Outcrossing Populations

Statistical Genetics Lab - Department of Genetics - Luiz de Queiroz College of Agriculture - University of São Paulo

The following example is intended to show the usage of `OneMap` functions for linkage mapping in outcrossing (non-inbred) populations. With basic knowledge of R syntax, one should have no problems using it. If you are not familiar with R, we recommend reading the vignette Introduction to R.

Hopefully, these examples will be clear enough to help any user to understand its functionality and start using it. You do not need to be an expert in R to build your linkage map, but some concepts are necessary and will help you through the process.

There is a GitHub `OneMap` version which is continuously improved, we strong recommend all users to try this version. In `augusto-garcia/onemap` GitHub page you can find instructions to install the package from GitHub and also more fancy tutorials.

Creating the data file

This step may be quite difficult because the data file is not very simple, and some errors can occur while reading it. The input file format is similar to that used by MAPMAKER/EXP (Lander et al., 1987), so experienced users of genetic analysis software should be already familiar with this scenario.

The input file is a text file, where the first line indicates the cross type, and the second line provides information about the number of individuals, the number of markers, the presence of physical marker locations, and the presence of phenotypic data. The third line contains the sample IDs. Then, the genotype information is included separately for each marker. The character `*` indicates the beginning of information input for a new marker, followed by the marker name. Next, there is a code indicating the marker type, according to Wu's et al. (2002a) notation. It is recommended to check Wu's et al. (2002a) paper before using `OneMap`.

Marker types must be one of the following: `A.1`, `A.2`, `A.3`, `A.4`, `B1.5`, `B2.6`, `B3.7`, `C.8`, `D1.9`, `D1.10`, `D1.11`, `D1.12`, `D1.13`, `D2.14`, `D2.15`, `D2.16`, `D2.17` or `D2.18`, each one corresponding to a row of the following table:

			Parent		Offspring	
		Crosstype	Cross	Observed bands	Observed bands	Segregation
A		1	$ab \times cd$	$ab \times cd$	ac, ad, bc, bd	$1:1:1:1$
		2	$ab \times ac$	$ab \times ac$	a, ac, ba, bc	$1:1:1:1$
		3	$ab \times co$	$ab \times c$	ac, a, bc, b	$1:1:1:1$
		4	$ao \times bo$	$a \times b$	ab, a, b, o	$1:1:1:1$
B	B_1	5	$ab \times ao$	$ab \times a$	$ab, 2a, b$	$1:2:1$
	B_2	6	$ao \times ab$	$a \times ab$	$ab, 2a, b$	$1:2:1$
	B_3	7	$ab \times ab$	$ab \times ab$	$a, 2ab, b$	$1:2:1$
C		8	$ao \times ao$	$a \times a$	$3a, o$	$3:1$
D	D_1	9	$ab \times cc$	$ab \times c$	ac, bc	$1:1$
		10	$ab \times aa$	$ab \times a$	a, ab	$1:1$
		11	$ab \times oo$	$ab \times o$	a, b	$1:1$
		12	$bo \times aa$	$b \times a$	ab, a	$1:1$
		13	$ao \times oo$	$a \times o$	a, o	$1:1$

		Parent		Offspring	
D_2	14	$cc \times ab$	$c \times ab$	ac, bc	$1:1$
	15	$aa \times ab$	$a \times ab$	a, ab	$1:1$
	16	$oo \times ab$	$o \times ab$	a, b	$1:1$
	17	$aa \times bo$	$a \times b$	ab, a	$1:1$
	18	$oo \times ao$	$o \times a$	a, o	$1:1$

Letters `A`, `B`, `C` and `D` indicate the segregation type (*i.e.*, `1:1:1:1`, `1:2:1`, `3:1` or `1:1`, respectively), while the number after the dot (*e.g.*, `A.1`) indicates the observed bands in the offspring. The paper cited above gives details with respect to marker types; we will not discuss them here, but it is easy to see that each marker is classified based on the band patterns of parents and progeny.

Finally, after each marker name, comes the genotype data for the segregating population. The coding for marker genotypes used by `OneMap` is also the same one proposed by Wu et al. (2002a), and the possible values vary according to the specific marker type. Missing data are indicated with the character `-` (minus sign), and an empty space separates the information for each individual. Phenotype information, if present, follows genotypic data with a similar structure. Details are found in the help of function `read_onemap`.

Here is an example of such a file for 10 individuals and 5 markers (the three zeros in the second line indicate that there is no chromosome information, physical position information or phenotypic data, respectively). It is very similar to a MAPMAKER/EXP file, but has additional information about the cross type.

```
data type outcross
10 5 0 0 0
I1 I2 I3 I4 I5 I6 I7 I8 I9 I10
*M1 B3.7 ab ab - ab b ab ab - ab b
*M2 D2.18 o - a a - o a - o o
*M3 D1.13 o a a o o - a o a o
*M4 A.4 ab b - ab a b ab b - a
*M5 D2.18 a a o - o o a o o o
```

Notice that once the marker type is identified, no variations of symbols presented on the table for the **observed bands** is allowed. For example, for `A.1`, only `ac`, `ad`, `bc` and `bd` genotypes are expected (plus missing values). **We notice in FAQs that this is a common mistake made by users, so please be careful**.

The input file must be saved in text format, with extensions like `.raw`. It is a good idea to open the text file called `onemap_example_out.raw` (available with `OneMap` and saved in the directory you installed it) to see how this file should be. You can see where `OneMap` is installed using the command

```r
system.file(package = "onemap")
```

Simulating the data file (new!)

`OneMap` offers wrapper functions for PedigreeSim (Voorrips and Maliepaard, 2012) software to simulate mapping populations.

Run PedigreeSim (new!)

First, download PedigreeSim java file. It will require java installed. You can run PedigreeSim directly and use its output files in `OneMap` or you can use the wrapper function named `run_pedsim` that facilitates this procedure.

The function do not provide every possibility offered by PedigreeSim software. If you want to change any parameter that is not available in the function, please use the PedigreeSim software directly. Here is the software documentation for more information.

```
# For outcrossing population
run_pedsim(chromosome = "Chr1", n.marker = 54,
           tot.size.cm = 100, centromere = 50,
           n.ind = 200,
           mk.types = c("A1", "A2", "A3", "A4", "B1.5", "B2.6", "B3.7",
                        "C.8", "D1.9", "D1.10", "D1.11", "D1.12", "D1.13",
                        "D2.14", "D2.15", "D2.16", "D2.17", "D2.18"),
           n.types = rep(3,18), pop = "F1",
           path.pedsim = "path/to/PedigreeSim/",
           name.mapfile = "mapfile.txt",
           name.founderfile="founderfile.gen",
           name.chromfile="sim.chrom",
           name.parfile="sim.par",
           name.out="sim_out")
```

The function allows creating outcrossing, f2 intercross, and backcross populations from bi-parental. You can define it in `pop` argument. You must change the `path.pedsim` to the path where the PedigreeSim.jar is stored in your system. You can also define the number of chromosomes (argument `chromosome`), the number of markers in each chromosome (`n.marker`), the total size of the groups in cM (`tot.size.cm`), the position of centromere (`centromere`), number of individuals in the population (`n.ind`), the marker types (`mk.types`, see the table in session `Creating the data file`) and the number of markers of each type (`n.types`).

We suggest you open the output files `founderfile`, `chromfile`, `mapfile` and `parfile` to check if they agree with your intentions before proceed to other analysis.

Convert PedigreeSim output to `OneMap` raw data (new!)

Once you run PedigreeSim (directly or using our `run_pedsim` function), you should have the output file `genotypes.dat` (see an example file called `sim_cod_out_genotypes.dat` in `OneMap` extdata directory). To convert this to `OneMap` raw file, you just need to specify the cross type (`cross`), which ones are the parents (`parent1` and `parent2`) and if you want to include missing genotypes (`miss.perc`). Only cross types `outcross` and `f2 intercross` are supported by now.

```
# For outcrossing population
pedsim2raw(genofile = "sim_out_genotypes.dat",
           cross="outcross", parent1 = "P1", parent2 = "P2",
           out.file = "sim_out.example.raw", miss.perc = 0)
```

```
# For outcrossing population
pedsim2raw(genofile = system.file("extdata/sim_out_genotypes.dat",
                                  package = "onemap"),
           cross="outcross", parent1 = "P1", parent2 = "P2",
           out.file = "sim_out.example.raw", miss.perc = 0)
```

Convert PedigreeSim output to VCF file (new!)

The same output file from PedigreeSim, the `genotypes.dat` can be used to simulate a VCF file together with the PedigreeSim `mapfile` and `chrom`. The advantages to simulate a VCF instead of `OneMap` raw file is that VCF is a standard file format and can store a lot of other information included the allele counts (usually in the field `AD` or `DPR`). The `pedsim2vcf` function can simulate the allele counts using negative binomial or updog distributions (argument `method`). The main parameters for the distributions are defined with arguments `mean.depth`, that defines the mean allele depth in the progeny, `p.mean.depth` that defines the mean allele depth in the parents, argument `disper.par` defines the dispersion parameter, `mean.phred` sets the mean Phred score of the sequencing technology used. The function can also simulate missing data (`miss.perc`). Through arguments `pos` and `chr` you can define vectors with physical position and chromosome of each marker. With argument `haplo.ref` you specify which one of the haplotypes in `genotypes.dat` will be considered the reference. Establishing `phase` as TRUE, the VCF will have phased genotypes. After allele counts are simulated, the genotypes are re-estimated using a binomial distribution. The VCF generated by this function only has one or two FORMAT fields, the GT and AD (if `counts = TRUE`). **Dominant markers are not supported by this function**.

```
# Dominant markers are not supported, then,
# we simulate other dataset with only codominant markers
run_pedsim(chromosome = "Chr1", n.marker = 42,
           tot.size.cm = 100, centromere = 50,
           n.ind = 200, mk.types = c("A1", "A2", "B3.7", "D1.9",
                                     "D1.10", "D2.14", "D2.15"),
           n.types = rep(6,7), pop = "F1",
           path.pedsim = "/home/rstudio/onemap/",
           name.mapfile = "mapfile_out.txt",
           name.founderfile="founderfile.gen",
           name.chromfile="sim_out.chrom", name.parfile="sim.par",
           name.out="sim_cod_out")
```

```
# For outcrossing population
pedsim2vcf(inputfile = "sim_cod_out_genotypes.dat",
           map.file = "mapfile_out.txt",
           chrom.file = "sim_out.chrom",
           out.file = "simu_cod_out.vcf",
           miss.perc = 0,
           counts = TRUE,
           mean.depth = 100,
           p.mean.depth = 100,
           chr.mb = 10,
           method = "updog",
           mean.phred = 20,
           bias=1,
           od=0.00001,
           pos=NULL,
           chr=NULL,
           phase = FALSE,
           disper.par=2)
```

```
#> Counts simulation changed 0.2828854 % of the given genotypes
```

The function prints on screen the percentage of genotypes changed between simulated by PedigreeSim and re-estimated using a binomial distribution and simulated allele counts. Notice that this percentage increase if you reduce the mean depth or increase bias or dispersion parameter.

In the session `Importing data from VCF file` below, you will see how to import VCF files as `OneMap` objects.

Importing data

Once the input file is created, the data can be loaded and saved into an R `onemap` object. The function used to import data is named `read_onemap`. Its usage is quite simple:

```r
onemap_example_out <- read_onemap(dir = "C:/workingdirectory",
                                  inputfile = "onemap_example_out.raw")
```

The first argument is the directory where the input file is located, so modify it accordingly. The second one is the data file name. In this example, an object named `onemap_example_out` was created. If you leave the argument `dir` blank, the file will be loaded from your `working directory`.

You can change the working directory in R using function `setwd()` or in the toolbar clicking `File -> Change dir`. If you set your working directory to the one containing the input file, you can just type:

```r
onemap_example_out <- read_onemap(inputfile = "onemap_example_out.raw")
```

If no error has occurred, a message will display some basic information about the data, such as number of individuals and number of markers:

```
#>  Working...
#>
#>  --Read the following data:
#>  Type of cross:           outcross
#>  Number of individuals:   100
#>  Number of markers:       30
#>  Chromosome information:  no
#>  Position information:    no
#>  Number of traits:        3
#>  Missing trait values:
#>   Pheno1: 0
#>   Pheno2: 3
#>   Pheno3: 0
```

Because this particular dataset is distributed along with the package, as an alternative you can load it typing

```r
data("onemap_example_out")
```

Loading the data creates an object of class `onemap`, which will further be used in the analysis. R command `print` recognizes objects of this class. Thus, if you type:

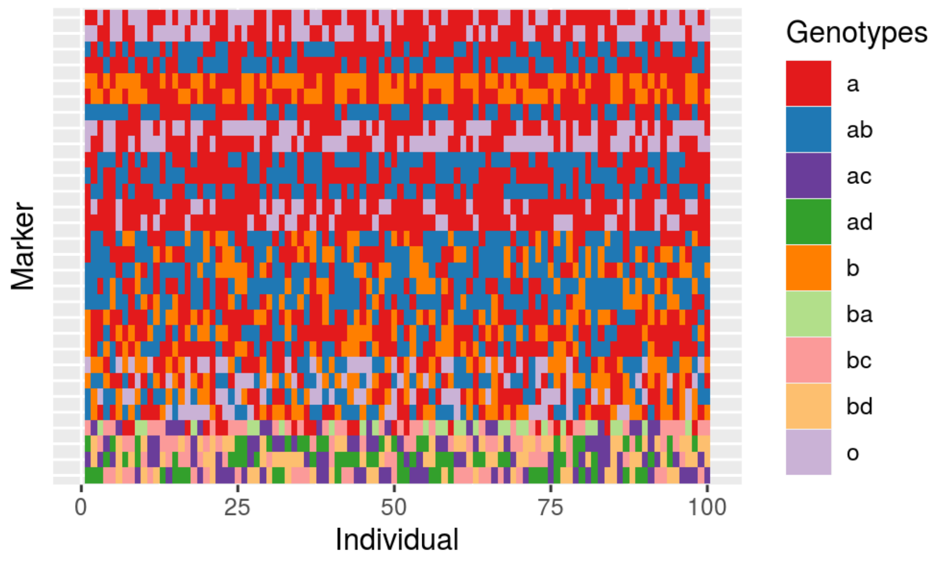
```r
onemap_example_out
```

you will see some information about the object:

```
#>   This is an object of class 'onemap'
#>     Type of cross:      outcross
#>     No. individuals:    100
#>     No. markers:        30
#>     CHROM information:  no
#>     POS information:    no
#>     Percent genotyped:  100
#>
#>     Segregation types:
#>                A.1 -->  3
#>                A.2 -->  1
#>                A.4 -->  4
#>               B1.5 -->  1
#>               B2.6 -->  2
#>                B3.7 -->  5
#>                 C.8 -->  2
#>               D1.10 -->  2
#>               D1.12 -->  1
#>               D1.13 -->  2
#>               D2.15 -->  1
#>               D2.16 -->  2
#>               D2.17 -->  2
#>               D2.18 -->  2
#>
#>     No. traits:          3
#>     Missing trait values:
#>   Pheno1: 0
#>   Pheno2: 3
#>   Pheno3: 0
```

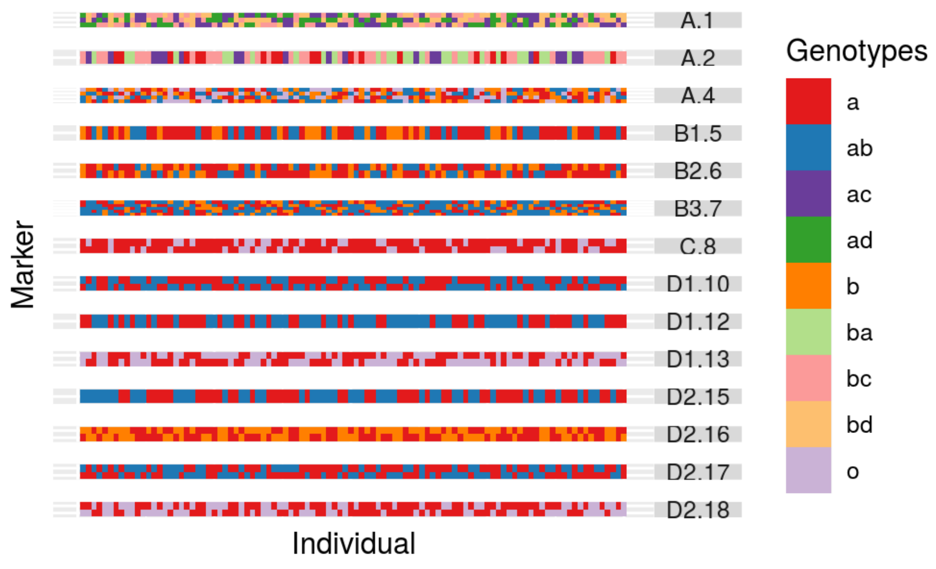
Also, you can use the `plot.onemap` function to see graphically markers genotypes:

```r
plot(onemap_example_out)
```



Changing the argument `all` to `FALSE`, the markers will be separated by their type. In this case, you can note that the graphic cell size will adapt to the number of markers of the same type. In other words, the higher is the number of markers with the same type, the lower is the cell for this type.

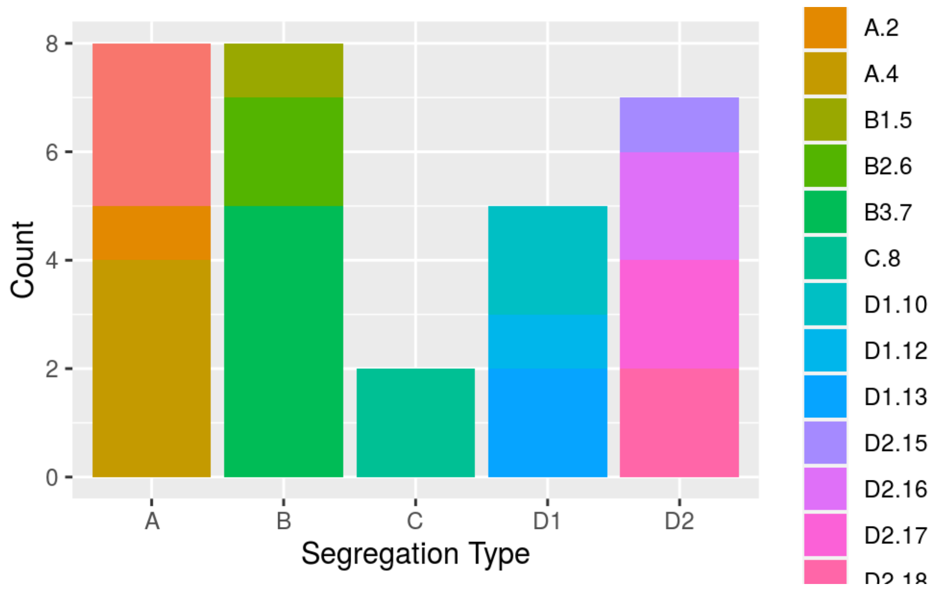
```r
plot(onemap_example_out, all = FALSE)
```



This function can take quite some time, depending on the number of markers involved. More information about this plot function can be found using `?plot.onemap`.

