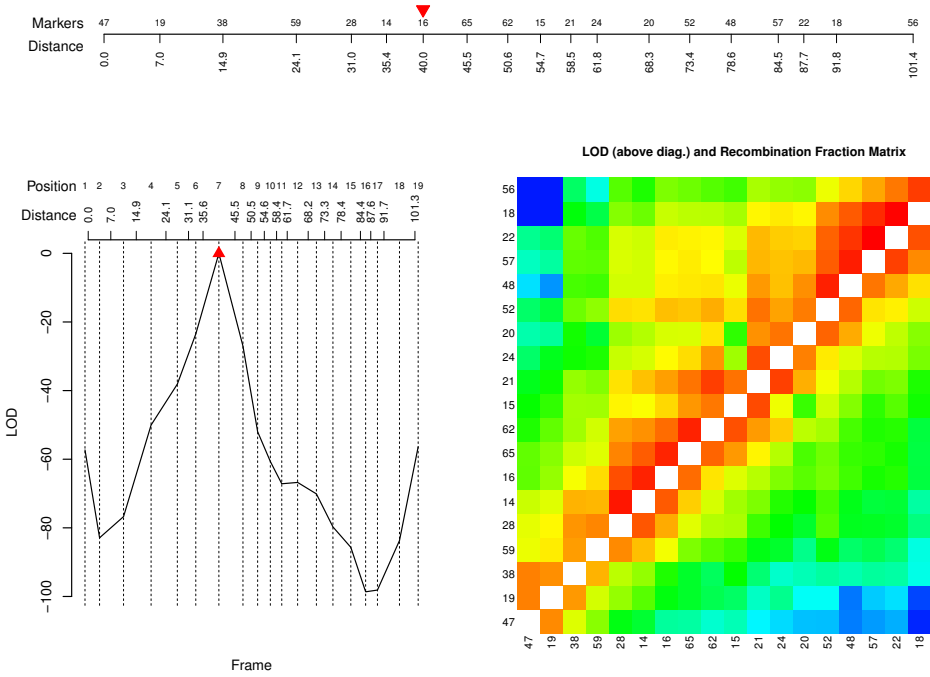
OneMap Tutorial

Software for constructing genetic maps in experimental crosses: full-sib, RILs, F_2 and backcrosses

Gabriel R A Margarido, Marcelo Mollinari and A Augusto F Garcia*



Department of Genetics
Escola Superior de Agricultura "Luiz de Queiroz" (ESALQ)
Universidade de São Paulo (USP) - Brazil
E-mail: augusto.garcia@usp.br

*corresponding author

December 6, 2012

Contents

1 Overview

OneMap is an environment for constructing linkage maps in several experimental crosses, including outcrossing (full-sib families derived from two non-homozygous parents), RILs, F_2 and backcrosses. It is implemented as a package to be used under the freely distributed R software, which is a language and environment for statistical computing (`www.r-project.org`). It is designed to be fully integrated with R/qtl package (Broman et al., 2008) and Windows QTL Cartographer (Wang et al., 2010) in order to do QTL mapping.

Wu et al. (2002a) proposed a methodology to construct genetic maps in outcrossing species, which allows the analysis of a mixed set of different marker types containing various segregation patterns. Also, it allows the simultaneous estimation of linkage and linkage phases between markers, and was successfully applied in the analysis of sugarcane (Garcia et al., 2006; Oliveira et al., 2007) and *Passiflora* (Oliveira et al., 2008) data sets. Actually, the analysis of these data sets motivated the implementation of the first release of *OneMap* (Margarido et al., 2007).

After extensively testing the software, we noticed that the construction of linkage maps could be greatly enhanced with the use of multipoint likelihood through Hidden Markov Models (HMM). Jiang and Zeng (1997) explained in detail this methodology, emphasizing its advantages and limitations for populations derived from inbred lines. Merging the ideas of Wu et al. (2002a) and the HMM framework, as done by Wu et al. (2002b), we then developed version 1.0-0 of *OneMap*, which could order markers using HMM-based algorithms for outcrossing species, in a similar way as implemented in MAPMAKER/EXP (Lander et al., 1987). We verified the great advantages of the new procedure through extensive simulations.

In version 2.0-0, we included several major modifications to take advantage of the fact that some segregation patterns that occur in outcrossing populations can also occur in populations derived from inbred lines (i.e. RILs, F_2 and backcrosses). For example, a marker that segregates in $1:2:1$ fashion in outcrossing context can be viewed as a co-dominant marker in F_2 populations. The main difference is that, for the later, there is no need to estimate linkage phases. Using these ideas, we adapted *OneMap* to also construct genetic maps in RILs, F_2 and backcross populations, taking advantage of *OneMap* facilities. Moreover, we also implemented three new ordination algorithms besides the ones included in version 1.0-0: Rapid Chain Delineation - RCD (Doerge, 1996) and TRY (Lander et al., 1987). They are Seriation - SER (Buetow and Chakravarti, 1987), recombination counting and ordering - RECORD (Van Os et al., 2005) and unidirectional growth - UG (Tan and Fu, 2006). They can be used for all experimental crosses included in *OneMap*, and can be chosen to give the best result for any situation faced

by the user (Mollinari et al., 2009)

OneMap is available as source code for Windows™ and Unix systems. It is released under the GNU General Public License, is open-source and the code can be changed freely. It comes with no warranty.

Although no advanced knowledge in R is required to use *OneMap*, in Section 2 we present a short introduction to R software, where we address the basic knowledge required to start using *OneMap*. People with some knowledge of R could just skip this part. In Section 3, information about *OneMap* installation is provided. In Section 4, we show the usage of *OneMap* functions for outcrossing (non-inbred) populations. In Section 5 we do the same for F_2 populations, which can also be applied to backcrosses and RILs. All sections could be read independently.

1.1 Citation

Margarido, G.R.A., Souza, A.P. and Garcia, A.A.F. OneMap: software for genetic mapping in outcrossing species. *Hereditas* 144: 78-79, 2007.

2 Introduction to R

R is a language and environment for statistical computing and graphics. To download R, please visit the Comprehensive R Archive Network (`cran.r-project.org`). Although we prefer and recommend the Linux version, in this tutorial, it is assumed that the user is running Windows™. Users of R under Linux or Mac® OS should have no difficult in following this tutorial.

After installing R, you can launch it by double-clicking the R icon created on your desktop during the installation process. You will see a window with the R Console (Figure 1).

2.1 Getting started

In Figure 1, you can see a *greater than* sign (">"), which means that R is waiting for a command. We call this *prompt*. Let us start with a simple example adding two numbers. Type "2+3" at the prompt then type the `Enter` key:

```
> 2+3
```

You can see the result directly on the screen. You can store this result into a variable for future use, applying the assignment operator `<-` (*less than* sign and *minus*, altogether):

```
> x<-2+3
```

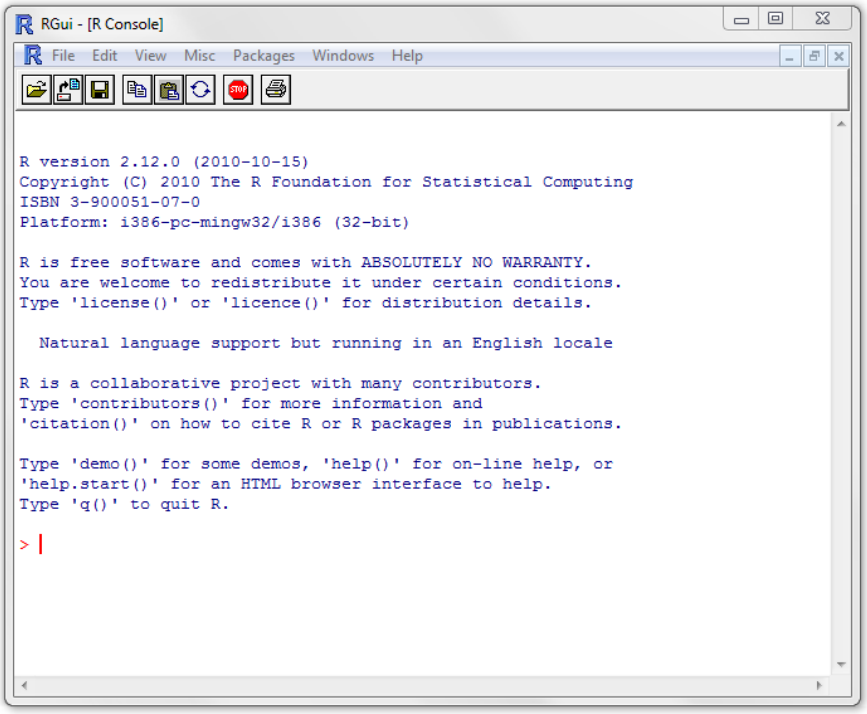


Figure 1: The R Console.

