UNIVERSITÉ DU QUÉBEC À MONTRÉAL

EXPLORATION D'UNE NOUVELLE MÉTHODE D'ESTIMATION DANS LE PROCESSUS DE COALESCENCE AVEC RECOMBINAISON

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES

PAR

HUGUES MASSÉ

AOÛT 2008

REMERCIEMENTS

Je tiens d'abord à remercier mon directeur Fabrice Larribe pour m'avoir supervisé. Malgré son horaire chargé, Fabrice a patiemment répondu à mes nombreuses questions. Il m'a facilité la vie en collaborant à l'élaboration d'un plan de travail et en fournissant des articles et outils informatiques appropriés. Merci Fabrice !

Ensuite, je tiens à remercier mes amis Simon Belzile pour ses précieux conseils informatiques et Alain Régnier pour son support.

Enfin, je veux remercier le CRSNG pour son aide financière et ma famille pour son appui tout au long de ma maîtrise.

TABLE DES MATIÈRES

LISTE DES FIGURES

RÉSUMÉ

L'estimation de paramètres génétiques est un problème important dans le domaine de la génétique mathématique et statistique. Il existe plusieurs méthodes s'attaquant à ce problème. Certaines d'entre elles utilisent la méthode du maximum de vraisemblance. Celle-ci peut être calculée à l'aide des équations exactes de Griffiths-Tavaré, équations de récurrence provenant du processus de coalescence.

Il s'agit alors de considérer plusieurs histoires possibles qui relient les données de l'échantillon initial de séquences d'ADN à un ancêtre commun. Habituellement, certaines des histoires possibles sont simulées, en conjonction avec l'application des méthodes Monte-Carlo. Larribe *et al.* (2002) utilisent cette méthode (voir chapitre IV).

Nous explorons une nouvelle approche permettant d'utiliser les équations de Griffiths-Tavaré de façon différente pour obtenir une estimation quasi exacte de la vraisemblance sans avoir recours aux simulations. Pour que le temps de calcul nécessaire à l'application de la méthode demeure raisonnable, nous devons faire deux compromis majeurs. La première concession consiste à limiter le nombre de recombinaisons permises dans les histoires. La seconde concession consiste à séparer les données en plusieurs parties appelées fenêtres. Nous obtenons ainsi plusieurs vraisemblances marginales que nous mettons ensuite en commun en appliquant le principe de vraisemblance composite.

À l'aide d'un programme écrit en C++, nous appliquons notre méthode dans le cadre d'un problème de cartographie génétique fine où nous voulons estimer la position d'une mutation causant une maladie génétique simple.

Notre méthode donne des résultats intéressants. Pour de très petits ensembles de données, nous montrons qu'il est possible de permettre un assez grand nombre de recombinaisons pour qu'il y ait convergence dans la courbe de vraisemblance obtenue. Aussi, il est également possible d'obtenir des courbes dont la forme et l'estimation du maximum de vraisemblance sont similaires à celles obtenues avec la méthode de Larribe *et al.* Cependant, notre méthode n'est pas encore applicable dans son état actuel parce qu'elle est encore trop exigeante en termes de temps de calcul.

Mots clés : équations exactes de Griffiths-Tavaré, paramètres génétiques, processus de coalescence, vraisemblance composite.

INTRODUCTION

En statistique génétique, plusieurs méthodes permettent d'estimer des paramètres génétiques. La méthode du maximum de vraisemblance en est une qui peut bien s'appliquer à ce type de problème. Celle-ci, à son tour, peut s'opérer de différentes façons. Entre autres, nous avons le choix d'effectuer les calculs de vraisemblance en considérant les généalogies ou les histoires (Felsenstein *et al.*, 1999, par exemple).

Cette dernière stratégie, effectuer les calculs de vraisemblance à l'aide des histoires, est une méthode courante depuis l'introduction des équations de Griffiths-Tavaré en 1994. Ce sont des équations de récurrence qui expriment la vraisemblance d'un ensemble de données en fonction des vraisemblances des ensembles de données antérieurs.

Les équations de Griffiths-Tavaré sont construites à partir d'un modèle d'évolution particulier : le processus de coalescence (Kingman, 1982). Il s'agit d'un processus stochastique servant à décrire l'historique de la transmission de séquences d'ADN dans une population. C'est un processus flexible qui permet d'inclure toutes sortes d'options comme celle qui consiste à considérer les évènements de recombinaison (Hudson, 1983 et Griffiths et Marjoram, 1996).

En 2002, Larribe *et al.* ont mis au point une méthode de cartographie génétique fine qui permet d'estimer la position d'une mutation causant une maladie génétique simple. C'est une méthode qui tient compte de la recombinaison et qui utilise les équations de Griffiths-Tavaré.

Dans leur méthode, Larribe *et al.* considèrent plusieurs histoires qui relient l'échantillon initial de séquences d'ADN à un ancêtre commun. Ils en simulent un grand nombre et la

vraisemblance est estimée à l'aide de méthodes Monte-Carlo.

Leur méthode comporte quelques inconvénients : le temps de calcul est assez grand, les courbes de vraisemblance sont différentes d'une simulation à l'autre et l'estimation fournie n'est pas toujours bonne. La recherche d'une méthode alternative est donc tout à fait justifiée. L'objectif principal de ce mémoire est d'explorer une nouvelle approche permettant possiblement d'obtenir de meilleurs résultats.

Cette approche consiste à considérer toutes les histoires qui relient l'échantillon initial de séquences d'ADN à l'ancêtre commun. Puisque le nombre d'histoires possibles est très grand, nous devons forcément faire des compromis pour que le temps de calcul demeure raisonnable. Ainsi, nous limitons le nombre de recombinaisons permises dans les histoires et nous séparons nos données en plusieurs parties appelées fenêtres. Nous regroupons par la suite les vraisemblances marginales obtenues pour chacune des fenêtres à l'aide du principe de vraisemblance composite (Lindsay, 1988 et Varin et Vidoni, 2005).

Ce mémoire est divisé en cinq chapitres. Les deux premiers chapitres fournissent la théorie nécessaire à l'application des méthodes d'estimation de vraisemblance considérées par la suite. Au chapitre I, nous présentons les notions pertinentes de génétique. Ensuite, au chapitre II, la théorie de la coalescence est expliquée. Plusieurs méthodes d'estimation de vraisemblance sont brièvement présentées au chapitre III. La méthode de Larribe *et al.* est exposée au chapitre IV. Finalement, au chapitre V, la nouvelle méthode que nous explorons dans ce mémoire est développée.

CHAPITRE I

NOTIONS DE GÉNÉTIQUE

Dans ce chapitre, quelques notions de génétique nécessaires à la compréhension du mémoire sont présentées.

1.1 Concepts élémentaires

L'être humain est composé d'ADN, une substance qui lui permet de synthétiser des protéines. L'ADN est composé d'une suite de nucléotides. Il existe quatre types de nucléotides : l'adénine (A), la cytosine (C), la guanine (G) et la thymine (T). Ces nucléotides sont arrangés en groupes de trois, appelés codons.

Chaque être humain a 46 chromosomes divisés en 23 paires. Nous disons alors que l'être humain est un être diploïde, contrairement aux êtres haploïdes qui n'ont qu'une seule copie de chaque chromosome. Présents dans le noyau de nos cellules, les chromosomes sont des matériaux génétiques composés d'ADN et de protéines.

Les gènes se présentent par paires sur les chromosomes à des positions particulières appelées loci. Les formes possibles que peuvent prendre un même gène sont appelées allèles.

Chaque caractère d'un être humain, comme le groupe sanguin ou la couleur des yeux, est déterminé par une ou plusieurs paires d'allèles tel que chaque paire est composée d'un allèle

provenant du père avec un allèle provenant de la mère. Les deux allèles se séparent durant la formation des cellules sexuelles, spermatozoïde ou ovule. Chaque cellule sexuelle a un des deux allèles avec équiprobabilité.

1.2 Polymorphisme, marqueur génétique et séquence d'ADN

Une séquence d'ADN détermine l'ordre des nucléotides qui composent un segment d'une molécule d'ADN. Il est à noter que dans ce mémoire, lorsque nous employons le terme séquence, nous sous-entendons une séquence d'ADN.

Un polymorphisme est une séquence d'ADN dont le code varie d'un individu à l'autre. Un marqueur génétique est un polymorphisme qui n'est pas forcément un gène et qui est situé à une certaine position sur le génome. Il peut être constitué d'un seul nucléotide nommé SNP (Single Nucleotide Polymorphism) ou d'une suite de nucléotides, comme dans le cas des microsatellites.

Aux chapitres IV et V, nous travaillons sur des séquences composées de marqueurs SNPs.

1.3 Mutation

Une mutation est un changement brusque dans le matériel génétique. C'est un évènement relativement rare, causé par exemple par une erreur dans la réplication. Ce sont les mutations qui font en sorte que différentes allèles existent à un locus.

Nous faisons trois hypothèses par rapport aux mutations :

1) Le MRCA (Most Recent Common Ancestor) est la séquence la plus récente à partir de laquelle toutes les séquences de l'échantillon sont directement descendantes. En fait, le MRCA constitue une référence. Nous supposons que le MRCA ne contient aucune mutation. Si par la suite une séquence descendante du MRCA a un allèle à un certain locus qui est

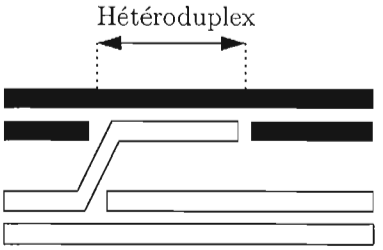


Figure 1.1 Invasion d'un brin de chromosome brisé.

différent de celui du MRCA, nous concluons que cet allèle est mutant.

2) Nous supposons que les mutations ne changent pas les chances de reproduction. Nous supposons donc qu'il n'y a pas de sélection. Ceci simplifiera nos calculs.

3) Nous travaillons avec le modèle de mutations "Infinite sites" (Kimura, 1969). Celui-ci décrit l'évolution de longues séquences d'ADN avec des taux de mutation faibles à chaque position. Nous négligeons la probabilité qu'une mutation se produise à la même position plus d'une fois. Ainsi, il y a toujours un ou deux états possibles à une position donnée dans un ensemble de séquences (un s'il n'y a jamais eu mutation à cette position et deux s'il y a eu mutation).

1.4 Recombinaison du point de vue biologique et conversion de gènes

Chez les eucaryotes, individus dont les cellules renferment un noyau, un bris accidentel d'un des deux brins d'un chromosome peut entraîner l'invasion de ce brin par le chromosome homologue. Ceci crée une région d'ADN hétéroduplex (voir figure 1.1)

Dans la région d'ADN hétéroduplex, il y a des incohérences entre les deux brins de chromosomes. Normalement, les nucléotides forment des paires de Watson-Crick (C avec G et A avec T) entre les deux brins. Ceci ne sera plus toujours le cas. Le système de réparation de l'ADN va reconnaître les incohérences et les corriger en utilisant un des deux brins comme

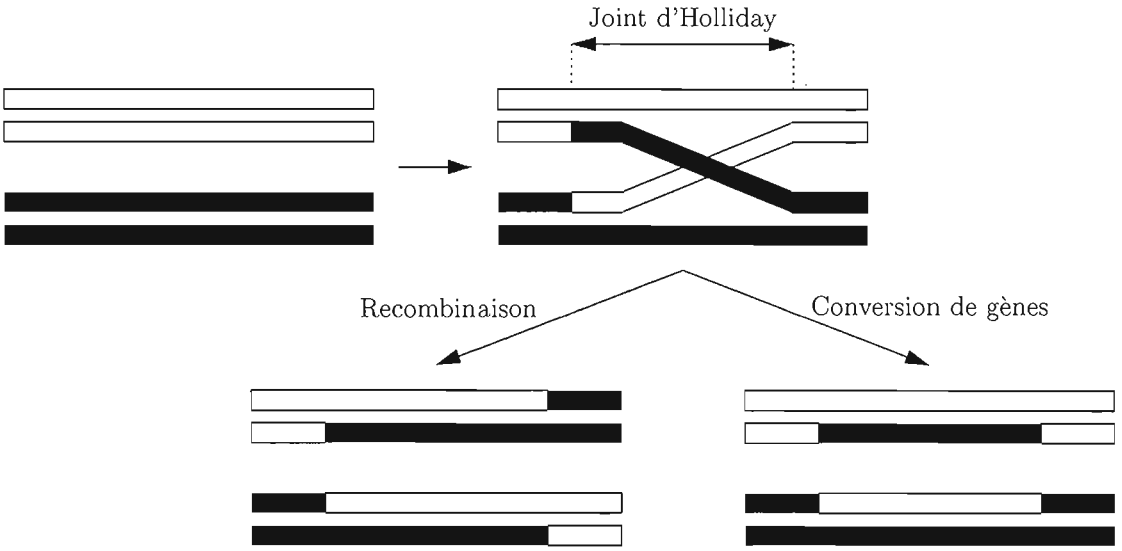


Figure 1.2 La résolution du joint d'Holliday peut mener à une recombinaison ou à une conversion de gènes.

référence pour la réparation.