Also, you can see the number of markers by segregation pattern with the `plot_by_segreg_type` function:

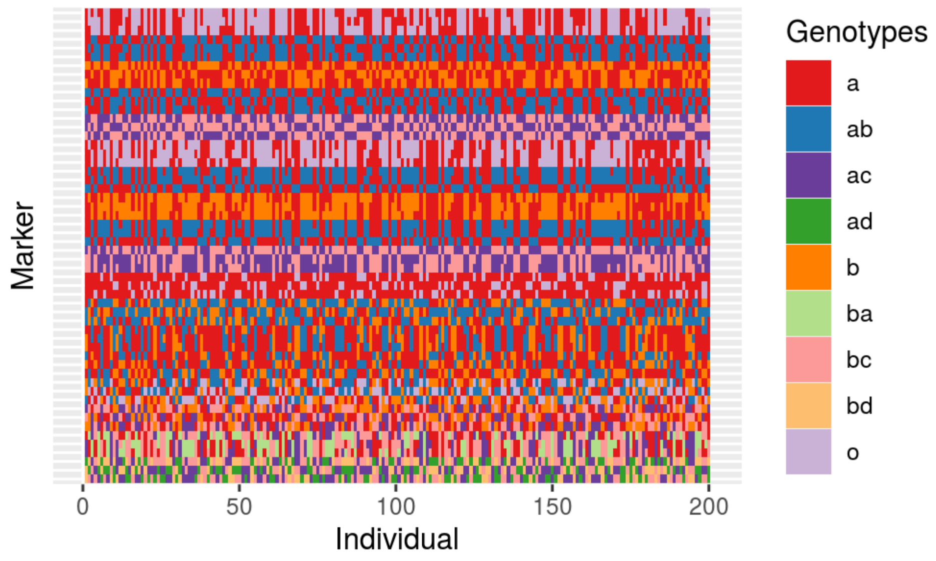
```
plot_by_segreg_type(onemap_example_out)
```



(new!)

Using the same functions to import the dataset, we can evaluate the simulated data by `run_pedsim` and `pedsim2raw`:

```
simulated_example <- read_onemap(inputfile = "sim_out.example.raw")
#>  Working...
#>
#>  --Read the following data:
#>  Type of cross:          outcross
#>  Number of individuals:   200
#>  Number of markers:       54
#>  Chromosome information:  no
#>  Position information:    no
#>  Number of traits:        0
plot(simulated_example)
```



Importing data from VCF file

You can import information from `VCF` to `OneMap` using `onemap_read_vcfR` function.

With the `onemap_read_vcfR` you can convert the object from `vcfR` package directly to `onemap`. The `onemap_read_vcfR` function keeps chromosome and position information for each marker in the onemap object generated.

We will use the example file `vcf_example_out.vcf` to show how it works, which contains markers from the same population of `onemap_example_out.raw`.

First, we convert the `VCF` file to `vcfR` object:

```
library(vcfR)
vcfR.object <- read.vcfR(system.file("extdata/vcf_example_out.vcf",
                                     package = "onemap"))
#> Scanning file to determine attributes.
#> File attributes:
#>   meta lines: 8
#>   header_line: 9
#>   variant count: 25
#>   column count: 103
#>
Meta line 8 read in.
#> All meta lines processed.
#> gt matrix initialized.
#> Character matrix gt created.
#>   Character matrix gt rows: 25
#>   Character matrix gt cols: 103
#>   skip: 0
#>   nrows: 25
#>   row_num: 0
#> 
Processed variant: 25
#> All variants processed
```

As described in the `vcfR` package vignette, memory use is an important consideration when using `vcfR`. Depending of your dataset, the object created can be huge and occupy a lot of memory.

After, you can use `onemap_read_vcfR` function to convert this object to `onemap` object. The parameters used are the `vcfR.object` (which was just created), the identification of each parent (here, you must define only one sample for each parent) and the cross type.

```
vcf_example_out <- onemap_read_vcfR(vcfR.object = vcfR.object,
                                    parent1 = "P1",
                                    parent2 = "P2",
                                    cross = "outcross")
```

Depending on your dataset, this function can take some time to run.

Note that the conversion filter out markers which are not informative for the informed cross type. For example, in outcrossing species, markers that have both parents homozygous (aa x bb) do not inform recombinations and are removed of the data set. Only markers types contained the table at Creating the data file are kept in the onemap object. Function `onemap_read_vcfR` print at the screen the reason why markers were filtered.

You can also have more missing data in the returned object compared with the VCF because the `onemap_read_vcfR` replace by missing data the genotypes that are not expected for that marker type. For example, for a marker type D1.10 (`ab` x `aa`), we only expect `aa` and `ab` genotypes, if there are `bb` genotypes they will be replaced by missing data. You can see the percentage of missing data at the resulted onemap object with:

```
vcf_example_out
#>   This is an object of class 'onemap'
#>     Type of cross:      outcross
#>     No. individuals:    92
#>     No. markers:        24
#>     CHROM information:  yes
#>     POS information:    yes
#>     Percent genotyped:  99
#> 
#>     Segregation types:
#>                B3.7 -->  18
#>               D1.10 -->  6
#> 
#>     No. traits:         0
```

After the conversion, we can save the `vcfR.object` as a `.RData` and remove it from the workspace, once it can occupy a lot of memory and turn the other process too slow.

```
save(vcfR.object, file = "vcfR.object.RData")
# rm(vcfR.object)
```

NOTE:From version 2.0.6 to 2.1.1005, `OneMap` had the `vcf2raw` function to convert `vcf` to `.raw`. Now, this function is defunct, but it can be replaced by a combination of `onemap_read_vcfR` and `write_onemap_raw` functions. See Exporting .raw file from onemap object session to further information about `write_onemap_raw`.

Filter onemap object by missing data

If your onemap object have too much missing genotypes you can face problems during the analysis. Check the percentage of missing genotypes in our data set printing the onemap object:

```
vcf_example_out
#>   This is an object of class 'onemap'
#>     Type of cross:      outcross
#>     No. individuals:    92
#>     No. markers:        24
#>     CHROM information:  yes
#>     POS information:    yes
#>     Percent genotyped:  99
#>
#>     Segregation types:
#>                B3.7 -->  18
#>               D1.10 -->  6
#>
#>     No. traits:         0
```

Our example has 1% of missing genotypes (99% are genotyped). If you want to filter markers according to its percentage of missing data, you can use the function `filter_missing`:

```
vcf_filtered <- filter_missing(vcf_example_out, threshold = 0.25)
#> Number of markers removed from the onemap object:  0
```

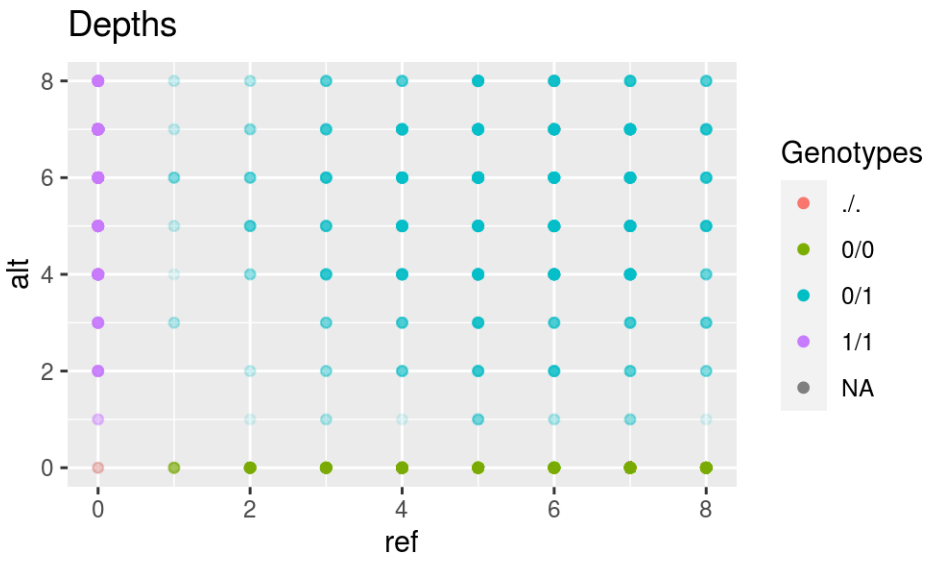
Any of our markers were filtered, because, in this example, we do not have many missing data.

Graphical view of genotypes and allele depths (new!)

Function `create_depth_profile` generates dispersion graphics with x and y-axis representing, respectively, the reference and alternative allele depths. The function is only available for biallelic markers in VCF files with allele counts information. Each dot represents a genotype for `mks` markers and `inds` individuals. If both arguments receive `NULL`, all markers and individuals are considered. Dots are colored according to the genotypes present in onemap object (`GTfrom = onemap`) or VCF file (`GTfrom = vcf`). A rds file is generated with the data in the graphic (`rds.file`). The `alpha` argument controls the transparency of the color of each dot. Control this parameter is a good idea when having a large number of markers and individuals. The `x_lim` and `y_lim` control the axis scale limits; by default, it uses the maximum value of the counts.

Here is an example for the `vcf_example_out` dataset.

```
# For outcrossing population
create_depths_profile(onemap.obj = vcf_example_out,
                      vcfR.object = vcfR.object,
                      parent1 = "P1",
                      parent2 = "P2",
                      vcf.par = "AD",
                      recovering = FALSE,
                      mks = NULL,
                      inds = NULL,
                      GTfrom = "vcf",
                      alpha = 0.1,
                      rds.file = "depths_out.rds")
#> Summary of reference counts:
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    1.00    4.00    6.00    5.43    7.00    8.00
#> Summary of alternative counts:
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   1.000   4.000   5.000   5.349   7.000   8.000
```

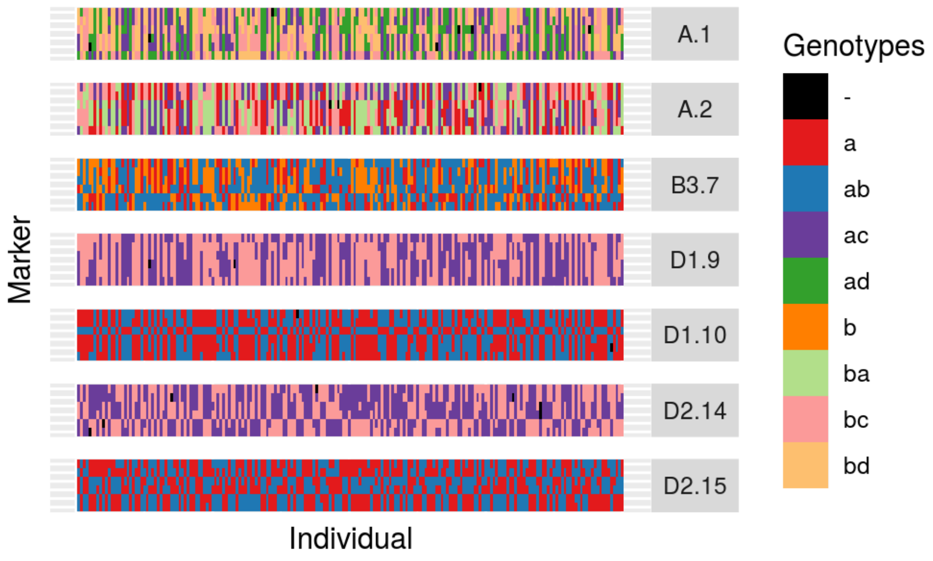


Because the genotypes are from VCF file, the legend points the VCF codification? `./.` represent missing data; `0/0` homozygotes for reference alleles; `0/1` heterozygotes; `1/1` homozygotes for alternative alleles. You can also have phased genotypes represented which have pipe `|` instead of bar `/`.

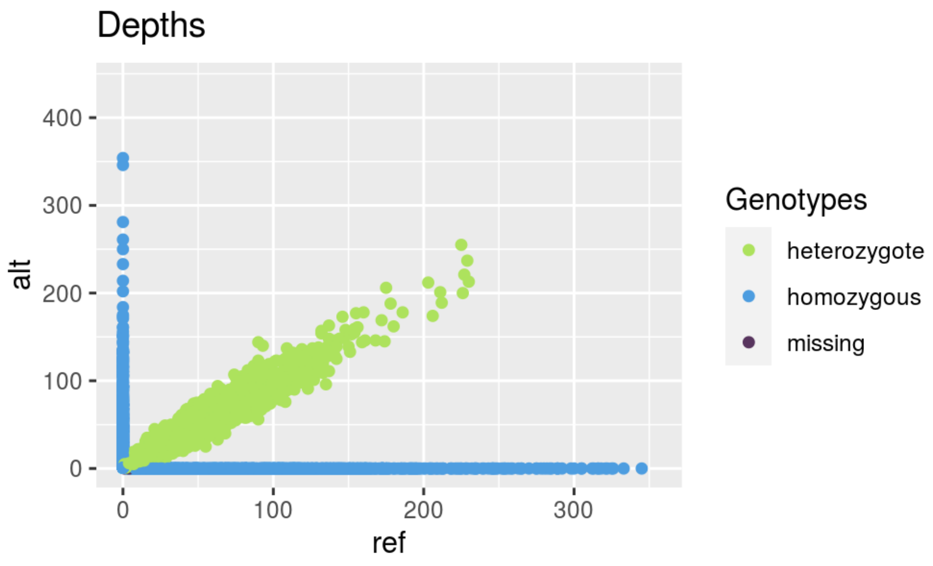
Using the same functions, we can import and check the simulated dataset generated by `run_pedsim` and `pedsim2vcf` above:

```
vcfR.object <- read.vcfR("simu_cod_out.vcf")
#> Scanning file to determine attributes.
#> File attributes:
#>   meta lines: 4
#>   header_line: 5
#>   variant count: 42
#>   column count: 211
#>
Meta line 4 read in.
#> All meta lines processed.
#> gt matrix initialized.
#> Character matrix gt created.
#>   Character matrix gt rows: 42
#>   Character matrix gt cols: 211
#>   skip: 0
#>   nrows: 42
#>   row_num: 0
#>
Processed variant: 42
#> All variants processed

simu_cod_out <- onemap_read_vcfR(vcfR.object = vcfR.object,
                                 parent1 = "P1",
                                 parent2 = "P2",
```

```r
                                           cross = "outcross",
                                           only_biallelic = FALSE)

plot(simu_cod_out, all=FALSE)
```



```r
create_depths_profile(onemap.obj = simu_cod_out,
                      vcfR.object = vcfR.object,
                      parent1 = "P1",
                      parent2 = "P2",GTfrom = "onemap")
#> Summary of reference counts:
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    1.00   28.00   52.00   66.64   89.00  441.00
#> Summary of alternative counts:
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    1.00   12.00   34.00   42.94   64.00  354.00
```



See that we had one warning message about the multiallelic markers in our dataset, they were not plotted.

Combining `OneMap` datasets

If you have more than one dataset of markers, all from the same mapping population, you can use the function `combine_onemap` to merge them into only one `onemap` object.

In our example, we have two datasets:

- first: `onemap_example_out` with 30 markers and 100 individuals
- second: `vcf_example_out` with 24 biallelic markers and 92 individuals.

The `combine_function` recognizes the correspondent individuals by the ID, thus, it is important to define the same IDs to respective individuals in both `raw` files. Compared with the first file, the second file does not have markers information for 8 individuals. The `combine_onemap` will complete this information with NA.

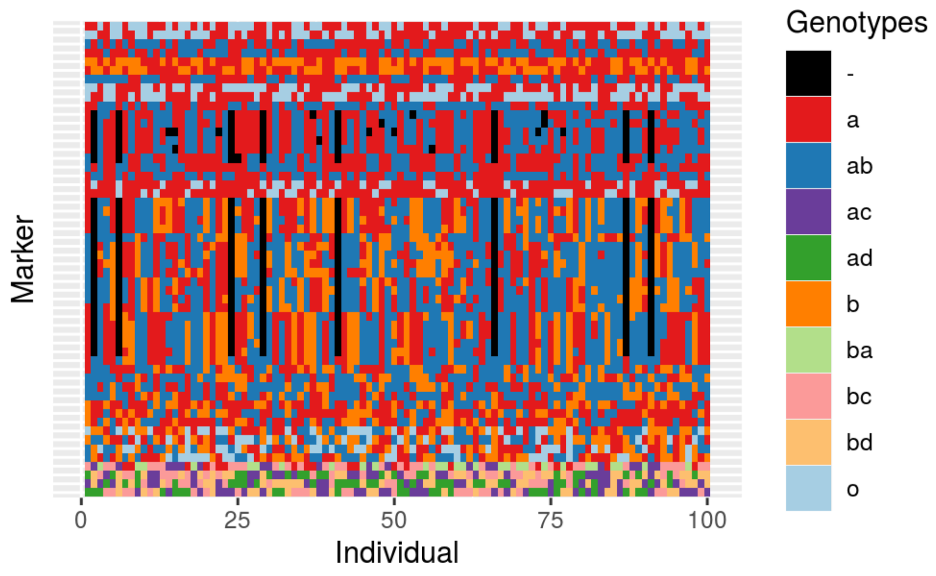
In our examples, we have only genotypic information, but the function can also merge the phenotypic information if it exists.

```
comb_example <- combine_onemap(onemap_example_out, vcf_example_out)
comb_example
#>   This is an object of class 'onemap'
#>     Type of cross:      outcross
#>     No. individuals:    100
#>     No. markers:        54
#>     CHROM information:  yes
#>     POS information:    yes
#>     Percent genotyped:  96
#>
#>     Segregation types:
#>                A.1 -->  3
#>                A.2 -->  1
#>                A.4 -->  4
#>               B1.5 -->  1
#>               B2.6 -->  2
#>               B3.7 -->  23
#>                C.8 -->  2
#>              D1.10 -->  8
#>              D1.12 -->  1
#>              D1.13 -->  2
#>              D2.15 -->  1
#>              D2.16 -->  2
#>              D2.17 -->  2
#>              D2.18 -->  2
#>
#>     No. traits:        3
#>     Missing trait values:
#>   Pheno1: 0
#>   Pheno2: 3
#>   Pheno3: 0
```

The function arguments are the names of the `onemap` objects you want to combine.

Plotting markers genotypes from the outputted `onemap` object, we can see that there are more missing data `-` (black vertical lines) for some individuals because they were missing in the second file.

```r
plot(comb_example)
```



Find redundant markers

It is possible that there are redundant markers in your dataset, especially when dealing with too many markers. Redundant markers have the same genotypic information that other markers because it didn't happen recombination events between each other. They will not increase information to the map, but will increase computational effort during the map building. Therefore, it is a good practice to remove them to build the map and, once the map is already built, they can be added again.

First, we use the function `find_bins` to group the markers into bins according to their genotypic information. In other words, markers with the same genotypic information will be in the same bin.

```r
bins <- find_bins(comb_example, exact = FALSE)
bins
#> This is an object of class 'onemap_bin'
#>     No. individuals:                          100
#>     No. markers in original dataset:          54
#>     No. of bins found:                        52
#>     Average of markers per bin:               1.038462
#>     Type of search performed:                 non exact
```

The first argument is the `onemap` object and the `exact` argument specify if only markers with same information will be at the same bin. Using `FALSE` at this second argument, missing data will not be considered, and the marker with the lowest amount of missing data will be the representative marker on the bin.

Our example dataset has only two redundant markers. We can create a new `onemap` object without them, using the `create_data_bins` function. This function keeps only the most representative marker of each bin from `bins` object.

```r
bins_example <- create_data_bins(comb_example, bins)
bins_example
#>   This is an object of class 'onemap'
#>     Type of cross:      outcross
#>     No. individuals:    100
#>     No. markers:        52
#>     CHROM information:  yes
#>     POS information:    yes
#>     Percent genotyped:  96
#>
#>     Segregation types:
#>                A.1 -->  3
#>                A.2 -->  1
#>                A.4 -->  4
#>               B1.5 -->  1
#>               B2.6 -->  2
#>               B3.7 -->  22
#>                C.8 -->  2
#>              D1.10 -->  7
#>              D1.12 -->  1
#>              D1.13 -->  2
#>              D2.15 -->  1
#>              D2.16 -->  2
#>              D2.17 -->  2
#>              D2.18 -->  2
#>
#>     No. traits:        3
#>     Missing trait values:
#>   Pheno1: 0
#>   Pheno2: 3
#>   Pheno3: 0
```

The arguments for `create_data_bins` function are the `onemap` object and the object created by `find_bins` function.

