack.samples* object generated by *compute.fstats* function for simulated Pool-Seq read count data with the functions *qpgraph* (A) function for simulated Pool-Seq read count data with the functions *qpgraph* (A) and *find_graphs* (B) from the *admixtools* package.

# 8 References

Akey J. M., Zhang G., Zhang K., Jin L., Shriver M. D., 2002 Interrogating a high-density SNP map for signatures of natural selection. Genome Research **12**: 1805–1814.

Collin F.-D., Durif G., Raynal L., Lombaert E., Gautier M., Vitalis R., Marin J.-M., Estoup A., 2021 Extending approximate bayesian computation with supervised machine learning to infer demographic history from genetic polymorphisms using DIYABC random forest. Molecular Ecology Resources **21**: 2598–2613.

Cruickshank T. E., Hahn M. W., 2014 Reanalysis suggests that genomic islands of speciation are due to reduced diversity, not reduced gene flow. Molecular Ecology **23**: 3133–3157.

Durand E. Y., Patterson N., Reich D., Slatkin M., 2011 Testing for ancient admixture between closely related populations. Molecular Biology and Evolution **28**: 2239–2252.

Ferretti L., Ramos-Onsins S. E., Pérez-Enciso M., 2013 Population genomics from pool sequencing. Molecular Ecology **22**: 5561–5576.

Garrison E., Marth G., 2012 Haplotype-based variant detection from short-read sequencing. arXiv: 1207.3907.

Gautier M., Foucaud J., Gharbi K., Cézard T., Galan M., Loiseau A., Thomson M., Pudlo P., Kerdelhué C., Estoup A., 2013 Estimation of population allele frequencies from next-generation sequencing data: Pool-versus individual-based genotyping. Molecular Ecology **22**: 3766–3779.

Gautier M., 2015 Genome-wide scan for adaptive divergence and association with population-specific covariates. Genetics **201**: 1555–1579.

Gautier M., Vitalis R., Flori L., Estoup A., 2022 *f*-statistics estimation and admixture graph construction with pool-seq or allele count data using the R package *poolfstat*. Molecular Ecology Resources **22**: 1394–1416.

Gautier M., Coronado-Zamora M., Vitalis R., 2024 Estimating hierarchical F–statistics from Pool–Seq data. bioRxiv: 2024.11.22.624688.

Green R. E., Krause J., Briggs A. W., Maricic T., Stenzel U., Kircher M., Patterson N., Li H., Zhai W., Fritz M. H.-Y., Hansen N. F., Durand E. Y., Malaspinas A.-S., Jensen J. D., Marques-Bonet T., Alkan C., Prüfer K., Meyer M., Burbano H. A., Good J. M., Schultz R., Aximu-Petri A., Butthof A., Höber B., Höffner B., Siegemund M., Weihmann A., Nusbaum C., Lander E. S., Russ C., Novod N., Affourtit J., Egholm M., Verna C., Rudan P., Brajkovic D., Kucan Ž., Gušic I., Doronichev V. B., Golovanova L. V., Lalueza-Fox C., Rasilla M. de la, Fortea J., Rosas A., Schmitz R. W., Johnson P. L. F., Eichler E. E., Falush D., Birney E., Mullikin J. C., Slatkin M., Nielsen R., Kelso J., Lachmann M., Reich D., Pääbo S., 2010 A draft sequence of the neandertal genome. Science **328**: 710–722.

Hivert V., Leblois R., Petit E. J., Gautier M., Vitalis R., 2018 Measuring genetic differentiation from pool-seq data. Genetics **210**: 315–330.

Iannone R., 2020 *DiagrammeR: Graph/network visualization.*

Karlsson E. K., Baranowska I., Wade C. M., Salmon Hillbertz N. H. C., Zody M. C., Anderson N., Biagi T. M., Patterson N., Pielberg G. R., Kulbokas E. J., Comstock K. E., Keller E. T., Mesirov J. P., Euler H. von, Kämpe O., Hedhammar A., Lander E. S., Andersson G., Andersson

L., Lindblad-Toh K., 2007 Efficient mapping of mendelian traits in dogs through genome-wide association. Nature Genetics **39**: 1321–1328.

Kass R. E., Raftery A. E., 1995 Bayes factors. Journal of the American Statistical Association **90**: 773–795.

Kelleher J., Etheridge A. M., McVean G., 2016 Efficient coalescent simulation and genealogical analysis for large sample sizes. PLoS Computational Biology **12**: e1004842.

Knaus B. J., Grünwald N. J., 2017 Vcfr: A package to manipulate and visualize variant call format data in R. Molecular Ecology Resources **17**: 44–53.

