1 Introduction

Multiparent recombinant inbred lines are a novel class of experimental design where the genotypes of the final progeny are mosaics of the genotypes of the 2^n recombinant inbred founder lines. These designs have found recent application in mice (**?**), Arabadopsis (**?**), barley (**?**), maize, rice, tomatoes (**?**) and wheat (**??**).

Existing software able to analyze multiparent designs includes **happy**, **qtl** and **mpMap** (the previous version of **mpMap2**). Packages **happy** and **qtl** are focused on qtl mapping, and do not provide the functionality necessary for map construction. Package **mpMap** provides map construction functionality for multiparent designs, but has significant limitations. Some of these limitations are computational, such as problems analysing the large data sets currently being generated. Others are statistical, such as the inability to model finite generations of selfing and residual heterozygosity.

These limitations motivated the development of **mpMap2**. Our goals for **mpMap2** were

1. To write functionality in C++ where required.
2. To make use of the S4 object system, to enable easier integration of C++ code.
3. To extend the package to biparental and 16-parent populations.
4. To allow for finite generations of selfing, and therefore incorporate heterozygous lines into the map construction process.
5. To allow the user to asses the computational resources required for an analysis.
6. To allow map construction to be performed visually and interactively.
7. To allow the simultaneous use of multiple experiments in the construction of a single map.
8. To use unit testing to speed up development.

2 Experimental designs

We first outline the most general experimental design that we wish to be able to analyse. We have 2^n inbred founder lines which are combined over the first n generations, resulting in a line whoose genetic material is a mosaic of the original 2^n founders. An example of the first n generations for $n = 2$ is given in Figure 1 and for $n = 3$ in Figure 2.

After the first n generations there is some number of generations of random intermating (possibly zero), and some number of generations of inbreeding by selfing (possibly zero). Mathematically it is possible to assume that the number of generations of inbreeding is infinite, and in this case the design is said to be a 2^n-way RIL (**?**). In practice this cannot be achieved, but it might be assumed for the purposes of analysing the population. If the number of generations of selfing is non-zero and the number of generations of inbreeding is assumed to be infinite, the design is said to be a 2^n-way intermated recombinant inbred population (IRIP) (**?**).

One complication is that different orders of the founders in the initial cross result in genetically different individuals at the nth generation. For example, the first three genotypes of $\{A, E\}, \{A, F\}$ and $\{A, G\}$ Figure 3 are possible at the third generation if the initial cross $\{A, B, C, D, E, F, G, H\}$ shown in Figure 2 is used. The remaining three genetoypes are $\{E, F\}, \{E, G\}$ and $\{E, H\}$, and are impossible using this initial cross. However, if founder lines D and E were swapped in the initial cross, then the first three genotypes become impossible, and the last three become possible.

The initial crosses are known as *funnels*. Accounting for symmetries, there are three different funnels for the 4 parent design, 315 different funnels for the 8-parent design and 638512875 different funnels for the 16-parent design. Two cases are mathematically tractable. In the first, only one funnel is ever used. In the second every funnel is chosen at random, which averages out the differences between the funnels.

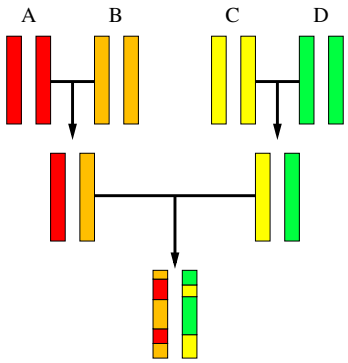

Figure 1: Combining four founders into a single line

3 Pedigrees

3.1 Biparental pedigrees

Package **mpMap2** provides code for the generation of a large number of pedigrees. The two simplest biparental design functions are `rilPedigree(populationSize, selfingGenerations)` which generates a biparental RIL pedigree, and `f2Pedigree(populationSize)` which generates an F2 population. Note that the RIL pedigree requires the specification of the number of generations of selfing, and the populations generated from this pedigree is likely to contain some residual heterozygosity. The pedigree object has a slot `selfing` that controls whether this heterozygosity is modelled in the analysis. The only possible values are `"finite"`, in which case heterozygosity is explicitly modelled, or `"infinite"` in which case the number of generations of selfing is assumed to be inifinite.

Both the F2 and RIL are special cases of a more general biparental design, generated by

```
twoParentPedigree(initialPopulationSize, selfingGenerations,
  nSeeds, intercrossingGenerations).
```

Input `initialPopulationSize` is the number of crosses of the founders, which by assumption are all genetically identical. Input `intercrossingGenerations` is the number of generations

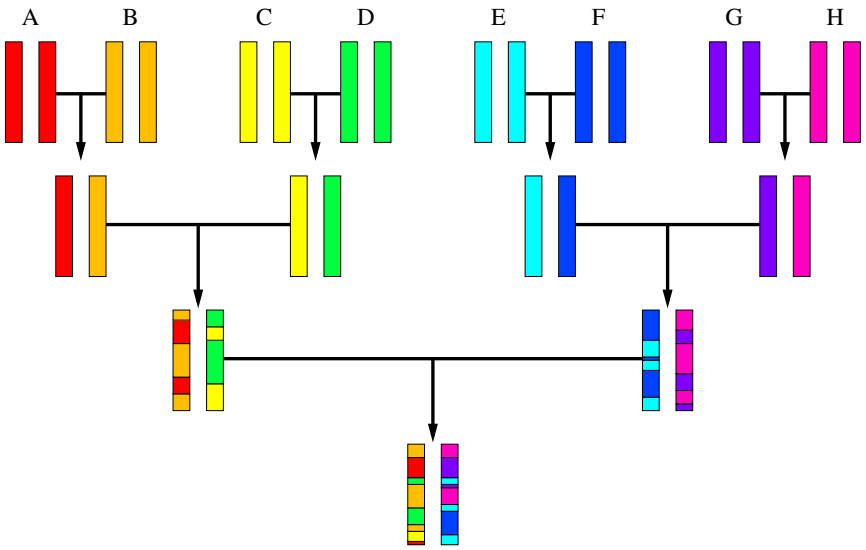

Figure 2: Combining eight founders into a single line

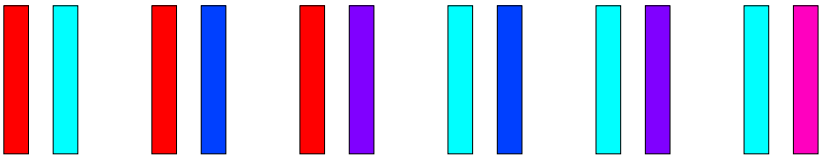

Figure 3: Example genotypes for the third generation of the eight-way cross, which will be possible or impossible, depending on the choice of initial cross.

of random intermating. Input `nSeeds` is the number of independent selfing lines generated from each individual after the random intermating. Input `selfingGenerations` is the number of generations of inbreeding by selfing.

3.2 Four-parent pedigrees

The functions for simulation of four parent RIL designs are `fourParentPedigreeSingleFunnel` and `fourParentPedigreeRandomFunnels`. In the first case only the funnel $\{A, B, C, D\}$ is used. In the second case each individual is drawn from a randomly chosen funnel. The signatures for these functions are

```
fourParentPedigreeRandomFunnels(initialPopulationSize, selfingGenerations,
    nSeeds, intercrossingGenerations),
fourParentPedigreeSingleFunnel(initialPopulationSize, selfingGenerations,
    nSeeds, intercrossingGenerations).
```

3.3 Higher order pedigrees

The functions for generating eight and sixteen parent designs have identical signatures and similar names, except with `four` replaced with `eight` or `sixteen`.