Exporting .raw file from onemap object

The functions `onemap_read_vcfR` generates new onemap objects without use a input `.raw` file. Also, the functions `combine_onemap` and `create_data_bins` manipulates the information of the original `.raw` file and creates a new dataset. In both cases, you do not have an input file `.raw` that contains the same information of the analyzed data. If you want to create a new input file with the dataset you are working after using these functions, you can use the function `write_onemap_raw`.

```
write_onemap_raw(bins_example, file.name = "new_dataset.raw")
```

The file `new_dataset.raw` will be generated in your working directory. In our example, it contains only non-redundant markers from `onemap_example_out` and `vcf_example_out` datasets.

Testing segregation pattern

For the map building process, it is also important to know which markers have deviations in the expected segregation pattern. It can be a good practice to remove them from the map building process, because they can adversely affect the map building, and, once the map is built, they can be inserted.

The function `test_segregation_of_a_marker` performs a chi-square test according to Mendelian segregation to check if a specific marker is following the expected segregation pattern.

```
test_segregation_of_a_marker(bins_example, 4)
#> $Hypothesis
#> [1] "1:1:1:1"
#>
#> $qui.quad
#> X-squared
#>      2.64
#>
#> $p.val
#> [1] 0.4505201
#>
#> $perc.genot
#> [1] 100
```

The arguments are the `onemap` object and the number of the marker you want to test.

You can also test all the markers in your `onemap` object using the `test_segregation` function. The results can be viewed by printing the output object of class `onemap_segreg_test`.

```
segreg_test <- test_segregation(bins_example)
print(segreg_test)
#>   Marker    H0 Chi-square   p-value % genot.
#> 1     M1 1:2:1       1.76 0.4147829      100
#> 2     M2   1:1       0.04 0.8414806      100
#>  [ reached 'max' / getOption("max.print") -- omitted 50 rows ]
```

The only argument of the function is a `onemap` object.

Once we have the `onemap_segreg_test`object, the function `select_segreg` can be used to show only the markers considered with/without segregation distortion. By default, it uses as a threshold for the test a global $\alpha = 0.05$, corrected for multiple tests with Bonferroni correction.

```
#to show the markers names with segregation distortion
select_segreg(segreg_test, distorted = TRUE)
#> [1] "M17" "M19" "M25"

#to show the markers names without segregation distortion
select_segreg(segreg_test, distorted = FALSE)
#>  [1] "M1"  "M2"  "M3"  "M4"  "M5"  "M6"  "M7"  "M8"  "M9"  "M10"
#>  [ reached getOption("max.print") -- omitted 39 entries ]
```

It is not recommended, but you can define a different threshold value changing the `threshold` argument of the function `select_segreg`.

For the next steps, it will be useful to know the numbers of each marker with segregation distortion, so then you can keep those out of your map building analysis. These numbers refer to the lines where markers are located on the data file.

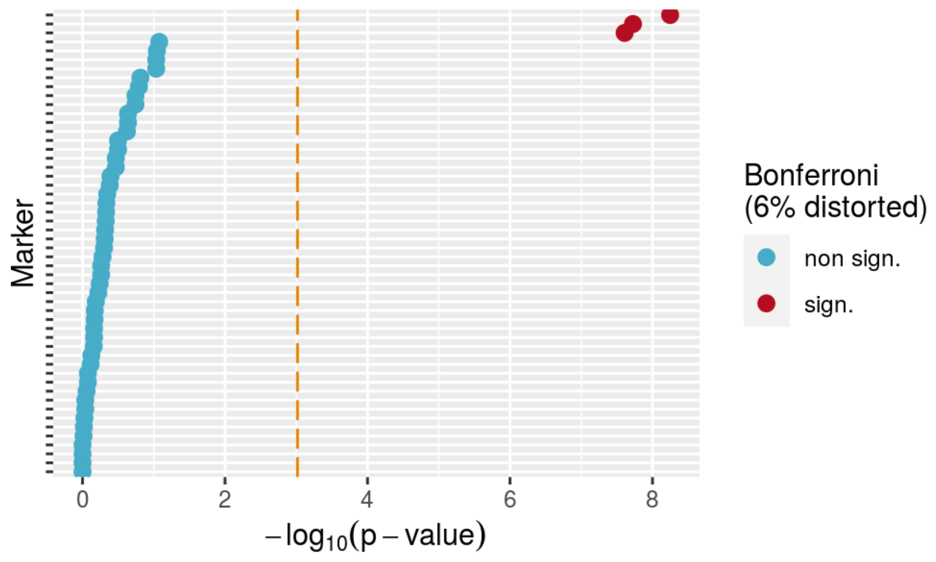
To access the corresponding number for of this markers you can change the `numbers` argument:

```
#to show the markers numbers with segregation distortion
dist <- select_segreg(segreg_test, distorted = TRUE, numbers = TRUE)
dist
#> [1] 17 19 25

#to show the markers numbers without segregation distortion
no_dist <- select_segreg(segreg_test, distorted = FALSE, numbers = TRUE)
no_dist
```

```
#>  [1]  1  2  3  4  5  6  7  8  9 10
#>  [ reached getOption("max.print") -- omitted 39 entries ]
```

You can also see the results graphically by:

```
plot(segreg_test)
```



Strategies for this tutorial example

Now, we start the map building analysis. In this example, we follow two different strategies:

- Using only recombinations information.
- Using the recombinations and also the reference genome information, once our example has `CHROM` and `POS` information for some of the markers.

First, we will apply the strategy using only recombinations information. In the second part of this tutorial, we show a way to use also reference genome information. At the and of our analysis, we will be able to compare these two strategies drawing the resulted genetic maps.

Using only recombinations information

Estimating two-point recombination fractions

The first step is estimating the recombination fraction between all pairs of markers, using two-point tests.

```
twopts <- rf_2pts(bins_example)
```

Although two-point tests were implemented in C language, which is usually much faster than R, this step can take quite some time, depending on the number of markers involved and their segregation type, because all combinations will be estimated and tested. Besides, the results use a a lot of memory and a rather powerful computer is needed.

When the two-point analysis is finished, an object of class `rf_2pts` is created. Typing

```
twopts
```

will show a message with the criteria used in the analysis and some other information:

```
#>   This is an object of class 'rf_2pts'
#>
#>   Criteria: LOD = 3 , Maximum recombination fraction = 0.5
#>
#>   This object is too complex to print
#>   Type 'print(object, c(mrk1=marker, mrk2=marker))' to see
#>     the analysis for two markers
#>     mrk1 and mrk2 can be the names or numbers of both markers
```

If you want to see the results for given markers, say `M1` and `M3`, the command is:

```
print(twopts, c("M1", "M3"))
#>   Results of the 2-point analysis for markers: M1 and M3
#>   Criteria: LOD =  3 , Maximum recombination fraction =  0.5
#>
#>           rf      LOD
#> CC 0.2954514 1.646878
#> CR 0.2954514 1.646878
#> RC 0.7045486 1.646878
#> RR 0.7045486 1.646878
```

Each line corresponds to a possible linkage phase. `CC` denotes the coupling phase in both parents, `CR` and `RC` denote coupling phase in parent 1 and 2, respectively, and repulsion in the other, and `RR` denotes the repulsion phase in both parents. Value `rf` is the maximum likelihood estimate of the recombination fraction, with its corresponding LOD Score.

Assigning markers to linkage groups

Once the recombination fractions and linkage phases for all pairs of markers have been estimated and tested, markers can be assigned to linkage groups. To do this, first use the function `make_seq` to create a sequence with the markers you want to assign.

The function `make_seq` is used to create sequences from objects of several kinds, as will be seen along this tutorial.

Here, the object is of class `rf_2pts` and the second argument specifies which markers one wants to use. If one wants to use only a subset of markers, say `M1` and `M2`, the option will be a vector with the corresponding numbers of the markers, as `c(1,2)`, you can also use a string `"all"` to specify that you want to analyze all markers. In our example, we will use the vector with the numbers of the markers with no segregation distortion.

```
mark_no_dist <- make_seq(twopts, c(no_dist))
```

Because the identification of the markers can be cumbersome, one should use the function `marker type` to see their numbers, names and types:

```
marker_type(mark_no_dist)
#>  [1] "  Marker 1 ( M1 ) is of type B3.7 \n"
#>  [2] "  Marker 2 ( M2 ) is of type D2.18 \n"
#>  [3] "  Marker 3 ( M3 ) is of type D1.13 \n"
#>  [4] "  Marker 4 ( M4 ) is of type A.4 \n"
#>  [5] "  Marker 5 ( M5 ) is of type D2.18 \n"
#>  [6] "  Marker 6 ( M6 ) is of type B3.7 \n"
#>  [7] "  Marker 7 ( M7 ) is of type D2.15 \n"
#>  [8] "  Marker 8 ( M8 ) is of type B3.7 \n"
#>  [9] "  Marker 9 ( M9 ) is of type D1.10 \n"
#> [10] "  Marker 10 ( M10 ) is of type D2.17 \n"
#>  [ reached getOption("max.print") -- omitted 39 entries ]
```

The grouping step is very simple and can be done by using the function `group`:

```
LGs <- group(mark_no_dist)
#>    Selecting markers:
#>    group    1
#>     ........................
#>    group    2
#>     ................
#>    group    3
#>     ......
```

For this function, optional arguments are `LOD` and `max.rf`, which define thresholds to be used when assigning markers to linkage groups. If none provided (default), it uses as default values of LOD Score `3` and maximum recombination fraction `0.50`.

Also, you can use the function `suggest_lod` to calculate a suggested LOD score considering that multiple tests are being performed.

```
LOD_sug <- suggest_lod(mark_no_dist)
LOD_sug
#> [1] 3.639312
```

And apply this suggested value to the two-point tests:

```
LGs <- group(mark_no_dist, LOD=LOD_sug)
#>    Selecting markers:
#>    group    1
#>     ........................
#>    group    2
#>     ................
#>    group    3
#>     ......
```

The previous command generates an object of class `group` and the command `print` for such object has two options. If you type:

```
LGs
```

you will get detailed information about the groups, that is, all linkage groups will be printed, displaying the names of markers in each one of them.

However, in case you just want to see some basic information (such as the number of groups, number of linked markers, etc), use

```
print(LGs, detailed = FALSE)
#>   This is an object of class 'group'
#>   It was generated from the object "mark_no_dist"
#>
#>   Criteria used to assign markers to groups:
#>     LOD = 3.639312 , Maximum recombination fraction = 0.5
#>
#>   No. markers:            49
#>   No. groups:             3
#>   No. linked markers:     49
#>   No. unlinked markers:   0
```

You can notice that all markers are linked to some linkage group. If the LOD Score threshold is changed to a higher value, some markers are kept unassigned:

```
LGs <- group(mark_no_dist, LOD = 6)
#>    Selecting markers:
#>    group    1
#>     .........................
#>    group    2
#>     ..........
#>    group    3
#>     .....
#>    group    4
#>     ....
```

Changing back to the previous criteria, now setting the maximum recombination fraction to 0.40:

```
LGs <- group(mark_no_dist, LOD = LOD_sug, max.rf = 0.4)
#>    Selecting markers:
#>    group    1
#>     .........................
#>    group    2
#>     ................
#>    group    3
#>     ......
```

Genetic mapping of linkage group 3

Once marker assignment to linkage groups is finished, the mapping step can take place. First of all, you must set the mapping function that should be used to display the genetic map throughout the analysis. You can choose between `Kosambi` or `Haldane` mapping functions. To use Haldane, type

```
set_map_fun(type = "haldane")
```

To use Kosambi

```
set_map_fun(type = "kosambi")
```

If you do not set one of these functions, the kosambi is used as default.

Now, you must define which linkage group will be mapped. In other words, a linkage group must be `extracted` from the object of class `group`, in order to be mapped. For simplicity, we will start here with the smallest one, which is linkage group 3. This can be easily done using the following code:

```
LG3 <- make_seq(LGs, 3)
```

The first argument (`LGs`) is an object of class `group` and the second is a number indicating which linkage group will be extracted, according to the results stored in object `LGs`. The object `LG3`, generated by function `make_seq`, is of class `sequence`, showing that this function can be used with several types of objects.

If you type

```
LG3
```

you will see which markers are comprised in the sequence, and also that no parameters have been estimated so far.

```
#>
#> Number of markers: 7
#> Markers in the sequence:
#> M7 M8 M13 M18 M22 SNP14 SNP16
#>
#> Parameters not estimated.
```

To order these markers, one can use a two-point based algorithm such as Seriation (Buetow and Chakravarti, 1987), Rapid Chain Delineation (Doerge, 1996), Recombination Counting and Ordering (Van Os et al., 2005) and Unidirectional Growth (Tan and Fu, 2006):

```
LG3_ser <- seriation(LG3)
LG3_rcd <- rcd(LG3)
LG3_rec <- record(LG3)
LG3_ug <- ug(LG3)
```

In this case, there are some differences between algorithms results (results not shown).

Alternatively, you can also use `mds_onemap` function to obtain a first draft for the order of the markers. The `mds_onemap` is a wrapper function that makes an interface between `OneMap` and `MDSMap` package. The ordering approach presented in `MDSMap` provides a faster and efficient way of ordering markers using multi-dimensional scaling. The method also provides diagnostics graphics and parameters to find outliers to help users to filter the dataset. You can find more informations in `MDSMap` vignette. Here we will show a simple example of how it can be used for ordering our example markers from an outcrossing population.

```
LG3_mds <- mds_onemap(LG3)
#> Stress: 0.225277102704001
#> Mean Nearest Neighbour Fit: 22.0492750538469
#> Markers omitted:
```

If you only specify the input sequence, mds_onemap will use the default parameters. It will generate MDSMap input file in `out.file` file and the `NULL.pdf` file with the diagnostic graphics in your work directory. You can use `out.file` in the MDSMap package to try other parameters too. The default method used is the principal curves, know more about using `?mds_onemap` and reading the MDSMap vignette.

Besides these algorithms use a two-point approach to order the markers, a multipoint approach is applied to estimate the genetic distances after the order is estimated. Thus, it can happen that some markers are not considered linked when evaluated by multipoint information, and the function will return an error like this:

```
ERROR: The linkage between markers 1 and 2 did not reach the OneMap
default criteria. They are probably segregating independently
```

You can automatically remove these markers setting argument `rm_unlinked = TRUE`. The marker will be removed, and the ordering algorithms will be restarted. Warning messages will inform which markers were removed. If you don't get warning messages, it means that any marker needed to be removed. This is our case in this example, but if you obtain an error or warning running your dataset, you already know what happened.

NOTE: (new!) If your sequence has many markers (more than 60), we suggest to speed up `mds`, `seriation`, `rcd`, `record` and `ug` using BatchMap parallelization approach. See section `Speed up analysis with parallelization` for more information.

To order by comparing all possible orders (exhaustive search), the function `compare` can be used:

```
LG3_comp <- compare(LG3)
```

WARNING: This algorithm can take some time to run, depending on marker types in the linkage group. If you are working in a personal computer, without high capacity, we recommend using a maximum of ten markers.

If you have more markers in your group, we suggest using the following explained approaches `order_seq`.

In the example, `LG3` contains only seven markers. Two of them are of type D1, and one is segregating in 3:1 fashion (type C). Thus, although the number of possible orders is relatively small (360), for each order, there are various possible combinations of linkage phases. Also, the convergence of the EM algorithm takes considerably more time, because markers of type C and D are not very informative.

The first argument to the `compare` function is an object of class `sequence` (the extracted group `LG3`), and the object generated by this function is of class `compare`.

To see the results of the previous step, type

```
LG3_comp
```

Remember that for outcrossing populations, one needs to estimate marker order and also linkage phases between markers for a given order. However, because two-point analysis provides information about linkage phases, this information is taken into consideration in the `compare` function, reducing the number of combinations to be evaluated. If a given linkage phase has LOD greater than 0.005 in the two-point analysis, we assume that this phase is very unlikely and so does not need to be evaluated in the multipoint procedure used by `compare`. We did extensive simulations, which showed that this is a good procedure.

By default, `OneMap` stores 50 orders, which may or may not be unique. The value of `LOD` refers to the overall LOD Score, considering all orders tested. `Nested LOD` refers to LOD Scores within a given order, that is, scores for different combinations of linkage phases for the same marker order.

For example, order 1 has the largest value of log-likelihood and, therefore, its LOD Score is zero for a given combination of linkage phases (CC, CC, RR, RR). For this same order and other linkage phases, LOD Score is -5.20. Analyzing the results for order 2, notice that its highest LOD Score is very close to zero, indicating that this order is also quite plausible. Notice also that `Nested LOD` will always contain at least one zero value, corresponding to the best combination of phases for markers in a given order. Due to the information provided by a two-point analysis, not all combinations are tested, and that is the reason why the number of Nested LOD values is different for each order.

Unless one has some biological information, it is a good idea to choose the order with the highest likelihood. The final map can then be obtained with the command.

```
LG3_final <- make_seq(LG3_comp, 1, 1)
```

The first argument is the object of class `compare`. The second argument indicates which order is chosen: 1 is for the order with the highest likelihood, 2 is for the second-best, and so on. The third argument indicates which combination of phases is chosen for a given order: 1 also means the combination with the highest likelihood among all combinations of phases (based on Nested LOD).

For simplicity, these values are defaults, so typing

```
LG3_final <- make_seq(LG3_comp)
```

have the same effect.

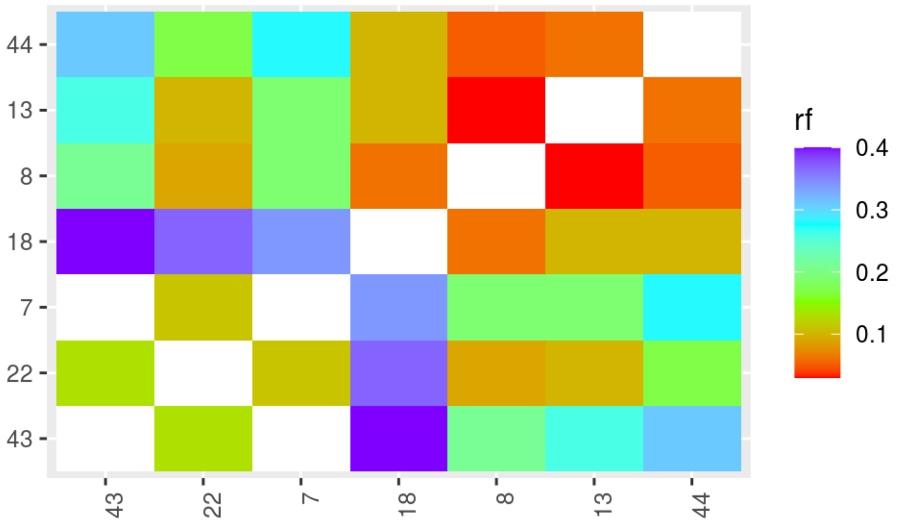
To see the final map, type

```
LG3_final
#> 
#> Printing map:
#> 
#> Markers             Position            Parent 1       Parent 2
#> 
#> 43 SNP14                0.00            a |    | b     a |    | a 
#> 22 M22                 13.51            a |    | b     a |    | a 
#>  7 M7                  24.56            a |    | a     a |    | b 
#> 18 M18                 65.59            a |    | b     c |    | d 
#>  8 M8                71.28          b |   | a        b |   | a
#> 13 M13               73.94          a |   | o        a |   | o
#> 44 SNP16             79.82          b |   | a        b |   | a
#>
#> 7 markers            log-likelihood: -398.6863
```

At the leftmost position, marker names are displayed. `Position` shows the cumulative distance using the Kosambi mapping function. Finally, `Parent 1` and `Parent 2` show the diplotypes of both parents, that is, the combination in which alleles are arranged in the chromosomes, given the estimated linkage phase. The notation is the same as that used by Wu et al. (2002a). Details about how ordering algorithms can be chosen and used are presented by Mollinari et al. (2009).