Lorsque les deux brins se brisent en même temps, un joint d'Holliday se forme. Plusieurs protéines s'occupent de la maintenance et de la résolution finale du joint d'Holliday. S'il y a enjambement (mécanisme d'échange de gènes entre chromosomes homologues) lors de la résolution du joint d'Holliday, nous nommons l'évènement une recombinaison. S'il n'y a pas enjambement lors de la résolution du joint d'Holliday, nous nommons l'évènement une conversion de gènes. La figure 1.2 schématise ces deux évènements.

Les mutations et les recombinaisons sont les évènements qui créent la variation dans l'évolution. Ces deux évènements sont modélisés dans notre méthode de cartographie génétique fine.

CHAPITRE II

THÉORIE DE LA COALESCENCE

Nous introduisons d'abord le processus de coalescence de base, en décrivant le modèle de Wright-Fischer et en examinant le processus de coalescence à temps discret et à temps continu.

Par la suite, nous définissons et nous voyons comment modéliser quatre types d'évènements dans un processus de coalescence : la coalescence identique, la coalescence distincte, la mutation et la recombinaison. Ce sont ces quatre évènements que nous considérons dans notre méthode.

Finalement, nous montrons comment obtenir certains résultats probabilistes importants dans le processus de coalescence. Certains de ces résultats sont directement utiles pour notre méthode de cartographie génétique.

2.1 Le processus de coalescence de base

Le processus de coalescence est un processus stochastique introduit par Kingman (1982) et servant à décrire l'historique de la transmission de certaines séquences dans une population. Un exemple de processus de coalescence est représenté par un arbre généalogique à la figure 2.1.

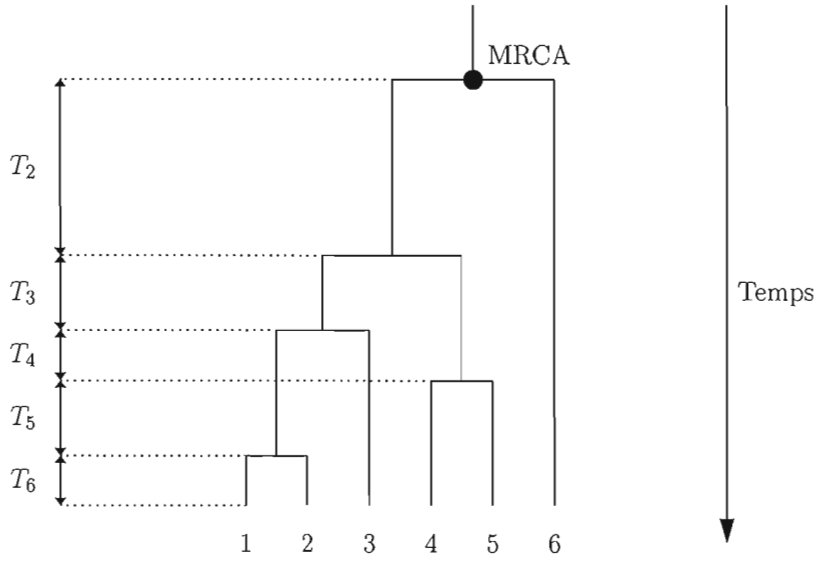


Figure 2.1 Un exemple du processus de coalescence dans un échantillon de six séquences.

Les séquences y sont numérotées de façon arbitraire de 1 à 6. Nous disons qu'il y a coalescence lorsque deux séquences sont une copie identique du même parent. Lorsqu'un tel évènement se produit, le nombre de séquences diminue de un dans l'échantillon si nous remontons dans le temps. Si nous remontons suffisamment dans le temps, jusqu'en haut du schéma, cinq coalescences ont eu lieu et nous retrouvons donc une seule séquence, le MRCA (Most Recent Common Ancestor). À la gauche du graphique, les variables T_s $(s = 2, \ldots, 6)$ représentent le temps écoulé avant qu'il y ait coalescence lorsqu'il y a s séquences dans l'échantillon.

En fait, un processus de coalescence est décrit par deux composantes : la topologie (quelles séquences coalescent ensemble) et les temps de coalescence. Si nous disposons d'un échantillon de n séquences, il est possible de simuler le processus de coalescence ayant engendré ces n séquences. À chaque fois qu'il y a coalescence parmi s séquences, la paire de séquences qui a coalescé est choisie au hasard parmi les $\binom{s}{2}$ paires possibles. Les temps de coales-

cence lorsqu'il y a s séquences, quant à eux, sont tirés d'une distribution exponentielle de paramètre $\binom{s}{2}$. Ceci découle du processus de coalescence à temps continu, une approximation du processus de coalescence à temps discret qui est obtenu à l'aide du modèle de Wright-Fisher. Examinons plus en détail ces concepts.

2.1.1 Le modèle de Wright-Fisher

Le modèle de Wright-Fisher a été introduit par Wright (1931) et Fisher (1930). Il s'agit d'un modèle simple décrivant les relations généalogiques entre des séquences. Ce modèle est encore utilisé dans de multiples contextes.

Nous supposons une population constante de $2N$ séquences à chaque génération. Cette taille de population est choisie pour faciliter la comparaison entre les modèles haploïdes et diploïdes. Ainsi, nous pouvons supposer que les séquences proviennent de $2N$ individus haploïdes ou de N individus diploïdes. Chaque séquence d'une nouvelle génération $t+1$ est une copie identique d'une des $2N$ séquences choisie au hasard à la génération précédente t. Une séquence peut donc ne pas être transmise ou être transmise une ou plusieurs fois. En fait, le nombre de fois qu'une certaine séquence est transmise est une variable aléatoire qui suit une loi binomiale de paramètres $2N$ (le nombre de descendants possibles) et $1/2N$ (la probabilité qu'un descendant soit une copie de la séquence considérée). La figure 2.2 schématise le modèle de Wright-Fisher.

Le modèle de Wright-Fisher fait appel aux hypothèses simplificatrices suivantes :

- Les générations sont discrètes et il n'y a pas d'interaction entre deux générations différentes.
- Un individu diploïde est considéré équivalent à deux individus haploïdes.
- La taille de la population est constante.
- Tous les individus ont les mêmes chances de se reproduire (pas de sélection).
- La population n'a pas de structure géographique.

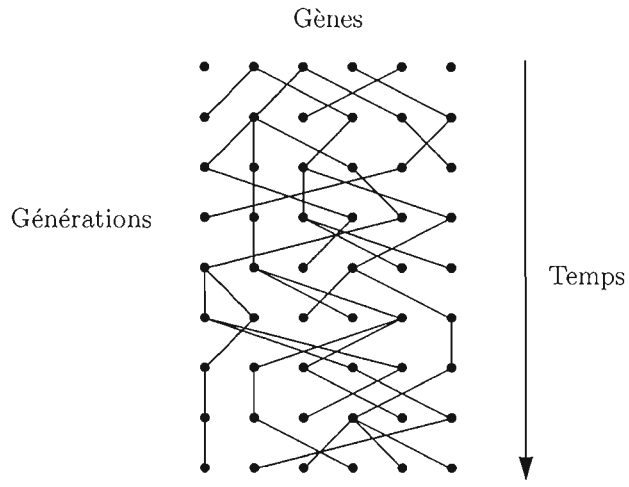


Figure 2.2 Schéma du modèle de Wright-Fisher avec six séquences et neuf générations.

– Les séquences dans la population ne se recombinent pas.

2.1.2 Le processus de coalescence à temps discret

Tel que mentionné précédemment, une coalescence a lieu lorsque deux séquences partagent un parent commun. Nous allons trouver la probabilité qu'un tel évènement se produise à une génération donnée. Tout d'abord, calculons la probabilité que pour une génération donnée, s ($\leq n$) séquences aient s ancêtres différents (c'est-à-dire qu'il n'y a pas de coalescence) sous le modèle de Wright-Fisher. Pour obtenir cette probabilité, il suffit de remarquer que la première séquence peut choisir n'importe quel parent parmi les $2N$ séquences de la génération précédente et que le choix de parents diminue de un pour les séquences subséquentes. Dans ce contexte, la probabilité qu'il n'y ait pas de coalescence pour une génération donnée est donc :

$$\frac{(2N-1)}{2N}\frac{(2N-2)}{2N}\cdots\frac{(2N-s+1)}{2N} = \prod_{i=1}^{s-1}\left(1-\frac{i}{2N}\right)$$

$$
\begin{aligned}
&= 1 - \sum_{i=1}^{s-1} \frac{i}{2N} + O\left(\frac{1}{N^2}\right) \\
&= 1 - \binom{s}{2}\frac{1}{2N} + O\left(\frac{1}{N^2}\right),
\end{aligned}
$$

où $O(1/N^2)$ représente tous les termes qui sont divisés par N^2 ou une puissance supérieure de N.

Puisque n est très petit par rapport à N, nous pouvons négliger le terme $O(1/N^2)$. Ceci est équivalent à ignorer la possibilité que plus d'une paire de séquences trouve un ancêtre commun à la même génération puisque la probabilité d'un tel évènement est aussi $O(1/N^2)$.

La probabilité qu'il n'y ait pas de coalescence est donc approximée par $1 - \binom{s}{2}/(2N)$. Son complément, $1 - \left(1 - \binom{s}{2}/(2N)\right) = \binom{s}{2}/(2N)$, approxime la probabilité qu'il y ait une et une seule coalescence.

Si T_s désigne le nombre de générations qu'il faut remonter dans le temps pour qu'une paire de séquences parmi s séquences coalesce, alors, par ce qui précède, T_s suit approximativement une loi géométrique de paramètre $\binom{s}{2}/(2N)$.

2.1.3 Le processus de coalescence à temps continu

Il existe plusieurs avantages à mesurer le temps de façon continue dans le processus de coalescence. Conceptuellement, il est plus naturel de mesurer le temps de cette façon et les calculs à l'ordinateur se font alors plus aisément. Afin de rendre le temps continu, nous définissons une unité de temps continu : elle est équivalente à $2N$ générations. En termes mathématiques, si j désigne le temps discret mesuré en générations et t désigne le temps continu, alors $t = j/(2N)$. Un avantage de définir ainsi le temps continu est que ce dernier est alors indépendant de la taille de la population.

En temps discret, puisque $T_s \sim \text{Géo}\left(\binom{s}{2}/(2N)\right)$ approximativement, $\left(1 - \binom{s}{2}/(2N)\right)$ représente la probabilité qu'il n'y ait pas coalescence pour une génération donnée. Nous obtenons donc :

$$P(T_s \geq j) = \left(1 - \frac{\binom{s}{2}}{2N}\right)^j.$$

En temps continu, puisque $j = 2Nt$,

$$\begin{aligned} P(T_s^c \geq t) &= \left(1 - \frac{\binom{s}{2}}{2N}\right)^{2Nt} \\ &\approx e^{-\binom{s}{2}t}, \qquad \text{car } \left(1 - \frac{x}{N}\right)^{Nt} \to e^{-tx} \textit{ lorsque } N \to \infty, \end{aligned}$$

où T_s^c désigne le nombre d'unités de temps continu qu'il faut remonter dans le temps pour qu'une paire de séquences parmi s séquences coalesce. Nous obtenons donc approximativement $T_s^c \sim \text{Exp}\left(\binom{s}{2}\right)$.

Puisque le processus de coalescence réfère habituellement au modèle à temps continu, dans un contexte général nous ne spécifions pas l'exposant c et T_s^c est dorénavant noté T_s.

2.2 Évènements dans le processus de coalescence

Le but de cette section est de montrer comment modéliser quatre types d'évènements dans le processus de coalescence. Afin d'arriver à ce but, nous présentons d'abord l'information considérée sur une séquence d'ADN et nous définissons les quatre évènements en question.

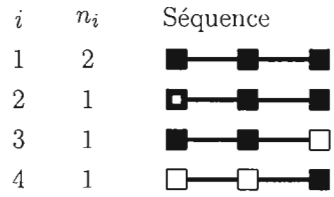


Figure 2.3 Exemple d'échantillon de séquences d'ADN : multiplicité de chaque séquence i (n_i) pour $i = 1, \ldots, 4$.

2.2.1 Information sur les séquences d'ADN

Pour appliquer notre méthode de cartographie génétique, nous aurons un échantillon de séquences d'ADN. La figure 2.3 est un exemple d'un tel échantillon. Nous y retrouvons $n = 5$ séquences de $d = 4$ types différents et de multiplicités $n_1 = 2$ et $n_2 = n_3 = n_4 = 1$. Nous remarquons que dans l'échantillon, nous avons de l'information sur les séquences d'ADN à plusieurs positions, là où se trouvent des marqueurs génétiques. Dans l'exemple, il y a trois marqueurs génétiques par séquence. Ceux-ci sont représentés par des petites boîtes carrées le long de la séquence. Dans notre méthode, de façon générale il y a L marqueurs génétiques par séquence : $L - 1$ marqueurs de positions connues et la mutation causant la maladie génétique, que nous considérons comme un marqueur génétique dont la position n'est pas connue. Nous supposons seulement qu'elle est située entre le premier et le dernier marqueur.