Koboldt D. C., Zhang Q., Larson D. E., Shen D., McLellan M. D., Lin L., Miller C. A., Mardis E. R., Ding L., Wilson R. K., 2012 VarScan 2: Somatic mutation and copy number alteration discovery in cancer by exome sequencing. Genome Research **22**: 568–576.

Kofler R., Orozco-terWengel P., De Maio N., Pandey R. V., Nolte V., Futschik A., Kosiol C., Schlötterer C., 2011 PoPoolation: A toolbox for population genetic analysis of next generation sequencing data from pooled individuals. PloS One **6**: e15925.

Korneliussen T. S., Albrechtsen A., Nielsen R., 2014 ANGSD: Analysis of next generation sequencing data. BMC Bioinformatics **15**: 356.

Li H., Handsaker B., Wysoker A., Fennell T., Ruan J., Homer N., Marth G., Abecasis G., Durbin R., Subgroup 1000. G. P. D. P., 2009 The sequence alignment/map format and SAMtools. Bioinformatics **25**: 2078–2079.

Lipson M., Loh P.-R., Levin A., Reich D., Patterson N., Berger B., 2013 Efficient moment-based inference of admixture parameters and sources of gene flow. Molecular Biology and Evolution **30**: 1788–1802.

Lipson M., 2020 Applying f-statistics and admixture graphs: Theory and examples. Molecular Ecology Resources **20**: 1658–1667.

Maier R., Flegontov P., Flegontova O., Isildak U., Changmai P., Reich D., 2023 On the limits of fitting complex models of population history to *f*-statistics. eLife **12**: e85492.

Malinsky M., Challis R. J., Tyers A. M., Schiffels S., Terai Y., Ngatunga B. P., Miska E. A., Durbin R., Genner M. J., Turner G. F., 2015 Genomic islands of speciation separate cichlid ecomorphs in an East African crater lake. Science **350**: 1493–1498.

Malinsky M., Matschiner M., Svardal H., 2021 Dsuite – fast D-statistics and related admixture evidence from VCF files. Molecular Ecology Resources **21**: 584–595.

Martin S. H., Davey J. W., Jiggins C. D., 2015 Evaluating the use of ABBA-BABA statistics to locate introgressed loci. Molecular Biology and Evolution **32**: 244–257.

McKenna A., Hanna M., Banks E., Sivachenko A., Cibulskis K., Kernytsky A., Garimella K., Altshuler D., Gabriel S., Daly M., DePristo M. A., 2010 The genome analysis toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. Genome Research **20**: 1297–1303.

Nei M., 1973 Analysis of gene diversity in subdivided populations. Proc. Natl. Acad. Sci. USA **70**: 3321–3323.

Paradis E., Claude J., Strimmer K., 2004 APE: Analyses of phylogenetics and evolution in r language.

Bioinformatics **20**: 289–290.

Patterson N., Moorjani P., Luo Y., Mallick S., Rohland N., Zhan Y., Genschoreck T., Webster T., Reich D., 2012 Ancient admixture in human history. Genetics **192**: 1065–1093.

Peter B. M., 2016 Admixture, population structure, and f-statistics. Genetics **202**: 1485–1501.

Pickrell J. K., Pritchard J. K., 2012 Inference of population splits and mixtures from genome-wide allele frequency data. PLoS Genetics **8**: e1002967.

Reich D., Thangaraj K., Patterson N., Price A. L., Singh L., 2009 Reconstructing indian population history. Nature **461**: 489–494.

Rousset F., 2007 Inferences from spatial population genetics. In: Balding DJ, Bishop M, Cannings C (Eds.), *Handbook of statistical genetics*, John Wiley; Sons, Ltd, Chichester, England, pp. 945–979.

Skoglund P., Jakobsson M., 2011 Archaic human ancestry in east asia. Proceedings of the National Academy of Sciences **108**: 18301–18306.

Vitalis R., Gautier M., Dawson K. J., Beaumont M. A., 2014 Detecting and measuring selection from gene frequency data. Genetics **196**: 799–817.

Weir B. S., Cockerham C. C., 1984 Estimating F-statistics for the analysis of population structure. Evolution **38**: 1358–1370.

Weir B. S., 1996 *Genetic data analysis II : Methods for discrete population genetic data.* Sinauer Associates, Sunderland, Mass.