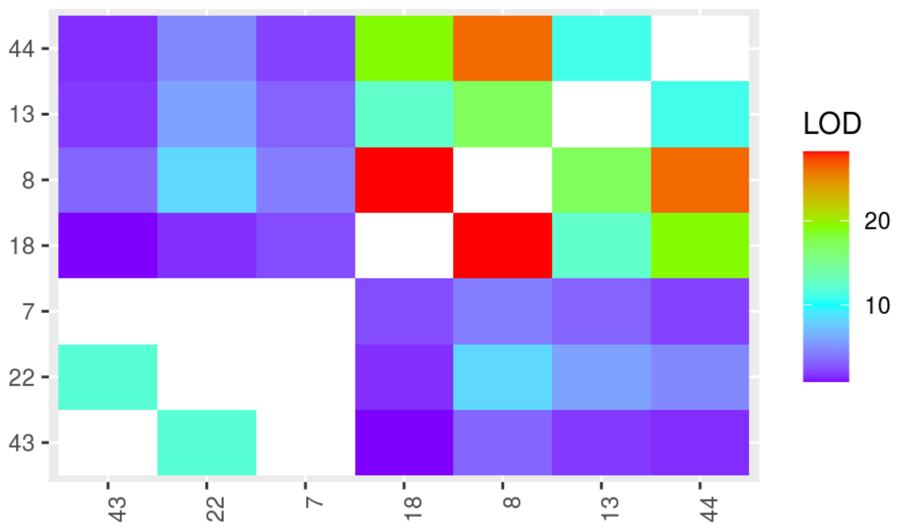
A careful examination of the results can be done using the function `rf_graph_table` to provide graphical view:

```
rf_graph_table(LG3_final)
```



With the default arguments, this function plots the recombination fractions between the markers pointed in the axes. You can change the number of colors from `rainbow` palette with the argument `n.colors`. Hot colors (more close to red) represent lower values of recombination fractions, as shown in the scale at the right side of the graphic. White cells indicate combinations of markers for which the recombination fractions cannot be estimated (D1 and D2). If you want to analyze the LOD values between the markers, use `graph.LOD = TRUE`.

```
rf_graph_table(LG3_final, graph.LOD = TRUE)
```



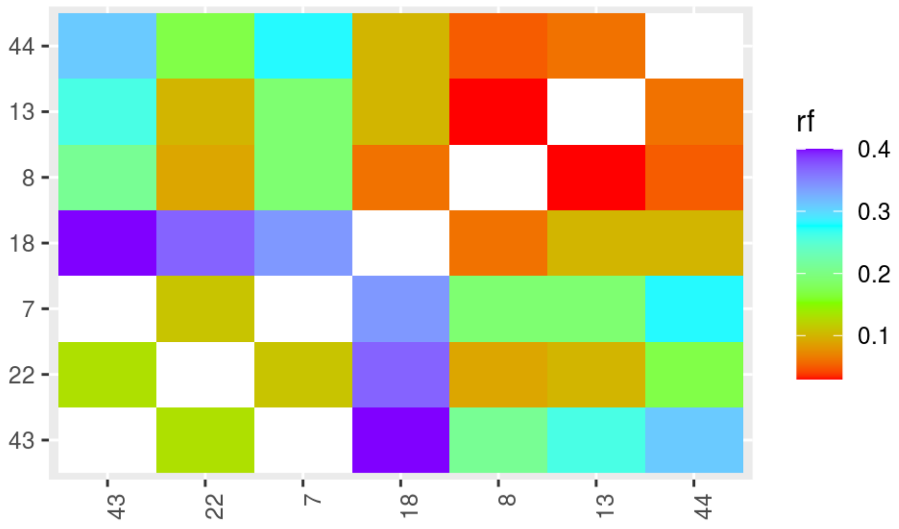
If you change the `inter` argument to `TRUE`, you should also specify an output HTML file name in `html.file`. This HTML contains an iterative plotly graphic. If you hover the mouse cursor over the cells it shows some extra information about cells, as a percentage of missing data, marker name and type. The output HTML file is generated in your work directory and opens automatically in your internet browser.

```
rf_graph_table(LG3_final, inter = TRUE, mrk.axis= "names", html.file = "LG3.html")
```

For example, passing on the cell corresponding to markers `8` and `13`, you can see their names (`M8` and `M13`), types (`B3.7` and `C.8`), recombination fraction (`rf = 0.03`) and LOD Scores for each possible linkage phase. This is quite useful in helping to interpret the results.

If you want to see corresponding marker numbers (not the names) in the axis, just change the argument `mrk.axis` to `numbers`. It can make the next steps easier.

```
rf_graph_table(LG3_final, inter = FALSE, mrk.axis = "numbers")
```

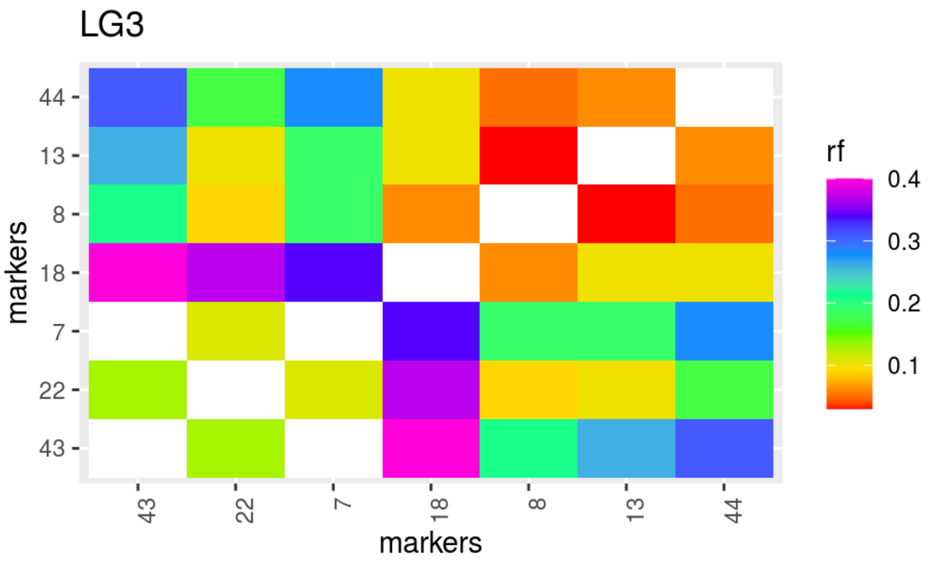


The `rf_graph_table` can also be used to check the order of markers based on the monotonicity of the matrix: as we get away from the secondary diagonal, the recombination fraction values should increase.

It is possible to see a gap between markers M7 and M18 (numbers 7 and 18). In some cases, gaps could indicate that the group must be divided at this position, but here SNP18 (number 43) also show linkage with M8, which points that probably it is only a gap. Adding more markers to these groups could fill this gap.

Changing other arguments of the function, you can add/remove labels of the axes ('lab.xy') and add a title to the graph ('main').

```
rf_graph_table(LG3_final, main = "LG3", inter = FALSE,
               n.colors = 7, lab.xy = c("markers", "markers"))
```



Genetic mapping of linkage group 2

Now, let us map the markers in linkage group number 2.

Again, extract that group from the object LGs:

```
LG2 <- make_seq(LGs, 2)
LG2
#>
#> Number of markers: 17
#> Markers in the sequence:
#> M4 M9 M16 M20 M21 M23 M24 M27 M29 SNP17 SNP18 SNP20 SNP21 SNP22 SNP23 SNP24
#> SNP25
#>
#> Parameters not estimated.
```

Note that there are more than 10 markers in this group, so it is infeasible to use the compare function with all of them because it will take a very long time to proceed.

First, use rcd to get a preliminary order estimate:

```
LG2_rcd <- rcd(LG2)
#>
#> order obtained using RCD algorithm:
#>
#>  27 47 16 20 4 21 23 45 46 9 48 51 50 24 49 52 29
#>
```

```
#> calculating multipoint map using tol =  1e-04 .
LG2_rcd
#>
#> Printing map:
#>
#> Markers           Position           Parent 1       Parent 2
#>
#> 27 M27              0.00           b |  | o       a |  | a
#> 47 SNP20           98.48           a |  | b       a |  | a
#> 16 M16            149.21           a |  | a       b |  | o
#> 20 M20            160.39           a |  | b       c |  | d
#>  4 M4             173.16           a |  | o       o |  | b
#> 21 M21            187.19           o |  | o       a |  | b
#> 23 M23            192.89           a |  | o       a |  | o
#> 45 SNP17          211.80           a |  | b       a |  | a
#> 46 SNP18          225.91           a |  | b       a |  | a
#>  9 M9             232.12           a |  | b       a |  | a
#> 48 SNP21          239.01           a |  | b       a |  | a
#> 51 SNP24          261.01           a |  | b       b |  | a
#> 50 SNP23          271.95           a |  | b       b |  | a
#> 24 M24            276.90           a |  | b       b |  | a
#> 49 SNP22          280.29           a |  | b       b |  | a
#> 52 SNP25          285.42           a |  | b       b |  | a
#> 29 M29            314.87           o |  | a       o |  | o
#>
#> 17 markers            log-likelihood: -902.103
```

Use the `marker_type` function to check the segregation types of all markers in this group:

```
marker_type(LG2)
#>  [1] "  Marker 4 ( M4 ) is of type A.4 \n"
#>  [2] "  Marker 9 ( M9 ) is of type D1.10 \n"
#>  [3] "  Marker 16 ( M16 ) is of type D2.17 \n"
#>  [4] "  Marker 20 ( M20 ) is of type A.1 \n"
#>  [5] "  Marker 21 ( M21 ) is of type D2.16 \n"
#>  [6] "  Marker 23 ( M23 ) is of type C.8 \n"
#>  [7] "  Marker 24 ( M24 ) is of type B3.7 \n"
#>  [8] "  Marker 27 ( M27 ) is of type D1.12 \n"
#>  [9] "  Marker 29 ( M29 ) is of type D1.13 \n"
#> [10] "  Marker 45 ( SNP17 ) is of type D1.10 \n"
#>  [ reached getOption("max.print") -- omitted 7 entries ]
```

Based on their segregation types and distribution on the preliminary map, markers `M4`, `M20`, `M24`, `SNP22`, `SNP23`, `SNP24` and `SNP25` are the most informative ones (type `A` is better, followed by type `B`). So, let us create a framework of ordered markers using `compare` for the most informative ones:

```
LG2_init <- make_seq(twopts, c(4, 20, 24, 49,50,51, 52))
```

Here there is a automatic way of obtain a new sequence only with markers selected by type: (new!)

```
LG2_init <- seq_by_type(sequence = LG2, mk_type = c("A", "B"))
marker_type(LG2_init)
#> [1] "  Marker 4 ( M4 ) is of type A.4 \n"
#> [2] "  Marker 20 ( M20 ) is of type A.1 \n"
#> [3] "  Marker 24 ( M24 ) is of type B3.7 \n"
#> [4] "  Marker 49 ( SNP22 ) is of type B3.7 \n"
#> [5] "  Marker 50 ( SNP23 ) is of type B3.7 \n"
#> [6] "  Marker 51 ( SNP24 ) is of type B3.7 \n"
#> [7] "  Marker 52 ( SNP25 ) is of type B3.7 \n"
# If I want to reduce even more the number of markers
# I can use drop_marker function
LG2_init <- drop_marker(input.seq = LG2_init, mrks = 52)
marker_type(LG2_init)
#> [1] "  Marker 4 ( M4 ) is of type A.4 \n"
#> [2] "  Marker 20 ( M20 ) is of type A.1 \n"
#> [3] "  Marker 24 ( M24 ) is of type B3.7 \n"
#> [4] "  Marker 49 ( SNP22 ) is of type B3.7 \n"
#> [5] "  Marker 50 ( SNP23 ) is of type B3.7 \n"
#> [6] "  Marker 51 ( SNP24 ) is of type B3.7 \n"
```

Now, the first argument to `make_seq` is an object of class `rf_2pts`, and the second argument is a vector of integers, specifying which molecular markers comprise the sequence.

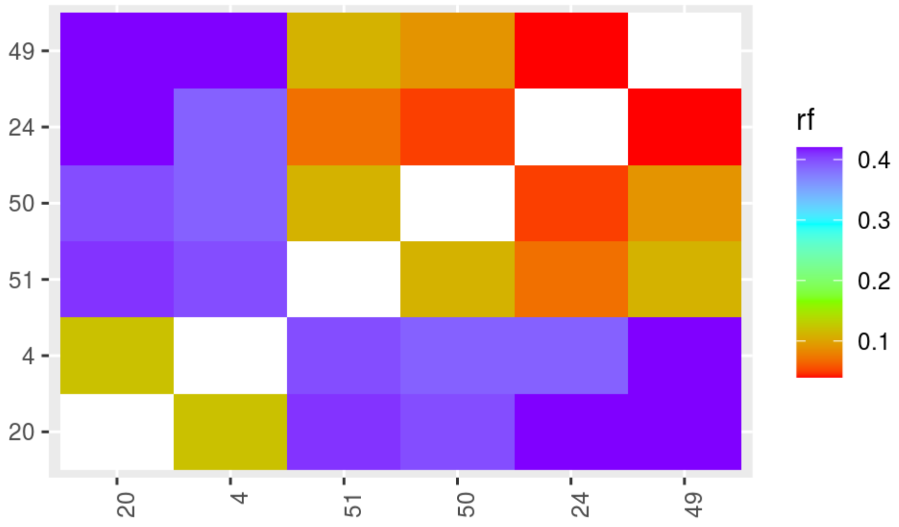
```
LG2_comp <- compare(LG2_init)
```

Select the best order:

```
LG2_frame <- make_seq(LG2_comp)
```

Also, we can obtain a useful diagnostic graphic using the function `rf_graph_table`.

```
rf_graph_table(LG2_frame, mrk.axis = "numbers")
```



The graphic shows that there are two groups of markers, once `M20` and `M4` are far from the other markers. These markers could be in other linkage groups, or they are distant in the same group. Adding more markers will give more information to solve this issue.

Next, let us try to map the remaining markers, one at a time. First, we will try to add the remaining most informative markers. Starting with `SNP25`:

```
LG2_extend <- try_seq(LG2_frame, 52)
```

```
LG2_extend
```

Based on the LOD Scores, marker SNP25 is probably better located after `SNP22` (number 49). Detailed results can be seen with

```
print(LG2_extend, 7)
#>
#> LOD is the overall LOD score (among all orders)
#>
#> NEST.LOD is the LOD score within the order
#>
#> Marker tested: 52
#> --------------
#> |     |      |
#> | 20  |      |
#> |     |  CR  |
#> |  4  |      |
#> |     |  CC  |
#> | 51  |      |
#> |     |  CC  |
#> | 50  |      |
#> |     |  CC  |
#> | 24  |      |
#> |     |  CC  |
#> | 49  |      |
#> |     |  CC  |
#> | 52  |      |
#> |     |      |
#> |------------|
#> | LOD |   0.0|
#> |------------|
#> |NEST.|      |
#> | LOD |   0.0|
#> --------------
```

The second argument indicates the position where to place the marker. Note that the first allele arrangement is the most likely one.

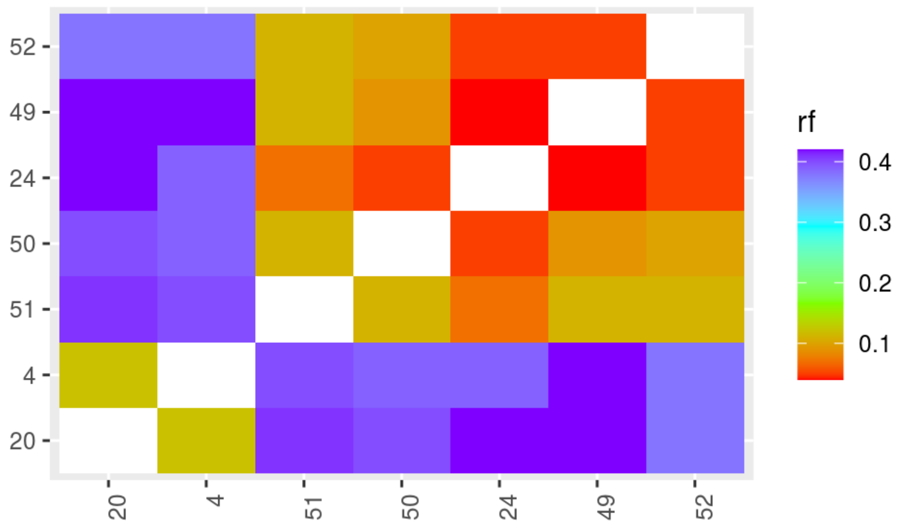
It should be pointed out that the framework created by the function `compare` (with `M20`, `M4`, `SNP24`, `SNP23`, `M24` and `SNP22`, or numbers `20`, `4`, `51`,`50`, `24` and `49`) could be in reverse order (`SNP22`, `M24`, `SNP23`, `SNP24`, `M4` and `M20`, or numbers `49`, `24`, `50`, `51`, `4`, `20`) and still represent the same map. Thus, the positioning of markers with the `try_seq` command can be different on your computer. For example, here marker `SNP25` (number `52`) was better placed at position 7; however, if you obtain a reversed order, marker `SNP25` would be better placed in position 1. In both cases, the best position for this marker is after `SNP22`.

We can better evaluate the order with `rf_graph_table`. It requires an object of `sequence` class with mapping information.

```
LG2_test <- make_seq(LG2_extend, 7, 1)
```

When using `make_seq` with an object of class `try`, the second argument is the position on the map (according to the scale on the right of the output) and the last argument indicates linkage phases (defaults to 1, higher nested LOD).

```
rf_graph_table(LG2_test, mrk.axis = "numbers")
```



We can see that `SNP25` (or marker 52) was positioned at the end of the sequence and the color pattern shows that it is strongly linked with its neighbors, indicating that it is well-positioned. We will maintain this marker at this position:

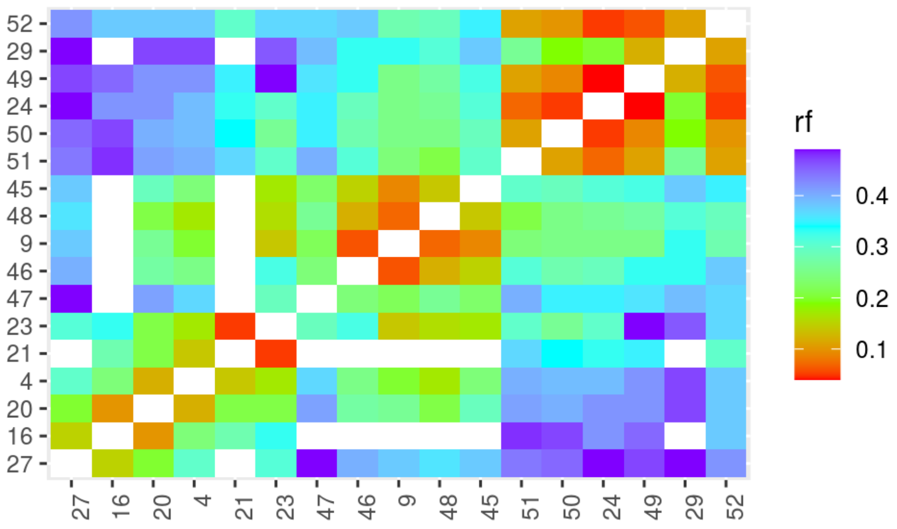
```
LG2_frame <- LG2_test
```

Adding other markers, one by one (output not shown):

```
LG2_extend <- try_seq(LG2_frame, 9)
LG2_frame <- make_seq(LG2_extend, 3)
LG2_extend <- try_seq(LG2_frame, 16)
LG2_frame <- make_seq(LG2_extend, 1)
LG2_extend <- try_seq(LG2_frame, 21)
LG2_frame <- make_seq(LG2_extend, 4)
LG2_extend <- try_seq(LG2_frame, 23)
LG2_frame <- make_seq(LG2_extend, 5)
LG2_extend <- try_seq(LG2_frame, 27)
LG2_frame <- make_seq(LG2_extend, 1)
LG2_extend <- try_seq(LG2_frame, 29)
LG2_frame <- make_seq(LG2_extend, 12)
LG2_extend <- try_seq(LG2_frame, 45)
LG2_frame <- make_seq(LG2_extend, 8)
LG2_extend <- try_seq(LG2_frame, 46)
LG2_frame <- make_seq(LG2_extend, 7)
LG2_extend <- try_seq(LG2_frame, 47)
LG2_frame <- make_seq(LG2_extend, 7)
LG2_extend <- try_seq(LG2_frame, 48)
LG2_final <- make_seq(LG2_extend, 10)
```

Checking graphically:

```
rf_graph_table(LG2_final)
```



The process of adding markers can be automated with the use of function `order_seq`.

```
LG2_ord <- order_seq(LG2, n.init = 5, THRES = 3)
```

This function automates what the `try_seq` function does, using some predefined rules. In the function, `n.init = 5` means that five markers (the most informative ones) will be used in the `compare` step; `THRES = 3` indicates that the `try_seq` step will only add markers to the sequence which can be mapped with LOD Score greater than 3.