3.4 Inputting pedigrees

Pedigrees from experiments can be input into **mpMap2** using the `pedigree` function.

```
pedigree(lineNames, mother, father, selfing, warnImproperFunnels = TRUE)
```

Input `lineNames` contains the names of the lines in the pedigree. Inputs `mother` and `father` are integer vectors giving the indices of the parents within `lineNames`. Lines with `mother` and `father` set to 0 are the initial lines of the cross, which are assumed to be inbred. The founding lines of the pedigree must appear at the start of the pedigree. Input `selfing` must have value `"finite"` or `"infinite"`, and determines whether any subsequent analysis using this pedigree should assume infinite generations of selfing. If input `warnImproperFunnels` is `TRUE`, then warnings will be generated about lines derived from funnels with repeated founders.

For example, consider the following pedigree with three founder lines.

```
> p <- pedigree(lineNames = c("A", "B", "C", "F1-1", "F1-2", "F1-3", "F1-4",
+     "F2-1", "F2-2", "F3"), mother = c(0, 0, 0, 1, 1, 1, 2, 4, 6, 8),
+     father = c(0, 0, 0, 2, 2, 3, 3, 5, 7, 9), selfing = "finite")
> plot(pedigreeToGraph(p))
```

This pedigree will be recognised as a special case of the eight-parent design where the founders are repeated within a funnel, so pedigrees of this type can be used for map construction.

4 Genetic data and genetic maps

4.1 Simulation

Once a pedigree has been created it can be used to generate genetic data. Note that for simulation of genotypes the pedigree is not restricted to those listed above, and arbitrary pedigrees are allowed. The signature of the simulation function is

`simulateMPCross(map, pedigree, mapFunction, seed).`

Input `map` is a genetic map object in the format used by package **qtl**. Input `pedigree` is a pedigree object and input `mapFunction` is a function that converts centiMorgan distances into recombination fractions. The two suggested values are `haldane` and `kosambi`. Input `seed` is the random seed used for random number generation in the simulation of the genetic data. The output is an S4 object of class `mpcross`.

As an example of the functions provided so far, we simulate from two four-parent designs of 1000 individuals with one generation of intercrossing and four generations of selfing. One set of simulated data uses randomly chosen funnels, while the other uses a single funnel. The same genetic map is used in both cases; there are 2 chromosomes of length 300 cM, each of which has 301 equally spaced markers.

```
> #Generate map
> map <- qtl::sim.map(len = rep(300, 2), n.mar = 301, anchor.tel = TRUE,
+     include.x = FALSE, eq.spacing = TRUE)
> #Generate random funnels pedigree
> pedigreeRF <- fourParentPedigreeRandomFunnels(initialPopulationSize = 1000,
+     nSeeds = 1, intercrossingGenerations = 1, selfingGenerations = 2)
> #Analysis pedigreeRF will assume finite generations of selfing (two)
> selfing(pedigreeRF) <- "finite"
> #Prefix line names with RF
> lineNames(pedigreeRF) <- paste0("RF", lineNames(pedigreeRF))
> #Generate single funnel pedigree
> pedigreeSF <- fourParentPedigreeSingleFunnel(initialPopulationSize = 1000,
+     nSeeds = 1, intercrossingGenerations = 1, selfingGenerations = 2)
> #Analysis pedigreeSF will assume finite generations of selfing (two)
> selfing(pedigreeSF) <- "finite"
> #Prefix line names with SF
> lineNames(pedigreeSF) <- paste0("SF", lineNames(pedigreeSF))
> crossSingleFunnel <- simulateMPCross(map = map, pedigree = pedigreeSF,
+     mapFunction = haldane, seed = 1)
> crossRandomFunnels <- simulateMPCross(map = map, pedigree = pedigreeRF,
+     mapFunction = haldane, seed = 1)
```

The simulated cross object has a single entry named `geneticData`, which is a list of S4 objects of class `geneticData`. This allows `mpcross` objects to contain data from multiple experiments. In the case of `crossSingleFunnel` and `crossRandomFunnels` the list has a single entry. Experiments can be combined using the addition operator to give a single object

containing the data from both. The line names involved in both experiments must be different, which is the reason for the prefixes "SF" and "RF".

```
> length(crossSingleFunnel@geneticData)

[1] 1

> length(crossRandomFunnels@geneticData)

[1] 1

> combined <- crossSingleFunnel + crossRandomFunnels
> length(combined@geneticData)

[1] 2
```

4.2 Summarising and subsetting

The number of markers, founder lines and final lines can be extracted using functions `nMarkers`, `nFounders` and `nLines`. The number of markers is standardised once the objects are combined, so the `nMarkers` function outputs only a single value. Functions `nFounders` and `nLines` output a value for each contained design.

```
> nMarkers(crossSingleFunnel)

[1] 602

> nFounders(crossSingleFunnel)

[1] 4

> nFounders(combined)

[1] 4 4

> nLines(crossSingleFunnel)

[1] 1000

> nLines(combined)

[1] 1000 1000
```

A summary of an `mpcross` object is generated using the print function.

```
> print(crossSingleFunnel)
```

```
-------------------------------------------------------
Summary of mpcross object
-------------------------------------------------------
0  markers had missing values in founders
0  markers had non-polymorphic founder genotypes
-------------------------------------------------------
0  markers were biallelic.
602  markers were multiallelic.
-------------------------------------------------------
0  markers had >5% missing data.
0  markers had >10% missing data.
0  markers had >20% missing data.
```

Subsets of the data in an `mpcross` object can be extracted using the `subset` function.

```
subset(mpcross, markers, chromosomes, lines, groups)
```

Input `markers` can be marker names of indices within `markers(mpcross)`. Input `chromosomes` is only valid if object `mpcross` has an associated map, and must refer to chromosomose by name. Input `lines` must refer to lines by name. Input `groups` is only valid if object `mpcross` has associated linkage groups (without an actual map).

4.3 Fully informative markers

When simulating data using `simulateMPCross` all markers are generated as fully informative. We can see this by inspecting the contained objects of class `geneticData`. Slot `founders` contains data about the founder alleles. The founders data can be accessed by the helper function `founders`. In the following case, each founder line carries a unique marker allele, for all five markers.

```
> #Equivalent to crossSingleFunnel@geneticData[[1]]@founders[,1:5]
> founders(crossSingleFunnel)[,1:5]
```

```
     D1M1 D1M2 D1M3 D1M4 D1M5
SFL1    1    1    1    1    1
SFL2    2    2    2    2    2
SFL3    3    3    3    3    3
SFL4    4    4    4    4    4
```

When simulating data using `simulateMPCross`, there are also marker heterozygotes, which by default are all simulated as being distinguishable. The simulated object contains information about how combinations of marker alleles from the founder lines are mapped to observed values for the final population. As the founders are assumed to be inbred, it is logical that a homozygote of each marker allele present in the founders should be encoded identically in the final population. For example, in the simulated population we are considering, there are alleles `1 - 4` for each marker, and values `1 - 4` for the final population represent homozygotes

of those alleles. This restriction is enforced by the package, so it is not possible to encode a homozygote of marker allele `2` in the founder lines, as `3` in the final population.