The result of the calculation was stored into the variable `x`. You can access this result typing "x" at the prompt:

```
> x
```

You can also use the variable `x` into another calculation, for example:

```
> x+4
```

2.2 Functions

Another fundamental aspect in R is the usage of *functions*. A function is a predefined routine used to do specific calculations. For example, to calculate the natural logarithm of 6.7, we can use the function `log`:

```
> log(6.7)
```

The function `log` contains a group of internal procedures to calculate the natural logarithm of a positive real number. The input values of a function are called *arguments*. In previous example, we provided only one argument to the function (6.7). Sometimes a function has more than one argument. For example, to obtain the logarithm of 6.7 to base 4, you can use:

```
> log(6.7,base=4)
```

It is possible to calculate the natural logarithm of a set of numbers by defining a vector and using it as the first argument of the function `log`. To do so we use the function `c`, that *combines* a set of values into a vector. Thus, to calculate the logarithm of the numbers 6.7, 3.2, 5.4, 8.1, 4.9, 9.7 and 2.5, we can use:

```
> y<-c(6.7, 3.2, 5.4, 8.1, 4.9, 9.7, 2.5)
> log(y)
```

2.3 Getting help

Every R function has a help page which can be accessed using a question mark before the name of the function. For example, to get help on function `log`, you would type:

```
> ?log
```

This command will open a help page in the default web browser of your system. The help page contains some important information about the function such its syntax, its arguments and some usage examples.

2.4 Packages

Although R has a huge amount of internal functions, for doing more specific computations, like constructing genetic linkage maps, it is necessary to use complementary functions. These functions can be obtained by installing a *package*. A package is a collection of related functions, help files and example data files that have been bundled together (Adler, 2010).

For example, let us assume you need to convert a set of recombination fractions into centimorgan distance using the Kosambi function. One possible way to do that, is to use the basic R functions to calculate the distances. Another way is use the *OneMap* package. To install *OneMap* you can type:

```
> install.packages("onemap")
```

You also can use the console menus: *Packages* $\rightarrow$ *Install package(s)*. After clicking, a box will pop-up asking you to choose the CRAN mirror. Choose the location nearest to you. Then, another box will pop-up asking you to choose the package you want to install. Select `onemap` then click *OK*. The package will be automatically installed on your computer. Returning to the console, you need to load *OneMap* by typing:

```
> library("onemap")
```

Let us enter some recombination fractions, for example, 0.01, 0.12, 0.05, 0.11, 0.21, 0.07, and save it into a variable called `rf`:

```
> rf<-c(0.01, 0.12, 0.05, 0.11, 0.21, 0.07)
```

Now, let us use the function `kosambi`, which belongs to *OneMap* package, to do the calculation:

```
> kosambi(rf)
```

You can also obtain help on the function `kosambi` using the question mark in the same way it was done with function `log`:

```
> ?kosambi
```

2.5 Importing and exporting data

So far, we entered the variables in R by typing them directly into the console. However, in real situations we usually *read* these values from a file or a data bank. To exemplify this procedure, copy and paste the following table into a text editor (for example, *notepad*) and save it to a file called `test.txt` into your *working directory* (such as *My Documents*).

```
     x       y
  2.13    4.50
  4.48    1.98
 10.95    9.29
 10.03   16.25
 12.72   27.38
 24.63   22.60
 22.57   36.87
 29.78   31.73
 19.54   10.42
  7.86   14.68
 11.75    8.68
 23.71   37.39
```

To read these data in R, first, we have to set the working directory using the function `setwd`. For example, if `"C:/Users/mmollina/Documents"` is the full path to *My Documents* directory, one should use:

```
> setwd("C:/Users/mmollina/Documents")
```

Every time you inform paths, directories or files you have to use double quotes (“ ”), which indicates a string of characters instead of a variable. You also can use the console menus to set the working directory: *File* → *Change Dir....* From here, every object will be read or saved to this directory.

Now let us read the file `test.txt` into R and store it in a variable called `dat` using the function `read.table`. The first argument is the name of the file. The second indicates if the file contains a header, i. e. if the first line of the file contains the names of the variables:

```
> (dat<-read.table(file="test.txt", header=TRUE))
```

Notice that the whole command line is limited by parenthesis. This indicates to R to show the results at the same time you store then into a variable. One could type the command without parenthesis and then type `dat` at the prompt, to produce the same result. Inspecting the object `dat` you can see a table with 12 rows and two columns. The names of the columns are `x` and `y`. We can access the variables in columns using the dollar sign followed by the column name:

```
> dat$x
> dat$y
```

It is also possible to use a function called `summary` to extract some information about the object `dat` or about each one of the columns separately::

```
> summary(dat)
> summary(dat$x)
> summary(dat$y)
```

The function `summary` provides some basic statistics about the variables in the dataset. If you want to export these information to a file you can use the function `write.table`:

```
> write.table(x=summary(dat), file="test_sum.txt", quote=FALSE)
```

The first argument is the output of the `summary` function. Note that is possible to use a function as an argument of another one. The second argument is the name of the file in which the summary is going to be written. Notice that the file will be written in the *working directory*, previously set. The third argument eliminates double quotes from the output file. After running the command, you can look for the file `test_sum.txt` in the *working directory*.

2.6 Classes and methods

In R, every object belongs to a *class*. For example, the object `dat` belongs to a class called `data.frame`. We can obtain this information using the function `class`:

```
> class(dat)
```

When we use the function `summary`, it recognizes the class of the `dat` and applies a specific procedure to the `data.frame` class, which in this case involves the computation of some descriptive statistics. This procedure is called *method*. However, another classes of objects can be used as arguments to function `summary` and the result will be different. For example, let us adjust a linear model using column `y` as the dependent variable and column `x` as independent. This can be done with the function `lm()`:

```
> ft.mod<-lm(dat$y~dat$x)
> ft.mod
```

Function `lm` is used to fit linear models and, by default, returns just a formula and the coefficients of the linear regression. Object `ft.mod` is of class `lm`:

```
> class(ft.mod)
```

To obtain more information about the fitted model, we can use the function `summary`:

```
> summary(ft.mod)
```

In this case, the function `summary` recognizes `lm.fit` as an object of class `lm` and applies a method which shows information about the fitted model such as distribution of the residuals, regression coefficients, t-tests, and the coefficient of determination (r^2), etc (significance stars not shown). Thus, it is possible to use the same function in different classes of object to obtain different results. This concept is very important in *OneMap*. For example, depending on the class of the dataset, which can be `outcross`, `f2.onemap`, `bc.onemap`, `riself.onemap` and `risib.onemap`, a certain set of procedures will be applied.