L'information que nous avons à chaque marqueur est très simple. Un marqueur peut être ancestral sans mutation, ancestral mutant ou non-ancestral. Un marqueur ancestral sans mutation est un marqueur qui n'a pas subi de mutation après l'ancêtre commun. Il est important de se rappeler que nous travaillons avec le modèle de mutations "Infinite sites" et donc qu'une mutation ne peut se produire qu'une seule fois en une position donnée. Ainsi, un marqueur ancestral sans mutation nous indique qu'il n'y a jamais eu mutation

alors qu'un marqueur ancestral mutant nous indique qu'il y a eu une et une seule mutation. Dans notre méhode, l'échantillon initial ne comprend que des marqueurs ancestraux, mutants ou non mutants. Lorsqu'un marqueur est non-ancestral, nous ignorons s'il s'agit d'un marqueur mutant ou non. Nous verrons dans la prochaine sous-section que les recombinaisons introduisent des marqueurs non-ancestraux, alors que les coalescences distinctes tendent à les faire disparaître.

Dans ce mémoire, une convention consiste à représenter graphiquement un marqueur ancestral sans mutation par un carré noir, un marqueur ancestral mutant par un carré noir avec un carré blanc à l'intérieur et un marqueur non-ancestral par un carré blanc (voir figure 2.3, par exemple). En langage informatique ou dans un texte, les trois sortes de marqueurs sont plutôt représentés par un caractère : "0" pour un marqueur ancestral sans mutation, "1" pour un marqueur ancestral mutant et "−" pour un marqueur non-ancestral.

2.2.2 Définition des quatre types d'évènements considérés

- **La coalescence identique C_i**

  La coalescence identique C_i se produit lorsque deux séquences identiques de type i trouvent un ancêtre commun qui est aussi de type i. En remontant dans le temps, pour que cet évènement se réalise, il faut donc nécessairement que la multiplicité du type de séquence impliqué soit supérieure à un. Il est aussi bon de noter que, si nous remontons dans le temps vers l'ancêtre commun, l'évènement C_i diminue de un le nombre de séquences en diminuant de un le nombre de séquences de type i. La figure 2.4 (a) illustre la coalescence identique C_1 des deux séquences de type 1 de la figure 2.3.

- **La coalescence distincte C_{ij}^k, avec $i \neq j$**

  La coalescence distincte C_{ij}^k se produit lorsque deux séquences de types différents i et j trouvent un ancêtre commun de type k. Le fait que deux séquences "compatibles", mais non identiques puissent avoir un ancêtre commun est rendu possible grâce aux marqueurs

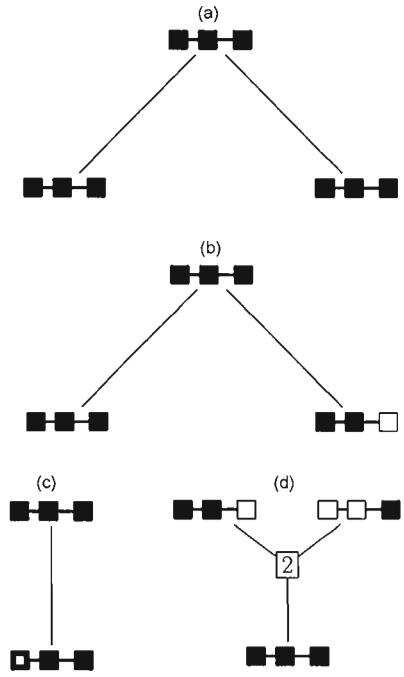


Figure 2.4 Exemple d'évènement de : (a) coalescence identique, (b) coalescence distincte, (c) mutation et (d) recombinaison.

non-ancestraux. Ceux-ci, contrairement aux marqueurs ancestraux, ne posent pas de restrictions sur les possibilités de coalescence. Par exemple, les deux séquences $(0, -, 1)$ et $(-, 0, 1)$ pourraient coalescer pour donner l'unique parent $(0, 0, 1)$, et cela même si l'information n'est pas la même d'une séquence à l'autre pour le premier et pour le second marqueur. En remontant dans le temps vers l'ancêtre commun, l'évènement C_{ij}^{k} diminue de un le nombre de séquences. Pour ce faire, le nombre de séquences de type i ainsi que le nombre de séquences de type j sont diminués de un et le nombre de séquences de type k est augmenté de un. La figure 2.4 (b) illustre la coalescence distincte C_{13}^{1} des séquences de type 1 et 3 de la figure 2.3. Cela produit une séquence de type 1.

- **La mutation** $M_i^j(m)$

  En remontant dans le temps jusqu'à l'ancêtre commun, il faut enlever toutes les mutations, car nous supposons que le MRCA est une séquence ne contenant aucune mutation. Or, selon notre modèle de mutations utilisé, une mutation survient en une position au plus une fois. Ainsi, pour enlever une mutation au marqueur m, il faut qu'il y ait une seule séquence qui possède cette mutation à ce marqueur. Un évènement de mutation ne change pas le nombre de séquences, seul le type d'une séquence est modifié. La figure 2.4 (c) illustre la mutation $M_1^2(1)$ au premier marqueur de la séquence de type 2 de la figure 2.3. Cela produit une séquence de type 1.

- **La recombinaison** $R_i^{jk}(p)$

  La recombinaison $R_i^{jk}(p)$ se produit lorsqu'une séquence de type i descend de deux parents de types j et k et que le point de recombinaison est dans l'intervalle p. Par convention, cet intervalle se situe entre le $p^{\text{ème}}$ et le $(p+1)^{\text{ème}}$ marqueur. Chacun des deux parents donne une partie du matériel génétique à l'enfant d'une façon bien précise : un des parents fournit le matériel génétique à la gauche du point de recombinaison et l'autre parent fournit le matériel génétique à la droite du point de recombinaison. Lorsqu'un évènement de recombinaison survient, des marqueurs non-ancestraux sont toujours introduits. Par exemple, une recombinaison de la séquence $(0,1,0)$ donne les parents $(0,-,-)$ et $(-,1,0)$ ou $(0,1,-)$ et $(-,-,0)$ lorsque le point de recombinaison est respectivement dans le premier ou le second intervalle. En remontant dans le temps vers l'ancêtre commun, l'évènement $R_i^{jk}(p)$ augmente de un le nombre de séquences. Pour ce faire, le nombre de séquences de type i est diminué de un et le nombre de séquences de types j ainsi que le nombre de séquences de type k sont augmentés de un. La figure 2.4 (d) illustre la recombinaison $R_1^{34}(2)$ d'une séquence de type 1 de la figure 2.3 avec un point de recombinaison choisi dans le deuxième intervalle. Cela produit les séquences 3 et 4.

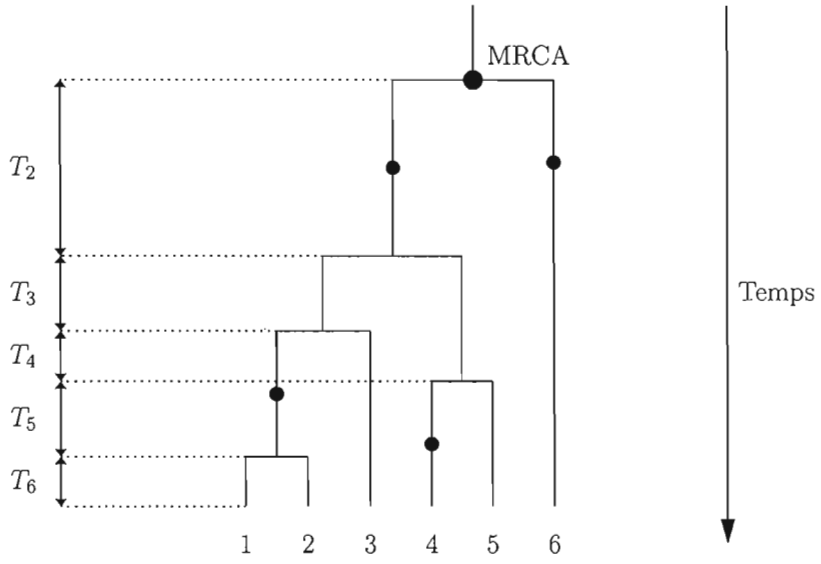


Figure 2.5 Un exemple du processus de coalescence avec mutations dans un échantillon de six séquences. Les petits points noirs y désignent des mutations.

2.2.3 Le processus de coalescence avec mutations

Les mutations se modélisent facilement dans le processus de coalesence. La raison en est que les évènements de mutations ne font que modifier le type d'une séquence. Ainsi, la représentation d'un processus de coalescence avec mutations demeure un arbre généalogique. La figure 2.5 en est un exemple. Il s'agit de la même généalogie que celle de la figure 2.1, à la seule différence que les mutations y sont indiquées à l'aide de petits points noirs.

2.2.4 Le processus de coalescence avec recombinaisons

Il ne reste donc plus qu'à inclure les recombinaisons dans le processus de coalescence pour que les quatre évènements décrits à la sous-section 2.2.2 y soient modélisés. Toutefois, à la base, le processus de coalescence est un modèle qui n'inclut pas les évènements de recombi-

naison. Cependant, puisque la recombinaison est fondamentale dans la transmission d'ADN d'une génération à l'autre chez la plupart des organismes vivants, il est particulièrement important de la considérer pour développer une méthode de cartographie génétique. Le processus de coalescence avec recombinaison est donc une extension importante du processus de coalescence de base. Nous nous servons de cette extension pour modéliser les quatre types d'évènements qui nous intéressent.

Hudson (1983) a d'abord présenté un modèle de base permettant de modéliser la recombinaison sans tenir compte de ses détails biologiques complexes. Par la suite, Griffiths et Marjoram (1996) ont décrit le graphe de recombinaison ancestral (ARG), qui permet également de modéliser les évènements de recombinaison. Il s'agit d'une extension de la coalescence de base couramment utilisée. La figure 2.6 est un exemple d'ARG. Il décrit la généalogie qui relie un échantillon de quatre séquences d'ADN à quatre marqueurs à l'ancêtre commun le plus récent (MRCA). Il est à noter que les carrés noirs, les carrés noirs avec un cercle blanc à l'intérieur et les carrés blancs y représentent respectivement un marqueur ancestral sans mutation, un marqueur ancestral mutant et un marqueur non-ancestral.

Les quatre évènements décrits à la sous-section 2.2.2 sont représentés dans l'ARG : une coalescence identique ou distincte lorsque deux séquences descendent du même parent, une mutation lorsque le matériel génétique à un marqueur en particulier est changé et une recombinaison lorsqu'une séquence descend de deux parents. Dans ce dernier cas, un point de recombinaison est choisi dans un des trois intervalles entre deux marqueurs consécutifs (dans l'ARG de la figure 2.6, le numéro de cet intervalle est indiqué pour chaque recombinaison). Les marqueurs à la gauche du point de recombinaison sont des copies identiques des marqueurs du parent "gauche" et les marqueurs à la droite du point de recombinaison sont des copies identiques des marqueurs du parent "droit". Il y a deux remarques intéressantes à faire sur l'ARG.

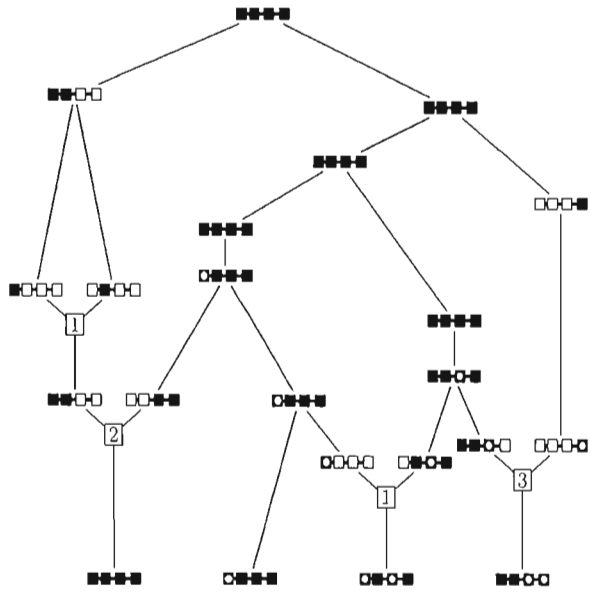


Figure 2.6 Un exemple d'ARG (tiré de Larribe *et al.*, 2002).

Premièrement, puisqu'avec un évènement de recombinaison il est maintenant possible que le nombre de séquences augmente, nous pourrions nous demander s'il est possible que nous n'atteignions jamais l'ancêtre commun. Afin de répondre à cette question, il faut comparer la fréquence des évènements de recombinaison (qui augmentent de un le nombre de séquences) à la fréquence des évènements de coalescence (qui diminuent de un le nombre de séquences). À la prochaine section, nous verrons que le nombre d'unités de temps continu qu'il faut remonter dans le temps avant le premier évènement de recombinaison suit une loi exponentielle de paramètre $n\rho/2$ (où ρ désigne le taux de recombinaison à l'échelle) alors que le nombre d'unités de temps continu qu'il faut remonter dans le temps avant le premier évènement de coalescence suit une loi exponentielle de paramètre $n(n-1)/2$. Puisque les évènements de recombinaison surviennent à un taux linéaire alors que les évènements de coalescence surviennent à un taux quadratique, le nombre de séquences demeurera toujours

fini et nous finirons toujours par trouver l'ancêtre commun.