The marker alleles for the founders do not *have* to be the values `1 - 4`. For example, another possibility is:

```
     D1M1 D1M2 D1M3 D1M4 D1M5
SFL1    1    1    1    1    1
SLF2   10   10   10   10   10
SLF3  100  100  100  100  100
SLF4  200  200  200  200  200
```

In this case values `1, 10, 100` and `200` for the final population must represent a homozygote of the corresponding marker allele.

Data about the encoding of heterozygotes is contained in the `hetData` slot. Heterozygote data for a marker is formatted in three columns. The first and second are marker alleles, and the third is the encoding of that combination of marker alleles, for a line in the final population. As mentioned previously, it is required that a homozygote for a marker allele m be encoded as m. For example, a row containing $1, 1$ and 0 would be invalid, as it attempts to encode a homozygote of marker allele 1 as 0 in the final population.

Helper function `hetData` can be used to access the heterozygote data. For example:

```
> #Equivalent to crossSingleFunnel@geneticData[[1]]@hetData[["D1M1"]]
> hetData(crossSingleFunnel, "D1M1")
```

```
      [,1] [,2] [,3]
 [1,]    1    1    1
 [2,]    2    1    5
 [3,]    3    1    6
 [4,]    4    1    7
 [5,]    1    2    5
 [6,]    2    2    2
 [7,]    3    2    8
 [8,]    4    2    9
 [9,]    1    3    6
[10,]    2    3    8
[11,]    3    3    3
[12,]    4    3   10
[13,]    1    4    7
[14,]    2    4    9
[15,]    3    4   10
[16,]    4    4    4
```

Columns one and two give a pair of marker alleles, and the third column gives the encoding of this combination in the final poulation. So observed values $1 - 4$ correspond to homozygotes for founder lines, and values $5 - 10$ correspond to different heterozygotes. We specified 2 generations of selfing and this is reflected in the distribution of observed values for the final population.

```
> table(finals(crossSingleFunnel)[,1])

  1   2   3   4   5   6   7   8   9  10
222 198 187 192  33  35  31  32  33  37
```

4.4 Less informative markers

The most common types of markers currently used are *Single Nucleotide Polymorphism* (SNP) markers. To convert our simulated data objects to these types of markers, we combine them with a call to `multiparentSNP`.

```
> combinedSNP <- combined + multiparentSNP(keepHets = TRUE)
```

This modification can also be applied on a per-dataset basis.

```
> combinedSNP <- combined
> combinedSNP@geneticData[[1]] <- combinedSNP@geneticData[[1]] +
+     multiparentSNP(keepHets = TRUE)
> combinedSNP@geneticData[[2]] <- combinedSNP@geneticData[[2]] +
+     multiparentSNP(keepHets = FALSE)
> founders(combinedSNP@geneticData[[1]])[, 1:5]

     D1M1 D1M2 D1M3 D1M4 D1M5
SFL1    0    1    1    0    0
SFL2    0    1    0    1    1
SFL3    0    1    0    1    1
SFL4    1    0    0    1    0

> hetData(combinedSNP, "D1M1")

[[1]]
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    1    1    1
[3,]    0    1    2
[4,]    1    0    2

[[2]]
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    1    1    1
```

The founders in object `combinedSNP` now have only two alleles (0 and 1) for every marker. In the first data set combinations of different marker alleles are coded as 2. For the second data set we specified `keepHets = FALSE` so these marker heterozygotes are replaced by `NA` in the data, and no encoding for heterozygotes is specified. The corresponding function for biparental designs is `biparentalSNP`.

4.5 Importing data

An `mpcross` object can be created using function `mpcross`.

```
mpcross(founders, finals, pedigree, hetData, fixCodingErrors = FALSE)
```

Input `founders` is the matrix of founder marker alleles, where rows correspond to lines and columns correspond to marker names. The number of rows must be equal to the number of initial lines in the pedigree (lines which have `mother` and `father` equal to 0). The row names of input `founders` must match the names of the initial lines in the pedigree.

Input `finals` is the matrix of marker alleles for the final population of genotyped lines, where rows correspond to lines and columns correspond to marker names. The row names of this matrix must be lines named in the pedigree. The column names must be the marker names, which must be identical to the markers given for `founders`.

Input `pedigree` must be a pedigree object, as described in Section 3.

Input `hetData` describes the encoding of the marker alleles for the final population, and must have class `hetData`. It is list, where the names of elements are marker names, and each entry is a three-column matrix, giving the marker encodings for that marker. In the simplest case we have `nMarkers` SNP markers without any heterozygote calls, so the `hetData` object can be constructed as follows.

```
> nMarkers <- 10
> hetData <- replicate(nMarkers, rbind(rep(0, 3), rep(1, 3)), simplify=FALSE)
> names(hetData) <- paste0("M", 1:10)
> hetData <- new("hetData", hetData)
> hetData[[1]]

     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    1    1    1
```

Specifying `hetData = infiniteSelfing` in the call to `mpcross` is a shortcut for this common case. Another common case is bi-allelic SNP markers with heterozygotes called, which is specified with `hetData = hetsForSNPMarkers` in the call to `mpcross`. The encoding for the heterozygotes can be automatically be determined from the data, as there is only a single heterozygote. An example of *manually* constructing the `hetData` object for SNP markers with heterozygotes is as follows.

```
> nMarkers <- 10
> hetData <- replicate(nMarkers, rbind(rep(0, 3), rep(1, 3), c(0, 1, 2),
+     c(1, 0, 2)), simplify=FALSE)
> names(hetData) <- paste0("M", 1:10)
> hetData <- new("hetData", hetData)
> hetData[[1]]

     [,1] [,2] [,3]
[1,]    0    0    0
```

```
[2,]    1    1    1
[3,]    0    1    2
[4,]    1    0    2
```

If `fixCodingErrors = TRUE`, then the function will remove invalid data. Invalid data is detected using the `listCodingErrors` function. See Section 4.6 for further details. The previously constructed object `crossSingleFunnel` can be constructed from its parts as

```
> founders <- founders(crossSingleFunnel)
> finals <- finals(crossSingleFunnel)
> hetData <- hetData(crossSingleFunnel)
> crossSingleFunnel <- mpcross(founders = founders, finals = finals,
+     pedigree = pedigreeSF, hetData = hetData)
```

4.6 Invalid data

Real data often contains some invalid data. **mpMap2** performs extensive checks, and will reject invalid data.

```
> #Put in two errors for the founders
> founders[3,3] <- 10
> founders[2,2] <- NA
> #Put in an error for the finals
> finals[1, 1] <- 100
> #Put in two errors for the hetData
> hetData[4] <- list(rbind(rep(0, 3), rep(1, 3)))
> hetData[[5]][1,1] <- NA
> error <- try(crossSingleFunnel <- mpcross(founders = founders, finals = finals,
+     pedigree = pedigreeSF, hetData = hetData))
> cat(error)
```

```
Error in validObject(.Object) :
  invalid class âĂIJgeneticDataâĂ object: 1: Coding error for marker D1M3: Founder allele
invalid class âĂIJgeneticDataâĂ object: 2: Coding error for marker D1M4: Founder allele 2
invalid class âĂIJgeneticDataâĂ object: 3: Coding error for marker D1M4: Founder allele 3
invalid class âĂIJgeneticDataâĂ object: 4: Coding error for marker D1M4: Founder allele 4
invalid class âĂIJgeneticDataâĂ object: 5: Coding error for marker D1M5: Founder allele 1
invalid class âĂIJgeneticDataâĂ object: 6: Omitting details of further coding errors
```