2.7 Saving a Workspace

You can save your analysis using the function `save.image`. For example, if you want to save your analysis in a file called `myworkspace.RData`, you should use:

```
> save.image("myworkspace.RData")
```

You can also use the console menus: *File* → *Save Workspace*. Now, you can load your analysis into R, using the function `load`:

```
> load("myworkspace.RData")
```

This is useful if you want to stop one session and continuing on the following day, etc.

3 Installation and Introduction to OneMap

OneMap can be installed by opening R and typing the command

```
> install.packages("onemap")
```

You also can use the console menus: *Packages* → *Install package(s)*. After clicking, a box will pop-up asking you to choose the CRAN mirror. Choose the location nearest to you. Then, another box will pop-up asking you to choose the package you want to install. Select `onemap` then click *OK*. The package will be automatically installed on your computer.

OneMap can also be installed by downloading the appropriate files directly at the CRAN web site and following the instructions given in the section "6.3 Installing Packages" of the "R Installation and Administration" manual (`http://cran.r-project.org/doc/manuals/R-admin.pdf`).

OneMap is comprised by set of functions (listed on Table 1). There are other functions used internally by the software. However, you do not need to use them directly.

After *OneMap* is installed, you can load it with

```
> library(onemap)
```

A list of packages and datasets that are available on your computer can be obtained with

```
> library()
> data()
```

Table 1: *OneMap* functions

Function type	Function name	Function description
Input	read.outcross	Read data from an outcross
	read.mapmaker	Read data from a Mapmaker raw file
Data manipulation	make.seq	Creates a sequence of markers based on objects of other types
	marker.type	Informs the segregation type of genetic markers
	add.marker	Adds markers to a sequence
	drop.marker	Drops markers from a sequence
Genetic mapping	rf.2pts	Estimates recombination fractions (two points)
	group	Assigns markers to linkage groups
	set.map.fun	Defines the default mapping function
	rcd	Orders markers in a sequence using RCD algorithm
	seriation	Orders markers in a sequence using SERIATION algorithm
	record	Orders markers in a sequence using RECORD algorithm
	ug	Orders markers in a sequence using UG algorithm
	compare	Compares all possible orders of markers in a sequence
	try.seq	Tries to map a marker into a given linkage group
	order.seq	Automates map construction through "compare" and "try.seq" functions
	ripple.seq	Compares alternative orders for a map and displays the plausible ones
	map	Constructs a multipoint linkage map for a sequence in a given order
	rf.graph.table	Plots a pairwise recombination fraction and LOD matrix using a color scale.
	draw.map	Draws a genetic map
Output	write.map	Writes a genetic map to a file to be used in other softwares (only for backcrosses, F_2 and RILs)
Defunct	def.rf.3pts	Estimates recombination fractions (three points method)

4 Outcrossing populations

The following example is intended to show the usage of *OneMap* functions for linkage mapping in **outcrossing** (non-inbred) species. With basic knowledge of R syntax, one should have no

big problems using it. If you are not familiar with R software, we recommend reading Section 2. It is assumed that the user is running Windows™. Hopefully these examples will be clear enough to help any user to understand its functionality and start using it.

1. Start R by double-clicking its icon.
2. Load *OneMap*, after installing it:

   ```
   > library(onemap)
   ```

3. To save your project anytime, type:

   ```
   > save.image("C:/.../yourfile.RData")
   ```

   or access the toolbar File → Save Workspace.

4.1 Creating the data file

This step might be quite difficult, since the data file is not very simple and many 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 it.

Basically, the input file is a text file, where the first line indicates the number of individuals and the number of markers. 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 (Table 2)

Actually, 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 table. The letter and the number before the dot indicate the segregation type (i.e., 1:1:1:1, 1:2:1, 3:1 or 1:1), while the number after the dot 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 on parents and progeny.

Table 2: Notation used to identify markers and genotypes

			Parent						Offspring	
		crosstype	Cross			Observed bands			Observed bands	Segregation
A		1	ab	×	cd	ab	×	cd	ac, ad, bc, bd	1:1:1:1
		2	ab	×	ac	ab	×	ac	a, ac, ba, bc	1:1:1:1
		3	ab	×	co	ab	×	c	ac, a, bc, b	1:1:1:1
		4	ao	×	bo	a	×	b	ab, a, b, o	1:1:1:1
B	B_1	5	ab	×	ao	ab	×	a	$ab, 2a, b$	1:2:1
	B_2	6	ao	×	ab	a	×	ab	$ab, 2a, b$	1:2:1
	B_3	7	ab	×	ab	ab	×	ab	$a, 2ab, b$	1:2:1
C		8	ao	×	ao	a	×	a	$3a, o$	3:1
D	D_1	9	ab	×	cc	ab	×	c	ac, bc	1:1
		10	ab	×	aa	ab	×	a	a, ab	1:1
		11	ab	×	oo	ab	×	o	a, b	1:1
		12	bo	×	aa	b	×	a	ab, a	1:1
		13	ao	×	oo	a	×	o	a, o	1:1
	D_2	14	cc	×	ab	c	×	ab	ac, bc	1:1
		15	aa	×	ab	a	×	ab	a, ab	1:1
		16	oo	×	ab	o	×	ab	a, b	1:1
		17	aa	×	bo	a	×	b	ab, a	1:1
		18	oo	×	ao	o	×	a	a, o	1:1

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 a comma separates the information for each individual.

Here is an example of such file for 10 individuals and 5 markers:

```
10 5
*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 that this is a common mistake made by users, so be careful.

The input file must be saved in text format, with extensions like ".txt". It is a good idea to open the text file called "example.out.txt" (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

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

4.2 Importing data

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

   ```
   > example.out<- read.outcross("C:/workingdirectory","example.out.txt")
   ```

   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 `example.out` was created. If you leave the argument `dir` blank, the file will be read from your *working directory*. To set a working directory, see Section 2.5.

2. 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:

   ```
   > example.out<- read.outcross(file="example.out.txt")
   ```

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

3. Because this particular data set is distributed along with the package, as an alternative you can load it typing

   ```
   > data(example.out)
   ```

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

   ```
   > example.out
   ```

   you will see some information about the object.

4.3 Estimating two-point recombination fractions

1. To start the analysis, the first step is estimating the recombination fraction between all pairs of markers, using two-point tests:

   ```
   > twopts <- rf.2pts(example.out)
   ```

   The function `rf.2pts` uses as default values of LOD Score 3 and maximum recombination fraction 0.50.

2. Different values for the criteria can be chosen using:

   ```
   > twopts <- rf.2pts(example.out, LOD=3, max.rf=0.4)
   ```

3. Although two-point tests were implemented in C language, which is much faster than R, this step can take quite some time, depending on the number of markers involved and their segregation type, since all combinations will be estimated and tested. Besides, the results use a lot of memory and a rather powerful computer is needed. For example, the analysis of a real data set with 1741 markers (segregating 3:1 and 1:1) took 2.8 hours, running under Windows™ on a Pentium® 4 CPU 3.00 GHz with 1 GB RAM memory.

4. 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:

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

   ```
   > print(twopts, "M1", "M3")
   ```

   Each line corresponds to a possible linkage phase. 1 denotes coupling phase in both parents (CC), 2 and 3 denote coupling phase in parent 1 and 2, respectively, and repulsion in the other (CR and RC), and 4 denotes repulsion phase in both parents (RR). Theta is the maximum likelihood estimate of the recombination fraction, with its LOD Scores.

4.4 Assigning markers to linkage groups

1. 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:

```
> mark.all <- make.seq(twopts, "all")
```

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. In this example, the argument `"all"` indicates that all markers will be analyzed. If one wants to use only a subset of markers, say `M1` and `M2`, the option will be `c(1,2)`. These numbers refer to the lines where markers are located on the data file. Since 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.all)
```

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

   ```
   > LGs <- group(mark.all)
   ```

   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 `twopts` are used.