NOTE: Although very useful, this function can be misleading, especially if there are not many fully informative markers, so use it carefully. Results can vary between multiple runs on the same markers, of course.

Check the final order:

```
LG2_ord
```

Note that markers `9`, `21`, `29`, `46`, `47`, `48`, `51` and `52` could not be safely mapped to a single position (`LOD Score > THRES` in absolute value). The output displays the `safe` order and the most likely positions for markers not mapped, where `***` indicates the most likely position and `*` corresponds to other plausible positions.

To get the safe order (*i.e.*, without markers `9`, `21`, `29`, `46`, `47`, `48`, `51` and `52`), use

```
LG2_safe <- make_seq(LG2_ord, "safe")
```

and to get the order with all markers, use

```
LG2_all <- make_seq(LG2_ord, "force")
LG2_all
#>
#> Printing map:
#>
#> Markers              Position             Parent 1       Parent 2
#>
#> 27 M27                   0.00             b |    | o       a |    | a
#> 16 M16                  11.76             a |    | a       b |    | o
#> 20 M20                  22.94             a |    | b       c |    | d
#>  4 M4                   35.71             a |    | o       o |    | b
#> 21 M21                49.40           o |   | o         a |   | b
#> 23 M23                55.18           a |   | o         a |   | o
#> 48 SNP21              67.73           a |   | b         a |   | a
#>  9 M9                 74.32           a |   | b         a |   | a
#> 46 SNP18              80.85           a |   | b         a |   | a
#> 45 SNP17              96.14           a |   | b         a |   | a
#> 47 SNP20             121.33           a |   | b         a |   | a
#> 29 M29               170.70           o |   | a         o |   | o
#> 50 SNP23             192.38           a |   | b         b |   | a
#> 24 M24               197.52           a |   | b         b |   | a
#> 49 SNP22             200.80           a |   | b         b |   | a
#> 52 SNP25             205.70           a |   | b         b |   | a
#> 51 SNP24             216.64           a |   | b         b |   | a
#>
#> 17 markers            log-likelihood: -871.9124
```

Notice that, for this linkage group, the `forced` map obtained with `order_seq` is different from that obtained with `compare` plus `try_seq`. It depends on which markers we choose to try to add first when doing manually.

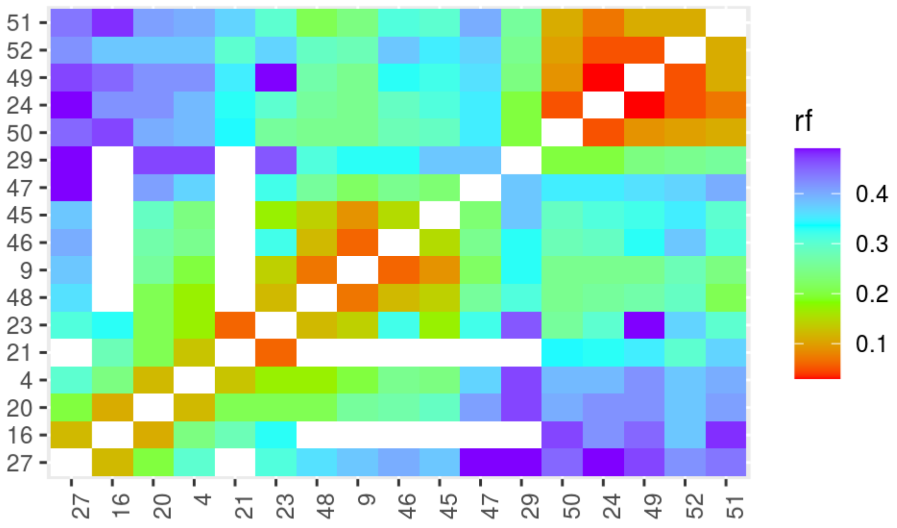
The `order_seq` function can also perform two rounds of the `try_seq` algorithms, first using `THRES` and then `THRES - 1` as a threshold. This generally results in safe orders with more markers mapped but may take longer to run. To do this, use the `touchdown` option:

```
LG2_ord <- order_seq(LG2, n.init = 5, THRES = 3, touchdown = TRUE)
```

```
LG2_ord
```

For this particular sequence, the `touchdown` step could map safely markers `46` and `52`, but this depends on the specific dataset.

```
rf_graph_table(LG2_all, mrk.axis = "numbers")
```



Finally, to check for alternative orders (because we did not use exhaustive search), use the `ripple_seq` function:

```
ripple_seq(LG2_all, ws = 4, LOD = LOD_sug)
```

We should do this to any of the orders we found, either using `try_seq` or `order_seq`. Here, we choose

`LG2_all` for didactic purposes only. The second argument, `ws = 4`, means that subsets (windows) of four markers will be permuted sequentially (`4!` orders for each window), to search for other plausible orders. The `LOD` argument means that only orders with LOD Score smaller than 3.68 will be printed.

The output shows sequences of four numbers, because `ws = 4`. They are followed by an `OK` if there is no alternative order with LOD Score smaller than `LOD = LOD_sug` in absolute value, or by a list of alternative orders. In the example, some sequences showed alternative orders with LOD smaller than `LOD = LOD_sug`. However, the best order was the original one (`LOD = 0.00`).

If there were an alternative order more likely than the original, one should check the difference between these orders (and linkage phases).

In some cases, even if there are no better alternative orders suggested by `ripple_seq`, the graphic showed color pattern different from the expected. Then, we can remove doubtful markers (for this groups markers `23` and `29`) and try to position them again. First, we use the function `drop_marker` to remove the selected marker of our sequence.

```
LG2_test_seq <- drop_marker(LG2_all, c(23,29))
```

The function will provide a sequence with the same order of the estimated map (`LG2_all`). After, we should estimate the map again using this predefined order (see section `Map estimation for an arbitrary order` for further information). For this we use the `map` function:

```
(LG2_test_map <- map(LG2_test_seq))
```

Warning: If you find an error message like:

```
Error in as_mapper(.f, ...) : argument ".f" is missing, with no default
```

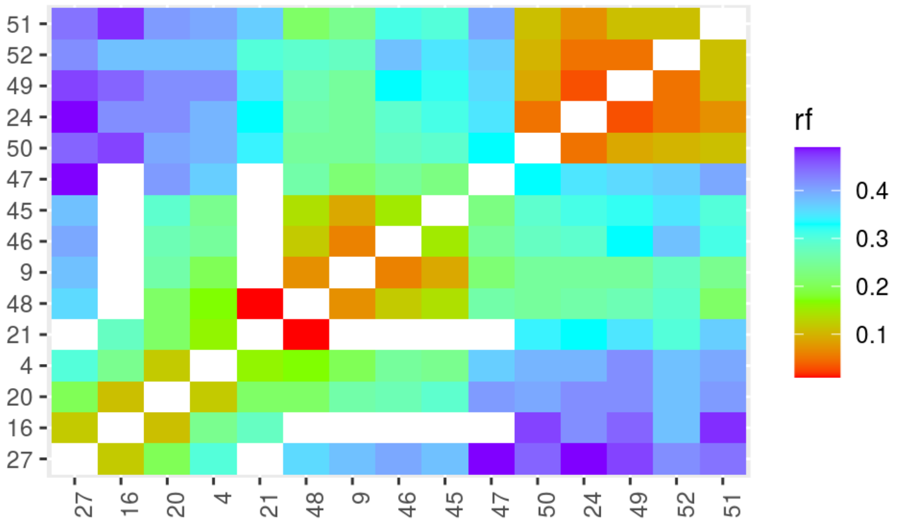
It's because the `map` function has a very common name, and you can have in your environment other function with the same name. In the case of the pointed error, R is using `map` function from `purrr` package instead of `OneMap`, to solve this, simply specify that you want `OneMap` function with `::` command from `stringr` package:

```
library(stringr)
(LG2_test_map <- onemap::map(LG2_test_seq))
#> 
#> Printing map:
#> 
#> Markers              Position            Parent 1       Parent 2
#> 
#> 27 M27                   0.00            b |   | o        a |   | a 
#> 16 M16                  11.75            a |   | a        o |   | b 
#> 20 M20                  22.93            a |   | b        d |   | c 
#>  4 M4                   35.70            a |   | o        b |   | o 
#> 21 M21                  52.67            o |   | o        b |   | a 
#> 48 SNP21                53.59            a |   | b        a |   | a 
#>  9 M9                   60.40            a |   | b        a |   | a 
#> 46 SNP18                66.83            a |   | b        a |   | a 
#> 45 SNP17                81.83            a |   | b        a |   | a 
#> 47 SNP20               106.46            a |   | b        a |   | a 
#> 50 SNP23               145.68            a |   | b        a |   | b 
#> 24 M24                 150.69            a |   | b        a |   | b 
#> 49 SNP22               153.97            a |   | b        a |   | b 
#> 52 SNP25               158.87            a |   | b        a |   | b 
#> 51 SNP24               169.81            a |   | b        a |   | b 
#> 
#> 15 markers            log-likelihood: -780.5746
```

NOTE: (new!) If your sequence has many markers (more than 60), we suggest to speed up `map` using BatchMap parallelization approach. See section `Speed up analysis with parallelization` for more information.

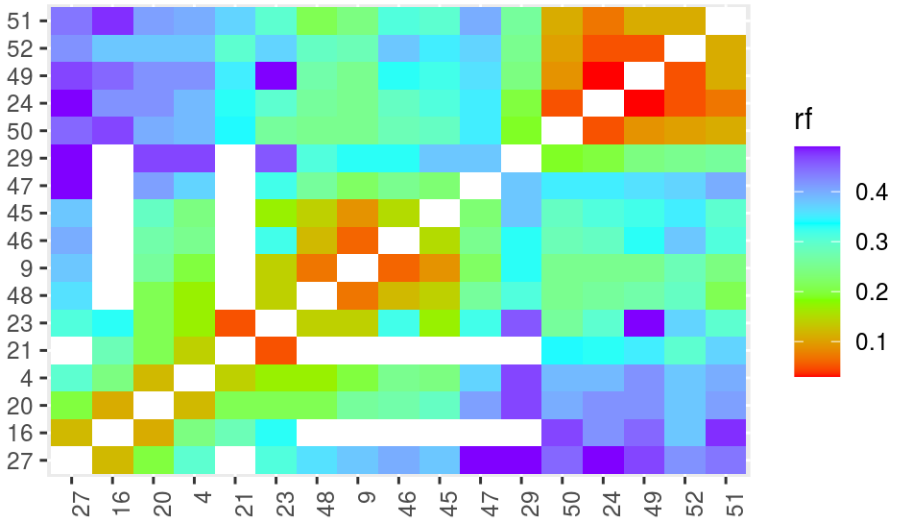
Now, we have the map without markers `23` and `51`.

```
rf_graph_table(LG2_test_map, mrk.axis = "numbers")
```



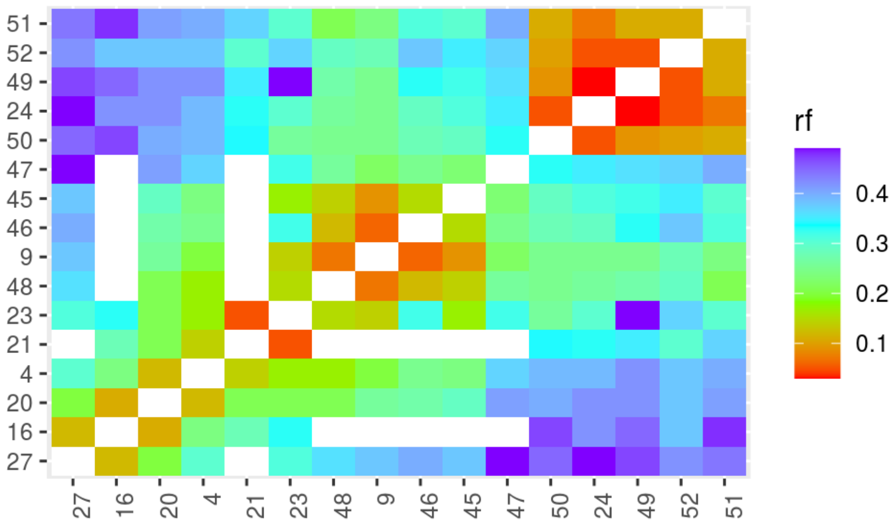
We use `try_seq` function to positioned them again:

```
LG2_test_seq <- try_seq(LG2_test_map, 23)
#> 23 --> M23   : .................
LG2_test_23 <- make_seq(LG2_test_seq, 6)
LG2_test_seq <- try_seq(LG2_test_23, 29)
#> 29 --> M29   : .................
LG2_test_23_29 <- make_seq(LG2_test_seq, 12)

rf_graph_table(LG2_test_23_29, mrk.axis = "numbers")
```



Marker 23 kept its previous position, but marker 29 was re-positioned, configuring now a gap between markers 47 and 50. We removed marker 29 of our map, because it color pattern is too different from expected. Then, our final map is:

```
LG2_final <- LG2_test_23
LG2_final
#>
#> Printing map:
#>
#> Markers           Position           Parent 1       Parent 2
#>
#> 27 M27                0.00           b |   | o       a |   | a
#> 16 M16               11.75           a |   | a       o |   | b
#> 20 M20               22.93           a |   | b       d |   | c
#>  4 M4                35.70           a |   | o       b |   | o
#> 21 M21               50.34           o |   | o       b |   | a
#> 23 M23               54.87           a |   | o       o |   | a
#> 48 SNP21             70.11           a |   | b       a |   | a
#>  9 M9                76.77           a |   | b       a |   | a
#> 46 SNP18             83.19           a |   | b       a |   | a
#> 45 SNP17             98.20           a |   | b       a |   | a
#> 47 SNP20            122.83           a |   | b       a |   | a
#> 50 SNP23            161.89           a |   | b       a |   | b
#> 24 M24              166.89           a |   | b       a |   | b
#> 49 SNP22            170.18           a |   | b       a |   | b
#> 52 SNP25            175.08           a |   | b       a |   | b
#> 51 SNP24            186.02           a |   | b       a |   | b
#>
#> 16 markers              log-likelihood: -811.7886

rf_graph_table(LG2_final, mrk.axis = "numbers")
```



Genetic mapping of linkage group 1

Finally, linkage group 1 (the largest one) will be analyzed. Extract markers:

```
LG1 <- make_seq(LGs, 1)
```

Construct the linkage map, by automatically using try algorithm:

```
LG1_ord <- order_seq(LG1, n.init = 6, touchdown = TRUE)
```

Notice that the second round of `try_seq` added markers `10`, `31`, `32`, `35`, `36` and `40`.

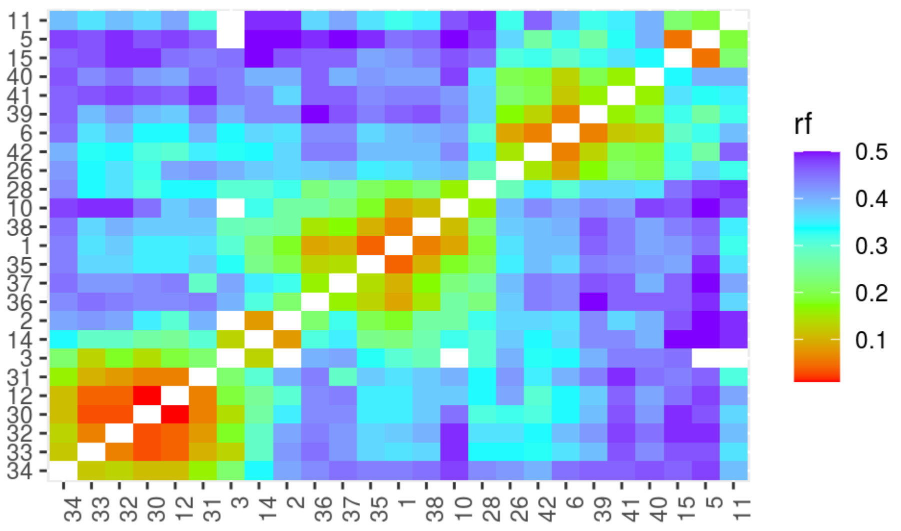
```
LG1_ord
```

Now, get the order with all markers:

```
(LG1_frame <- make_seq(LG1_ord, "force"))
#>
#> Printing map:
#>
#> Markers            Position           Parent 1       Parent 2
#>
#> 34 SNP5              0.00            a |   | b        a |   | b
#> 33 SNP3             12.27            a |   | b        a |   | b
#> 32 SNP2             18.45            a |   | b        a |   | b
#> 30 M30              21.21            a |   | b        a |   | b
#> 12 M12              22.22            b |   | a        c |   | a
#> 31 SNP1             27.77            a |   | b        a |   | b
#>  3 M3               51.05            o |   | a        o |   | o
#> 14 M14              64.18            a |   | o        b |   | o
#>  2 M2               72.05            o |   | o        o |   | a
#> 36 SNP7             95.86            b |   | a        b |   | a
#> 37 SNP8            112.04            b |   | a        b |   | a
#> 35 SNP6            126.75            b |   | a        b |   | a
#>  1 M1              131.15            b |   | a        b |   | a
#> 38 SNP9            137.28            b |   | a        b |   | a
#> 10 M10             148.79            a |   | a        o |   | b
#> 28 M28             165.35            a |   | o        b |   | o
#> 26 M26             197.72            a |   | b        d |   | c
#> 42 SNP13           212.84            b |   | a        b |   | a
#>  6 M6              219.19            b |   | a        b |   | a
#> 39 SNP10           225.36            b |   | a        b |   | a
#> 41 SNP12           241.82            b |   | a        b |   | a
#> 40 SNP11           258.95            b |   | a        b |   | a
#> 15 M15             301.19            o |   | a        o |   | b
#>  5 M5              306.20            o |   | o        o |   | a
#> 11 M11             326.21            o |   | o        b |   | a
#>
#> 25 markers            log-likelihood: -1554.628
```

Check the map graphically:

```
rf_graph_table(LG1_frame, mrk.axis = "numbers")
```



Check for alternative orders:

```r
ripple_seq(LG1_frame)
```

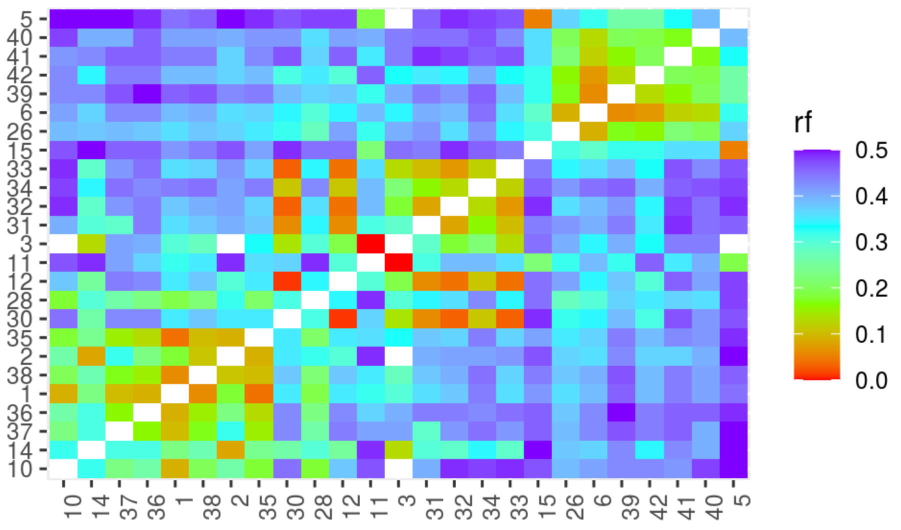
No better order was observed.