Deuxièmement, dans un ARG chaque séquence d'ADN a un ou deux parents dans la génération précédente (deux s'il y a recombinaison et un sinon). C'est pour cela que nous obtenons un graphe et non un arbre. Cependant, si nous considérons un seul marqueur par séquence, nous éliminons les évènements de recombinaison puisqu'une recombinaison n'est pas possible à l'intérieur d'un marqueur. En représentant la généalogie des séquences à un marqueur, nous obtenons donc un arbre. En fait, il s'agit d'un arbre partiel qui est inclus dans l'ARG. Les arbres partiels pour chacun des quatre marqueurs de la figure 2.6 sont mis en évidence à la figure 2.7.

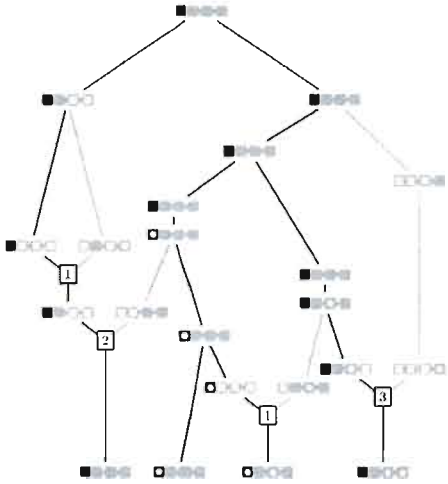
2.3 Paramètres et résultats importants du processus de coalescence

Dans cette section, nous montrons comment obtenir certains résultats probabilistes importants dans le processus de coalescence. Nous obtenons d'abord plusieurs résultats par rapport au temps requis pour passer de l'échantillon initial à l'ancêtre commun. Ensuite, nous calculons les probabilités qu'un évènement qui s'est produit dans le processus de coalescence soit une coalescence, une mutation ou une recombinaison. Les calculs que nous effectuons nécessitent que nous introduisions plusieurs paramètres génétiques. Ainsi, avant d'aborder ces calculs, nous présentons les paramètres génétiques utilisés dans cette section. Du coup, nous en profitons également pour présenter les paramètres génétiques utilisés dans les autres sections.

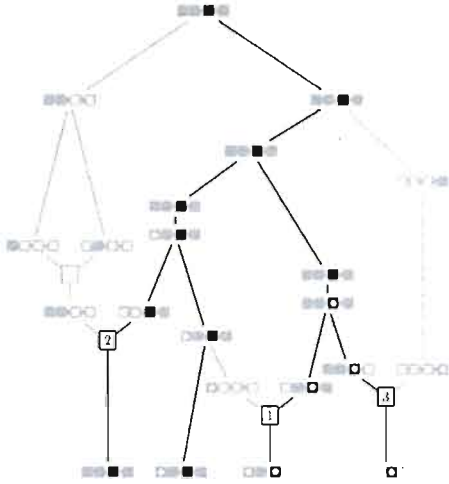
2.3.1 Paramètres génétiques

Tout au long du mémoire, nous utilisons plusieurs paramètres génétiques que nous présentons maintenant.

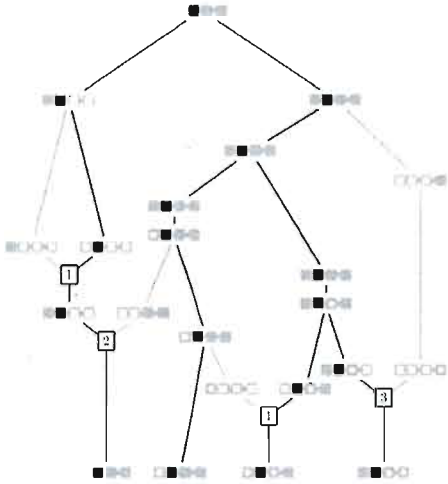
Nous utilisons deux paramètres génétiques pour la taille de la population. La taille réelle



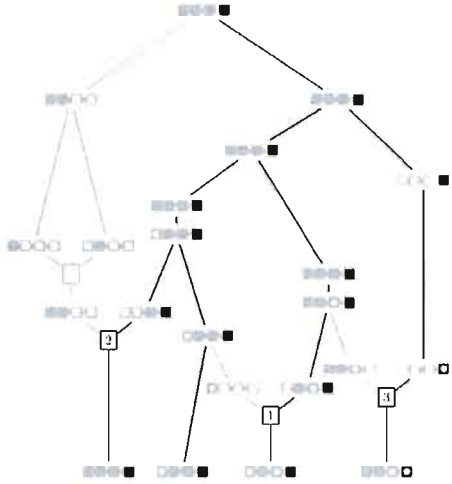
(a) Arbre partiel pour le marqueur 1



(c) Arbre partiel pour le marqueur 3



(b) Arbre partiel pour le marqueur 2



(d) Arbre partiel pour le marqueur 4

Figure 2.7 Arbres partiels pour l'ARG de la figure 2.6 (tiré de Larribe *et al.*, 2002).

de la population est notée N, alors que la taille effective de la population est notée N_e. Cette dernière correspond à la taille que la population doit avoir pour qu'elle se reproduise selon le modèle de Wright-Fisher (voir sous-section 2.1.1). Utiliser la taille réelle de la population dans ce modèle n'a pas de sens (Hein *et al.*, 2005, par exemple). Comme taille de population, nous utilisons N lorsque nous présentons de la théorie et N_e dans le cadre d'une méthode d'estimation de vraisemblance.

Nous considérons également deux paramètres génétiques pour le taux de mutation. Tout d'abord, le taux de mutation par séquence par génération est noté μ. Il s'agit de la probabilité qu'il y ait mutation sur une séquence donnée pour une génération quelconque. Il est intéressant de remarquer que, s'il y a L marqueurs par séquence et que nous supposons que les mutations surviennent à chacun des marqueurs avec équiprobabilité, alors μ/L est la probabilité qu'il y ait mutation à un marqueur particulier d'une séquence donnée pour une génération quelconque. Le taux de mutation à l'échelle, quant à lui, est noté θ. Il s'agit d'un paramètre qui tient compte du taux de mutation et de la taille réelle ou effective de la population. Tout dépendant du paramètre génétique choisi pour la taille de la population, nous le calculons à l'aide de l'une de ces deux expressions :

$$\theta = 4N\mu \quad \text{ou} \quad \theta = 4N_e\mu.$$

En génétique mathématique, le facteur $4N$ (ou $4N_e$) est préféré à $2N$ (ou $2N_e$) pour mettre à l'échelle μ, parce que cela permet de simplifier plusieurs formules.

Puisque le but de notre méthode est d'estimer la position d'une mutation, nous utilisons des paramètres génétiques de position. La position de la mutation est notée r_T. Ce paramètre désigne la distance génétique en morgans entre le premier marqueur de la séquence et le marqueur de mutation. La distance génétique entre deux marqueurs est définie par la probabilité, par génération, d'obtenir une recombinaison entre ces marqueurs. La distance entre deux marqueurs consécutifs est donnée par r_p. Il s'agit en fait de la distance

génétique en morgans entre le $p^{\text{ème}}$ et le $(p+1)^{\text{ème}}$ marqueur. S'il y a L marqueurs par séquence, que nous connaissons $r_1, r_2, \ldots, r_{L-1}$ et que r désigne la distance entre le premier et le dernier marqueur, nous pouvons trouver $r = \sum_{p=1}^{L-1} r_p$ si nous supposons que les distances génétiques sont additives (c'est-à-dire qu'il n'y a pas d'interférence génétique). C'est une hypothèse vraisemblable lorsque les distances génétiques sont petites. En effet, dans ce cas-là, les distances génétiques sont approximativement proportionnelles aux distances physiques. Puisque les distances physiques sont évidemment additives, alors les distances génétiques le sont aussi approximativement. Comme nous travaillons avec de petites distances génétiques, nous appliquons l'hypothèse d'additivité des distances génétiques. La distance génétique entre le premier et le dernier marqueur, r, correspond à la probabilité qu'il y ait recombinaison pour une séquence donnée et une génération quelconque. Ce paramètre est le taux de recombinaison par séquence par génération. Nous pouvons mettre ce taux à l'échelle et nous obtenons ρ, le taux de recombinaison à l'échelle. Il s'agit d'un paramètre qui tient compte du taux de recombinaison et de la taille réelle ou effective de la population. Tout dépendant du paramètre génétique choisi pour la taille de la population, nous le calculons à l'aide de l'une de ces deux expressions :

$$\rho = 4Nr \quad \text{ou} \quad \rho = 4N_e r.$$

2.3.2 Quelques résultats probabilistes importants du processus de coalescence

Une fois les paramètres génétiques définis, nous pouvons maintenant effectuer plusieurs calculs dans le cadre du processus de coalescence. Nous obtenons d'abord l'espérance et la variance du temps requis pour passer de l'échantillon initial au MRCA. Ensuite, nous calculons les probabilités qu'un évènement qui s'est produit dans le processus de coalescence soit une coalescence, une mutation ou une recombinaison.

Commençons donc par l'étude du temps requis, en unités de temps continu (une unité

de temps continu équivaut à $2N$ générations, voir sous-section 2.1.3), pour passer de l'échantillon initial à l'ancêtre commun. Nous notons cette quantité T_{MRCA}. Si nous travaillons dans le cas général où l'échantillon initial comporte n séquences, nous obtenons alors :

$$T_{MRCA} = \sum_{s=2}^{n} T_s.$$

En effet, cette quantité est la somme des temps écoulés avant qu'il y ait coalescence lorsqu'il y a s séquences dans l'échantillon, T_s, pour $s = 2, \ldots, n$. Le lecteur peut se référer à la figure 2.5 pour une représentation des variables T_s avec un échantillon initial de $n = 6$ séquences.

L'espérance du temps avant de remonter au MRCA est :

$$\begin{aligned} E(T_{MRCA}) &= E\left(\sum_{s=2}^{n} T_s\right) \\ &= \sum_{s=2}^{n} E(T_s) \\ &= \sum_{s=2}^{n} \frac{2}{s(s-1)}, \qquad \text{car } T_s \sim \text{Exp}(s(s-1)/2) \\ &= 2\sum_{s=2}^{n} \left(\frac{1}{s-1} - \frac{1}{s}\right) \\ &= 2\left(1 - \frac{1}{n}\right). \end{aligned}$$

D'une part, nous constatons que le temps moyen avant de remonter au MRCA est inférieur à 2. D'autre part, nous remarquons que $E(T_2) = 1$, car $T_2 \sim \text{Exp}(1)$. Cela signifie qu'en moyenne l'échantillon comporte seulement deux séquences pendant plus de la moitié du temps total.

La variance de T_{MRCA} se calcule de la manière suivante :

$$\begin{aligned} Var(T_{MRCA}) &= Var\left(\sum_{s=2}^{n} T_s\right) \\ &= \sum_{s=2}^{n} Var(T_s), \qquad \text{par indépendance des } T_s \\ &= 4\sum_{s=2}^{n} \frac{1}{s^2(s-1)^2}, \qquad \text{car } T_s \sim \text{Exp}(s(s-1)/2). \end{aligned}$$

Il est possible de montrer que $\lim_{n\to\infty} Var(T_{MRCA}) = 4\pi^2/3 - 12$ (Hein *et al.*, 2005, par exemple). Nous remarquons aussi que $Var(T_2) = 1$, car $T_2 \sim \text{Exp}(1)$. Cela signifie que la majorité de la variance de T_{MRCA}, qui est la somme des variances des T_s, provient de celle de T_2.

Passons maintenant aux calculs de probabilités d'évènements. Si T_n désigne le nombre d'unités de temps continu qu'il faut remonter dans le temps pour qu'une paire de séquences parmi n séquences coalesce, nous avons vu (voir sous-section 2.1.3) que $T_n \sim \text{Exp}\,(n(n-1)/2)$ approximativement. Il faut également trouver les lois des variables T_m et T_r qui représentent respectivement le nombre d'unités de temps continu qu'il faut remonter dans le temps avant le premier évènement de mutation ou de recombinaison.

Nous allons d'abord trouver les lois de T_{m1} et T_{r1} (le nombre d'unités de temps continu avant le premier évènement de mutation ou de recombinaison à une séquence donnée). Puisque μ, le taux de mutation par séquence par génération, est la probabilité qu'il y ait mutation sur une séquence donnée pour une génération quelconque, il en découle que $T_{m1} \sim \text{Géo}(\mu)$. Ainsi, $P(T_{m1} \geq j) = (1-\mu)^j$ car il faut alors qu'il n'y ait pas mutation pendant au moins j générations. En temps continu ($t = j/(2N)$),

$$\begin{aligned} P(T_{m1} \geq t) &= (1-\mu)^{2Nt} \\ &= \left(1 - \frac{\theta/2}{2N}\right)^{2Nt} \\ &\approx e^{-\frac{\theta}{2}t}, \qquad \text{car } \left(1 - \frac{x}{N}\right)^{Nt} \to e^{-tx} \text{ lorsque } N \to \infty. \end{aligned}$$

Donc $T_{m1} \sim \text{Exp}\,(\theta/2)$ approximativement, si la population est assez grande. S'il y a n séquences, nous avons $T_m \sim \text{Exp}\,(n\theta/2)$ approximativement puisque les séquences évoluent de façon indépendante les unes des autres. Ceci découle du théorème stipulant que si $U \sim \text{Exp}(a)$ et $V \sim \text{Exp}(b)$ avec U et V indépendants, alors $\min(U, V) \sim \text{Exp}(a+b)$.