A more computer-friendly list of most of thees errors is available using the `listCodingErrors` function.

```
> errors <- listCodingErrors(founders = founders, finals = finals, hetData = hetData)
> errors$invalidHetData
```

```
     Marker Row Column
[1,]      3   3      1
```

```
 [2,]      3   7      1
 [3,]      3   9      2
 [4,]      3  10      2
 [5,]      3  11      1
 [6,]      3  11      2
 [7,]      3  12      2
 [8,]      3  15      1
 [9,]      4   1      1
[10,]      4   1      2
[11,]      5   1      1
```

```
> errors$null
```

```
[1] 2
```

```
> head(errors$finals)
```

```
     Row Column
[1,]   1      1
[2,]   1      4
[3,]   3      4
[4,]   5      4
[5,]   6      4
[6,]   7      4
```

We begin with `errors$invalidHetData`. There are errors in the `hetData` for the third marker, because marker allele 3 has been removed for this marker. This means that heterozygotes between this allele and other marker alleles are now invalid. There are errors in the `hetData` for the fourth marker, because there is no marker allele 0. There is an error in the `hetData` for the fifth marker, because `NA` is invalid here.

The entry `errors$null` indicates that the second marker has a missing allele for a founder. For these markers the observed marker alleles for the final population must be `NA`, and the corresponding `hetData` entry must have zero rows.

The entry `errors$finals` indicates that line 1 has an invalid value of 100 for marker 1. It also indicates that every marker allele except 1 is invalid for marker 4, due to the modification of `hetData[[4]]`.

4.7 Genetic maps

The format used for genetic maps in this package is identical to that in package **qtl**. A genetic map is a named list, where the name of each entry is the name of that chromosome. Each entry contains the names and positions of each marker, in increasing order. The overall object must have class `"map"`. We now give an example of the structure of a simulated map.

```
> simulatedMap <- qtl::sim.map(len = rep(100, 2), n.mar = 11, anchor.tel = TRUE,
+     include.x = FALSE, eq.spacing = FALSE)
> #map object has class "map"
> class(simulatedMap)
```

```
[1] "map"

> #Names of entries are chromosomoe names
> names(simulatedMap)

[1] "1" "2"

> #Markers are in increasing order.
> simulatedMap[["1"]]

      D1M1       D1M2       D1M3       D1M4       D1M5       D1M6       D1M7
  0.000000   3.690961  10.556573  25.683635  45.513211  57.730927  61.346645
      D1M8       D1M9      D1M10      D1M11
 66.263965  82.381991  85.227520 100.000000
attr(,"class")
[1] "A"
```

5 Estimation of recombination fractions

5.1 Methodology

For any pair of genetic locations there is a probability model govering the joint distribution of the sources of the inherited alleles. That is, a genotyped final line will have an allele at marker M_1 inherited from some founder line, and another allele at marker M_2 inherited from a (potentially different) founder line. We ignore the fact that different founders may have identical alleles; it is the source of the allele that is important.

These joint distributions are governed by the *identity-by-descent* (IBD) probabilities, which have been calculated for a variety of different designs (**????**). These probabilities are a function of the recombination fraction r between the two markers. The relevant probabilities for more complicated designs (especially those with finite generations of selfing) are too complicated to give here, but can be calculated with the help of a computer algebra system such as Octave or Mathematica.

If two markers are fully informative, then the probability model is informative for the parameter r, which can be estimated using numerical maximum likelihood. However this may no longer be true when the markers are less informative. For example, assume we have a four-parent design with a single funnel and infinite generations of selfing, and markers M_1 and M_2 with the following distribution of marker alleles for the founders.

```
           M1 M2
Founder 1  1  0
Founder 2  0  1
Founder 3  0  0
Founder 4  1  1
```

In this case every combination of marker alleles occurs with probability $\frac{1}{4}$, regardless of the parameter r. For four-parent designs this combination of marker allele distributions is the

only one that may be non-informative for r. Note that for four-parent designs with finite generations of selfing this combination may in fact be informative.

The situation appears to be more complicated the larger the number of founders. For the eight-way design there are combinations of marker allele distributions that are completely uninformative, similar to the four parent design. However there are also marker allele distributions which are *approximately uninformative* for the parameter r. For example, consider the following marker allele distributions with a single funnel and infinite generations of selfing.

```
           M1 M2
Founder 1   1  0
Founder 2   0  0
Founder 3   0  1
Founder 4   1  0
Founder 5   1  0
Founder 6   0  0
Founder 7   1  1
Founder 8   1  1
```

In this case the likelihood is approximately (but not exactly) flat. The marker probabilities as a function of r are shown in Figure 4. For comparison, we consider the following marker allele distribution to be informative.

```
           M1 M2
Founder 1   1  0
Founder 2   1  1
Founder 3   0  0
Founder 4   1  0
Founder 5   1  0
Founder 6   0  0
Founder 7   1  1
Founder 8   1  1
```

The marker probabilities for this informative case are shown in Figure 5. There are also cases where the likelihood is approximately symmetric. For example, consider the following marker allele distribution.

```
           M1 M2
Founder 1   1  0
Founder 2   1  1
Founder 3   0  1
Founder 4   1  0
Founder 5   1  0
Founder 6   0  0
Founder 7   1  1
Founder 8   1  1
```

The marker probabilities for this case are shown in Figure 6. In this case $r = 0$ is indistinguishable from $r = 0.5$.

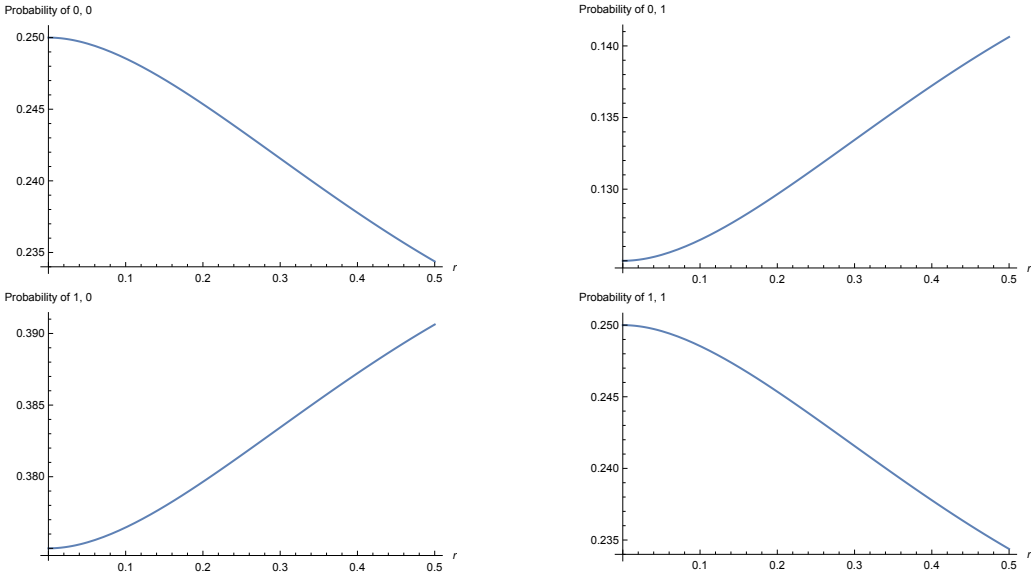

Figure 4: Joint marker probabilities for an approximately uninformative pair of markers. The design used is an eight-parent cross with a single funnel, zero generations of intercrossing and infinite generations of selfing.