# A Apprendix

## A.1 Block-Jackknife estimation of standard errors

Standard-error of genome-wide estimates of $F_{ST}$ and other $f$-statistics can estimated using a block-jackknife sampling approach (see Patterson *et al.* 2012 and references therein). The algorithm implemented in *poolfstat* consists of dividing the genome into contiguous chunks of a pre-defined number of SNPs (specified by the argument *nsnp.per.bjack.block* of the *computeFST*, *compute.pairwiseFST* or *compute.fstats* functions, see sections 3.1.1, 3.2 and 4 respectively) and then removing each block in turn to quantify the variability of the estimates. If $n_b$ blocks are available and $\widehat{f_i}$ is the estimate of the statistics when removing all the SNPs belonging to block $i$, the standard error $\widehat{\sigma_f}$ of the estimator $\widehat{f}$ of the statistics of interest is computed as:

$$\widehat{\sigma_f} = \sqrt{\frac{n_b - 1}{n_b} \sum_{i=1}^{i=n_b} \left(\widehat{f_i} - \widehat{\mu_f}\right)^2}$$

where $\widehat{\mu_f} = \frac{1}{nb} \sum_{i=1}^{i=nb} \widehat{f_i}$ which may be slightly different from the estimator obtained with all the markers since the latter may include SNPs that are not eligible for block-jackknife sampling (e.g., those at the chromosome/scaffolds boundaries or those belonging to chromosome/scaffolds with less than *nsnp.per.bjack.block* SNPs). Finally, block-jackknife sampling may also be used to obtain estimates of the covariance between the estimates $\widehat{f^a}$ and $\widehat{f^b}$ as (using similar notations):

$$\widehat{\mathbb{C}\mathrm{ov}}\left(\widehat{f^a}; \widehat{f^b}\right) = \frac{n_b - 1}{n_b} \sum_{i=1}^{i=n_b} \left(\widehat{f_i^a} - \widehat{\mu_{f^a}}\right)\left(\widehat{f_i^b} - \widehat{\mu_{f^b}}\right)$$

## A.2 On the $D$- and other related statistics

### A.2.1 Comparing various definitions of the $D$-statistic

The $D$-statistic was originally introduced as a site-pattern–based approach (*a.k.a.*, the *ABBA*-*BABA* framework) to detect deviations from a strictly bifurcating population history caused by gene flow (Green *et al.* 2010; Durand *et al.* 2011). In this original formulation, it considered aligned polymorphic sites across four individual chromosomes each sampled from four populations arranged as a rooted quartet $P_1, P_2; P_3, P_O$, where $P_O$ is an outgroup used to infer the ancestral allele (i.e., not involved in gene flow with the other three populations in their recent history) and $P_3$ is a potential source of introgression. At each (biallelic) SNP, alleles are classified as ancestral $A$ or derived $B$ according to their state in the $P_O$ sampled chromosome. Two discordant site patterns are then of particular interest: *ABBA* sites, where the chromosomes representing populations $P_1$, $P_2$ carry alleles $A$ and $B$, respectively; and *BABA* sites, where they carry alleles $B$ and $A$; the chromosomes representing $P_3$ and $P_O$ carrying in both cases $B$ and $A$, respectively. Indeed, under the null hypothesis of no gene flow, *ABBA* and *BABA* patterns are expected to occur with equal frequency, reflecting incomplete lineage sorting alone. A systematic excess of one pattern over the other indicates asymmetric allele sharing between $P_3$ and either $P_1$ or $P_2$, providing evidence for introgression. The ABBA–BABA framework thus forms the conceptual basis of the $D$-statistic and its various algebraic reformulations described below.

The $D$-statistic was formulated as a comparison of the numbers of sites displaying the *ABBA* and *BABA* patterns over the genome (Green *et al.* 2010, eq. S15.1; Durand *et al.* 2011, eq. 1):

$$D_{dur} = \frac{\sum\limits_{i=1}^{n} \left[C_{ABBA}(i) - C_{BABA}(i)\right]}{\sum\limits_{i=1}^{n} \left[C_{ABBA}(i) + C_{BABA}(i)\right]} = \frac{n_{ABBA} - n_{BABA}}{n_{ABBA} + n_{BABA}}$$

where $C_{ABBA}(i)$ and $C_{BABA}(i)$ are indicator variables that take value 0 or 1 depending on whether an ABBA or BABA pattern is seen at each base $i$ over the $n$ bases the genome; $n_{ABBA} = \sum\limits_{i=1}^{n} C_{ABBA}(i)$ and $n_{BABA} = \sum\limits_{i=1}^{n} C_{BABA}(i)$.