Par un raisonnement similaire, nous trouvons approximativement que $T_{r1} \sim \text{Exp}(\rho/2)$ et $T_r \sim \text{Exp}(n\rho/2)$. En résumé, nous avons :

$$T_n \sim \text{Exp}\left(\frac{n(n-1)}{2}\right),\ T_m \sim \text{Exp}\left(\frac{n\theta}{2}\right),\ T_r \sim \text{Exp}\left(\frac{n\rho}{2}\right).$$

Ces trois distributions exponentielles sont indépendantes et donc le temps avant le prochain évènement suit une loi exponentielle de paramètre

$$\frac{n(n-1)}{2} + \frac{n\theta}{2} + \frac{n\rho}{2} = \frac{n(n-1+\theta+\rho)}{2}.$$

Ainsi, les probabilités d'une coalescence, d'une mutation ou d'une recombinaison lorsqu'un de ces trois évènements survient sont :

$$P(\text{Co} \mid \cdot) = \frac{n(n-1)/2}{n(n-1+\theta+\rho)/2} = \frac{n-1}{n-1+\theta+\rho},$$

$$P(\text{Mu} \mid \cdot) = \frac{n\theta/2}{n(n-1+\theta+\rho)/2} = \frac{\theta}{n-1+\theta+\rho},$$

$$P(\text{Re} \mid \cdot) = \frac{n\rho/2}{n(n-1+\theta+\rho)/2} = \frac{\rho}{n-1+\theta+\rho},$$

où "$\cdot$" signifie qu'un évènement de coalescence, de mutation ou de recombinaison a eu lieu. Ces probabilités nous sont utiles pour construire l'équation de récurrence de Griffiths-Tavaré utilisée dans notre méthode.

CHAPITRE III

MÉTHODES D'ESTIMATION DE LA VRAISEMBLANCE

Le but des méthodes des chapitres IV et V est d'estimer la position (r_T) d'une mutation causant une maladie génétique. Pour ce faire, il s'agit d'évaluer la vraisemblance de r_T en utilisant le processus de coalescence avec recombinaison pour modéliser l'histoire des séquences. Les méthodes des chapitres IV et V sont donc des méthodes d'estimation de la vraisemblance qui sont appliquées pour estimer le paramètre génétique (r_T).

Dans ce chapitre, nous nous intéressons à plusieurs méthodes d'estimation de la vraisemblance de divers paramètres génétiques. Nous commençons par définir la vraisemblance des paramètres génétiques. Ensuite, nous voyons les formules exactes pour la calculer lorsque nous travaillons avec les généalogies et lorsque nous travaillons avec les histoires. Dans ce dernier cas, nous utilisons alors les équations de Griffiths-Tavaré, équations de récurrence qui expriment la vraisemblance en fonction de la vraisemblance pour des ensembles de données qui ont résulté d'un évènement évolutif de moins. Pour les deux méthodes, nous voyons comment calculer une approximation de la vraisemblance en utilisant l'échantillonnage pondéré (importance sampling). Par la suite, nous examinons d'autres façons d'estimer la vraisemblance. Nous voyons d'abord une façon alternative d'utiliser l'échantillonnage pondéré (Fearnhead et Donnelly, 2001). Ensuite, nous étudions une façon de synthétiser les données pour sauver du temps de calcul (Fearnhead et Donnelly, 2002). Nous en profitons pour mentionner comment, dans ce cas, l'estimateur de vraisemblance

composite correspondant peut être obtenu. Nous terminons la section en présentant un autre estimateur de vraisemblance composite, tiré d'Hudson (2001).

3.1 Définition de la vraisemblance des paramètres génétiques

Tout d'abord, afin de définir la vraisemblance des paramètres génétiques, il est nécessaire de connaître le principe sur lequel s'appuient les méthodes d'inférence de vraisemblance à partir de données. Ce principe est que les données observées proviennent d'un processus aléatoire quelconque (ou modèle) dont certaines quantités (des paramètres génétiques) sont inconnues. Il est alors naturel de définir la vraisemblance des paramètres génétiques comme la probabilité d'observer les données D si les paramètres du modèle prennent la valeur ψ :

$$L(\psi) = P(D|\psi).$$

Examinons maintenant comment calculer $P(D|\psi)$ lorsque nous travaillons avec les généalogies et aussi lorsque nous travaillons avec les histoires.

3.2 Calcul de la vraisemblance avec les généalogies

Lorsque nous considérons les généalogies pour calculer la vraisemblance, nous travaillons directement avec le processus de coalescence de base. Nous tenons compte de la topologie (quelles séquences coalescent ensemble) et des temps de coalescence (voir section 2.1). Puisque les temps de coalescence sont des variables continues, il existe une infinité de généalogies. La formule théorique de la vraisemblance implique donc une intégration sur les généalogies. Par exemple, la vraisemblance des paramètres génétiques N_e et μ est donnée par :

$$P(D|N_e, \mu) = \int_{G'} f(G'|N_e) P(D|G', \mu) dG',$$

où G' est une généalogie quelconque. Il s'agit en fait d'une moyenne, sur l'infinité de généalogies possibles, des vraisemblances selon la généalogie considérée ($P(D|G', \mu)$), pondérées par la fonction de densité de cette généalogie ($f(G'|N_e)$). Nous allons voir comment calculer les deux termes de la formule.

Le calcul de $f(G'|N_e)$, la densité d'une généalogie, implique d'une part, les probabilités que les bonnes séquences aient coalescé ensemble et d'autre part, les fonctions de densité des temps de coalescence. S'il y a n séquences, nous devons considérer ces probabilités pour $2, 3, \ldots, n$ séquences. Nous avons déjà vu que $T_s \sim \text{Exp}\,(s(s-1)/2)$ approximativement, où T_s désigne le nombre d'unités de temps continu qu'il faut remonter dans le temps pour qu'une paire de séquences parmi s séquences coalesce (voir sous-section 2.1.3). Nous allons maintenant considérer le temps en générations. Rappelons qu'une unité de temps continu correspond à $2N_e$ générations. En appliquant le changement d'échelle, il suffit de multiplier les temps par $2N_e$. Ainsi, le nombre de générations qu'il faut remonter dans le temps pour qu'une paire de séquences parmi s séquences coalesce suit une loi exponentielle de paramètre

$$\frac{s(s-1)}{2} \cdot \frac{1}{2N_e} = \frac{s(s-1)}{4N_e}.$$

Pour appliquer le changement d'échelle, il faut diviser par $2N_e$ le paramètre de l'exponentielle puisque maintenant, en moyenne, nous avons besoin de plus d'unités de temps avant qu'il y ait coalescence et que la moyenne d'une exponentielle est l'inverse de son paramètre. Aussi, la probabilité que deux séquences i et j (où i peut être égal à j) coalescent ensemble, étant donné qu'un évènement de coalescence a eu lieu, est :

$$P(C_{ij}^k \mid Co) = \frac{1}{\binom{s}{2}} = \frac{2}{s(s-1)},$$

puisqu'il y a une chance sur le nombre de paires de séquences possibles que deux séquences en particulier coalescent. Pour calculer $f(G'|N_e)$, il suffit simplement de multiplier les fonctions de densité des temps de coalescence lorsqu'il y a $2, 3, \ldots, n$ séquences, pondérées par la probabilité que les bonnes séquences coalescent. Ainsi,

$$\begin{aligned} f(G'|N_e) &= \prod_{s=2}^{n} P(C_{ij}^{k} \mid Co) \cdot g_{T_s'}(t_s') \\ &= \prod_{s=2}^{n} \frac{2}{s(s-1)} \cdot \frac{s(s-1)}{4N_e} \exp\left(\frac{-s(s-1)}{4N_e} t_s'\right) \\ &= \prod_{s=2}^{n} \frac{2}{4N_e} \exp\left(\frac{-s(s-1)}{4N_e} t_s'\right), \end{aligned}$$

où $g_{T_s'}(t_s')$ est la fonction de densité de la variable qui représente le nombre de générations qu'il faut remonter dans le temps avant qu'il y ait une coalescence parmi les s séquences.

Pour combiner les deux paramètres N_e et μ, nous changeons l'échelle de temps et nous exprimons les temps en unités de $1/\mu$ génération. Nous multiplions les temps par μ, divisons par μ le paramètre de l'exponentielle et $f(G'|N_e)$ devient :

$$g(G|\theta) = \prod_{s=2}^{n} \frac{2}{\theta} \exp\left(\frac{-s(s-1)}{\theta} t_s\right),$$

où $\theta = 4N_e\mu$. Ainsi, la fonction de vraisemblance devient :

$$P(D|\theta) = \int_G g(G|\theta) P(D|G) dG. \tag{3.1}$$

Afin d'approximer cette intégrale, nous pourrions utiliser l'approche Monte-Carlo classique. Cette approche permet d'estimer l'espérance d'une fonction $g(X)$ d'une variable continue X par la moyenne de M réalisations de cette fonction :

$$\begin{aligned}\int f(x)g(x)dx &= E_f(g(X)) \\ &\approx \frac{1}{M}\sum_{i=1}^{M} g\left(X^{(i)}\right).\end{aligned}$$

Si nous appliquons la méthode de Monte-Carlo à notre problème, nous obtenons :

$$\begin{aligned}P(D|\theta) &= \int_G g(G|\theta)P(D|G)dG \\ &= E_g(P(D|G)) \\ &\approx \frac{1}{M}\sum_{i=1}^{M} P(D|G_i).\end{aligned}$$

Dans notre cas, il y aurait un problème à appliquer directement l'approche Monte-Carlo. Seulement un petit nombre de termes $P(D|G_i)$ contribueraient de façon significative à la somme. La méthode serait inefficace. Afin de remédier à ce problème, nous pouvons utiliser l'échantillonnage pondéré (importance sampling). Il s'agit d'une méthode qui cherche à concentrer la simulation sur les généalogies les plus importantes, celles qui sont les plus compatibles avec les données. Pour ce faire, il suffit simplement de multiplier et de diviser par une distribution appelée la distribution proposée. Dans notre cas, nous choisissons la distribution proportionnelle à $g(G|\theta_0)P(D|G)$ comme distribution proposée. Dans cette expression, θ_0 est une valeur conductrice de θ. Il s'agit d'une valeur de θ vraisemblable a priori. En appliquant l'échantillonnage pondéré, l'intégration Monte-Carlo donne maintenant :

$$\begin{aligned}P(D|\theta) &= \int_G \frac{g(G|\theta)P(D|G)}{g(G|\theta_0)P(D|G)} g(G|\theta_0)P(D|G)dG \\ &= \int_G \frac{g(G|\theta)}{g(G|\theta_0)} g(G|\theta_0)P(D|G)dG \\ &= E\left(\frac{g(G|\theta)}{g(G|\theta_0)}\right)\end{aligned}$$

$$\approx \frac{1}{M}\sum_{i=1}^{M}\frac{g(G_i|\theta)}{g(G_i|\theta_0)}.$$

L'échantillonnage est maintenant plus efficace. Il est possible d'échantillonner les G_i d'une distribution proportionnelle à $g(G|\theta_0)P(D|G)$ en utilisant l'algorithme Metropolis-Hastings de la méthode MCMC, tel que proposé par Felsenstein *et al.* (1999).

3.3 Calcul de la vraisemblance avec les histoires

Les équations de Griffiths-Tavaré (1994) se prêtent bien au calcul de la vraisemblance avec les histoires. Une histoire, notée (H), est une suite d'évènements permettant de faire la transition entre l'échantillon de départ et l'ancêtre commun. Une histoire peut donc être représentée par un ARG. À la différence d'une généalogie, nous ne nous intéressons pas aux temps des évènements dans une histoire. La figure 3.1 donne un exemple d'histoire. Celle-ci est composée de trois coalescences et de deux mutations qui permettent de passer de l'échantillon initial à l'ancêtre commun. Si nous considérons des histoires où il n'y a pas d'évènements de recombinaison, il y a un nombre fini d'histoires qui mènent à un ensemble de données particulier. La formule de la vraisemblance d'un ou de plusieurs paramètres génétiques (θ par exemple) implique alors une sommation :

$$P(D|\theta) = \sum_{H} P(D|H)P(H|\theta).$$

Remarquons que cette formule est analogue à (3.1). Le terme $P(D|H)$ est égal à 1 si l'histoire est compatible avec les données ; sinon, il est égal à 0. Nous nous intéressons seulement aux histoires compatibles avec les données et donc $P(D|H)$ est toujours égal à 1. L'autre terme, $P(H|\theta)$, est le produit des probabilités des évènements de H multiplié par la probabilité que la séquence initiale d'ADN soit telle qu'elle est. Dans notre exemple,