Let's check how it behaves with MDS approach:

```r
LG1_mds <- mds_onemap(LG1, rm_unlinked = TRUE)
#> Stress: 0.303935806009325
#> Mean Nearest Neighbour Fit: 30.2669402204895
#> Markers omitted:
```

NOTE: (new!) If your sequence has many markers (more than 60), we suggest to speed up `mds_onemap` using BatchMap parallelization approach. See section `Speed up analysis with parallelization` for more information.

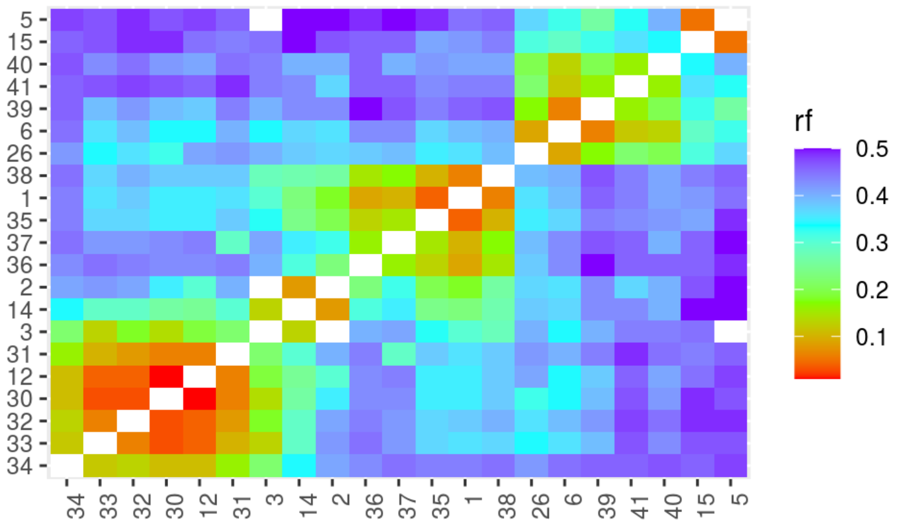
```r
rf_graph_table(LG1_mds)
```



Based on the drafts from `order_seq` or/and `mds_onemap`, we can remove some doubtful markers accordingly with the graphic, try to position them again and decide if and where we will maintain them.

```
LG1_test_seq <- drop_marker(LG1_frame, c(10,11,28,42))
LG1_test_map <- onemap::map(LG1_test_seq)
```

```
rf_graph_table(LG1_test_map, mrk.axis = "numbers")
```

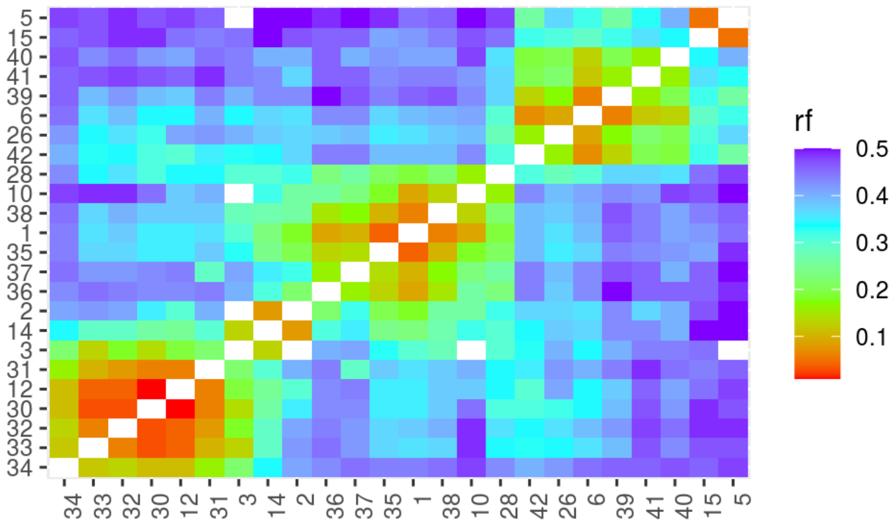


```
LG1_extend <- try_seq(LG1_test_map,10)
LG1_test_map <- make_seq(LG1_extend,15)
LG1_extend <- try_seq(LG1_test_map,11)
LG1_test <- make_seq(LG1_extend,23) # We choose to remove this marker
LG1_extend <- try_seq(LG1_test_map,28)
LG1_test_map <- make_seq(LG1_extend,16)
LG1_extend <- try_seq(LG1_test_map,42)
LG1_final <- make_seq(LG1_extend,17)
```

Print it:

```
LG1_final
#>
#> Printing map:
#>
#> Markers            Position           Parent 1       Parent 2
#>
#> 34 SNP5                0.00           a |    | b       a |    | b
#> 33 SNP3               12.27           a |    | b       a |    | b
#> 32 SNP2               18.45           a |    | b       a |    | b
#> 30 M30                21.21           a |    | b       a |    | b
#> 12 M12                22.22           b |    | a       c |    | a
#> 31 SNP1               27.77           a |    | b       a |    | b
#>  3 M3                 51.05           o |    | a       o |    | o
#> 14 M14                64.18           a |    | o       b |    | o
#>  2 M2                 72.10           o |    | o       o |    | a
#> 36 SNP7               95.93           b |    | a       b |    | a
#> 37 SNP8              112.12           b |    | a       b |    | a
#> 35 SNP6              126.83           b |    | a       b |    | a
#>  1 M1              131.23          b |   | a       b |   | a
#> 38 SNP9            137.34          b |   | a       b |   | a
#> 10 M10             150.29          a |   | a       o |   | b
#> 28 M28             166.58          a |   | o       b |   | o
#> 42 SNP13           203.65          b |   | a       b |   | a
#> 26 M26             219.73          a |   | b       d |   | c
#>  6 M6              228.71          b |   | a       b |   | a
#> 39 SNP10           234.88          b |   | a       b |   | a
#> 41 SNP12           251.25          b |   | a       b |   | a
#> 40 SNP11           268.27          b |   | a       b |   | a
#> 15 M15             309.06          o |   | a       o |   | b
#>  5 M5              314.07          o |   | o       o |   | a
#>
#> 24 markers              log-likelihood: -1518.748
```

```
rf_graph_table(LG1_final)
```



As an option, different algorithms to order markers could be applied:

```
LG1_ser <- seriation(LG1)
LG1_rcd <- rcd(LG1)
LG1_rec <- record(LG1)
LG1_ug  <- ug(LG1)
```

There are some differences between the results. Seriation did not provide good results in this case. See Mollinari et al. (2009) for an evaluation of these methods.

NOTE: (new!) If your sequence has many markers (more than 60), we suggest to speed up `seriation`, `rcd`, `record` and `ug` using BatchMap parallelization approach. See section `Speed up analysis with parallelization` for more information.

Using the recombinations and the reference genome information

In our example, we have reference genome chromosome and position information for some of the markers; here, we will exemplify one method of using this information to help build the genetic map.

With the CHROM information in the input file, you can identify markers belonging to some chromosome using the function make_seq with the rf_2pts object. For example, assign the string "1" for the second argument to get chromosome 1 makers. The output sequence will be automatically ordered by POS information.

```
CHR1 <- make_seq(twopts, "1")
CHR1
#>
#> Number of markers: 12
#> Markers in the sequence:
#> SNP1 SNP2 SNP3 SNP5 SNP6 SNP7 SNP8 SNP9 SNP10 SNP11 SNP12 SNP13
#>
#> Parameters not estimated.
CHR2 <- make_seq(twopts, "2")
CHR3 <- make_seq(twopts, "3")
```

Here we use string "1" because it is our chromosome ID, you can have a different string as ID, check this with:

```
unique(bins_example$CHROM)
#> [1] NA  "1" "2" "3"
```

We can see that we have markers without chromosome information (NA) and markers with chromosome ID "1", "2" and "3".

Adding markers with no reference genome information

According to CHROM informations we have three defined linkage groups, now we can try to group the markers without chromosome informations to them using recombination informations. For this, we can use the function group_seq:

```
CHR_mks <- group_seq(input.2pts = twopts, seqs = "CHROM",
                     unlink.mks = mark_no_dist,
                     repeated = FALSE)
#>    Selecting markers:
#>    group    1
#>     ........................
#>    group    2
#>     ........
#>    group    3
#>     ....
#>    Selecting markers:
#>    group    1
#>     ......
#>    group    2
#>     ............
#>    group    3
#>     ........
#>    Selecting markers:
#>    group    1
#>     ................
#>    group    2
#>     ............
#>    group    3
#>     ....
```

The function works as the function group, but considering pre-existing sequences. Setting seqs argument with the string "CHROM", it will consider the pre-existing sequences according to CHROM information. You can also indicate other pre-existing sequences if they make sense for your study. For that, you should inform a list of objects of class `sequences`, as the example:

```
CHR_mks <- group_seq(input.2pts = twopts,
                     seqs = list(CHR1=CHR1, CHR2=CHR2, CHR3=CHR3),
                     unlink.mks = mark_no_dist, repeated = FALSE)
```

In this case, the command had the same effect of the previous because we indicate chromosome sequences, but other sequences can be used.

The `unlink.mks` argument receives a object of class `sequence`; this defines which markers will be tested to group with the sequences in `seqs`. In our example, we will indicate only the markers with no segregation distortion, using the sequence `mark_no_dist`. It is also possible to use the string `"all"` to test all the remaining markers at the `rf_2pts` object.

In some cases, the same marker can group to more than one sequence; those markers will be considered `repeated`. We can choose if we want to remove or not (`FALSE/TRUE`) them of the output sequences, with the argument `repeated`. Anyway, their numbers will be informed at the list `repeated` in the output object. In the example case, there are no repeated markers. However, if they exist, it could indicate that their groups actually constitute the same group. Also, genotyping errors can generate repeated markers. Anyway, they deserve better investigations.

We can access detailed information about the results, just printing:

```
CHR_mks
```

Also, we can access the numbers of repeated markers with:

```
CHR_mks$repeated
#> [1] NA
```

We have no repeated markers.

The same way, we can access the output sequences:

```
CHR_mks$sequences$CHR1
#>
#> Number of markers: 25
#> Markers in the sequence:
#> SNP1 SNP2 SNP3 SNP5 SNP6 SNP7 SNP8 SNP9 SNP10 SNP11 SNP12 SNP13 M1 M2 M3 M5 M6
#> M10 M11 M12 M14 M15 M26 M28 M30
#>
#> Parameters not estimated.
# or
CHR_mks$sequences[[1]]
#>
#> Number of markers: 25
#> Markers in the sequence:
#> SNP1 SNP2 SNP3 SNP5 SNP6 SNP7 SNP8 SNP9 SNP10 SNP11 SNP12 SNP13 M1 M2 M3 M5 M6
#> M10 M11 M12 M14 M15 M26 M28 M30
#>
#> Parameters not estimated.
```

For this function, optional arguments are `LOD` and `max.rf`, which define thresholds to be used when assigning markers to linkage groups. If none provided (default), criteria previously defined for the object `rf_2pts` are used.

Now we can order the markers in each group as we made before in (Genetic mapping of linkage group 1,2 and 3). As shown, we can choose different approaches to order the markers.

To order those groups, first, we will use `order_seq` function to access a preliminary order, and after, we will edit some markers position or remove some of them according with their color pattern in `rf_graph_table` graphic, and other parameters as likelihood and map size.

Genetic mapping of linkage group 1 (with reference genome informations)

```
CHR1_frame <- mds_onemap(CHR_mks$sequences$CHR1)
# or
CHR1_ord <- order_seq(CHR_mks$sequences$CHR1)
CHR1_frame <- make_seq(CHR1_ord, "force")
```

```
rf_graph_table(CHR1_frame) # graphic not shown
```

NOTE: (new!) If your sequence has many markers (more than 60), we suggest to speed up `mds_onemap` using BatchMap parallelization approach. See section `Speed up analysis with parallelization` for more information.

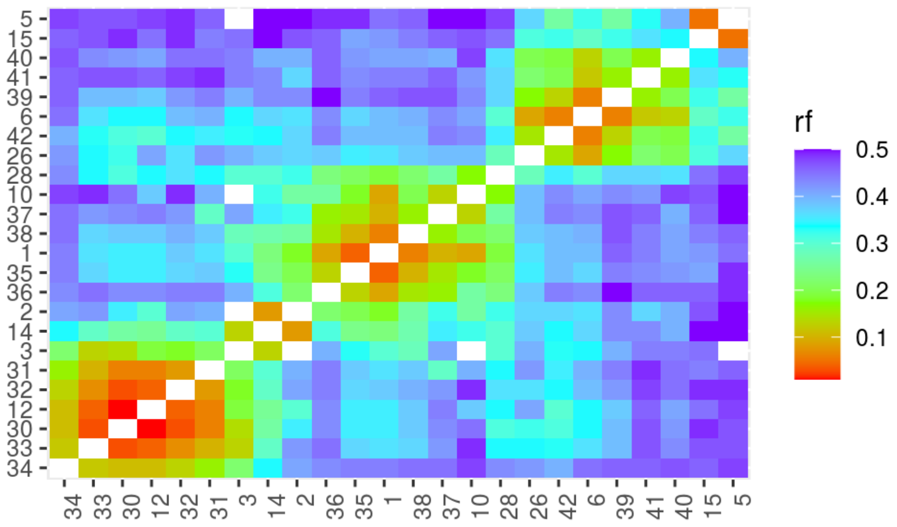
The group is similar to that built before with only recombinations information. We will better explore differences in the later step. Only marker `11` does not follow the expected color pattern; then, we will try to reposition it.

```
CHR1_test_seq <- drop_marker(CHR1_frame, 11)
CHR1_test_map <- onemap::map(CHR1_test_seq)
CHR1_add11_seq <- try_seq(CHR1_test_map, 11)
#> 11 --> M11    : .........................
CHR1_add11 <- make_seq(CHR1_add11_seq, 25)
# marker 11 was placed at the same position as before
```

Based in those results, we decide not to include marker 11 in our map.

```
CHR1_test_map
#>
#> Printing map:
#>
#> Markers           Position           Parent 1       Parent 2
#>
#> 34 SNP5               0.00           a |  | b        a |  | b
#> 33 SNP3              12.27           a |  | b        a |  | b
#> 30 M30               15.58           a |  | b        a |  | b
#> 12 M12               16.59           b |  | a        c |  | a
#> 32 SNP2              20.15           a |  | b        a |  | b
#> 31 SNP1              28.31           a |  | b        a |  | b
#>  3 M3                51.17           o |  | a        o |  | o
#> 14 M14               64.54           a |  | o        b |  | o
#>  2 M2                72.44           o |  | o        o |  | a
#> 36 SNP7              96.32           b |  | a        b |  | a
#> 35 SNP6             109.13           b |  | a        b |  | a
#>  1 M1               113.53           b |  | a        b |  | a
#> 38 SNP9             119.59           b |  | a        b |  | a
#> 37 SNP8             136.22           b |  | a        b |  | a
#> 10 M10              149.87           a |  | a        o |  | b
#> 28 M28              167.09           a |  | o        b |  | o
#> 26 M26              199.46           a |  | b        d |  | c
#> 42 SNP13            214.58           b |  | a        b |  | a
#>  6 M6               220.93           b |  | a        b |  | a
#> 39 SNP10              227.10          b |   | a        b |   | a
#> 41 SNP12              243.56          b |   | a        b |   | a
#> 40 SNP11              260.69          b |   | a        b |   | a
#> 15 M15                302.93          o |   | a        o |   | b
#>  5 M5                 307.95          o |   | o        o |   | a
#>
#> 24 markers              log-likelihood: -1507.316
```

```
rf_graph_table(CHR1_test_map)
```



```
CHR1_final <- CHR1_test_map
```

Checking for better orders:

```
ripple_seq(CHR1_final)
```

Genetic mapping of linkage group 2 (with reference genome informations)

```
CHR2_frame <- mds_onemap(CHR_mks$sequences$CHR2)
# or
CHR2_ord <- order_seq(CHR_mks$sequences$CHR2)
CHR2_frame <- make_seq(CHR2_ord, "force")
```

```
rf_graph_table(CHR2_frame) # graphic not shown
```

As did before, we will not change markers positions of this group.

```
CHR2_final <- CHR2_frame
```

Genetic mapping of linkage group 2 (with reference genome information)

```
CHR2_frame <- mds_onemap(CHR_mks$sequences$CHR2)
# or
```

```
CHR2_ord <- order_seq(CHR_mks$sequences$CHR2)
CHR2_frame <- make_seq(CHR2_ord, "force")
```

```
rf_graph_table(CHR2_frame) # graphic not shown
```

As did before, we will not change the markers positions of this group.

```
CHR2_final <- CHR2_frame
```

Genetic mapping of linkage group 3 (with reference genome information)

```
CHR3_frame <- mds_onemap(CHR_mks$sequences$CHR3)
# or
CHR3_ord <- order_seq(CHR_mks$sequences$CHR3)
CHR3_frame <- make_seq(CHR3_ord, "force")
```

```
rf_graph_table(CHR3_frame, mrk.axis = "numbers") # graphic not shown
```

Here, marker 29 have color pattern too different of the expected, removing it could be influential in other markers ordering. Then we will remove them and search for a new order.