From a population genetics perspective, the meaning of the indicator variables $C_{ABBA}(i)$ and $C_{BABA}(i)$ can be extended to rather specify a topological pattern of allele sharing across the quartet $P_1, P_2; P_3, P_O$, not to restrict a particular allelic state where $B$ is the "derived" allele. The *ABBA* (resp. *BABA*) site patterns will then refer to a configuration in which populations $P_2$ and $P_3$ (resp. $P_1$ and $P_3$) share one allelic state while $P_1$ and $P_4 = P_O$ (resp. $P_2$ and $P_4$) share the other (see Patterson *et al.* 2012). This allows considering SNPs that are polymorphic in the "outgroup" $P_4 = P_O$, for which ancestral-state assignment may be uncertain.

Hence, if $\hat{p}_{ij}$ is the sample (reference) allele frequency at SNP $i$ in population $P_j$, $n_{ABBA}$ and $n_{BABA}$ can be estimated as (see also Malinsky *et al.* 2021, eq. 4):

- $n_{ABBA} = \sum\limits_{i}^{n} \left[\hat{p}_{i1}(1-\hat{p}_{i2})(1-\hat{p}_{i3})\hat{p}_{i4} + (1-\hat{p}_{i1})\hat{p}_{i2}\hat{p}_{i3}(1-\hat{p}_{i4})\right]$
- $n_{BABA} = \sum\limits_{i}^{n} \left[(1-\hat{p}_{i1})\hat{p}_{i2}(1-\hat{p}_{i3})\hat{p}_{i4} + \hat{p}_{i1}(1-\hat{p}_{i2})\hat{p}_{i3}(1-\hat{p}_{i4})\right]$

Note that these estimators are invariant to allele label switching.

With these estimates, the numerator of the $D_{dur}$ are algebraically equal to the opposite of the (unbiased) covariance-based (Patterson *et al.* 2012) and IIS (Gautier *et al.* 2022) estimator of the $F_4\left(P_1, P_2; P_3, P_4\right)$ statistics:

$$\begin{aligned}
S\left(P_1,P_2;P_3,P_4\right) = n_{ABBA} - n_{BABA} &= -\sum_i^n (\hat{p}_{i2}-\hat{p}_{i1})(\hat{p}_{i4}-\hat{p}_{i3}) \\
&= -f_4\left(P_1,P_2;P_3,P_4\right) \\
&= -\frac{1}{2}\left(\widehat{Q}_2^{P_1,P_3} + \widehat{Q}_2^{P_2,P_4} - \widehat{Q}_2^{P_1,P_4} - \widehat{Q}_2^{P_2,P_3}\right)
\end{aligned}$$

Likewise, the denominator of $D_{dur}$ is equal to:

$$\begin{aligned}
n_{ABBA} + n_{BABA} &= \sum_i^n (\hat{p}_1 + \hat{p}_2 - 2\hat{p}_1\hat{p}_2)(\hat{p}_3 + \hat{p}_4 - 2\hat{p}_3\hat{p}_4)) \\
&= (1 - \hat{p}_1\hat{p}_2 - (1-\hat{p}_1)(1-\hat{p}_2))(1 - \hat{p}_3\hat{p}_4 - (1-\hat{p}_3)(1-\hat{p}_4)) \\
&= (1 - \widehat{Q}_2^{P_1,P_2})(1 - \widehat{Q}_2^{P_3,P_4})
\end{aligned}$$

As a result:

$$\begin{aligned}
D_{dur} = -D_{pat} &= -\frac{\sum_i^n (\hat{p}_{i2}-\hat{p}_{i1})(\hat{p}_{i4}-\hat{p}_{i3})}{(1 - \hat{p}_1\hat{p}_2 - (1-\hat{p}_1)(1-\hat{p}_2))(1 - \hat{p}_3\hat{p}_4 - (1-\hat{p}_3)(1-\hat{p}_4))} \\
= -D_{pf} &= -\frac{f_4\left(P_1,P_2;P_3,P_4\right)}{(1 - \widehat{Q}_2^{P_1,P_2})(1 - \widehat{Q}_2^{P_3,P_4})}
\end{aligned}$$

i.e., the $D-$statistics $D_{dur}$ defined in Green *et al.* (2010) and Durand *et al.* (2011), is the opposite of the estimators of $D_{pat}$ defined in Patterson *et al.* (2012), who explicitly defined the numerator as $P_{BABA} - P_{ABBA}$, and $D_{pf}$ defined in Gautier *et al.* (2022). It is easier to see with this original definition that by construction $-1 \leq D \leq 1$ since $D = \frac{n_{ABBA}-n_{BABA}}{(n_{ABBA}-n_{BABA})+2n_{BABA}}$ i.e. $D = 1$ (resp. $D = -1$) if $n_{BABA} = 0$ (resp. $n_{ABBA} = 0$). In addition, the same unbiased estimators can be obtained for the $D$-statistic, including its numerator $S\left(P_1,P_2;P_3,P_4\right)$ (providing all population sample are distinct) by estimating population allele frequencies from allele or read (i.e., for Pool-Seq data without accounting for haploid sample sizes) (Gautier *et al.* 2022).