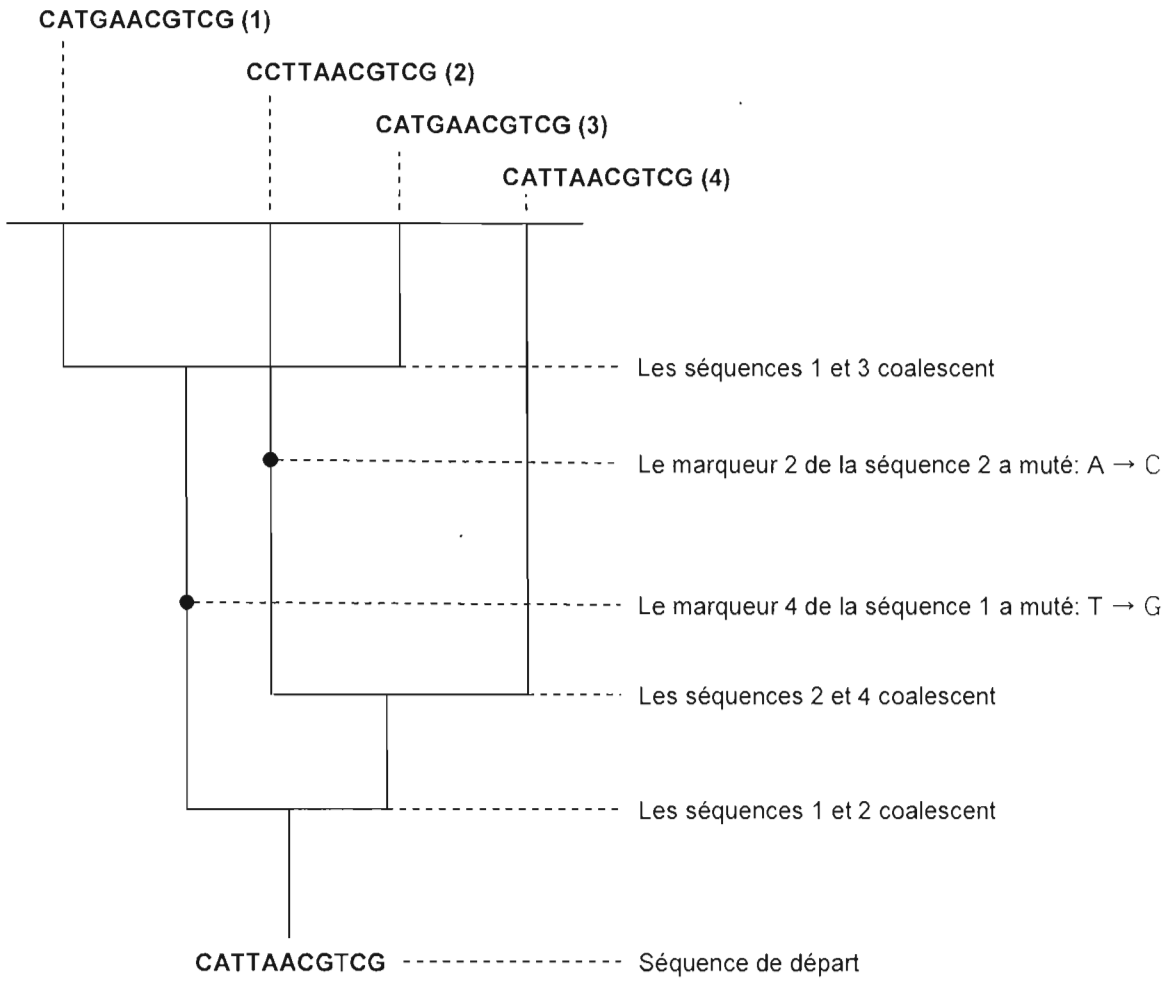


Figure 3.1 Exemple d'une histoire (tiré de Felsenstein *et al.*, 1999).

$P(H|\theta)$ est le produit des probabilités des cinq évènements de l'histoire multiplié par la probabilité d'obtenir "CATTAACGTCG" comme séquence unique. Nous verrons au chapitre IV comment utiliser les équations de Griffiths-Tavaré pour calculer les probabilités qu'un évènement particulier se produise. En supposant l'indépendance entre les nucléotides et l'équiprobabilité de chacune des sortes de nucléotides, la probabilité d'obtenir "CATTAACGTCG" comme séquence unique est quant à elle égale à $(1/4)^{11}$ puisque qu'il y a quatre nucléotides possibles à chaque position et que la séquence en contient 11.

Puisque le nombre d'histoires est fini, il est théoriquement possible de calculer de façon exacte la vraisemblance. C'est ce qui est fait dans la méthode du chapitre V. Pour que le temps de calcul soit raisonnable, nous y considérons les séquences par morceaux et ensuite nous combinons le tout à l'aide de la vraisemblance composite. Au chapitre IV, le calcul de la vraisemblance n'est pas fait de façon exacte. L'échantillonnage pondéré avec $P(H|\theta_0)$ comme distribution proposée, où θ_0 est une valeur conductrice de θ, combiné à l'intégration Monte-Carlo y est utilisé. Ceci donne :

$$\begin{aligned} P(D|\theta) &= \sum_H P(D|H)P(H|\theta) \\ &= \sum_H P(D|H)\frac{P(H|\theta)}{P(H|\theta_0)}P(H|\theta_0) \\ &= E\left(\frac{P(H|\theta)}{P(H|\theta_0)}\right) \\ &\approx \frac{1}{M}\sum_{i=1}^{M}\frac{P(H_i|\theta)}{P(H_i|\theta_0)}. \end{aligned}$$

Pour obtenir l'estimation de la vraisemblance, il s'agit donc de simuler M histoires compatibles avec les données et tirées de la distribution proposée $P(H|\theta_0)$.

3.4 Méthode de Fearnhead et Donnelly (2001)

En 2001, Fearnhead et Donnelly ont proposé une façon alternative d'estimer la vraisemblance avec l'échantillonnage pondéré lorsque nous travaillons avec les histoires. D'une façon similaire à ce que nous venons de voir, en utilisant l'échantillonnage pondéré avec distribution proposée $q(H)$, ils arrivent à l'estimation suivante de la vraisemblance des paramètres génétiques ρ et θ :

$$L(\rho, \theta) \approx \frac{1}{M} \sum_{i=1}^{M} \frac{P(H_i \mid \rho, \theta)}{q(H_i)}.$$

Tout comme dans la méthode de Larribe *et al.* du chapitre IV, une réalisation de la distribution proposée est constituée d'une série d'états échantillonnaux $\mathbf{H}_0, \ldots, \mathbf{H}_\tau$ (où $\mathbf{H}_0$ correspond à l'échantillon de départ et $\mathbf{H}_\tau$ au MRCA) avec probabilités de transition vers l'ancêtre commun $q(\mathbf{H}_{i+1} \mid \mathbf{H}_i)$ proportionnelles aux probabilités de transition connues $q(\mathbf{H}_i \mid \mathbf{H}_{i+1})$. Fearnhead et Donnelly utilisent cependant une formule différente pour obtenir les probabilités de transition vers le MRCA :

$$q(\mathbf{H}_{i+1} \mid \mathbf{H}_i) = p(\mathbf{H}_i \mid \mathbf{H}_{i+1})\pi(\mathbf{H}_{i+1})/\pi(\mathbf{H}_i). \tag{3.2}$$

Dans cette équation, le rapport $\pi(\mathbf{H}_{i+1})/\pi(\mathbf{H}_i)$ peut être simplifié parce que pour toutes les transitions possibles, la majorité des chromosomes ne sont pas affectés. Par exemple, pour un évènement de coalescence identique de deux chromosomes de types α, $\mathbf{H}_{i+1}$ est égal à $\mathbf{H}_i$, à la seule différence qu'il y a un chromosome de type α en moins. Nous utilisons alors la notation $\mathbf{H}_i - \alpha$ pour désigner $\mathbf{H}_{i+1}$. Dans ce cas,

$$\pi(\mathbf{H}_{i+1})/\pi(\mathbf{H}_i) = 1/\pi(\alpha \mid \mathbf{H}_i - \alpha),$$

où $\pi(\alpha \mid \mathbf{H}) = \pi(\{\mathbf{H}, \alpha\})/\pi(\mathbf{H})$ est la distribution conditionnelle du dernier chromosome dans un échantillon, étant donné que les autres chromosomes sont de type $\mathbf{H}$. Nous pouvons donc réécrire (3.2) en fonction des probabilités connues $p(\mathbf{H}_i \mid \mathbf{H}_{i+1})$ et des densités $\pi(\alpha \mid \mathbf{H})$. Ainsi, pour échantillonner de la distribution proposée, il suffit seulement d'être en mesure d'approximer $\pi(\alpha \mid \mathbf{H})$. Stephens et Donnelly (2000) proposent une façon d'obtenir l'approximation dans le cas où $\rho = 0$ et ce résultat peut être généralisé au cas où $\rho \neq 0$ (Fearnhead et Donnelly, 2001).

3.5 Méthode de Fearnhead et Donnelly (2002)

Fearnhead et Donnelly ont proposé, en 2002, une méthode intéressante d'approximation de la vraisemblance qui permet de réaliser des économies dans les temps de calcul. Plutôt que de considérer en entier l'échantillon initial, ils proposent d'en extraire l'information la plus importante.

Soit D un ensemble de données, S l'ensemble des marqueurs auxquels la fréquence du marqueur le plus rare est supérieure à un seuil préspécifié, D_S le sous-ensemble de données de D comprenant seulement les marqueurs dans S et S_0 le nombre de marqueurs dans l'ensemble de données D qui ne sont pas dans S. Notons que D_S est très informatif sur ρ et le nombre total de marqueurs l'est sur θ. Conditionnellement à H, D_S et S_0 sont indépendants. Ainsi, $L_M(\rho, \theta)$, la vraisemblance de la synthèse des données considérée, peut s'obtenir d'une façon similaire à ce que nous avons vu :

$$L_M(\rho, \theta) = \sum_H P(D_S \mid H) P(S_0 \mid H, \theta) P(H \mid \rho, \theta).$$

En sommant seulement les histoires H' des marqueurs dans S qui sont compatibles avec les données et en utilisant l'échantillonnage pondéré, nous obtenons :

$$L_M(\rho,\theta) = \sum_{H'} \frac{P(S_0 \mid H',\theta)P(H' \mid \rho,\theta)}{q(H')} q(H'). \tag{3.3}$$

Comme distribution proposée $q(H')$, nous pouvons entre autres choisir celle proposée par Fearnhead et Donnelly en 2001. Il ne restera donc plus qu'à approximer $P(S_0 \mid H',\theta)$. Une façon de procéder est de supposer que la généalogie de chaque marqueur est la même que la généalogie du marqueur de S qui est le plus près. Ainsi, en utilisant cette approximation de $P(S_0 \mid H',\theta)$ dans (3.3), nous obtenons une approximation marginale de la vraisemblance.

L'estimateur de vraisemblance composite correspondant s'obtient en séparant les séquences en R sous-régions. Pour $r = 1,\ldots,R$, soit D_r l'ensemble de données de la $r^{\text{ème}}$ sous-région. L'estimateur de vraisemblance composite est alors défini par :

$$L_C(\rho,\theta) = \prod_{r=1}^{R} p(D_r \mid \rho,\theta).$$

Dans cette formule, $p(D_r \mid \rho,\theta)$ peut être estimé via l'approximation marginale de la vraisemblance que nous venons d'étudier.

3.6 Estimateur de vraisemblance composite d'Hudson (2001)

La méthode d'Hudson (2001) permet également d'obtenir un estimateur de vraisemblance composite. D'abord, un estimateur de vraisemblance est obtenu dans le cas où nous considérons un échantillon où les séquences comportent chacune deux marqueurs génétiques :

$$L(\mathbf{n_1},\mathbf{n_2},\ldots,\mathbf{n_s};\rho_{bp}) \approx \prod_{i}^{s} q_c(\mathbf{n_i};\rho_{bp}d_i), \tag{3.4}$$

où $\mathbf{n_i}$ représente la configuration de la $i^{\text{ème}}$ séquence à deux marqueurs, $q_c(\mathbf{n_i}; \rho_{bp} d_i)$ la fonction de densité de $\mathbf{n_i}$ conditionnelle à $\rho_{bp} d_i$, ρ_{bp} le taux de recombinaison à l'échelle par paire de base et d_i la distance, en paire de bases, entre la $i^{\text{ème}}$ paire de marqueurs.

Pour généraliser cet estimateur dans le cas où plus de deux marqueurs sont définis par séquence, la vraisemblance composite est alors tout à fait appropriée. Dans (3.4), il suffit de prendre le produit sur toutes les paires de marqueurs. Il est à noter que l'estimateur de vraisemblance composite ainsi obtenu a le défaut de ne pas tenir compte de la dépendance statistique entre les paires de marqueurs sur les mêmes séquences.

CHAPITRE IV

MÉTHODE DE LARRIBE *ET AL.*

En 2002, Larribe, Lessard et Schork ont présenté une nouvelle méthode basée sur le déséquilibre de liaison et faisant appel à plusieurs loci pour localiser la position d'un gène mutant sur un chromosome. Un aspect intéressant de leur méthode est qu'elle tient compte, dans le cadre du processus de coalescence, des évènements de recombinaison lors de la modélisation de l'historique des séquences de l'échantillon. L'historique est donc représentée par un graphe, le graphe de recombinaison ancestral, et non par un arbre. Ceci complexifie la méthode mais du même coup la rend potentiellement plus puissante.

La distance, en centimorgans, entre le début d'une séquence et la position du gène mutant est estimée par la distance qui maximise sa vraisemblance. La courbe de vraisemblance est estimée à l'aide d'un algorithme de Monte-Carlo basé sur les probabilités de transition entre tous les états possibles de l'échantillon.