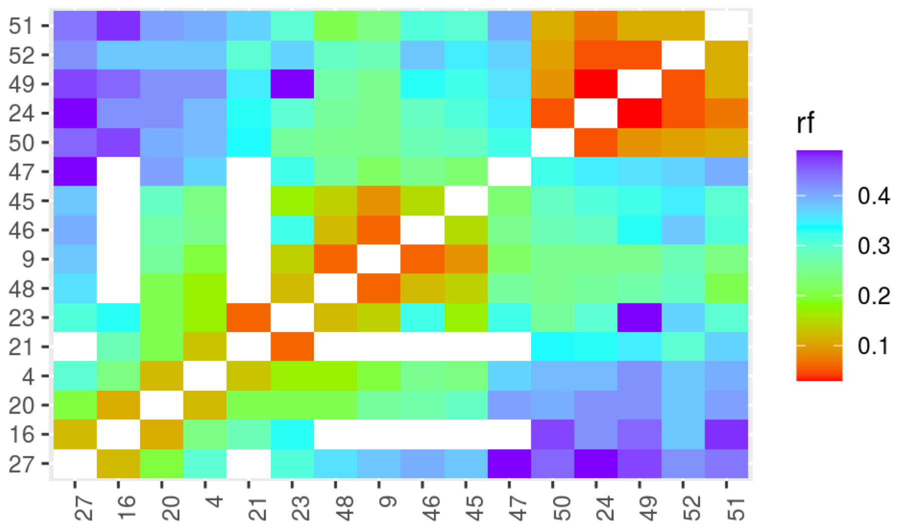
```
CHR3_test_seq <- drop_marker(CHR3_frame, c(29))
CHR3_test_ord <- order_seq(CHR3_test_seq)
CHR3_test_map <- make_seq(CHR3_test_ord, "force")
```

```
rf_graph_table(CHR3_test_map, mrk.axis = "numbers") #graphic not shown
```

Trying to add marker 29 again.

```
CHR3_add29_seq <- try_seq(CHR3_test_map, 29)
CHR3_add29 <- make_seq(CHR3_add29_seq, 12)
# Marker 29 increase the map size disproportionately, it was removed from the map
```

```
CHR3_final <- CHR3_test_map
rf_graph_table(CHR3_final, inter = FALSE)
```



Checking for better orders:

```r
ripple_seq(CHR3_final)
```

Drawing the genetic maps

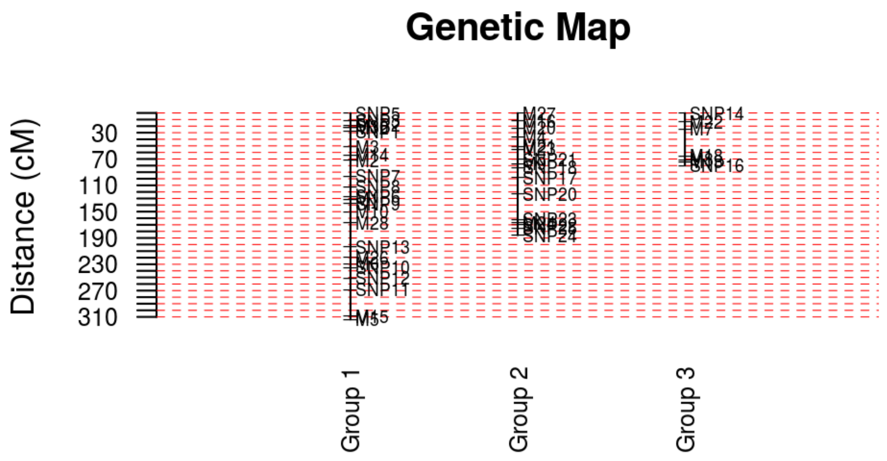
Once all linkage groups were obtained using both strategies, we can draw a map for each approach using the function `draw_map`. Since version 2.1.1007, `OneMap` has a new version of `draw_map`, called `draw_map2`. The new function draws elegant linkage groups and presents new arguments to personalize your draw.

If you prefer the old function, we also keep it. Follow examples on how to use both of them.

Draw_map

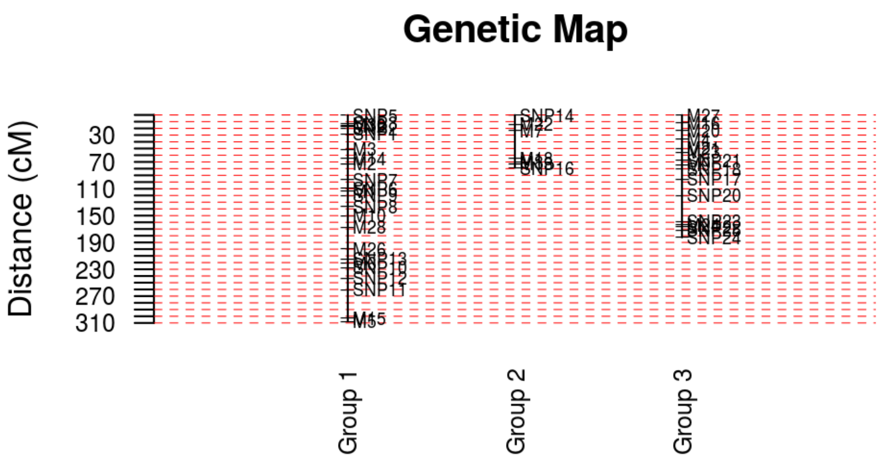
Drawing the map, which was built with only recombinations information.

```r
map1 <- list(LG1_final, LG2_final, LG3_final)
draw_map(map1, names = TRUE, grid = TRUE, cex.mrk = 0.7)
```



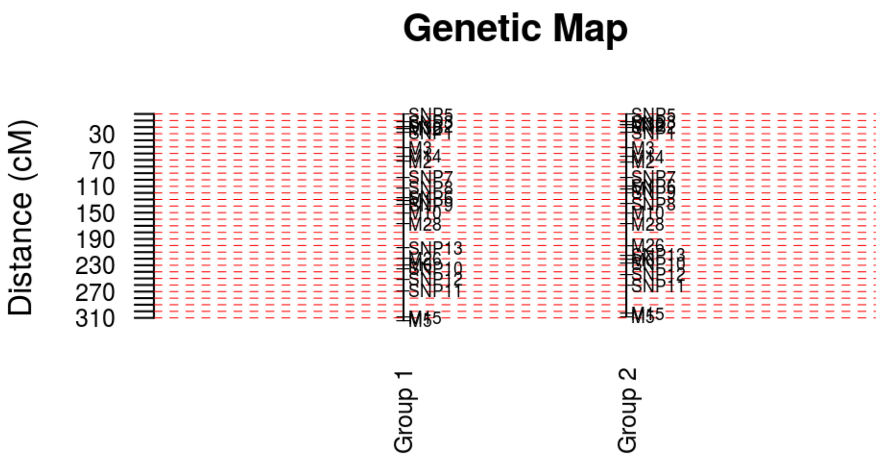
Drawing the map, which built with reference genome and recombinations information

```r
map2 <- list(CHR1_final, CHR2_final, CHR3_final)
draw_map(map2, names = TRUE, grid = TRUE, cex.mrk = 0.7)
```

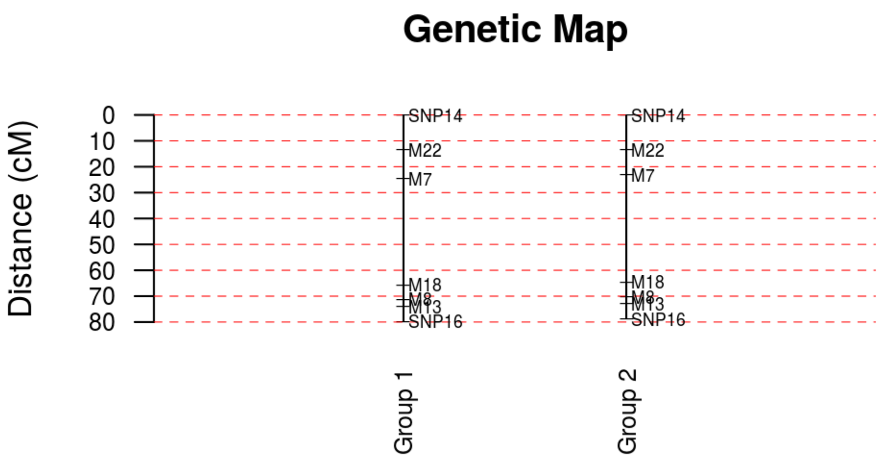


We also can draw a maps comparing corresponding linkage groups in each strategy:

```
CHR1_comp <- list(LG1_final, CHR1_final)
draw_map(CHR1_comp, names = TRUE, grid = TRUE, cex.mrk = 0.7)
```

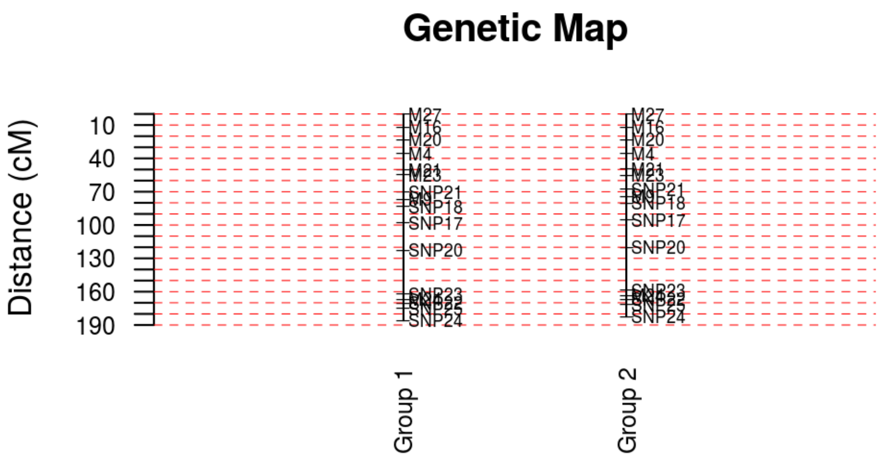


```
CHR2_comp <- list(LG3_final, CHR2_final)
draw_map(CHR2_comp, names = TRUE, grid = TRUE, cex.mrk = 0.7)
```



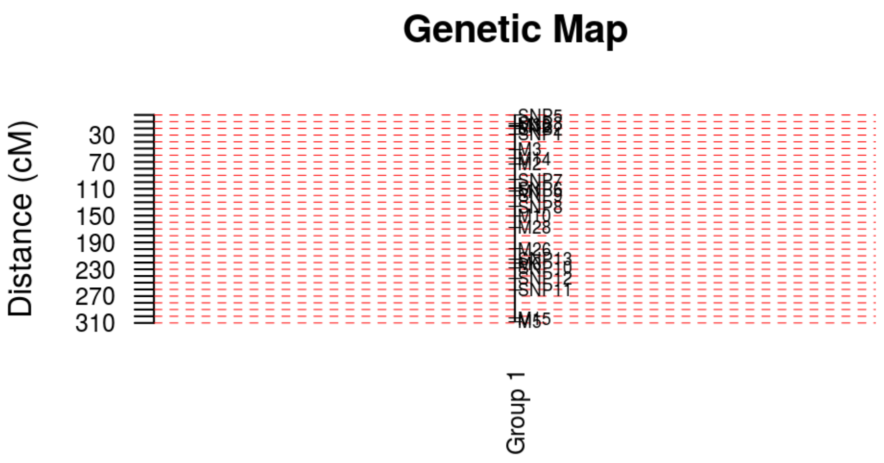
Both strategies produced the same result for CHR2 (the map is only inverted).

```
CHR3_comp <- list(LG2_final, CHR3_final)
draw_map(CHR3_comp, names = TRUE, grid = TRUE, cex.mrk = 0.7)
```



Or groups alone:

```
draw_map(CHR1_final, names = TRUE, grid = TRUE, cex.mrk = 0.7)
```



Function `draw_map` draws a straightforward graphic representation of the genetic map. More recently, we developed a new version called `draw_map2` that brings a more sophisticated figure. Furthermore, once the distances and the linkage phases are estimated, other map figures can be drawn by the user with any appropriate software. There are several free software that can be used, such as `MapChart` (Voorrips, 2002).

Draw_map2

The same figures did with `draw_map` can be done with `draw_map2` function. But it has different capacities and arguments. Here are some examples, but you can find more options in the help page `?write_map2`.

Drawing the map, which was built with only recombinations information

```r
draw_map2(LG1_final, LG2_final, LG3_final,
          main = "Only with linkage information",
          group.names = c("LG1", "LG2", "LG3"))
```

The figure will be saved in your work directory with the default name `map.eps`. You can change the file name and extension specifying them in the argument `output`.

Drawing the map, which was built with reference genome and recombinations information

```r
draw_map2(CHR1_final, CHR2_final, CHR3_final, output= "map_ref.pdf",
          col.group = "#58A4B0",
          col.mark= "#335C81")
```

With the argument `tag`, we can highlight some markers with other colors. The arguments `col.group`, `col.mark` and `col.tag` can be changed to personalize the color of the groups, the markers, and the highlighted markers, respectively.

We also can draw a maps comparing corresponding linkage groups in each strategy:

```r
draw_map2(LG1_final, CHR1_final, output = "map_comp.pdf", tag = c("M1","SNP2"))
```

When defining marker names in `tag` argument, all markers with these name will be highlighted no matter in which group it/they is/are.

```r
draw_map2(LG2_final, CHR3_final, tag= c("SNP17", "SNP18", "M29"),
          main = "Chromosome 3",
          group.names = c("Only linkage", "With genome"),
          centered = TRUE, output = "map_comp2.pdf")
```

Map estimation for an arbitrary order

If for any reason, one wants to estimate parameters for a given linkage map (*e.g.*, for other orders on published papers), it is possible to define a sequence and use the `map` function. For example, for markers `M30`, `M12`, `M3`, `M14` and `M2`, in this order, use:

```
any_seq <- make_seq(twopts, c(30, 12, 3, 14, 2))
(any_seq_map <- map(any_seq))
#>
#> Printing map:
#>
#> Markers           Position           Parent 1       Parent 2
#>
#> 30 M30                0.00           a |  | b       a |  | b
#> 12 M12                1.00           b |  | a       c |  | a
#>  3 M3                20.57           o |  | a       o |  | o
#> 14 M14               33.63           a |  | o       b |  | o
#>  2 M2                41.70           o |  | o       o |  | a
#>
#> 5 markers            log-likelihood: -320.9012
```

NOTE: (new!) If your sequence has many markers (more than 60), we suggest to speed up `map` using BatchMap parallelization approach. See section `Speed up analysis with parallelization` for more information.

Warning: If you find an error message like:

```
Error in as_mapper(.f, ...) : argument ".f" is missing, with no default
```

It's because the `map` function has a very common name, and you can have in your environment other function with the same name. In the case of the pointed error, R is using `map` function from `purrr` package instead of `OneMap`, to solve this, simply specify that you want `OneMap` function with `::` command from `stringr` package::

```
library(stringr)
(any_seq_map <- onemap::map(any_seq))
```

This is a subset of the first linkage group. When used this way, the `map` function searches for the best combination of phases between markers and prints the results.

(new!)

Warning: It is not our case in this example, but, sometimes, it can happen that some markers in your sequence don't reach the `OneMap` linkage criteria when linkage are estimated by HMM multipoint approach using `map`, it will produce an error like this:

```
ERROR: The linkage between markers 1 and 2 did not reach
the OneMap default criteria. They are probably segregating independently
```

You can evaluate the marker manually, or you can remove them automatically using `map` argument `rm_unlinked = TRUE`. The `map` function will return a vector with marker numbers excluding the problematic marker; then, you can repeat the process without the marker, using `make_seq` to create a new sequence and repeat the `map`. You can also do it automatically using the `map_avoid_unlinked` function:

```
LG2_test_map <- map_avoid_unlinked(any_seq)
```

Using this, if `map` finds a problematic marker, it will print a warning pointing the marker number, which was removed and will automatically repeat the analysis without it.

Furthermore, a sequence can also have user-defined linkage phases. The next example shows (incorrect) phases used for the same order of markers:

```
any_seq <- make_seq(twopts, c(30, 12, 3, 14, 2), phase = c(4, 1, 4, 3))
(any_seq_map <- map(any_seq))
```

If one needs to add or drop markers from a predefined sequence, functions `add_marker` and `drop_marker` can be used. For example, to add markers 4 to 8 to `any_seq`.

```
(any_seq <- add_marker(any_seq, 4:8))
#>
#> Number of markers: 10
#> Markers in the sequence:
#> M30 M12 M3 M14 M2 M4 M5 M6 M7 M8
#>
#> Parameters not estimated.
```

Removing markers 3, 4, 5, 12 and 30 from `any_seq`:

```
(any_seq <- drop_marker(any_seq, c(3, 4, 5, 12, 30)))
#>
#> Number of markers: 5
#> Markers in the sequence:
#> M14 M2 M6 M7 M8
#>
#> Parameters not estimated.
```

After that, the map needs to be re-estimated.

Speed up analysis with parallelization (new!)

Warning: By now, only available for Unix-like operating systems and for outcrossing and f2 intercross populations.

As already mentioned, `OneMap` uses HMM multipoint approach to estimate genetic distances, a very robust method, but it can take time to run if you have many markers. In 2017, Schiffthaler et al. release an `OneMap` fork with modifications in CRAN and in GitHub with the possibility of parallelizing the HMM chain dividing markers in batches and use different cores for each phase. Their approach speeds up our HMM and keeps the genetic distance estimation quality. It allows us to divide the job in a maximum of four cores according to the four possible phases for outcrossing mapping populations. We add this parallelized approach to the functions: `map`, `mds_onemap`, `seriation`, `rcd`, `record` and `ug`. For better efficiency, batches must be composed of 50 markers or more; therefore, this approach is only recommended for linkage groups with many markers.

Here we will show an example of how to use the BatchMap approach in some functions that requires HMM. For this, we will simulate a group with 294 markers (we don't want this vignette takes too much time to run, but usually maps with markers from high-throughput technologies result in larger groups). Before start, you can see the time spent on each approach in this example:

	Without parallelization (h)	With parallelization (h)
rcd	0.6700558	0.1612458
record_map	1.4368436	0.2907308

	Without parallelization (h)	With parallelization (h)
ug_map	0.7145778	0.1884214
mds_onemap	1.0643083	0.2827314
map	2.0994486	0.6107456

Now, let's simulate the mapping population and markers.

```r
run_pedsim(chromosome = "Chr1", n.marker = 294,
           tot.size.cm = 100, centromere = 50,
           n.ind = 200,
           mk.types = c("A1", "A2", "B3.7", "D1.9",
                        "D1.10", "D2.14", "D2.15"),
           n.types = rep(42,7), pop = "F1", path.pedsim = "./",
           name.mapfile = "mapfile.txt",
           name.founderfile="founderfile.gen",
           name.chromfile="sim.chrom", name.parfile="sim.par",
           name.out="simParall_out")

# Do the conversion

pedsim2raw(cross="outcross",
           genofile = "simParall_out_genotypes.dat",
           parent1 = "P1", parent2 = "P2",
           out.file = "simParall_out.raw",
           miss.perc = 25)

# Import to R environment as onemap object

simParallel <- read_onemap("simParall_out.raw")
plot(simParallel, all=FALSE)
```

```r
# Calculates two-points recombination fractions
twopts <- rf_2pts(simParallel)

seq_all <- make_seq(twopts, "all")

# There are no redundant markers
find_bins(simParallel)

# There are no distorted markers
p <- plot(test_segregation(simParallel))
```

To prepare the data with defined bach size, we use function `pick_batch_sizes`. It selects a batch size that splits the data into even groups. Argument `size` defines the batch size next to which an optimum size will be searched. `overlap` defines the number of markers that overlap between the present batch and next. This is used because pre-defined phases at these overlap markers in the present batch are used to start the HMM in the next batch. The `around` argument defines how much the function can vary around the defined number in `size` to search for the optimum batch size.