More generally, at each SNP $i$, the $ABBA$ or $BABA$ pattern observed in the data might be viewed as resulting from sampling alleles from populations with some (unknown) allele frequencies. Under such a random sampling of alleles from populations with allele frequencies $\mathbf{p}_i = \{p_{i1}, p_{i2}, p_{i3}, p_{i4}\}$, the probability of observing an $ABBA$ and $BABA$ pattern at locus $i$ are therefore:

$$\begin{aligned}
P_{ABBA} &= \mathbb{E}\left[C_{ABBA}(i) \mid \mathbf{p}_i\right] = p_{i1}(1-p_{i2})(1-p_{i3})p_{i4} + (1-p_{i1})p_{i2}p_{i3}(1-p_{i4}) \\
P_{BABA} &= \mathbb{E}\left[C_{BABA}(i) \mid \mathbf{p}_i\right] = (1-p_{i1})p_{i2}(1-p_{i3})p_{i4} + p_{i1}(1-p_{i2})p_{i3}(1-p_{i4})
\end{aligned}$$

and $\mathbb{E}\left[C_{ABBA}(i) - C_{BABA}(i) \mid \mathbf{p}_i\right] = -(p_{i2}-p_{i1})(p_{i4}-p_{i3})$

As mentioned above, these formulae are invariant to allele label switching, which guarantees i) label invariance making expectation not depending on which allele is called "derived"; and ii) consistency under sampling making expectation matching the site-pattern definition when populations are replaced by single haplotypes. As a consequence, the unconditional expectation (formally integrating over all possible realizations of the allele frequencies at a SNP $i$ from the evolutionary process) can be obtained by the law of iterated expectations as:

$$\mathbb{E}\left[\mathbb{E}\left[C_{ABBA}(i) - C_{BABA}(i) \mid \mathbf{p}_i\right]\right] = -\mathbb{E}\left[(p_{i2}-p_{i1})(p_{i4}-p_{i3})\right]$$

Further assuming that the sequence $(C_{ABBA}(i) - C_{BABA}(i)_{i\geq 1})$ is exchangeable across loci, i.e. that each locus is a realization of the same evolutionary process, given which the ABBA–BABA pattern at each locus is drawn from the same probability distribution, the (weak) law of large numbers gives:

$$\frac{1}{n}\sum_{i=1}^{n}\left[C_{ABBA}(i)-C_{BABA}(i)\right]\xrightarrow[n\to\infty]{P}-\mathbb{E}\left[(p_2-p_1)(p_4-p_3)\right]=-F_4\left(P_1,P_2;P_3,P_4\right)$$

where the expectation is taken with respect to the joint distribution of allele frequencies induced by the evolutionary history. In other words,

$$\mathbb{E}\left[S\left(P_1,P_2;P_3,P_4\right)\right]=-F_4\left(P_1,P_2;P_3,P_4\right)$$

### A.2.2 The $F_h$, $F_d$ and $F_{dM}$ statistics

As shown by Martin *et al.* (2015), the $D$-statistic may be misleading for detecting local introgression along the genome. To overcome this limitation, several alternative measure have been proposed.

#### A.2.2.1 $F_h$ statistic

The $F_h$ statistic was proposed by Martin *et al.* (2015) as an approximation of the $F_G$ statistic originally (introduced by Durand *et al.* 2011, eq. 8). $F_h$ is defined as:

$$F_h(P_1,P_2;P_3,O)=\frac{S(P_1,P_2;P_3,O)}{S(P_1,P_3;P_3,O)}$$

where $S(P_1,P_2;P_3,O)=n_{ABBA}-n_{BABA}$ measures the difference between the number of sites displaying the $ABBA$ and $BABA$ patterns (see A.2.1). From this definition and above derivation, it directly follows that $F_h(P_1,P_2;P_3,O)$ is algebraically strictly equivalent to a ratio of $F_4$ and $F_3$ statistics:

$$F_h(P_1,P_2;P_3,O)=\frac{S(P_1,P_2;P_3,O)}{S(P_1,P_3;P_3,O)}=\frac{F_4(P_1,P_2;P_3,O)}{F_4(P_1,P_3;P_3,O)}=-\frac{F_4(P_1,P_2;P_3,O)}{F_3(P_3;P_1,O)}$$

since by definition and following notation in Section 4:

$$F_4(A,B;B,C)=\mathbb{E}\left((p_B-p_A)(p_C-p_B)\right)=-\mathbb{E}\left((p_B-p_A)(p_B-p_C)\right)=-F_3(B;A,C)$$