Dans ce chapitre, nous présentons d'abord les détails et hypothèses du modèle utilisé. Ensuite, nous étudions les concepts mathématiques sur lesquels se base la méthode. Nous voyons comment construire l'équation de récurrence de Griffiths-Tavaré qui nous intéresse, comment définir la chaîne de Markov donnant les probabilités de transition d'un état à un autre qui est plus près du MRCA et, finalement, comment utiliser l'échantillonage pondéré pour obtenir les vraisemblances.

4.1 Détails et hypothèses du modèle

Nous présentons maintenant les détails et les hypothèses du modèle utilisé. Il est à noter que ces détails et hypothèses sont les mêmes pour la méthode développée au chapitre V. D'abord, les haplotypes sont connus. Cela signifie que l'information pour chaque marqueur de chacune des séquences de notre échantillon est connue. En ce qui concerne la taille de la population, celle-ci demeure constante : elle ne varie pas dans le temps. Nous supposons aussi qu'il n'y a pas d'immigration : la population n'a pas de structure géographique qui pourrait influencer la reproduction. Nous simplifions également le modèle en supposant qu'il y a pénétrance complète et qu'il n'y a pas de phénocopie. La pénétrance complète signifie que si un individu possède la mutation, alors cet individu est affecté par la maladie. Le fait qu'il n'y ait pas de phénocopie nous assure que tous les individus qui ne possèdent pas la mutation ne sont pas affectés par la maladie. Nous supposons qu'une séquence de notre échantillon qui possède la mutation provient d'un individu diploïde malade ayant la mutation sur chacun de ses allèles et qu'une séquence n'ayant pas la mutation provient d'un individu diploïde non malade qui n'a pas la mutation sur aucun de ses allèles. Il s'agit d'une version simple d'un modèle de maladie récessive rare.

En ce qui a trait aux mutations, nous faisons les trois hypothèses suivantes (voir le chapitre I pour plus d'informations) :

- Le MRCA est une séquence ne contenant aucune mutation. Cette hypothèse peut sembler très restrictive, mais en fait elle ne l'est pas. Il s'agit seulement d'une question de notation. Nous ne faisons que supposer que le code génétique de l'ancêtre est connu. Pour ce faire, nous pouvons prendre les allèles ayant la plus grande fréquence dans l'échantillon ou encore utiliser des études animales pour déterminer ce que doit être le code de l'ancêtre.
- Les mutations ne changent pas les chances de reproduction, il n'y a donc pas de sélection.
- Nous travaillons avec le modèle de mutation "Infinite sites" (Kimura, 1969), voir section

1.3.

Finalement, notre dernière hypothèse est que les distances génétiques sont additives. Comme nous l'avons vu à la sous-section 2.3.1, il en découle que si nous connaissons la distance génétique entre chacun des marqueurs, alors nous pouvons trouver la distance génétique entre le premier et le dernier marqueur. Cette dernière quantité est en fait le taux de recombinaison (r) puisque par définition, la distance génétique entre deux marqueurs est la probabilité d'obtenir une recombinaison entre ces deux marqueurs. Il est à noter que l'hypothèse d'additivité est vraisemblable lorsque nous travaillons avec de petites distances génétiques, ce qui est le cas.

4.2 Équation de récurrence

Soit $\mathbf{H}_\tau$ l'ensemble des séquences ancestrales dans l'échantillon juste après que le $\tau^{\text{ème}}$ évènement, en remontant dans le temps, ait lieu ($\tau = 0, \ldots, \tau^*$). Une séquence ancestrale est une séquence qui n'est pas composée uniquement de marqueurs non-ancestraux. $\mathbf{H}_0$ correspond donc à l'échantillon de départ et $\mathbf{H}_{\tau^*}$ correspond au MRCA. Soit $Q(\mathbf{H}_\tau)$ la fonction de densité associée à $\mathbf{H}_\tau$.

Nous allons développer une équation de récurrence qui exprime $Q(\mathbf{H}_\tau)$ en fonction de $Q(\mathbf{H}_{\tau+1})$. Pour ce faire, utilisons l'égalité suivante bien connue en probabilités :

$$P(A) = \sum_{B_i} P(A \cap B_i), \text{ où } \sum_{B_i} P(B_i) = 1.$$

Puisque $P(A \cap B) = P(A \mid B)P(B)$, l'égalité peut aussi s'écrire de cette façon :

$$P(A) = \sum_{B_i} P(A \mid B_i)P(B_i), \text{ où } \sum_{B_i} P(B_i) = 1.$$

En posant $A = \mathbf{H}_\tau$, $B_i = \mathbf{H}_{\tau+1}$ et en remplaçant $P(.)$ par $Q(.)$ lorsqu'il s'agit d'une fonction de densité, nous obtenons :

$$Q(\mathbf{H}_\tau) = \sum_{\mathbf{H}_{\tau+1}} P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1}) Q(\mathbf{H}_{\tau+1}).$$

Nous décomposons la somme en considérant tous les états possibles juste après le $(\tau+1)^{\text{ème}}$ évènement :

$$\mathbf{H}_{\tau+1} = \begin{cases} \mathbf{H}_\tau + C_i & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est une coalescence de deux séquences de type } i, \\ \mathbf{H}_\tau + C_{ij}^k \text{ (avec } i \neq j) & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est une coalescence d'une séquence de type } i \text{ et d'une séquence de type } j \text{ vers une séquence de type } k, \\ \mathbf{H}_\tau + M_i^j(m) & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est une mutation au marqueur } m \text{ d'une séquence de type } i \text{ vers une séquence de type } j, \\ \mathbf{H}_\tau + R_i^{jk}(p) & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est une recombinaison d'une séquence de type } i \text{ dans l'intervalle entre le } p^{\text{ème}} \text{ et le } (p+1)^{\text{ème}} \text{ marqueur vers des séquences de type } j \text{ et } k, \\ \mathbf{H}_\tau & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est une mutation non-ancestrale ou une recombinaison non-ancestrale.} \end{cases}$$

Nous supposons qu'au temps τ il y a n_i séquences de type i, n_j séquences de type j, n_k séquences de type k et n séquences au total.

Puisque $\mathbf{H}_{\tau+1}$ est exprimé comme une fonction de $\mathbf{H}_\tau$ nous renseignant sur la nature du $(\tau+1)^{\text{ème}}$ évènement, nous connaissons aussi le nombre de séquences de chaque type au

temps $(\tau+1)$. Pour trouver $P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1})$, il suffit donc de calculer la probabilité du $(\tau+1)^{\text{ème}}$ évènement qui permet de passer de $\mathbf{H}_{\tau+1}$ à $\mathbf{H}_\tau$.

Pour faire nos calculs, nous utilisons la probabilité d'une coalescence, d'une mutation ou d'une recombinaison lorsque l'un de ces trois évènements a lieu. Rappelons que ces probabilités sont :

$$\begin{aligned} P(\text{Co} \mid \cdot) &= \frac{n-1}{n-1+\theta+\rho}, \\ P(\text{Mu} \mid \cdot) &= \frac{\theta}{n-1+\theta+\rho}, \\ P(\text{Re} \mid \cdot) &= \frac{\rho}{n-1+\theta+\rho}. \end{aligned}$$

Calculons maintenant $P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1})$ pour les cinq cas possibles. Pour ce faire, nous utilisons le résultat de probabilités stipulant que si A et B sont deux évènements indépendants, alors $P(A \cap B) = P(A) \cdot P(B)$.

1. Le $(\tau+1)^{\text{ème}}$ évènement est une coalescence identique C_i

Au temps $(\tau+1)$, il y a $(n_k + 1 - \delta_{ik} - \delta_{jk})$ séquences de type k, où

$$\delta_{ik} = \begin{cases} 1 & \text{si } i = k, \\ 0 & \text{sinon} \end{cases}$$

et

$$\delta_{jk} = \begin{cases} 1 & \text{si } j = k, \\ 0 & \text{sinon.} \end{cases}$$

En effet, la coalescence produit une séquence de type k et requiert aucune, une ou deux de ces séquences car i et/ou j peuvent être égaux à k (si i et j sont égaux à k, cela nous

ramène à une coalescence identique). Il y a également $(n-1)$ séquences au total au temps $(\tau+1)$. Pour illustrer cela, supposons que $i \neq j \neq k$ et que les séquences i et j coalescent vers une séquence k pour le $(\tau+1)^{\text{ème}}$ évènement. Le nombre de séquences k au temps $(\tau+1)$ est alors n_k+1, car nous avons n_k séquences de type k au temps τ et l'évènement de coalescence en crée une autre. Les δ_{ik} et δ_{jk} ajustent le nombre de séquences k si et seulement si $i=k$ et/ou $j=k$. Nous obtenons donc :

$$\begin{aligned}
P(\mathbf{H}_\tau \mid \mathbf{H}_\tau + C_{ij}^k) &= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est } C_{ij}^k) \\
&= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement implique une séquence de type } k \\
&\quad \text{et le } (\tau+1)^{\text{ème}} \text{ évènement est une coalescence}) \\
&= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement implique une séquence de type } k) \\
&\quad \cdot P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est une coalescence}) \\
&= \frac{n_k+1-\delta_{ik}-\delta_{jk}}{n-1} \cdot \frac{n-1}{n-1+\theta+\rho} \\
&= \frac{n_k+1-\delta_{ik}-\delta_{jk}}{n-1+\theta+\rho}.
\end{aligned}$$

2. Le $(\tau+1)^{\text{ème}}$ évènement est une coalescence distincte C_{ij}^k

Il s'agit du cas particulier d'une coalescence des séquences de types i et j vers une séquence de type k où $i=j=k$. Nous avons donc $\delta_{ik}=\delta_{jk}=1$ et

$$P(\mathbf{H}_\tau \mid \mathbf{H}_\tau + C_i) = \frac{n_i-1}{n-1+\theta+\rho}.$$

3. Le $(\tau+1)^{\text{ème}}$ évènement est une mutation $M_i^j(m)$

Au temps $(\tau+1)$, il y a (n_j+1) séquences de type j et n séquences au total, car le nombre de séquences ne change pas lors d'un évènement de mutation. Alors,

$$
\begin{aligned}
P(\mathbf{H}_\tau \mid \mathbf{H}_\tau + M_i^j(m)) &= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est } M_i^j(m)) \\
&= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement implique une séquence de type } j, \\
&\quad \text{le marqueur } m \text{ et le } (\tau+1)^{\text{ème}} \text{ évènement est une mutation}) \\
&= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement implique une séquence de type } j) \\
&\quad \cdot P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement implique le marqueur } m) \\
&\quad \cdot P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est une mutation}) \\
&= \frac{n_j+1}{n} \cdot \frac{1}{L} \cdot \frac{\theta}{n-1+\theta+\rho} \\
&= \frac{\theta(n_j+1)}{nL(n-1+\theta+\rho)}.
\end{aligned}
$$

Notons qu'il serait possible de tenir compte de taux de mutations variables à chaque marqueur, en remplaçant θ/L par θ_m pour le marqueur m, de façon à ce que $\sum_m \theta_m = \theta$.

4. Le $(\tau+1)^{\text{ème}}$ évènement est une recombinaison $R_i^{jk}(p)$

Au temps $(\tau+1)$, il y a (n_j+1) séquences de type j, (n_k+1) séquences de type k et $(n+1)$ séquences au total. Ainsi,

$$
\begin{aligned}
P(\mathbf{H}_\tau \mid \mathbf{H}_\tau + R_i^{jk}(p)) &= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est } R_i^{jk}(p)) \\
&= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement implique une séquence de type } j \text{ et} \\
&\quad \text{une séquence de type } k \text{ (où } j \text{ est choisi en premier), un point} \\
&\quad \text{de recombinaison dans l'intervalle } p \text{ et le } (\tau+1)^{\text{ème}} \text{ évènement} \\
&\quad \text{est une recombinaison})
\end{aligned}
$$

$$
\begin{aligned}
&= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement implique une séquence de type } j \text{ et} \\
&\quad \text{une séquence de type } k) \\
&\quad \cdot P(\text{la séquence } j \text{ est choisie en premier}) \\
&\quad \cdot P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement implique un point de} \\
&\quad \text{recombinaison dans l'intervalle } p) \\
&\quad \cdot P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est une recombinaison}) \\
&= \frac{\binom{n_j+1}{1}\binom{n_k+1}{1}}{\binom{n+1}{2}} \cdot \frac{1}{2} \cdot \frac{r_p}{r} \cdot \frac{\rho}{n-1+\theta+\rho} \\
&= \frac{\rho(n_j+1)(n_k+1)r_p}{n(n+1)(n-1+\theta+\rho)r}.
\end{aligned}
$$

5. Le $(\tau+1)^{\text{ème}}$ évènement est un évènement non-ancestral

Rappelons que $\mathbf{H}_\tau$ désigne seulement l'ensemble des séquences ancestrales. Pour que $\mathbf{H}_{\tau+1}$ soit égal à $\mathbf{H}_\tau$, il faut donc que le $(\tau+1)^{\text{ème}}$ évènement ait été un évènement qualifié de "non-ancestral". Il y a deux cas possibles. Premièrement, une mutation peut survenir sur un marqueur non-ancestral. Ensuite, il peut y avoir recombinaison dans laquelle une des deux parties de la séquence est constituée seulement de marqueurs non-ancestraux. Dans ce cas, une des deux nouvelles séquences produite par la recombinaison est constituée uniquement de marqueurs non-ancestraux alors que l'autre nouvelle séquence est une copie identique de la séquence originale.