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, i.e., 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):

   ```
   > print(LGs, detailed=FALSE)
   ```

4. 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.all, LOD=6)
   > LGs
   ```

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

```
> LGs <- group(mark.all, LOD=3, max.rf=0.4)
> LGs
```

4.5 Genetic mapping of linkage group 3

1. 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 through 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")
   ```

   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.

2. If you type

   ```
   > LG3
   ```

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

3. 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, all algorithms provided the same results (results not showed).

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

   ```
   > LG3.comp <- compare(LG3)
   ```

   This order step can take some time, depending on marker types in the linkage group. In the example, `LG3` contains one marker of type D1 and one of type D2, besides one marker segregating in 3:1 fashion (type C). Thus, although the number of possible orders is relatively small (60), for each order there are various possible combinations of linkage phases. Also, the convergence of the EM algorithm takes considerably more time, since markers of type C are not very informative.

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

5. 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, since two point analysis also provided information about linkage phases, this information was taken into consideration in the `compare` function, reducing the number of combinations to be evaluated. If at least one linkage phase has LOD equals to 0.005 in the two point analysis, we assumed that this phase is very unlikely and so do not need to be evaluated in the multipoint procedure used by compare. We did extensive simulations that 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, i.e., 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 -2.43. 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 two-point analysis, not all combinations are tested and that is the reason why the number of Nested LOD is different for each order.

6. 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 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 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)
   ```

   will have the same effect.

7. To see the final map type

   ```
   > LG3.final
   ```

   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 manner in which alleles are arranged in the chromosomes, given the estimated linkage phase. 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).

4.6 Genetic mapping of linkage group 2

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

1. Again, "extract" that group from the object `LGs`:

   ```
   > LG2 <- make.seq(LGs, 2)
   > LG2
   ```

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

2. First, use `rcd` to get a preliminary order estimate:

   ```
   > LG2.rcd <- rcd(LG2)
   > LG2.rcd
   ```

3. Use the `marker.type` function to check the segregation types of all markers in this group:

   ```
   > marker.type(LG2)
   ```

4. Based on their segregation types and distribution on the preliminary map, markers M4, M23, M19, M20 and M24 are the most informative ones (type A is the 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(4,23,19,20,24))
   > LG2.comp <- compare(LG2.init)

   > LG2.comp
   ```

   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 will be in the sequence.

5. Select the best order:

   ```
   > LG2.frame <- make.seq(LG2.comp)
   ```

6. Next, let us try to map the remaining markers, one at a time. Since there are more markers of type D1 than D2, the latter will be tried later. Starting with `M9`:

```
> LG2.extend <- try.seq(LG2.frame,9)
> LG2.extend
```

Based on the LOD Scores, marker M9 is probably better located between markers M23 and M24. However, the "*" symbol indicates that more than one linkage phase is possible. Detailed results can be seen with

```
> print(LG2.extend,5)
```

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

Also, we can obtain some useful diagnostic graphics using the argument `draw.try=TRUE` when using function `try.seq`:

```
> LG2.extend <- try.seq(LG2.frame, 9, draw.try=TRUE)
```

The top figure represents the new genetic map obtained with the insertion of marker 9 between markers M23 and M24 (most likely one). The left bottom figure represents the frame map M24 - M23 - M4 - M19 - M20 on x-axis and the LOD Scores of the linkage maps obtained with the insertion of marker 9 at the beginning, between markers and at the end of the frame map. The red triangle indicates the most likely position, where the marker 9 it is supposed to be placed. The right bottom figure is the recombination fraction matrix based on a color scale using the function `rf.graph.table`. See Section 4.9 for details. The diagnostic graphics show an almost monotonic recombination fraction matrix (the values are bigger as their distance from diagonal increases). This pattern is typical of ordered linkage groups. We can see that the position between markers `M23` and `M24` is the most likely one for positioning marker `M9`

7. Finally, the best order can be obtained with:

   ```
   > LG2.frame <- make.seq(LG2.extend,5,1)
   ```

   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).

   It should be pointed out that the framework created by the function `compare` with (`M20`, `M4`, `M19`, `M23` and `M24`) could be in reverse order (`M24`, `M23`, `M19`, `M4` and `M20`) and still be

the same map. Thus, the positioning of markers by command `try.seq` can be different in your computer. For example, here, marker `M9` was better placed in position 5, however if you obtain a reverse order, marker `M9` would be better placed in position 2. In both cases the best position is between markers `M24` and `M23`.

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

```
> LG2.extend <- try.seq(LG2.frame,29)
> LG2.frame <- make.seq(LG2.extend,7)
> LG2.extend <- try.seq(LG2.frame,27)
> LG2.frame <- make.seq(LG2.extend,1)
> LG2.extend <- try.seq(LG2.frame, 16)
> LG2.frame <- make.seq(LG2.extend,2)
> LG2.extend <- try.seq(LG2.frame,21)
> LG2.final <- make.seq(LG2.extend,6)
```

8. The process of adding markers sequentially can be automated with the use of function `order.seq`.

   ```
   > LG2.ord <- order.seq(LG2, n.init=5, THRES=3, draw.try=TRUE, wait=1)
   ```

   Basically, this function automates what the `try.seq` function does, using some pre-defined 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; `draw.try=TRUE` will display a diagnostic graphic for each `try.seq` step; `wait=1` indicates the minimum time interval in seconds to display the diagnostic graphic.

   NOTE: Although very useful, this function can be misleading, specially if there are not many fully informative markers, so use it carefully. Results can vary for each running, of course.

9. Check the final order:

   ```
   > LG2.ord
   ```

   Note that markers `21` and `29` 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.

10. To get the safe order (i.e. without markers 21 and 29), 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
    ```

    Notice that, for this linkage group, the "forced" map obtained with `order.seq` is the same as that obtained with `compare` plus `try.seq`, but *this is not always the case*.

11. The `order.seq` function can also performs two rounds of the `try.seq` algorithms, first using `THRES` and then `THRES - 1` as threshold. This generally results in safe orders with more markers mapped, but may take longer to run. To do this use the `touchdown` options:

    ```
    > LG2.ord <- order.seq(LG2, n.init=5, THRES=3, touchdown=TRUE)
    > LG2.ord
    ```

    For this particular sequence, the `touchdown` step could not map any additional marker, but this depends on the specific dataset.

12. Finally, to check for alternative orders (since we did not use exhaustive search), use the `ripple.seq` function:

    ```
    > ripple.seq(LG2.all, ws=4, LOD=3)
    ```

    We should do this to any of the orders we found, either using `try.seq` or `order.seq`. Here, we choose `LG2.all` only for didactic purpose. The second argument, `ws = 4`, means that subsets (windows) of four markers will be permutated 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 will be printed.

    The output shows sequences of four numbers, since `ws = 4`. They will be followed by an `OK`, if there is no alternative orders with LOD Scores smaller than `LOD = 3` in absolute value, or by a list of alternative orders. On the example, just the last sequence showed an

alternative order with LOD smaller than `LOD=3` (2.06, in absolute value). However, the best order was the previous one (LOD=0.00).

If there was an alternative order most likely than the original, one should check the difference between these orders (and linkage phases) and change it using, for example, the function `drop.marker` (see Section 4.8) and `seq.try` or typing the new order. You can use `$seq.num` and `$seq.phases` after the name of the sequence (for example, `LG2.all$seq.num` and `LG2.all$seq.phases`) to obtain the original order and linkage phases, make the necessary changes (by copying and paste) and then use the function `map` (see Section 4.8) to reestimate the genetic map for the new order.

Here, the function `ripple.seq` showed that the final order obtained is indeed the best for this linkage group. The map can then be printed using

```
> LG2.all
```

4.7 Genetic mapping of linkage group 1

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

   ```
   > LG1 <- make.seq(LGs, 1)
   ```

2. Construct the linkage map, by automatic using try algorithm:

   ```
   > LG1.ord <- order.seq(LG1, n.init=6, touchdown=TRUE)
   > LG1.ord
   ```

   Notice that the second round of `try.seq` added markers `M5` and `M25`.