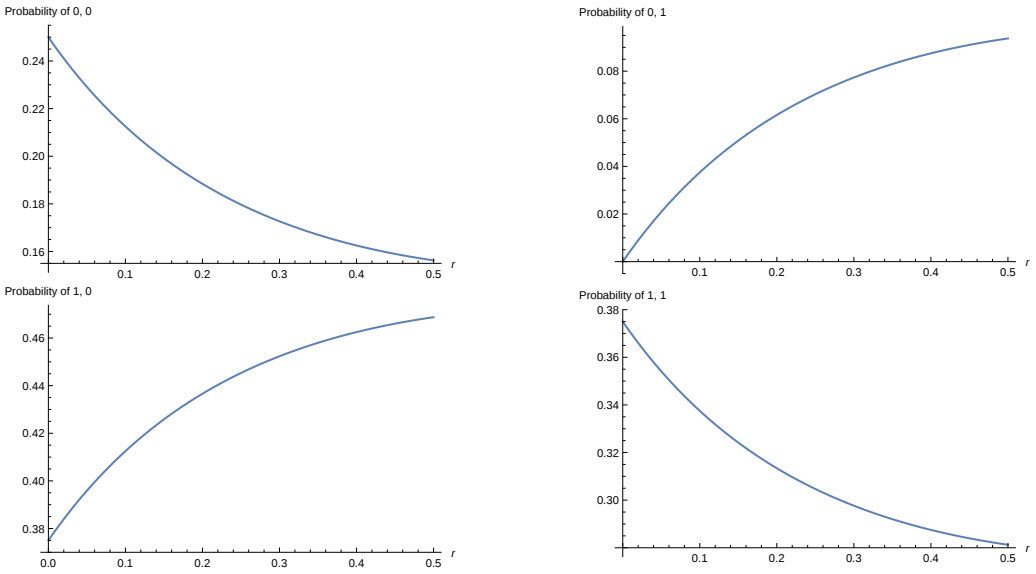

Figure 5: Joint marker probabilities for an informative pair of markers. The design used is an eight-parent cross with a single funnel, zero generations of intercrossing and infinite generations of selfing.

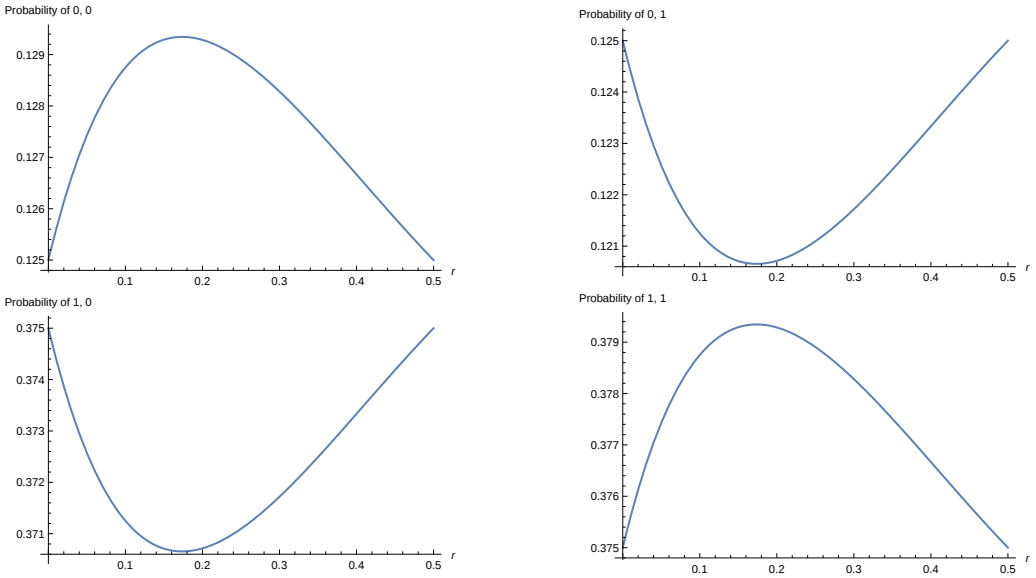

Figure 6: Joint marker probabilities for an uninformative pair of markers. The design used is an eight-parent cross with a single funnel, zero generations of intercrossing and infinite generations of selfing.

We can test whether a pair of markers is completely non-informative (in the sense of a flat likelihood) by testing whether the derivative of the likelihood is identically zero. This was the approach originally used in **mpMap**, however it appears to only be practical for designs involving infinite generations of selfing. This approach *cannot* be used to identify marker pairs like those in Figure 4, which are approximately non-informative, or those in Figure 6, which have an approximately symmetric likelihood. For this reason it is necessary to use a numerical test for non-informative and approximately non-informative marker pairs.

The marker probabilities are computed for a large number of equally spaced values of r. Let $\{P_i(r)\}$ be the set of marker probabilities at some recombination value r. For a pair of SNP markers, i would take values in $\{\{0,0\},\{0,1\},\{1,0\},\{1,1\}\}$. If there are recombination values r_1 and r_2 with $|r_1 - r_2| > 0.06$ so that the L^1 distance $\sum_i |P_i(r_1) - P_i(r_2)|$ is less than 0.003, then the pair of markers will be declared uninformative. This heuristic is computationally expensive, but has the advantage of detecting both uninformative and approximately uninformative pairs of markers.

Although the number of markers may be large, there are only a finite number of *different* marker allele distributions that are possible. Therefore we collect a list of distinct marker allele distributions and run the heuristic on all pairs. The computational cost of the heuristic is therefore a fixed cost, independent of the number of lines or number of markers. In the context of large data sets this computational cost will likely be insignificant.

5.2 Implementation

The function `estimateRF` estimates the recombination fractions between all pairs of markers in an `mpcross` object using numerical maximum likelihood and a simple grid search. It accounts for all the data sets contained in the object when performing the estimation, and

uses the numerical test mentioned at the end of Section 5.1 to return a value of `NA` where the relevant probability model is uninformative or approximately uninformative.

The signature of the function is

```
estimateRF(object, recombValues, lineWeights, gbLimit = -1, keepLod = FALSE,
    keepLkhd = FALSE, verbose = FALSE, markerRows = 1:nMarkers(object),
    markerColumns = 1:nMarkers(object))
```

Input `object` is an object of class `mpcross` and input `recombValues` is the set of recombination fraction values to test in the grid search. Input `lineWeights` allows correction for segregation distortion and is beyond the scope of this document, see `?` for further details. Input `gbLimit` specifies the maximum amount of memory to be used during the comutation. Input `keepLkhd` determines whether the value of the maximum likelihood is computed. Input `keepLod` determines whether the likelihood ratio statistic for testing the hypothesis $r \neq \frac{1}{2}$ is computed. Input `verbose` outputs diagnostic information such as the current progress, and the amount of memory used. Inputs `markerRows` and `markorColumns` are used to compute only part of the full recombination fraction matrix. When using these values, only the part in the upper-right triangle is computed.