Due the $F_3$ in the denominator, this definition makes it apparent the need for careful correction of the estimators when considering Pool-Seq data (see, Gautier *et al.* 2022). It also follows from this definition, that $F_h$ can be interpreted under a simple demographic scenario as the proportion of the $P_2$ genome originating from $P_3$. Indeed, assuming a phylogeny of the form $(((P_1,P_2),P_3),O)$ and a single introgression event $t_{23}$ generations ago from $P_3$ into $P_2$ affecting a fraction $f$ of the $P_2$ genome, $F_h(P_1,P_2;P_3,O)$ can be approximated as: $F_h(P_1,P_2;P_3,O)\simeq f\frac{t_3}{t_3-t_{23}}\simeq f$ when $t_{23}\ll t_3$, where $t_3$ is the length of the branch leading to $P_3$ (i.e., the introgression is recent relative to the split of $P_3$ and the ancestral $P_2$ population). This actually reproduces in a simplified form eq. 8 in Durand *et al.* (2011).

#### A.2.2.2 $F_d$ statistic

Martin *et al.* (2015) further proposed a modified version of $F_h$, called $F_d$, consisting of defining "dynamically" as the donor population between $P_2$ or $P_3$, the one with the higher derived allele frequency i.e. $P_D=P_3$ when $p_3\geq p_2$ and $P_D=P_2$ when $p_3<p_2$. The $F_d$ statistic was then defined in eq. 6 of Martin *et al.* (2015) and can be redefined (as above for $F_h$) in terms of $F_4$ and $F_3$ statistics as:

$$F_d(P_1,P_2;P_3,O)=\begin{cases}\dfrac{S(P_1,P_2;P_3,O)}{S(P_1,P_3;P_3,O)}=-\dfrac{F_4(P_1,P_2;P_3,O)}{F_3(P_3;P_1;O)}, & \text{if } p_2\leq p_3,\\ \dfrac{S(P_1,P_2;P_3,O)}{S(P_1,P_2;P_2,O)}=-\dfrac{F_4(P_1,P_2;P_3,O)}{F_3(P_2;P_1;O)}, & \text{if } p_2>p_3.\end{cases}$$

In their original formulation, Martin *et al.* (2015) only considered variants that were non polymorphic in the outgroup population $O$ and defined the derived allele as the one not present in $O$. Following Malinsky *et al.* (2021), the definition is generalized to also include variants polymorphic in the outgroup by doing allele polarization according to the outgroup $O$ (i.e., the derived allele is defined as the outgroup minor allele).

#### A.2.2.3 The $F_{dM}$ statistic

Malinsky *et al.* (2015) introduced the $F_{dM}$ statistic (in their Supp. Mat., p. 8) to further stabilize the estimate across windows[41] More precisely, the $F_{dM}$ statistic extends $F_d$ to all configurations of the derived allele frequencies in $P_1$, $P_2$ and $P_3$ (polarization being done according to the observed frequencies in $O$, as mentioned above) can be defined and expressed (as above for $F_h$ and $F_d$ statistics) in terms of $F_4$ and $F_3$ statistics as:

$$F_{dM}(P_1,P_2;P_3,O) = \begin{cases} F_d(P_1,P_2;P_3,O), & \text{si } p_1 \leq p_2, \\ -\dfrac{S(P_1,P_2;P_3,O)}{S(P_1,P_2;P_1,O)} = -\dfrac{F_4(P_1,P_2;P_3,O)}{F_3(P_1;P_2;O)}, & \text{if } p_1 > p_2 \text{ and } p_1 \geq p_3 \\ -\dfrac{S(P_1,P_2;P_3,O)}{S(P_3,P_2;P_3,O)} = -\dfrac{F_4(P_1,P_2;P_3,O)}{F_3(P_3;P_2;O)}, & \text{if } p_1 > p_2 \text{ and } p_1 < p_3 \end{cases}$$

since $-\dfrac{S(P_1,P_2;P_3,O)}{S(P_D,P_2;P_D,O)} = -\dfrac{-F_4(P_1,P_2;P_3,O)}{-F_4(P_D,P_2;P_D,O)} = -\dfrac{F_4(P_1,P_2;P_3,O)}{F_3(P_D;P_2,O)}$.