3. Now, get the order with all markers:

   ```
   > (LG1.final <- make.seq(LG1.ord,"force"))
   ```

4. Check the final map:

   ```
   > ripple.seq(LG1.final)
   ```

   No better order was observed.

5. Print it

   ```
   > LG1.final
   ```

6. As an option, different algorithms to order markers should 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.

4.8 Map estimation for an arbitrary order

1. 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))
```

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

2. 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))
```

3. 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))
```

   Removing markers 3, 4, 5, 12 and 30 from `any.seq`:

```
> (any.seq <- drop.marker(any.seq, c(3,4,5,12,30)))
```

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

4.9 Plotting the recombination fraction matrix

For a given sequence, it is possible to plot the recombination fraction matrix and LOD Scores based on a color scale using the function `rf.graph.table`. This matrix can be useful to make some diagnostics about the map.

1. For example, using the function `group` with `LOD=2.5`:

   ```
   > (LGs <- group(mark.all, LOD=2.5))
   ```

   Due to the small value used for the LOD Score (2.5, not adequate and resulting in false positives), markers from different groups were placed together.

2. Ordering markers (results not shown):

   ```
   > LG.err<-make.seq(LGs, 2)
   > LG.err.ord<-order.seq(LG.err)
   ```

   The map using option "force":

   ```
   > (LG.err.map<-make.seq(LG.err.ord, "force"))
   ```

3. A careful examination of the results shows that there are problems on the map. This can be done by plotting the recombination fraction matrix:

   ```
   > rf.graph.table(LG.err.map)
   ```

   The recombination fractions are plotted below the diagonal and the LOD Scores are plotted above the diagonal. The color scale varies from red (small distances or big LODs) to dark blue. This color scale follows the "rainbow" color palette with `start` argument equals to 0 and `end` argument equals to 0.65. White cells indicate for combinations of markers whose recombination fractions cannot be estimated (D_1 and D_2).

   Clicking on the cell corresponding to two markers (off secondary diagonal), you can see some information about them. For example, clicking on the cell corresponding to markers `M4` and `M19` you can see their names, types (`A.4` and `B1.5`), recombination fraction (`rf=0.02281`) and LOD Scores for each possible linkage phase. Clicking in a cell on the diagonal, some information about the corresponding marker is shown, including percent of missing data. We think this is quite useful in helping to interpret the results.

Looking at the matrix, it is possible to see two groups: one with markers from `LG2` (`M27`, `M16`, `M20`, `M4`, `M19`, `M21`, `M23`, `M9`, `M24`, and `M29`) and other with markers from `LG3` (`M22`, `M7`, `M18`, `M8` and `M13`). There is a gap between markers `M22` and `M29` (`rf=0.4594`). At this position, the group should be divided, that is, a higher LOD Score should be used. Notice that these two groups were placed together due to a false linkage (false positive) detected between markers `M4` and `M22` (LOD Score 2.9) due to the fact of not using appropriated LOD threshold (more conservative value).

The `rf.graph.table` can also be used to check the order of markers based on the monotonicity of the matrix, i.e. as we get away from the secondary diagonal, the recombination fraction values should increase. For another example of function `rf.graph.table`, see Section 5.9.

4.10 Drawing the genetic map

1. Once all linkage groups were obtained, we can draw a simple map using the function `draw.map`. We can draw a genetic map for all linkage groups:

```
> maps<-list(LG1.final, LG2.final, LG3.final)
> draw.map(maps, names= TRUE, grid=TRUE, cex.mrk=0.7)
```

2. For a specific linkage group:

```
> draw.map(LG1.final, names= TRUE, grid=TRUE, cex.mrk=0.7)
```

   It is obvious that function `draw.maps` draws a very simple graphic representation of the genetic map. But once the distances and the linkage phases are estimated, better map figures can be drawn by the user using any appropriate software. There are several free softwares that can be used, such as MapChart (Voorrips, 2002).

5 F_2 example

Starting in version 2.0-0, *OneMap* can also deal with inbred-based populations (F_2, backcrosses and RILs). In this section we explain how to proceed the analysis in an F_2 population. This procedure can be used for backcrosses and RILs as well. If you are not familiar with R software, we recommend the reading of Section 2. Most of the steps for constructing an F_2 genetic map

are the same as those used in the outcrossing example, thus details can be obtained on Section 4, However, this section could be read alone.

5.1 Creating the data file

For F_2, backcrosses and RILs we used exactly the same raw file used by MAPMAKER/EXP (Lander et al., 1987). Therefore, one should have no difficult in using data sets already available for MAPMAKER/EXP. This raw file can contain phenotypic information in the same way as a MAPMAKER/EXP file, but this will not be used during the map construction. This file, combined with the map file produced by *OneMap*, can be readily used for QTL mapping using *R/qtl* (Broman et al., 2008) or *QTL Cartographer* (Wang et al., 2010), among others. Here, we briefly present how to set up this data file. For more detailed information see the MAPMAKER/EXP manual (Lincon et al., 1993).

The first line of your data file should be:

```
data type xxxx
```

where `xxxx` is one of the following data types:

`f2 backcross`	for backcrosses
`f2 intercross`	for F_2
`ri self`	for RILs by selfing
`ri sib`	for RILs by sib mating

The second line should contain the number of individuals on the progeny, the number of markers and the number of quantitative traits. Then, the genotype information is included for each marker. The character "*" indicates the beginning of information of a marker, followed by the marker name. The codification for genotypes is the following:

- `A`: homozygous for allele A (from parental 1 - AA)
- `B`: homozygous for allele B (from parental 2 - BB)
- `H`: heterozygous carrying both alleles (AB)
- `C`: Not homozygous for allele A (Not AA)
- `D`: Not homozygous for allele B (Not BB)
- `-`: Missing data for the individual at this marker

The "symbols" option, used in MAPMAKER/EXP files, is also accepted (please, see the manual).

The quantitative trait data should come after the genotypic data and has a similar format, except the trait values for each individual must be separate by at least one space, a tab or a line break. A dash (-) indicates missing data. Here is an example of such file for an F_2 population, 10 individuals, 5 markers and 2 quantitative traits:

```
data type f2 intercross
10 5 2

*M1 A B H H A - B A A B
*M2 C - C C C - - C C A
*M3 D B D D - - B D D B
*M4 C C C - A C C A A C
*M5 C C C C C C C C C C

*weight 10.2 - 9.4 11.3 11.9 8.9 - 11.2 7.8 8.1
*length 1.7 2.1 - 1.8 2.0 1.0 - 1.7 1.0 1.1
```

This file must be saved in plain text format using a simple text editor such as *notepad*. Historically, MAPMAKER/EXP uses the ".raw" extension for this file, however, you can use other extensions, for example, ".txt". If you want to see an example how this file should be, you can open "fake.bc.onemap.raw" and "fake.f2.onemap.raw", both available with *OneMap* and saved in the directory you installed it (use `system.file(package="onemap")` to see where it is).

. Now, let us load *OneMap*:

1. Start R by double-clicking its icon.

2. Load *OneMap* (after installing it; for details see Sections 2.4 and 3):

   ```
   > library(onemap)
   ```

3. To save your project anytime, type:

   ```
   > save.image("C:/.../yourfile.RData")
   ```

   specifying where to have and naming the file, or access the toolbar File → Save Workspace.

5.2 Importing data

1. Once you created your data file, you can use the function `read.mapmaker` to import it to *OneMap*.

```
> fake.f2.onemap <- read.mapmaker(dir="C:/workingdirectory",
+                                 file="your_data_file.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 `fake.f2.onemap` was created. Notice that if you leave the argument `dir` blank, the file will be read from your *working directory*. To set a working directory, see Section 2.5.