The value returned by the function is an object of class `mpcrossRF`. It has the same `geneticData` slot as the object of class `mpcross`, but also contains a slot named `rf` with the results of the computation. The main result is the matrix of recombination fraction estimates, which is stored in slot `@rf@theta`. If this was stored as a numeric matrix it would require hundreds of gigabytes of storage space for some data sets. Fortunately, this matrix is symmetric, and each entry is one of the values specified in input `recombValues`. This matrix is therefore stored as an object of class `rawSymmetricMatrix`, which stores each value in the upper triangle of the matrix as a single byte. Each byte is interpreted as the index into `recombValues` which gives the estimated value. If n is the number of markers we require only $\frac{n(n+1)}{2}$ bytes of storage, a 16-fold reduction in storage requirements compared to the storage of a similar object in **mpMap**. The value of `0xff` is interpreted as being `NA`. This requires input `recombValues` to always have less than 256 values, but this is not a siginificant limitation.

Input `keepLod` instructs the function to compute the matrix of likelihood ratio statistics for testing whether the recombination fractions are different from $\frac{1}{2}$. Input `keepLkhd` instructs the function to return the maximum value of the likelihood for every pair of markers. The values contained in these extra symmetric matrices are not restricted to a small number of levels, so they are stored as objects of class `dspMatrix` (dense symmetric matrix in packed storage) from package **Matrix**. These matrices require $4n(n+1)$ bytes of storage, which becomes infeasible very quickly. For example, with $n = 10^5$ markers each of these matrices occupy 40 gb. In generaly we suggest that these matrices *not* be computed.

The intermediate stages of the computation require significantly more memory than the final result. It may be necessary to perform the computation of the recombination fraction matrix in parts to avoid running out of memory. Input `gbLimit` allows the user to specify the maximum amount of memory (in gigabytes) to be used at any one time.

Our package makes more extensive use of lookup tables and pre-computation than **mpMap**. As a result we can analyse large data sets using only OpenMP multi-threading. Package **mpMap** required the use of much more complicated MPI or CUDA multi-threading to achieve acceptable performance, and this code was much harder to maintain and use.

To demonstrate this function, we apply it to the object `combined` which we created previously. In general option `verbose` would be set to `TRUE` or `FALSE`. In this case we need output suitable for a document, so we use `list(progressStyle=1)`. This specifies that that argument `style` of `txtProgressBar` should be 1, giving output suitable for a document instead of a console.

```
> rf <- estimateRF(object = combinedSNP, verbose = list(progressStyle = 1))
```

```
Allocating results matrix of 88573464 bytes = 0 gb
Total lookup table size of 177632 bytes = 0 gb
================================================================================
```

6 Construction of linkage groups

Partitioning the markers into linkage groups can be performed using the function `formGroups`. It has signature

```
formGroups(mpcrossRF, groups, clusterBy = "theta", method = "average",
    preCluster = FALSE)
```

Input `mpcrossRF` is an object of class `mpcrossRF`. Input `groups` is the number of linkage groups to construct. Input `method` is the choice of linkage method, and must be one of `"average"`, `"complete"` or `"single"`. Input `clusterBy` is the choice of dissimilarity matrix to use for clustering. It can be either `"theta"` for the recombination fraction matrix, `"lod"` for the matrix of likelihood ratio test statistics, or `"combined"`. In the last case, let ∇ be the minimum distance between any pair of recombination fraction values used for the numerical maximum likelihood step, let $\mathbf{L}$ be the matrix of likelihood ratio statistics, let l be maximum of the values of $\mathbf{L}$ and let Θ be the matrix of recombination fractions. Then the dissimilarity matrix used in the `"combined"` case is

$$\Theta + \frac{\mathbf{L}}{l}\nabla.$$

Intuitively, this means that the values in $\mathbf{L}$ are used to break ties between equal values in Θ. We emphasise that specifying `"combined"` or `"lod"` for input `clusterBy` requires the matrix of likelihood ratio statistics to be computed using `estimateRF`, and as mentioned this may require an infeasible amount of storage.

Internally `formGroups` uses hierarcichal clustering, and this requires that the full dissimilarity matrix be stored in memory (as opposed to in a compressed form, such as an object of type `rawSymmetricMatrix`). This means that if there are a large number of markers `formGroups` may require a large amount of working memory. Specifying `preCluster = TRUE` attempts to reduce the amount of memory required by identifying groups of markers where the recombination fractions between them are all zero. These markers are grouped before the hierarchical clustering is performed, reducing the dimension of the dissimilarity matrix and therefore the required working memory.

7 Ordering of chromosomes

Ordering of chromosomes is performed using *simulated annealing*. We use a simulated annealing method (**?**) known as *Anti-Robinson seriation*. The implementation originally comes from the **seriation** package (**?**), and has been adapted to our data structures. The simulated annealing algorithm is based on two types of transformations. The first is swapping a pair of random markers, and is computationally fast. The second chooses a random marker and moves it to a random position. This second transformation can be computationally expensive, especially if the random position is very far away from the previous position.

The ordering function is

```
orderCross(mpcrossLG, cool = 0.5, tmin = 0.1, nReps = 1, maxMove = 0,
    effortMultiplier = 1, randomStart = TRUE, verbose = FALSE)
```

Input `cool` is the rate of cooling for the simulated annealing algorithm. Smaller values lead to slower cooling, and higher computational effort. Input `tmin` is the minimum temperature for the algorithm. Input `nReps` is the number of independent repetitions of the algorithm to perform. If `nReps > 1` then the best ordering is returned. If `randomStart = TRUE` then each of these repetitions starts from a random ordering, otherwise they are all started using the current ordering of the `mpcrossLG` object.

Input `maxMove` indicates the maximum possible distance to move a marker using the "move" transformation mentioned previously. A value of `0` indicates no limit, so the chosen marker can be shifted to any location. A value of `1` means the chosen marker can be shifted left one position or right one position, etc. Input `effortMultiplier` increases the amount of computational effort. So a value of 2 will double the amount of computational time, but hopefully result in a better ordering. If `verbose = TRUE` then a progress bar is displayed.

Although simulated annealing performs extremely well on smaller data sets, it is prohibitively expensive on larger data sets and cannot easily be parallelized. Fortunately, it is not generally necessary to order the entire chromosome in one pass. It is acceptable to use hierarchical cluster to form k groups, and order these k groups using simulated annealing. The hierarchical clustering ordering function is

```
clusterOrderCross(mpcrossLG, cool = 0.5, tmin = 0.1, nReps = 1,
    maxMove = 0, effortMultiplier = 1, randomStart = TRUE, nGroups)
```

After using `clusterOrderCross`, finer scale ordering can be performed using `orderCross` with `randomStart = FALSE` and setting `maxMove` to a relatively small value, which ensures only local changes are made to the ordering.

8 Estimation of map distances

Estimation of map distances is non-trivial, even if the correct ordering of the markers is known. One possibility is to take the estimated recombination fractions between adjacent markers, and convert them to centiMorgan distances. However, as demonstrated in Section 5, for some experimental designs and some specific marker allele distributions, the corresponding recombination fraction is very hard to estimate. In some cases the data can even be completely

uninformative about the recombination fraction. This approach is also wasteful in terms of the available information; we have recombination fractions estimates between all pairs of markers, not just all adjacent pairs.