#### A.2.2.4 Multi-locus estimators of $F$-statistics

As described in Gautier *et al.* (2022) for other statistics defined as ratios of $f$-statistics, the multi-locus (window-based) estimators of $F_h$, $F_d$, and $F_{dM}$ implemented in the *sliding.window.fstat* function are computed using unbiased estimators of the underlying IIS probabilities (4) that define their numerator $F_4$ and denominator $F_3$. Importantly, the multi-locus estimator is calculated as the ratio of the averaged numerator and denominator, rather than as the average of single-locus ratios. This formulation ensures that the estimator remains unbiased and consistent across loci (e.g., Gautier *et al.* 2022), even when the denominator varies between SNPs, as is the case for $F_d$ and $F_{dM}$.

[41] In p. 8 of the Supp. Mat. by Malinsky *et al.* (2015), $p_{i1} < p_{i2}$ should read $p_{i2} < p_{i1}$ (see Malinsky *et al.* 2021). Also eq. 8 in Malinsky *et al.* (2021) is ambiguous since $P_D$ is not well defined for the case when $p_1 > p_2$. We thus refer here, for the definition of $F_{dM}$, to eq. 8 of Malinsky *et al.* (2021) but use the original $P_D$ definition of Malinsky *et al.* (2015) stating that in this case: "$P_D$ is the population (either P1 or P3) that has the higher frequency of the derived allele".

## A.3 Symbolic representation of the $F$ parameters, admixture graph equations and the scaled covariance matrix $\mathbf{\Omega}$ with *graph.params2symbolic.fstats*

Given a graph topology relating the populations stored in a *graph.params* object, the *graph.params2symbolic.fstats* functions provide symbolic representation of the model equations used to fit the underlying admixture graph and all the $F_2$, $F_3$ and $F_4$ parameters together with the scaled covariance matrix of population allele frequencies called $\mathbf{\Omega}$ after GAUTIER (2015). Such representation may be useful for a closer examination of graph properties (or education purposes). The output objects consists of a list with the following elements:

- a character matrix named *model.matrix* consisting of the matrix $\mathbf{M}$ relating the parameters underlying the basis f-statistics and graph edge lengths in the model equations defined as $\mathbf{f} = \mathbf{M}\mathbf{b}$ where $\mathbf{f}$ is the vector of the basis f-statistics (row names of the *model.matrix* $\mathbf{M}$) and $\mathbf{b}$ is the vector of graph edges (column names of *model.matrix* $\mathbf{M}$).
- a character matrix named *omega* consisting of the the scaled covariance matrix of population allele frequencies $\mathbf{\Omega}$ (see e.g., GAUTIER 2015).
- a character vector *F2.equations* consisting of the symbolic representations of all the $\frac{1}{2}n_l(n_l-1)$ $F_2$ parameters (with edge and admixture proportion parameter names as defined in the *graph.params* object)
- a character vector *F3.equations* consisting of the symbolic representations of all the $\frac{1}{2}n_l(n_l-1)(n_l-2)$ $F_3$ parameters (with edge and admixture proportion parameter names as defined in the *graph.params* object)
- a character vector *F4.equations* consisting of the symbolic representations of all the $\frac{1}{8}n_l(n_l-1)(n_l-2)(n_l-3)$ $F_4$ parameters (with edge and admixture proportion parameter names as defined in the *graph.params* object)