2. For this example, we will use a simulated data set from an F_2 population which is distributed along with the *OneMap* package. Since this particular data set is distributed along with the package, you can load it typing

```
> data(fake.f2.onemap)
> fake.f2.onemap
```

   The data consists in a sample of 200 individuals genotyped for 66 markers (36 co-dominant (AA, AB or BB), 15 dominant (Not AA or AA) and 15 dominant (Not BB or BB) with 15% of missing data. You also can see that there is phenotypic information on the data set.

5.3 Estimating two-point recombination fractions

1. Let us start the analysis estimating the recombination fraction between all pairs of markers using two-point tests:

```
> twopts.f2 <- rf.2pts(fake.f2.onemap)
```

   There are two optional arguments in function `rf.2pts`: `LOD` and `max.rf` which indicate the minimum LOD Score and the maximum recombination fraction to declare linkage (defaults to 3.0 and 0.5).

2. If you want to see the results for any given markers, say `M12` and `M42`, use:

```
> print(twopts.f2, "M12", "M42")
```

5.4 Assigning markers to linkage groups

1. To assign markers to linkage groups, first use the function `make.seq` to create a sequence with all markers:

   ```
   > mark.all.f2 <- make.seq(twopts.f2, "all")
   ```

   The function `make.seq` is used to create sequences from objects of several kinds. Here, the first argument is of class `rf.2pts` and the second argument specifies which markers one wants to use (`"all"` indicates that all markers will be analyzed). To subset markers, say `M1`, `M3` and `M7`, use:

   ```
   > mrk.subset<-make.seq(twopts.f2, c(1,3,7))
   ```

2. You can assign markers to linkage groups using the function `group`:

   ```
   > (LGs.f2 <- group(mark.all.f2, LOD=3, max.rf=0.5))
   ```

   The arguments `LOD` and `max.rf` define thresholds to be used when assigning markers to linkage groups. If none provided (default), criteria previously defined for the object `twopts` are used. We can see that the markers were assigned to three linkage groups with 27, 16 and 23 markers, with no unlinked markers.

5.5 Genetic mapping of linkage group 2

After the assignment of markers to linkage groups, the next step is to order the markers within each group.

1. First, let us choose the mapping function used to display the genetic map. We 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")
   ```

2. To define which linkage group will be mapped, we must "extract" it from the object of class `group`. Let us extract the group 2 using:

```
> LG2.f2 <- make.seq(LGs.f2, 2)
```

The first argument is an object of class `group` and the second is a number indicating which linkage group will be extracted. In this case, the object `LGs.f2`, generated by function `group`, is of class `group`, showing this function can handle different classes of objects.

3. If you type

   ```
   > LG2.f2
   ```

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

4. 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):

   ```
   > LG2.ser.f2 <- seriation(LG2.f2)
   > LG2.rcd.f2 <- rcd(LG2.f2)
   > LG2.rec.f2 <- record(LG2.f2)
   > LG2.ug.f2 <- ug(LG2.f2)
   ```

   For this particular data set, the algorithms provided different results (results not show here). For an evaluation and comparison of these methods, see Mollinari et al. (2009).

   Now, let us use a multipoint approach to order markers within group 2. We could use the following: for each possible order of this group, we calculate the multipoint likelihood, and then compare all of them, choosing the most likely one (high likelihood). For a moderate number of markers (up to 10 or 11), this is feasible. This procedure is implemented in the function `compare`. Although feasible, with up to 7 markers the function `compare` could take a very long time, depending on the data set and computational resources used. A detailed use of this function can be seen in Section 4.5. It is important to say that for F_2 populations, we do not need to estimate the linkage phases, therefore, we can use a slightly large number of markers in function `compare`. However, for 16 markers, which is the number of markers in group 2, the use of function `compare` is unfeasible, and we should use another approach.

Thus we will apply the same procedure used in Section 4.6. We will choose a moderate number of markers, say 6, to create a framework using the function `compare` and then positioning the remaining markers using the function `try.seq`. The way we choose these markers in inbred-based populations (F_2, backcrosses and RILs) is somewhat different from outcrossing populations.

We recommend two methods: i) randomly choose a number of markers and calculate the multipoint likelihood of all possible orders (using the function `compare`). If the LOD Score of the second best order is greater than a threshold, say 3, then take the best order to proceed with the next step. If not, repeat the procedure. ii) use some two-point based algorithm to construct a map; then, take equally spaced markers from this map. Then, create a framework of ordered markers using the function `compare`. Next, try to map the remaining markers, one at a time, beginning with co-dominants (most informative ones), then add the dominants. You can do this procedure manually, like shown in Section 4.6; this procedure is also automated in function `order.seq` which we will use here for the latter procedure:

```
> LG2.f2.ord <- order.seq(input.seq=LG2.f2, n.init = 5,
+                         subset.search = "twopt",
+                         twopt.alg = "rcd", THRES = 3,
+                         draw.try = TRUE, wait = 1)
>
```

The first argument is an object of class sequence. `n.init = 5` means that five markers will be used in the `compare` step. The argument `subset.search = "twopt"` indicates that these five markers should be chosen by using a two point method, which will be Rapid Chain Delineation, as indicated by the argument `twopt.alg = "rcd"`. `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. `draw.try=TRUE` will display a diagnostic graphic for each `try.seq` step (see Section 4.6). `wait=1` indicates the minimum time interval in seconds to display the diagnostic graphic. NOTE: Although very useful, this function can be misleading, specially if there are a considerable amount of missing data and dominant markers, use it carefully.

5. Check the final order:

```
> LG2.f2.ord
```

Note that markers 11 and 45 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.

6. To get the "safe" order, use

```
> LG2.f2.safe <- make.seq(LG2.f2.ord,"safe")
```

and to get the order with all markers (i.e. including the ones not mapped to a single position), use:

```
> (LG2.f2.all <- make.seq(LG2.f2.ord,"force"))
```

Which places markers 11 and 45 into their most likely positions (between markers 2 and 43 and 32 and 54, respectively).

7. The order.seq function can perform two rounds of the try.seq step, first using THRES and then THRES - 1 as threshold. This generally results in safe orders with more markers mapped, but takes longer to run. To do this,type:

```
> LG2.f2.ord <- order.seq(input.seq=LG2.f2, n.init = 5,
+                         subset.search = "twopt",
+                         twopt.alg = "rcd", THRES = 3,
+                         draw.try = TRUE, wait = 1,
+                         touchdown=TRUE)
```

The output is too big to be included here, so please try to see what happened. In short, for this particular sequence, the touchdown step could not map any additional marker, but this depends on the dataset. Since there is no other reason to change position of markers 11 and 45 (e.g. biological information), let us use the order with all markers as suggested by the function order.seq:

```
> (LG2.f2.final<-make.seq(LG2.f2.ord, "force"))
```

8. Finally, to check for alternative orders, use the ripple.seq function:

```
> ripple.seq(LG2.f2.final, ws=5, LOD=3)
```

The second argument, `ws = 5`, means that subsets (windows) of five markers will be permutated sequentially (5! orders for each window), to search for other plausible orders. The `LOD` argument means that only orders with LOD Score smaller than 3 will be printed.

The output shows sequences of four numbers, since `ws = 5`. They can be followed by an `OK`, if there is no alternative orders with LOD Scores smaller than `LOD = 3` in absolute value, or by a list of alternative orders.

On the example, the six first sequences showed alternative orders with LOD smaller than `LOD=3`. However, the best order was that obtained with the `order.seq` function (LOD=0.00). If there was an alternative order most likely than the original, one should check the difference between these orders and if necessary change it using, for example, the function `drop.marker` (see Section 5.8) and `seq.try`, or simple typing the new order.Use `LG2.f2.final$seq.num` to obtain the original order; then make the necessary changes (by copying and paste) and use the function `map` (see Section 5.8) to reestimate the genetic map for the new order.

9. The `ripple.seq` command showed that the final order obtained is indeed the best for this linkage group. The map can then be printed using