Our map estimation process uses a matrix of constraints, involving not just the recombination fractions between adjacent markers, but any pair of markers that are sufficiently close, in terms of the chosen ordering. This matrix equation is then solved by non-linear least squares, using the **nnls** (**?**) package. Consider the case where there are three markers. With the markers in the (assumed) correct order, the matrix of estimated genetic distances (obtained from the estimated recombination fractions) is

$$\begin{pmatrix} 0 & 5 & 15 \\ 5 & 0 & 7 \\ 15 & 7 & 0 \end{pmatrix}.$$

Then the matrix equation to be solved is

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 15 \\ 7 \end{pmatrix},$$

where a_1 is the distance between the first and second markers, and a_2 is the distance between the second and third markers.

If *all* pairs of markers are used, then the matrix equation becomes large very quickly. As a compromise, we use only pairs of markers that are close, in terms of position within the specified ordering. For example, assume that there are five markers, and we wish to estimate the map distances, using only pairs of markers that are separated by at most two other genetic markers. Assume that the estimated matrix of pairwise genetic distances is

$$\begin{pmatrix} 0 & 5 & 9 & 17 & 24 \\ 5 & 0 & 7 & 11 & 15 \\ 9 & 7 & 0 & 6 & 9 \\ 17 & 11 & 6 & 0 & 3 \\ 24 & 15 & 9 & 3 & 0 \end{pmatrix}.$$

Then the corresponding matrix equation is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \begin{pmatrix} 5 \\ 9 \\ 17 \\ 7 \\ 11 \\ 15 \\ 6 \\ 9 \\ 3 \end{pmatrix}.$$

Note that the constraint $a_1 + a_2 + a_3 + a_4 = 24$ is not used.

The function to estimate a genetic map using this approach is

```
estimateMap (mpcrossLG, mapFunction = rfToHaldane, maxOffset = 1,
    maxMarkers = 2000, verbose = FALSE)
```

Input `mpcrossLG` is a mpcross object with assigned linkage groups. Input `mapFunction` is a map funtion that turns recombination fractions into centiMorgan distances. Input `maxOffset` is the maximum ordering difference between two markers, so that the estimated distance between those markers will be used. For example, in the five marker example just given, this input was 3, because the distance between markers 1 and 4 were used, and so was the distance between markers 2 and 5. But the distance between markers 1 and 5 was not used. Input `maxMarkers` is the maximum number of markers, for which the genetic map will be estimated in a single pass. If there is a larger number of markers in a single linkage group, the map will be estimated in smaller parts, and these parts are then combined. If input `verbose` is `TRUE`, then logging output is generated.

9 Probability models for imputation and probability computations

Imputation of founder genotypes is performed by assuming a Hidden Markov Model (HMM) for the underlying genotypes. Strictly speaking this is incorrect, as the founder genotypes do not form a Markov Chain. For example, consider the biparental recombinant inbred design. **?** gives the probability of the two-loci recombinant genotype AB as $\frac{r}{1+2r}$ and the probabilities of the non-recombinant genotype AA as $\frac{1}{2(1+2r)}$. If the IBD genotypes formed a Markov Chain then the probability of the three equally spaced loci having the IBD genotype AAA would be

$$2\left(\frac{1}{2(1+2r)}\right)^2.$$

This value is in fact (**?**)

$$\frac{1+2r-4r^2-2cr^2+4cr^3}{2(1+2r)(1+4r-4cr^2)},$$

where $c = r^{-2}\mathbb{P}$ (double recombinant). As shown in Figure 7 the approximation is very good, especially over shorter genetic distances. Assuming that the IBD genotype forms a Markov Chain goverened by its two-locus probabilities is unlikely to cause any problems.

10 Imputation

The underlying genotypes can then be imputed using the Viterbi algorithm. This imputation method is implemented by the function `imputeFounders`, which has signature

```
imputeFounders(mpcrossMapped, homozygoteMissingProb = 1,
    heterozygoteMissingProb = 1, errorProb = 0, extraPositions = list())
```

Input `extraPositions` is a list gives extra (non-marker) positions for which to perform imputation. These positions can be given explicitly, in the format shown below, or using the

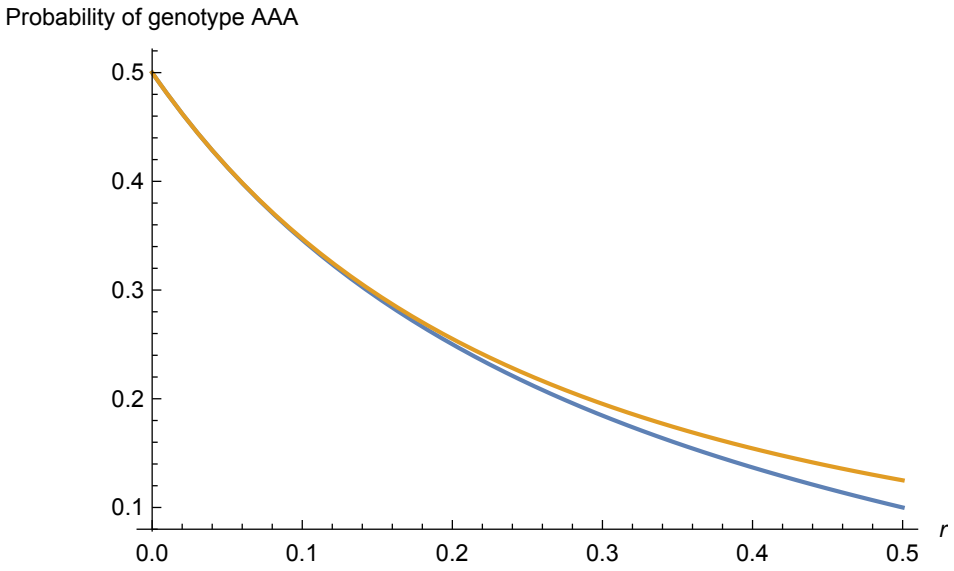

Figure 7: The true three-point probability of genotype AAA for a recombinant inbred line at three equally spaced locations, and the Markov Chain approximation.

convenience function `generateGridPositions(s)`. Specifying this convenience function for `extraPositions` generates a grid of points for each chromosome, equally spaced with distance `s`.