These different equations can also be printed in an output text file (with name specified by the *outfile* argument). The following example shows results obtained using the *graph.params* object *sim.graph.params* generated in 5.1.1 (Figure 12) that specifies the simulation scenario (Figure 1):

```
sim.fstats.sym<-graph.params2symbolic.fstats(sim.graph.params,outfile = "Fstats_equations")
```

```
Equations will be printed in file Fstats_equations
```

```
#Model equations matrix
sim.fstats.sym$model.matrix
```

```
             P7<->P1 s1<->P2 s2<->P3 S<->P6 P8<->s2     P7<->s1 P9<->P4 P9<->P5 P8<->P7     R<->P8 R<->P9
F2(P1,P2)    "1"     "1"     "0"     "0"    "0"         "1"     "0"     "0"     "0"         "0"    "0"
F2(P1,P3)    "1"     "0"     "1"     "0"    "1"         "0"     "0"     "0"     "1"         "0"    "0"
F2(P1,P4)    "1"     "0"     "0"     "0"    "0"         "0"     "1"     "0"     "1"         "1"    "1"
F2(P1,P5)    "1"     "0"     "0"     "0"    "0"         "0"     "0"     "1"     "1"         "1"    "1"
F2(P1,P6)    "1"     "0"     "0"     "1"    "a^2-2*a+1" "a^2"   "0"     "0"     "a^2-2*a+1" "0"    "0"
F3(P1;P2,P3) "1"     "0"     "0"     "0"    "0"         "0"     "0"     "0"     "0"         "0"    "0"
F3(P1;P2,P4) "1"     "0"     "0"     "0"    "0"         "0"     "0"     "0"     "0"         "0"    "0"
F3(P1;P2,P5) "1"     "0"     "0"     "0"    "0"         "0"     "0"     "0"     "0"         "0"    "0"
F3(P1;P2,P6) "1"     "0"     "0"     "0"    "0"         "a"     "0"     "0"     "0"         "0"    "0"
F3(P1;P3,P4) "1"     "0"     "0"     "0"    "0"         "0"     "0"     "0"     "1"         "0"    "0"
F3(P1;P3,P5) "1"     "0"     "0"     "0"    "0"         "0"     "0"     "0"     "1"         "0"    "0"
F3(P1;P3,P6) "1"     "0"     "0"     "0"    "1-a"       "0"     "0"     "0"     "1-a"       "0"    "0"
F3(P1;P4,P5) "1"     "0"     "0"     "0"    "0"         "0"     "0"     "0"     "1"         "1"    "1"
F3(P1;P4,P6) "1"     "0"     "0"     "0"    "0"         "0"     "0"     "0"     "1-a"       "0"    "0"
F3(P1;P5,P6) "1"     "0"     "0"     "0"    "0"         "0"     "0"     "0"     "1-a"       "0"    "0"
```

```
#scaled covariance matrix of allele frequencies (Omega)
sim.fstats.sym$omega
```

```
   P1                       P2                               P3
P1 "P7<->P1+P8<->P7+R<->P8" "P8<->P7+R<->P8"                 "R<->P8"
P2 "P8<->P7+R<->P8"         "s1<->P2+P7<->s1+P8<->P7+R<->P8" "R<->P8"
P3 "R<->P8"                 "R<->P8"                         "s2<->P3+P8<->s2+R<->P8"
P6 "P8<->P7*a+R<->P8"       "(P7<->s1+P8<->P7)*a+R<->P8"     "P8<->s2+R<->P8-P8<->s2*a"
P4 "0"                      "0"                              "0"
P5 "0"                      "0"                              "0"
```

```
   P6                                                             P4               P5
P1 "P8<->P7*a+R<->P8"                                             "0"              "0"
P2 "(P7<->s1+P8<->P7)*a+R<->P8"                                   "0"              "0"
P3 "P8<->s2+R<->P8-P8<->s2*a"                                     "0"              "0"
P6 "S<->P6+(P8<->s2+P7<->s1+P8<->P7)*a^2-2*P8<->s2*a+P8<->s2+R<->P8" "0"              "0"
P4 "0"                                                            "P9<->P4+R<->P9" "R<->P9"
P5 "0"                                                            "R<->P9"         "P9<->P5+R<->P9"
```

```
#F2 statistics (first five)
head(sim.fstats.sym$F2.equations)
```

```
[1] "F2(P1,P2) = P7<->P1+s1<->P2+P7<->s1"
[2] "F2(P1,P3) = P7<->P1+P8<->P7+s2<->P3+P8<->s2"
[3] "F2(P1,P6) = P7<->P1+(a^2-2*a+1)*P8<->P7+S<->P6+(a^2-2*a+1)*P8<->s2+a^2*P7<->s1"
[4] "F2(P1,P4) = P7<->P1+P8<->P7+R<->P8+P9<->P4+R<->P9"
[5] "F2(P1,P5) = P7<->P1+P8<->P7+R<->P8+P9<->P5+R<->P9"
[6] "F2(P2,P3) = s1<->P2+P7<->s1+P8<->P7+s2<->P3+P8<->s2"
```

```
#F3 statistics (first five)
head(sim.fstats.sym$F3.equations)
```

```
[1] "F3(P1;P2,P3) = P7<->P1"
[2] "F3(P1;P2,P6) = P7<->P1+a*P7<->s1"
[3] "F3(P1;P2,P4) = P7<->P1"
[4] "F3(P1;P2,P5) = P7<->P1"
[5] "F3(P1;P3,P6) = P7<->P1+(1-a)*P8<->P7+(1-a)*P8<->s2"
[6] "F3(P1;P3,P4) = P7<->P1+P8<->P7"
```

```
#F4 statistics (first five)
head(sim.fstats.sym$F4.equations)
```

```
[1] "F4(P1,P2;P3,P6) = P7<->s1*a"
[2] "F4(P1,P2;P3,P4) = 0"
[3] "F4(P1,P2;P3,P5) = 0"
[4] "F4(P1,P2;P6,P4) = -P7<->s1*a"
[5] "F4(P1,P2;P6,P5) = -P7<->s1*a"
[6] "F4(P1,P2;P4,P5) = 0"
```