```
> LG2.f2.final
```

5.6 Genetic mapping of linkage group 1

1. Let us analyze linkage group 1. Extract markers from object `LGs`:

```
> LG1.f2 <- make.seq(LGs.f2, 1)
```

2. Construct the linkage map, by automatic usage of `try` algorithm:

```
> LG1.f2.ord <- order.seq(input.seq=LG1.f2, n.init = 5,
+                         subset.search = "twopt",
+                         twopt.alg = "rcd", THRES = 3,
+                         draw.try = TRUE, wait = 1,
+                         touchdown=TRUE)
```

The second round of `try.seq` added markers `M9`, `M44` and `M48` (try it; results not shown).

3. Get the order with all markers:

```
> (LG1.f2.final <- make.seq(LG1.f2.ord,"force"))
```

4. Check the final map (results not shown):

```
> ripple.seq(ws=5, LG1.f2.final)
```

No better order was observed (please, try it to see).

5. Print it

```
> LG1.f2.final
```

5.7 Genetic mapping of linkage group 3

1. Extract markers from object `LGs.f2`:

```
> LG3.f2 <- make.seq(LGs.f2, 3)
```

2. Construct the linkage map, by automatic usage of try algorithm and drawing some useful graphics (not shown):

```
> LG3.f2.ord <- order.seq(input.seq=LG3.f2, n.init = 5,
+                         subset.search = "twopt",
+                         twopt.alg = "rcd", THRES = 3,
+                         draw.try = TRUE, wait = 1,
+                         touchdown=TRUE)
```

We can see that in the second round of `try.seq` marker `M56` was added (please, try it). A careful examination of the graphics can be a good source of information about how markers where placed. For more details about how to interpret it, see Section 4.6

3. Now, get the order with all markers:

```
> (LG3.f2.final <- make.seq(LG3.f2.ord,"force"))
```

4. Check the final map:

```
> ripple.seq(ws=5, LG3.f2.final)
```

No better alternative order was observed.

5. Print it

```
> LG3.f2.final
```

5.8 Map estimation for an arbitrary order

1. If you have some information about the order of the markers, for example, from a previous published paper, you can define a sequence of those markers (using the function `make.seq`) and then use the function `map` to estimate the genetic map. For example, for markers `M47`, `M38`, `M59`, `M16`, `M62`, `M21`, `M20`, `M48` and `M22`, in this order, use:

```
> LG3seq.f2 <- make.seq(twopts.f2,c(47,38,59,16,62,21,20,48,22))
> (LG3seq.f2.map <- map(LG3seq.f2))
```

   To see relation between marker names and numbers, use

```
>  marker.type(LG3seq.f2.map)
```

2. 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 `M18`, `M56` and `50` in the end of `LG3seq.f2.map`

```
> (LG3seq.f2.map <- add.marker(LG3seq.f2.map, c(18,56,50)))
```

   Removing markers `M59` and `21` from `LG3seq.f2.map`:

```
> (LG3seq.f2.map <- drop.marker(LG3seq.f2.map, c(59,21)))
```

5.9 Plotting the recombination fraction matrix

It is possible to plot the recombination fraction matrix and LOD Scores based on a color scale using the function `rf.graph.table`. This matrix can be useful to make some diagnostics about the map.

1. Let us place `M38` in the end of linkage group 3 (wrong position):

```
> temp.seq<-drop.marker(LG3.f2.final, 38)
```

```
> (temp.seq<-add.marker(temp.seq, 38))
> (LG3.f2.wrong<-map(temp.seq))
```

   Examining the results, we can see there is a big gap in the end of linkage group 3 (between markers `M50` and `M38` as expected.

2. Now let us plot the recombination fraction matrix:

```
> rf.graph.table(LG3.f2.wrong)
```

The recombination fractions are plotted under the diagonal and the LOD Scores are plotted upper the diagonal. The color scale varies from red (small distances big LODs) to dark blue. Clicking on the cell corresponding to two markers, you can see some information about them. For example, clicking on the cell corresponding to markers `M47` and `M19` you can see their names, types (co-dominant and dominant), recombination fraction (rf = 0.07323) and LOD Score (LOD = 23). Clicking in a cell on the diagonal, some information about the corresponding marker is shown, including percentage of missing data.

We clearly see a different pattern for marker `M38`. The blue cell, corresponding to markers `M50` and `M38`, indicates a big recombination fraction between these markers as seen before (by clicking, rf = 0.4049). Moreover, we can see a group of red cells corresponding to marker `M38` and markers `M59`, `M49`, `M39` and `M19`. This pattern indicates small recombination fractions between marker `M38` and other markers. Thus `M38` is suppose to be close to them on the map.

3. Since we have enough evidence that marker `M38` is misplaced, let us drop this marker and try to position it using the function `try.seq`:

```
> temp.seq <- drop.marker(LG3.f2.wrong,38)
> temp.map <- map(temp.seq)
> temp.try <- try.seq(temp.map, 38, draw.try=TRUE)
```

We can see that the most likely position for marker `M38` is between markers `M39` and `M49` (position 4). The patterns on the color matrix are better now. Therefore:

```
> (LG3.f2.final<-make.seq(temp.try, 4))
```

4. For another example of using function `rf.graph.table`, see Section 4.9.

5.10 Drawing the genetic map

1. We can draw a genetic map for all linkage groups using the function `draw.map`. First we have to create a list of ordered linkage groups:

```
> maps.list<-list(LG1.f2.final, LG2.f2.final, LG3.f2.final)
```

Then use it in function `draw.map`:

```
> draw.map(maps.list, names= TRUE, grid=TRUE, cex.mrk=0.7)
```

2. We also can draw a map for a specific linkage group:

```
> draw.map(LG1.f2.final, names= TRUE, grid=TRUE, cex.mrk=0.7)
```

Function `draw.map` draws a very simple graphic representation of the genetic map. But, once the distances and the linkage phases are estimated, better map figures can be drawn by the user using any appropriate software. Also, there are several free softwares that can be used, such as MapChart (Voorrips, 2002).

5.11 Exporting data to R/qtl and QTL Cartographer

Possibly one of the most important applications for a genetic map is its use in QTL mapping studies. In populations such as RILs, F_2 and backcrosses, there are a lot of softwares for doing this analysis. Here, we illustrate how to export the genetic map from *OneMap* to the widely used and excellent packages *R/qtl* (Broman et al., 2008) and to *QTL Cartographer* (Wang et al., 2010).

1. Using the function `write.map`, let us export the list `maps.list`, defined in previous section, to a file named `"fake.f2.onemap.map"`:

```
> write.map(maps.list, "fake.f2.onemap.map")
```

Notice that the file will be written on the *working directory*, unless specified by the second argument. To set a working directory, see Section 2.5.

2. Now, let us install the *R/qtl* package:

```
> install.packages("qtl")
```

Choose the nearest server location and proceed with the installation. Then, load *R/qtl*:

```
> library("qtl")
```

3. To read the data in *R/qtl* we will use the MAPMAKER/EXP format. Two files are needed: the first one is the map file (`"fake.f2.onemap.map"` in our case); the second one is the raw file written in MAPMAKER/EXP style, which was used in the beginning of this example. This file *must* contain phenotypic information. The simulated data `fake.f2.onemap` contains that information. The location of the raw file can be obtained using:

```
> raw.file<-paste(system.file("example",package="onemap"),
+                 "fake.f2.onemap.raw", sep="/")
```

4. Now we can read the data using the *R/qtl* function `read.cross`:

```
> fake.f2.qtl <- read.cross("mm", file=raw.file, mapfile="fake.f2.onemap.map")
```

   The first argument specifies the format of the data. In our case we used "`mm`" which stands for MAPMAKER. The second argument (`file`) indicates the raw file in MAPMAKER/EXP style and the third argument `mapfile` indicates the map file produced by *OneMap*

5. Then we can proceed with the analysis. *R/qtl* has several function to check the map. For example, re-estimating the genetic map within *R/qtl*:

```
> newmap <- est.map(fake.f2.qtl, tol=1e-6, map.function="kosambi")
```

   A comparison of the output of both software can be done with:

```
> plot.map(fake.f2.qtl, newmap)
```

   For each one of the three chromosomes, the left vertical line represents the map estimated by *OneMap* and the right vertical line represents the map estimated by *R/qtl*. The lines linking these two maps indicates the position of the markers. Thus, we can see that the two maps are almost identical.

6. Finally, we can run an interval mapping analysis for these data using the R/qtl function called `scanone` (for details, see *R/qtl* tutorial):

```
> fake.f2.qtl <- calc.genoprob(fake.f2.qtl, step=2)
> out.em <- scanone(fake.f2.qtl, method="em")
> out.hk <- scanone(fake.f2.qtl, method="hk")
> plot(out.em, out.hk, col=c("blue","red"))
```

Here we performed an interval mapping using two methods: mixture models with EM algorithm and Haley-Knott regression. The blue lines indicate the first one and the red lines indicate the second.

7. We can use *R/qtl* to generate *QTL Cartographer* input files.

```
> write.cross(fake.f2.qtl, format="qtlcart", filestem="fake.f2.onemap")
```

Again, the file will be written on the *working directory*, unless you specify differently in argument `filestem`. The files produced this way are ready to be used in *QTL Cartographer*.

6 Final comments

At this point it should be clear that any potential *OneMap* user must have some knowledge about genetic mapping and also the R language, since the analysis is not done with *only one mouse click*. In the future, perhaps a graphical interface will be made available to make this software a lot easier to use.

We do hope that *OneMap* should be useful to any researcher interested in genetic mapping in outcrossing or inbred-based populations. Any suggestions and critics are welcome.

7 References

Adler, J. *R in a Nutshell* A Desktop Quick Reference, 2009.

Broman, K. W., Wu, H., Churchill, G., Sen, S., Yandell, B. *qtl: Tools for analyzing QTL experiments* R package version 1.09-43, 2008. (`http://www.rqtl.org/`)

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.

Garcia, A.A.F., Kido, E.A., Meza, A.N., Souza, H.M.B., Pinto, L.R., Pastina, M.M., Leite, C.S., Silva, J.A.G., Ulian, E.C., Figueira, A. and Souza, A.P. Development of an integrated genetic map of a sugarcane (*Saccharum spp.*) commercial cross, based on a maximum-likelihood approach for estimation of linkage and linkage phases. *Theoretical and Applied Genetics* 112, 298-314, 2006.

Haldane, J. B. S. The combination of linkage values and the calculation of distance between the loci of linked factors. *Journal of Genetics* 8, 299-309, 1919.

Jiang, C. and Zeng, Z.-B. Mapping quantitative trait loci with dominant and missing markers in various crosses from two inbred lines. *Genetica* 101, 47-58, 1997.

Kosambi, D. D. The estimation of map distance from recombination values. *Annuaire of Eugenetics* 12, 172-175, 1944.

Lander, E. S. and Green, P. Construction of multilocus genetic linkage maps in humans. *Proc. Natl. Acad. Sci. USA* 84, 2363-2367, 1987.

Lander, E.S., Green, P., Abrahanson, J., Barlow, A., Daly, M.J., Lincoln, S.E. and Newburg, L. MAPMAKER, An interactive computing package for constructing primary genetic linkage maps of experimental and natural populations. *Genomics* 1, 174-181, 1987.

Lincoln, S. E., Daly, M. J. and Lander, E. S. Constructing genetic linkage maps with MAPMAKER/EXP Version 3.0: a tutorial and reference manual. *A Whitehead Institute for Biomedical Research Technical Report* 1993.

Margarido, G. R. A., Souza, A.P. and Garcia, A. A. F. OneMap: software for genetic mapping in outcrossing species. *Hereditas* 144, 78-79, 2007.

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.

Oliveira, K.M., Pinto, L.R., Marconi, T.G., Margarido, G.R.A., Pastina, M.M., Teixeira, L.H.M., Figueira, A.M., Ulian, E.C., Garcia, A.A.F., Souza, A.P. Functional genetic linkage map on EST-markers for a sugarcane (*Saccharum* spp.) commercial cross. *Molecular Breeding* 20, 189-208, 2007.

Oliveira, E. J., Vieira, M. L. C., Garcia, A. A. F., Munhoz, C. F.,Margarido, G. R.A., Consoli, L., Matta, F. P., Moraes, M. C., Zucchi, M. I., and Fungaro,M. H. P. An Integrated Molecular Map of Yellow Passion Fruit Based on Simultaneous Maximum-likelihood Estimation of Linkage and Linkage Phases *J. Amer. Soc. Hort. Sci.* 133, 35-41, 2008.

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.

Wang S., Basten, C. J. and Zeng Z.-B. Windows QTL Cartographer 2.5. Department of Statistics, North Carolina State University, Raleigh, NC, 2010. (`http://statgen.ncsu.edu/qtlcart/WQTLCart.htm`)

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.

Apendix

8 DEFUNCT - Checking the map with three-point analysis

For historical reasons, three-point analysis are maintained in *OneMap*, but the same (and a lot more) can be done using the multipoint approach.

1. The function `def.rf.3pts` is used as follows:

   ```
   > def.rf.3pts(example, "M18", "M8", "M13")
   ```

   The first argument is the object with the input data, of class `outcross`. Then, three ordered markers are specified.

   In this case, the assignments "`A11`", "`A12`", ..., have similar meanings to those of the two-point analysis: `1` means coupling/coupling, `2` is for coupling/repulsion, `3` is for repulsion/coupling and `4` is for repulsion/repulsion. The first number is the linkage phase between markers $\texttt{M}_{\texttt{i}}$ and $\texttt{M}_{\texttt{i+1}}$, while the second number is the linkage phase between markers $\texttt{M}_{\texttt{i+1}}$ and $\texttt{M}_{\texttt{i+2}}$.

2. Take a look at the default criteria used by this function: LOD = 5, maximum recombination fraction between adjacent markers = 0.35 and maximum recombination fraction between markers on the two ends = 0.55. Considering, for example, three markers A - B - C, in that order, the last criterion indicates the maximum recombination fraction acceptable between markers A and C. These values are used by the software to decide the most probable assignment and can be changed by the user:

   ```
   > def.rf.3pts(example, "M18", "M8", "M13", LOD=10, max.rf=0.4)
   > def.rf.3pts(example, "M18", "M8", "M13", max.rf=0.4, max.nolink=0.60)
   ```

   The arguments `max.rf` and `max.nolink` correspond to the maximum recombination fraction between adjacent markers and the maximum recombination fraction between markers on the two ends, respectively.

3. Do this step for all triplets of markers in linkage group 1:

```
> def.rf.3pts(example, "M18", "M8", "M13")
> def.rf.3pts(example, "M8", "M13", "M7")
> def.rf.3pts(example, "M13", "M7", "M22")
```

This last command line shows that the order `M13 - M7 - M22` is possibly incorrect, and a warning message is displayed. However, the HMM-based analysis use information from every marker in the sequence and, therefore, the order obtained through `compare` is likely to be the best order. Anyway, we had noticed that changing the positions of markers M7 and M22 resulted in an order with LOD Score -0.02, which is very close to zero. This probably happens because M7 is of type D2 and M22 is of type D1.

These three-point analysis were formerly used to check the final linkage map. In this new version, the best way to do this is using the new function `ripple.seq`.