Some aspects should be considered to define these arguments because if the batch size were set too high, there would be less gain in execution time. If the overlap size were too small, phases would be incorrectly estimated, and large gaps would appear in the map, inflating its size. In practice, these values will depend on many factors such as population size, marker quality, and species. BatchMap authors recommended to try several configurations on a subset of data and select the best performing one.

```r
batch_size <- pick_batch_sizes(input.seq = seq_all,
                               size = 80,
                               overlap = 30,
                               around = 10)

batch_size
```

Speed up two-points ordering approaches

To use parallelized approach you just need to include the arguments when using the functions:

```r
# Without parallelization
rcd_map <- rcd(input.seq = seq_all)

# With parallelization
rcd_map_par <- rcd(input.seq = seq_all,
                   phase_cores = 4,
                   size = batch_size,
                   overlap = 30)
```

```r
# Without parallelization
record_map <- record(input.seq = seq_all)

# With parallelization
record_map_par <- record(input.seq = seq_all,
                         phase_cores = 4,
                         size = batch_size,
                         overlap = 30)
```

```r
# Without parallelization
ug_map <- ug(input.seq = seq_all)

# With parallelization
ug_map_par <- ug(input.seq = seq_all,
                 phase_cores = 4,
                 size = batch_size,
                 overlap = 30)
```

Speed up mds_onemap

```r
# Without parallelization ok
map_mds <- mds_onemap(input.seq = seq_all)

# With parallelization
map_mds_par <- mds_onemap(input.seq = seq_all,
                          phase_cores = 4,
                          size = batch_size,
                          overlap = 30)
```

Speed up map estimation for an arbitrary order (map function)

Because we simulate this dataset, we know the correct order. We can use `map_overlapping_batches` to estimate genetic distance in this case. This is equivalent to `map`, but with a parallelized process.

Similarly, with `map`, using argument `rm_unlinked = TRUE` the function will return a vector with marker numbers without the problematic marker. To repeat the analysis removing automatically all problematic markers use `map_avoid_unlinked`:

```
# Without parallelization
batch_map <- map_avoid_unlinked(input.seq = seq_all)

# With parallelization
batch_map_par <- map_avoid_unlinked(input.seq = seq_all,
                                    size = batch_size,
                                    phase_cores = 4,
                                    overlap = 30)
```

As you can see in the above maps, heuristic ordering algorithms do not return an optimal order result, mostly if you don't have many individuals in your population. Because of the erroneous order, generated map size is not close to the simulated size (100 cM) and their heatmaps don't present the expected color pattern. Two of them get close to the color pattern, they are the ug and the MDS method. They present the right global ordering but not local. If you have a reference genome, you can use its position information to rearrange the local order.

Export estimated parents haplotypes (new!)

In the older version, users could only access the estimated linkage phase observing the print in the console:

```
CHR3_final
#>
#> Printing map:
#>
#> Markers           Position           Parent 1       Parent 2
#>
#> 27 M27                0.00           b |   | o       a |   | a
#> 16 M16               11.76           a |   | a       b |   | o
#> 20 M20               22.94           a |   | b       c |   | d
#>  4 M4                35.71           a |   | o       o |   | b
#> 21 M21               49.41           o |   | o       a |   | b
#> 23 M23               55.24           a |   | o       a |   | o
#> 48 SNP21             67.50           a |   | b       a |   | a
#>  9 M9                74.02           a |   | b       a |   | a
#> 46 SNP18             80.43           a |   | b       a |   | a
#> 45 SNP17             95.40           a |   | b       a |   | a
#> 47 SNP20            119.95           a |   | b       a |   | a
#> 50 SNP23            158.43           a |   | b       b |   | a
#> 24 M24              163.43           a |   | b       b |   | a
#> 49 SNP22            166.72           a |   | b       b |   | a
#> 52 SNP25            171.62           a |   | b       b |   | a
#> 51 SNP24            182.56           a |   | b       b |   | a
#>
#> 16 markers            log-likelihood: -811.5279
```

Now, you can export this information into a data.frame using:

```
(parents_haplot <- parents_haplotypes(CHR3_final))
#>       group mk.number mk.names dist P1_1 P1_2 P2_1 P2_2
#> 1 Group - 1        27      M27    0    b    o    a    a
#>  [ reached 'max' / getOption("max.print") -- omitted 15 rows ]
```

```
write.table(parents_haplot, "parents_haplot.txt")
```

The data.frame contains: group ID (group), marker number (mk.number) and names (mk.names), position in centimorgan (dist) and parents haplotypes (P1_1, P1_2, P2_1, P2_2).

You can also obtain a data.frame with a list of sequences and personalize the group names:

```
parents_haplotypes(CHR2_final,CHR3_final, group_names=c("CHR2","CHR3"))
#>   group mk.number mk.names dist P1_1 P1_2 P2_1 P2_2
#> 1  CHR2        43    SNP14    0    a    b    a    a
#>  [ reached 'max' / getOption("max.print") -- omitted 22 rows ]
```

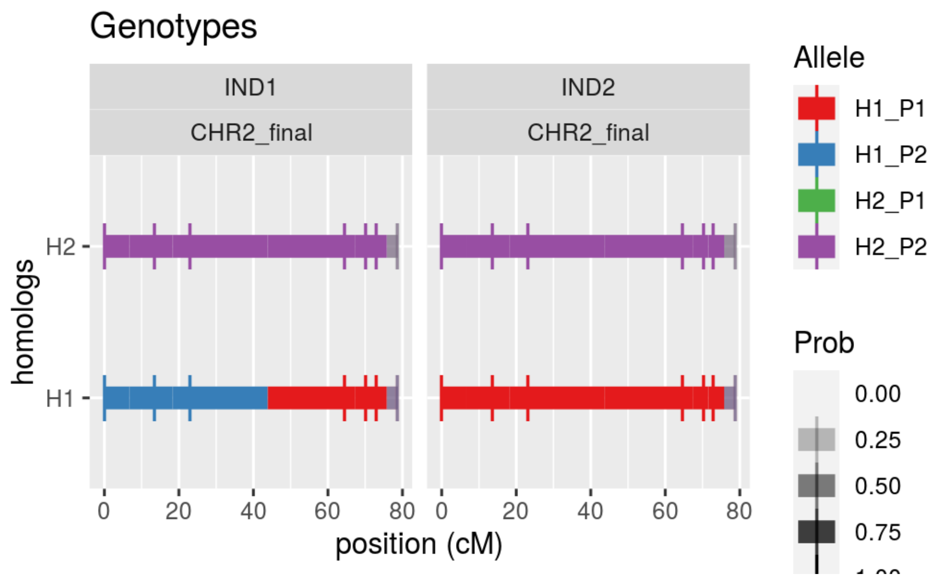
Export estimated progeny haplotypes (new!)

Function `progeny_haplotypes` generates a data.frame with progeny phased haplotypes estimated by `OneMap` HMM. For progeny, the HMM results in probabilities for each possible genotypes, then the generated data.frame contains all possible genotypes. If `most_likely = TRUE`, the most likely genotype receives 1 and the rest 0 (if there are two most likely both receive 0.5), if `most_likely = FALSE` genotypes probabilities will be according with the HMM results. You can choose which individual to be evaluated in `ind`. The data.frame is composed by the information: individual (ind) and group (grp) ID, position in centimorgan (pos), progeny homologs (homologs), and from each parent the allele came (parents).

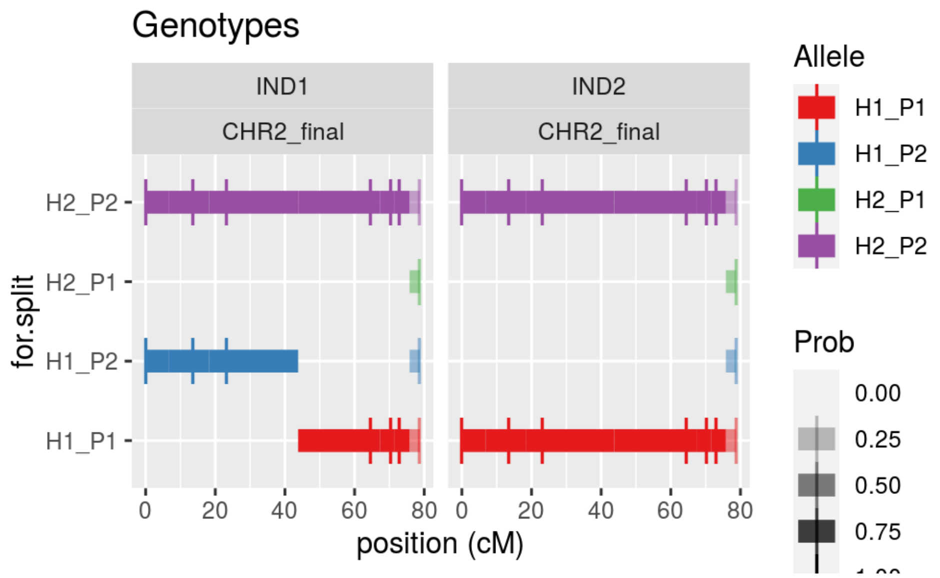
```
(progeny_haplot <- progeny_haplotypes(CHR2_final, most_likely = TRUE,
                                      ind = c(1,2),
                                      group_names = "CHR2_final"))
#>    ind        grp pos prob homologs parents
#> 1 IND1 CHR2_final   0    0       H1      P1
#>  [ reached 'max' / getOption("max.print") -- omitted 55 rows ]
```

You can also have a view of progeny estimated haplotypes using `plot`. It shows which markers came from each parent's homologs. `position` argument defines if haplotypes will be plotted by homologs (`stack`) or alleles (`split`). `split` option is a good way to view the likelihoods of each alleles.

```
plot(progeny_haplot, position = "stack")
```



```r
plot(progeny_haplot, position = "split")
```



Comparing estimated and simulated haplotypes

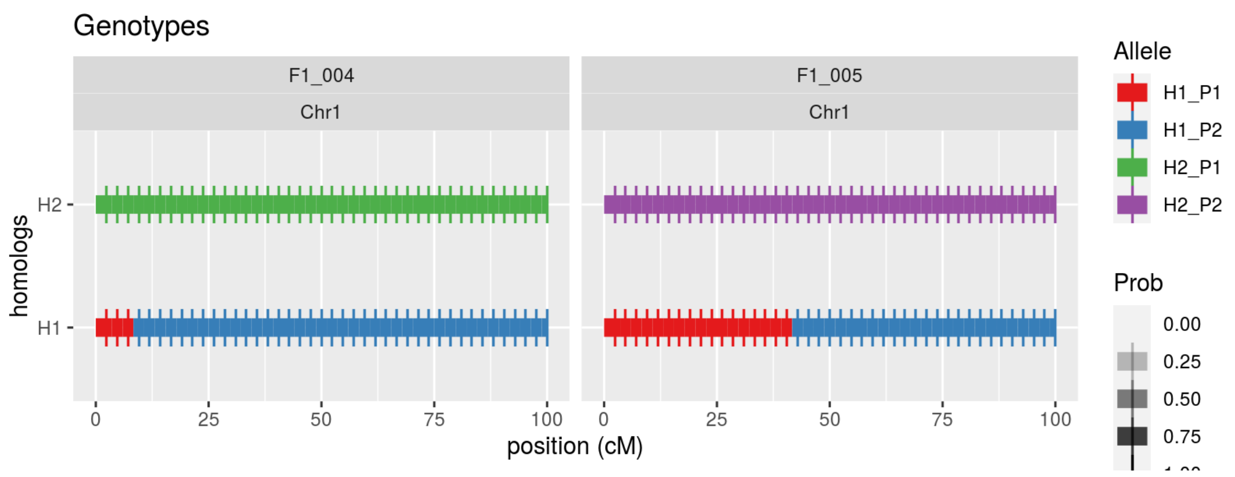
The simulations here included are very useful to test the efficiency of our algorithms, we made them available in case to be useful for other purposes. Using functions `run_pedsim`, `pedsim2vcf` with argument phased `TRUE`, we can convert the phased VCF using `vcf2progeny_haplotypes`, which keeps the original haplotype. With that, we can compare the haplotypes simulated and the ones estimated by the onemap approach.

Here we will use the file `sim_out_cod_genotypes.dat` to get one phased VCF. See that now we don't want to simulate read counts or genotype errors; we just want the original haplotypes in VCF format, then we set `counts = FALSE` and don't need to define the related arguments.

```r
pedsim2vcf(inputfile = system.file("extdata/sim_cod_out_genotypes.dat",
                                   package = "onemap"),
           map.file = system.file("extdata/mapfile_out.txt",
                                   package = "onemap"),
           chrom.file = system.file("extdata/sim_out.chrom",
                                     package = "onemap"),
           out.file = "simu_out_phased.vcf",
           miss.perc = 0,
           counts = FALSE,
           chr.mb = 10,
           pos="cM",
           chr=NULL,
           phase = TRUE)
```

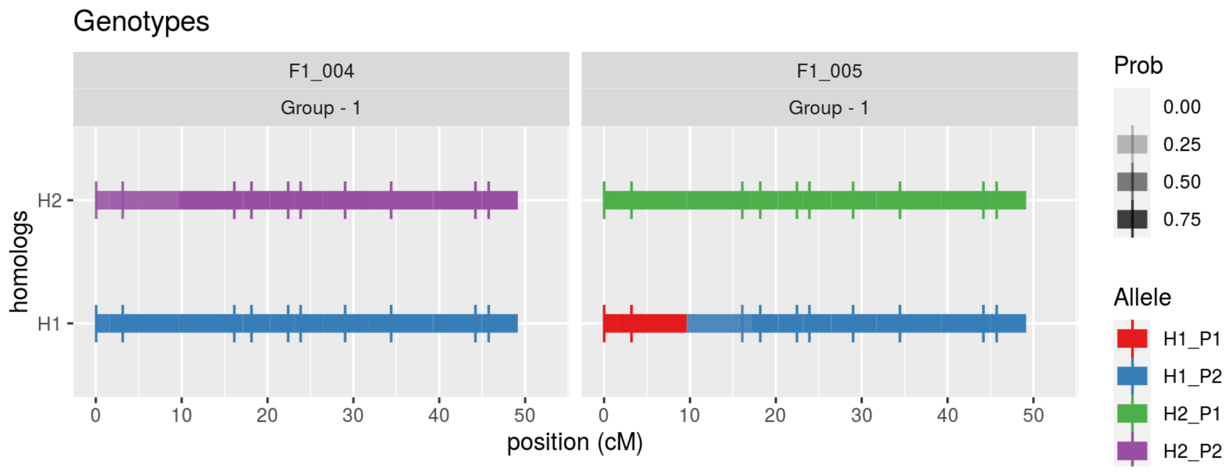
Now, we can convert the phased VCF to onemap_progeny_haplotypes object using function `vcf2progeny_haplotypes`. The position, the group names, and individuals id will be according to the ones contained in VCF file. This function can take some time to run if you select too many individuals or groups.

```
vcfR.object <- read.vcfR("simu_out_phased.vcf")
#> Scanning file to determine attributes.
#> File attributes:
#>   meta lines: 4
#>   header_line: 5
#>   variant count: 42
#>   column count: 211
#>
Meta line 4 read in.
#> All meta lines processed.
#> gt matrix initialized.
#> Character matrix gt created.
#>   Character matrix gt rows: 42
#>   Character matrix gt cols: 211
#>   skip: 0
#>   nrows: 42
#>   row_num: 0
#>
Processed variant: 42
#> All variants processed

progeny_dat <- vcf2progeny_haplotypes(vcfR.object = vcfR.object,
                                      ind.id = c("F1_004", "F1_005"),
                                      group_names = "Chr1",
                                      parent1 = "P1", parent2 = "P2",
                                      crosstype = "outcross")
plot(progeny_dat)
```



This is the simulated dataset. If OneMap algorithm could get all information needed, it could reproduce these haplotypes exactly how they were in simulated data. Let's see how OneMap estimates these haplotypes:

```
onemap_test <- onemap_read_vcfR(vcfR.object = vcfR.object,
                                cross = "outcross", parent1 = "P1",
                                parent2 = "P2", only_biallelic = FALSE)

onemap_test # Just to remember of which dataset we are talking about
#>   This is an object of class 'onemap'
#>     Type of cross:      outcross
#>     No. individuals:    200
#>     No. markers:        11
#>     CHROM information:  yes
#>     POS information:    yes
#>     Percent genotyped:  90
#>
#>     Segregation types:
#>                A.1 -->  6
#>              D1.10 -->  5
#>
#>     No. traits:         0

twopts <- rf_2pts(onemap_test)
#> Computing 55 recombination fractions ...
seq1 <- make_seq(twopts, "all")
map_test <- map(seq1)
progeny_est <- progeny_haplotypes(map_test, ind = c(4,5),
                                  most_likely = FALSE)
plot(progeny_est)
```



See that, besides markers are not in their exact position, the recombination breakpoints are the same as the ones simulated. The marker position could only be the same as the ones simulated if we have a very big (or maybe infinite) population size. Still, here we have only 200 individuals, but this is not a problem once we could reproduce the haplotypes.

Session Info

```
sessionInfo()
#> R version 3.6.1 (2019-07-05)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 16.04.6 LTS
#>
#> Matrix products: default
#> BLAS:   /usr/lib/openblas-base/libblas.so.3
#> LAPACK: /usr/lib/libopenblasp-r0.2.18.so
#>
#> locale:
#>  [1] LC_CTYPE=pt_BR.UTF-8    LC_NUMERIC=C            LC_TIME=pt_BR.UTF-8
#>  [4] LC_COLLATE=en_US.UTF-8  LC_MONETARY=pt_BR.UTF-8 LC_MESSAGES=en_US.UTF-8
#>  [7] LC_PAPER=pt_BR.UTF-8    LC_NAME=C               LC_ADDRESS=C
#> [10] LC_TELEPHONE=C
#>  [ reached getOption("max.print") -- omitted 2 entries ]
#>
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   base
#>
#> other attached packages:
#> [1] stringr_1.4.0  vcfR_1.8.0     rmarkdown_1.18 knitr_1.29     onemap_2.2.0
#>
#> loaded via a namespace (and not attached):
#>  [1] colorspace_1.4-1          ggsignif_0.6.0
#>  [3] ellipsis_0.3.1            class_7.3-15
#>  [5] rio_0.5.16                htmlTable_2.0.1
#>  [7] RcppArmadillo_0.9.900.2.0 base64enc_0.1-3
#>  [9] rstudioapi_0.11           mice_3.10.0.1
#>  [ reached getOption("max.print") -- omitted 116 entries ]
```

References

Buetow, K. H., Chakravarti, A. Multipoint gene mapping using seriation. I. General methods. ***American Journal of Human Genetics*** 41, 180-188, 1987.

Doerge, R.W. Constructing genetic maps by rapid chain delineation. ***Journal of Agricultural Genomics*** 2, 1996.

Mollinari, M., Margarido, G. R. A., Vencovsky, R. and Garcia, A. A. F. Evaluation of algorithms used to order markers on genetics maps. ***Heredity*** 103, 494-502, 2009.

Schiffthaler, B., Bernhardsson, C., Ingvarsson, P. K., & Street, N. R. BatchMap: A parallel implementation of the OneMap R package for fast computation of F1 linkage maps in outcrossing species. ***PLoS ONE***, 12(12), 1–12, 2017.

Tan, Y., Fu, Y. A novel method for estimating linkage maps. ***Genetics*** 173, 2383-2390, 2006.

Van Os H, Stam P, Visser R.G.F., Van Eck H.J. RECORD: a novel method for ordering loci on a genetic linkage map. ***Theor Appl Genet*** 112, 30-40, 2005.

Voorrips, R.E. MapChart: software for the graphical presentation of linkage maps and QTLs. ***Journal of Heredity*** 93, 77-78, 2002.

Voorrips, R. E., Maliepaard, C. A. The simulation of meiosis in diploid and tetraploid organisms using various genetic models. ***BMC Bioinformatics***, 13(1), 248, 2012.

Wu, R., Ma, C.X., Painter, I. and Zeng, Z.-B. Simultaneous maximum likelihood estimation of linkage and linkage phases in outcrossing species. ***Theoretical Population Biology*** 61, 349-363, 2002a.

Wu, R., Ma, C.-X., Wu, S. S. and Zeng, Z.-B. Linkage mapping of sex-specific differences. ***Genetical Research*** 79, 85-96, 2002b.