We apply this function to the object `combinedSNP`, which contains two data sets.

```
> mappedSNP <- new("mpcrossMapped", combinedSNP, map = map)
> imputed <- imputeFounders(mappedSNP, extraPositions =
+     list("2" = c("a" = 3.14, "b" = 66)))
```

The extra positions are specified to be positions 3.14 named `a` and 66 named `b`. There are no extra positions on chromosome `1`. Alternatively, we could specify a grid of points, separated by 10 cM.

```
> imputed <- imputeFounders(mappedSNP, extraPositions = generateGridPositions(10))
```

As we originally specified the `selfing` slot to have value `"finite"`, the imputed values will contain heterozygotes. The encoding of heterozygotes is given in an entry named `key`, which can be extracted using the `imputationKey` function. Comparing the imputed data to the original data (before the markers were converted to SNP markers) shows good agreement for the first data set, even for the heterozygotes.

```
> imputed <- imputeFounders(mappedSNP)
> imputationKey(imputed, experiment = 1)
```

```
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    2    5
```

```
 [3,]    1    3    6
 [4,]    1    4    7
 [5,]    2    1    5
 [6,]    2    2    2
 [7,]    2    3    8
 [8,]    2    4    9
 [9,]    3    1    6
[10,]    3    2    8
[11,]    3    3    3
[12,]    3    4   10
[13,]    4    1    7
[14,]    4    2    9
[15,]    4    3   10
[16,]    4    4    4
```

```
> table(imputationData(imputed, experiment = 1), finals(combined)[[1]])
```

```
       1      2      3      4      5      6      7      8      9     10
1  123651   1847   1134    797    797    698    570      4      2      5
2     413 114445    975   1047    794      8      9    653    606      1
3     236    347 121590   1904     10    744      7    666     13    776
4     160    254    249 118324      2      7    529      2    673    743
5      68     61      0      0  16414    208    135    125    111      1
6      24      0     35      2     88  18212    230    325      2     32
7      17      0      0     33     69     55  17623     19    341     35
8       1     83     82      0     74     70      7  17119    221     46
9       0     32      0     34     46      0     50     48  16540     81
10      0      1     34     35      2    105    161    143    173  16855
```

The pattern of imputation errors for the heterozygotes makes sense; value 5 is a heterozygote of founders 1 and 2, and the most frequent imputation error is to classify it as a homozygote of founder 1 or 2.

For the second data set no heterozygotes are imputed. This is because no heterozygote markers were called, so heterozygotes are either missing or consistent with being homozygotes, in which case the homozygote is always more likely. The only clue in the data is that missing values are always heterozygotes in this case. Setting `heterozygoteMissingProb` to 1 and `homozygoteMissingProb` to 0.05 gives acceptable results.

```
> table(imputationData(imputed, experiment = 2), finals(combined)[[2]])
```

```
       1      2      3      4      5      6      7      8      9     10
1 121810   1657   1387   1766  11398  11211  11635    417    570    362
2    326 121116   1444   1474   7856    275    195  10892  11175    434
3    357    330 116287   1376     86   7652    156   7678    180   9972
4    469    286    245 117495    132    125   7439     77   7970   6288
```

```
> imputed <- imputeFounders(mappedSNP, heterozygoteMissingProb = 1,
+     homozygoteMissingProb = 0.05)
> table(imputationData(imputed, experiment = 2), finals(combined)[[2]])
```

	1	2	3	4	5	6	7	8	9	10
1	121966	1479	1233	1546	990	851	1089	17	21	17
2	282	121304	1264	1300	842	8	8	932	1020	17
3	270	248	116609	1218	7	799	13	887	8	857
4	321	213	172	117944	7	3	868	6	948	751
5	39	45	0	1	17403	177	139	150	236	2
6	42	0	42	0	50	17254	147	161	5	176
7	42	0	0	49	46	56	17061	2	194	163
8	0	64	25	1	73	77	2	16834	157	138
9	0	36	0	25	54	1	61	35	17235	144
10	0	0	18	27	0	37	37	40	71	14791

The miss-classifications still demonstrate the same problem to a lesser extent. Value 5 is a heterozygote of founders 1 and 2, and just under 50% of these values are miss-classified as homozygotes of founders 1 or 2.

11 Example

11.1 No intecrossing or selfing

We begin with an example showing that **mpMap2** can construct correct maps from unusual experimental designs. In this case we use a four-parent cross with randomly chosen funnels and no intercrossing and no selfing. The underlying genotypes for this design are all heterozygotes.

```
> pedigree <- fourParentPedigreeRandomFunnels(initialPopulationSize = 800,
+     intercrossingGenerations = 0, selfingGenerations = 0, nSeeds = 1)
> selfing(pedigree) <- "finite"
> map <- qtl::sim.map(len = rep(300, 3), n.mar = 101, anchor.tel = TRUE,
+     include.x = FALSE, eq.spacing = FALSE)
> cross <- simulateMPCross(pedigree = pedigree, map = map,
+     mapFunction = haldane, seed = 1)
> crossSNP <- cross + multiparentSNP(keepHets=TRUE)
> table(finals(cross))
```

```
    5     6     7     8     9    10
41825 41089 40692 39893 39590 39311
```

The estimation of recombination fractions is somewhat slow due to the amount of precalculation, as every distinct funnel is treated separately. This precalculation cost does not grow with the total number of markers.

```
> #Randomly rearrange markers
> crossSNP <- subset(crossSNP, markers = sample(markers(cross)))
> rf <- estimateRF(crossSNP, verbose = list(progressStyle = 1))
```

```
Allocating results matrix of 22475328 bytes = 0 gb
Total lookup table size of 368928 bytes = 0 gb
================================================================================
```

The next steps are forming linkage groups, ordering chromosomes and imputing missing recombination fraction values.

In this case we disable parallelisation in the marker ordering step. As this is a small example, on a computer with a large number of threads parallelisation can significantly slow down the runtime. On the other hand, when doing local reordering on large datasets, parallelisation can help significantly.

```
> grouped <- formGroups(rf, groups = 3, method = "average", clusterBy="theta")
> try(omp_set_num_threads(1), silent = TRUE)

NULL

> ordered <- orderCross(grouped, effortMultiplier = 2)
> imputedTheta <- impute(ordered, verbose = list(progressStyle = 1))

Starting imputation for group 1
================================================================================
Starting imputation for group 2
================================================================================
Starting imputation for group 3
================================================================================
```

The next step is to estimate the map. Our estimated map is significantly longer than the true map. Note the use of the function `jitterMap`. This function spaces out markers that have been assigned to the same location. This is necessary for the purposes of imputation, as `estimateMap` is capable of estimating a map where markers which are observed to have at least one recombination event between them are assigned the same location. This makes the map incompatible with the data, and would cause problems during the imputation step, unless a non-zero `errorProb` is specified.

```
> estimatedMap <- estimateMap(imputedTheta, maxOffset = 10)
> estimatedMap <- jitterMap(estimatedMap)
> #match up estimated chromosomes with original chromosomes
> estChrFunc <- function(x) which.max(unlist(lapply(estimatedMap,
+     function(y) length(intersect(names(y), names(map[[x]]))))))
> estimatedChromosomes <- sapply(1:3, estChrFunc)
> tail(estimatedMap[[estimatedChromosomes[[1]]]])

    D1M6     D1M4     D1M3     D1M5     D1M2     D1M1
286.3170 290.8652 293.2176 295.5700 297.9224 298.3933

> tail(map[[1]])
```

```
   D1M96    D1M97    D1M98    D1M99   D1M100   D1M101
286.9760 287.1171 288.5251 292.6399 292.8494 300.0000
```

The reason for constructing a genetic map is often to search for quantitative trait loci (QTL). Therefore it is not the overall length that is important, but the accurate imputation of the underlying founder genotypes. In this case the imputation is highly accurate.

```
> mappedObject <- new("mpcrossMapped", imputedTheta, map = estimatedMap)
> imputedFounders <- imputeFounders(mappedObject)
> summary <- table(imputedFounders@geneticData[[1]]@imputed@data[,markers(cross)],
+     finals(cross))
> sum(diag(summary))/sum(summary)

[1] 0.9503424
```