Soit $|A_i|$ le nombre de marqueurs ancestraux sur une séquence de type i, A_i l'ensemble des marqueurs ancestraux sur une séquence de type i, d le nombre de types différents de séquences, L le nombre de marqueurs par séquence et r la longueur d'une séquence (la distance entre le premier et le dernier marqueur). Alors $a = \sum_{i=1}^{d} n_i|A_i|$ compte le nombre total de marqueurs ancestraux pour les n séquences et

$$b = \sum_{i=1}^{d} n_i \cdot (\max x_m;\ \text{marqueur } m \in A_i - \min x_m;\ \text{marqueur } m \in A_i)$$

compte la distance totale, pour les n séquences, sur laquelle une recombinaison est ancestrale.

Ainsi, sur les nL marqueurs, a d'entre eux sont ancestraux et sur la somme totale nr des longueurs des séquences, une recombinaison est ancestrale sur une longueur b. Le nombre de marqueurs non-ancestraux est donc $nL - a$, ce qui implique que $(nL - a)/nL$ donne la probabilité qu'un marqueur soit non-ancestral. Il s'agit aussi de la probabilité qu'une mutation soit non-ancestrale. La longueur totale sur laquelle une recombinaison est non-ancestrale est donc $nr - b$, ce qui implique que $(nr - b)/nr$ donne la probabilité qu'une recombinaison soit non-ancestrale. Ainsi, nous avons donc :

$$\begin{aligned}
P(\mathbf{H}_\tau \mid \mathbf{H}_\tau) &= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est une mutation non-ancestrale}) \\
&\quad + P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est une recombinaison non-ancestrale}) \\
&= P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est non-ancestral} \mid \text{le } (\tau+1)^{\text{ème}} \text{ évènement} \\
&\quad \text{est une mutation}) \cdot P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est une mutation}) \\
&\quad + P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est non-ancestral} \mid \text{le } (\tau+1)^{\text{ème}} \\
&\quad \text{évènement est une recombinaison}) \cdot P(\text{le } (\tau+1)^{\text{ème}} \text{ évènement est une} \\
&\quad \text{recombinaison}) \\
&= \frac{nL-a}{nL} \cdot \frac{\theta}{n-1+\theta+\rho} + \frac{nr-b}{nr} \cdot \frac{\rho}{n-1+\theta+\rho} \\
&= \frac{\theta(nL-a)}{nL(n-1+\theta+\rho)} + \frac{\rho(nr-b)}{nr(n-1+\theta+\rho)}.
\end{aligned}$$

Nous avons réussi à calculer $P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1})$ dans tous les cas possibles :

$$P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1}) = \begin{cases} \frac{n_i - 1}{n-1+\theta+\rho} & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est } C_i, \\ \frac{n_k+1-\delta_{ik}-\delta_{jk}}{n-1+\theta+\rho} & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est } C_{ij}^k \text{ (avec } i \neq j), \\ \frac{\theta(n_j+1)}{nL(n-1+\theta+\rho)} & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est } M_i^j(m), \\ \frac{\rho(n_j+1)(n_k+1)r_p}{n(n+1)(n-1+\theta+\rho)r} & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est } R_i^{jk}(p), \\ \frac{\theta(nL-a)}{nL(n-1+\theta+\rho)} & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est une mutation} \\ & \text{non-ancestrale,} \\ \frac{\rho(nr-b)}{nr(n-1+\theta+\rho)} & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est une recombinaison} \\ & \text{non-ancestrale.} \end{cases}$$

Pour calculer $Q(\mathbf{H}_\tau) = \sum_{\mathbf{H}_{\tau+1}} P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1}) Q(\mathbf{H}_{\tau+1})$, nous sommons sur tous les cas possibles pour $\mathbf{H}_{\tau+1}$.

Lorsque le $(\tau+1)^{\text{ème}}$ évènement est C_i, nous sommons sur tous les types i qui peuvent coalescer. Puisqu'une coalescence requiert deux séquences, nous sommons sur tous les $i = 1, \ldots, d$ tels que $n_i > 1$.

Lorsque le $(\tau+1)^{\text{ème}}$ évènement est C_{ij}^k (avec $i \neq j$), nous sommons sur toutes les paires i, j sans ordre telles que $i \neq j$ et telles que les types i et j possèdent le même ensemble de mutations dans le matériel ancestral (ce qui est une condition pour qu'il puisse y avoir coalescence). Nous multiplions la sommation par 2 (car $C_{ij}^k = C_{ji}^k$) pour dénombrer avec ordre.

Lorsque le $(\tau+1)^{\text{ème}}$ évènement est $M_i^j(m)$, nous sommons d'abord sur tous les types i tels que $n_i = 1$, puisque selon notre modèle une mutation ne peut survenir sur un marqueur plus d'une fois. Ensuite, nous sommons sur tous les marqueurs ancestraux qui ont une mutation et telle que cette mutation est la seule, parmi toutes les séquences, à cette position.

Lorsque le $(\tau+1)^{\text{ème}}$ évènement est $R_i^{jk}(p)$, nous sommons d'abord sur tous les types i car n'importe quel type peut se recombiner. Ensuite, nous sommons sur tous les intervalles

dans lesquels le point de recombinaison peut se trouver pour que la recombinaison soit ancestrale. Si γ_i et κ_i sont respectivement les numéros des premiers et derniers intervalles de la séquence de type i où une recombinaison affecte le matériel ancestral, alors nous sommons de $p = \gamma_i$ à $p = \kappa_i$.

Lorsque le $(\tau + 1)^{\text{ème}}$ évènement est une mutation ou une recombinaison non-ancestrale, nous n'avons pas besoin de sommer puisqu'il n'est pas nécessaire de connaître la nature des types impliqués dans l'évènement.

L'équation de récurrence est donc :

$$
\begin{aligned}
Q(\mathbf{H}_\tau) = & \frac{1}{n-1+\theta+\rho} \sum_{i:n_i>1} (n_i - 1) Q(\mathbf{H}_\tau + C_i) \\
& + \frac{2}{n-1+\theta+\rho} \sum_{i \neq j \ \ compatibles} (n_k + 1 - \delta_{ik} - \delta_{jk}) Q(\mathbf{H}_\tau + C_{ij}^k) \\
& + \frac{\theta}{nL(n-1+\theta+\rho)} \sum_{i:n_i=1} \sum_{m \in A_i \ \ unique} (n_j + 1) Q(\mathbf{H}_\tau + M_i^j(m)) \\
& + \frac{\rho(n_j+1)(n_k+1)}{n(n+1)(n-1+\theta+\rho)} \sum_{i=1}^{d} \sum_{p=\gamma_i}^{\kappa_i} \frac{r_p}{r} Q(\mathbf{H}_\tau + R_i^{jk}(p)) \\
& + \frac{\theta(nL-a)}{nL(n-1+\theta+\rho)} Q(\mathbf{H}_\tau) \\
& + \frac{\rho(nr-b)}{nr(n-1+\theta+\rho)} Q(\mathbf{H}_\tau).
\end{aligned}
$$

Il est possible de simplifier cette équation en isolant $Q(\mathbf{H}_\tau)$. Nous obtenons alors :

$$
\begin{aligned}
&\left[1- \frac{\theta(nL-a)}{nL(n-1+\theta+\rho)} - \frac{\rho(nr-b)}{nr(n-1+\theta+\rho)}\right] Q(\mathbf{H}_\tau) \\
&\quad = \left[1 - \frac{\theta}{n-1+\theta+\rho} - \frac{\rho}{n-1+\theta+\rho} + \frac{\frac{a}{nL}\theta}{n-1+\theta+\rho} + \frac{\frac{b}{nr}\rho}{n-1+\theta+\rho}\right] Q(\mathbf{H}_\tau) \\
&\quad = \left[1 + \frac{\frac{a}{nL}\theta + \frac{b}{nr}\rho - \theta - \rho}{n-1+\theta+\rho}\right] Q(\mathbf{H}_\tau) \\
&\quad = \left[\frac{n-1+\frac{a}{nL}\theta + \frac{b}{nr}\rho}{n-1+\theta+\rho}\right] Q(\mathbf{H}_\tau) \\
&\quad = \frac{1}{n-1+\theta+\rho} \sum_{i:n_i>1} (n_i-1) Q(\mathbf{H}_\tau + C_i) \\
&\qquad + \frac{2}{n-1+\theta+\rho} \sum_{i\neq j \ compatibles} (n_k+1-\delta_{ik}-\delta_{jk}) Q(\mathbf{H}_\tau + C_{ij}^k) \\
&\qquad + \frac{\theta}{nL(n-1+\theta+\rho)} \sum_{i:n_i=1} \sum_{m\in A_i \ unique} (n_j+1) Q(\mathbf{H}_\tau + M_i^j(m)) \\
&\qquad + \frac{\rho(n_j+1)(n_k+1)}{n(n+1)(n-1+\theta+\rho)} \sum_{i=1}^{d} \sum_{p=\gamma_i}^{\kappa_i} \frac{r_p}{r} Q(\mathbf{H}_\tau + R_i^{jk}(p)).
\end{aligned}
$$

En posant $\alpha = a/nL$, $\beta = b/nr$ et $D_{\mathbf{H}_\tau} = n-1+\alpha\theta+\beta\rho$, la formule de récurrence devient donc :

$$
\begin{aligned}
Q(\mathbf{H}_\tau) &= \frac{1}{D_{\mathbf{H}_\tau}} \sum_{i:n_i>1} (n_i-1) Q(\mathbf{H}_\tau + C_i) \\
&\quad + \frac{2}{D_{\mathbf{H}_\tau}} \sum_{i\neq j \ compatibles} (n_k+1-\delta_{ik}-\delta_{jk}) Q(\mathbf{H}_\tau + C_{ij}^k) \\
&\quad + \frac{\theta}{nLD_{\mathbf{H}_\tau}} \sum_{i:n_i=1} \sum_{m\in A_i \ unique} (n_j+1) Q(\mathbf{H}_\tau + M_i^j(m)) \\
&\quad + \frac{\rho(n_j+1)(n_k+1)}{n(n+1)D_{\mathbf{H}_\tau}} \sum_{i=1}^{d} \sum_{p=\gamma_i}^{\kappa_i} \frac{r_p}{r} Q(\mathbf{H}_\tau + R_i^{jk}(p)).
\end{aligned}
$$

C'est cette forme que nous considérons puisque nous ne nous intéressons pas aux évènements non-ancestraux. Cette équation est également à la base du calcul des vraisemblances dans le processus de coalescence avec recombinaison.

4.3 Chaîne de Markov

Nous prenons $Q(\mathbf{H}_0)$ comme vraisemblance de l'échantillon. Notons toutefois que c'est un abus de notation : en fait, la vraisemblance s'écrit habituellement $L(r_T) = Q(\mathbf{H}_0 \mid r_T)$. La formule de $Q(\mathbf{H}_0)$ se trouve à l'aide de l'équation de récurrence que nous venons de développer. Par la suite, il n'est malheureusement pas possible de calculer $Q(\mathbf{H}_0)$ de façon exacte puisque ce serait beaucoup trop long.

À la place, nous utilisons une stratégie Monte-Carlo introduite pour la première fois par Griffiths et Tavaré (1994). Pour ce faire, définissons d'abord une chaîne de Markov avec probabilités de transition de $\mathbf{H}_\tau$ à $\mathbf{H}_{\tau+1}$. Nous venons de calculer $P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1})$, mais nous ignorons comment calculer l'inverse. Nous utilisons une idée de Griffiths, qui consiste à proposer que $P(\mathbf{H}_{\tau+1} \mid \mathbf{H}_\tau)$ est tout simplement $c \times P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1})$, où c est une constante de normalisation pour en faire une distribution de probabilité.

Dans notre cas,

$$c = \frac{1}{\sum_{\mathbf{H}_{\tau+1}} P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1})}.$$

Ainsi,

$$P(\mathbf{H}_{\tau+1} \mid \mathbf{H}_\tau) = \frac{P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1})}{\sum_{\mathbf{H}_{\tau+1}} P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1})}.$$

Notons que les auteurs (Larribe *et al.*, 2002) ne prétendent pas que ce résultat est vrai. $P(\mathbf{H}_{\tau+1} \mid \mathbf{H}_\tau)$ joue le rôle d'une distribution proposée, et à ce titre, il était possible d'utiliser n'importe quelle distribution. La distribution proposée idéale, si elle était connue, serait évidemment $P(\mathbf{H}_{\tau+1} \mid \mathbf{H}_\tau)$.

Nous pouvons simplifier en divisant par le facteur $D_{\mathbf{H}_\tau} = n - 1 + \alpha\theta + \beta\rho$ au numérateur et au dénominateur. Aussi, nous laissons tomber le facteur $(n_j+1)(n_k+1)$ de la recombinaison. Nous perdons ainsi un peu de précision, mais cela va nous permettre de faire nos calculs beaucoup plus rapidement.

Soit $S_{\mathbf{H}_\tau}$ la somme des $P(\mathbf{H}_\tau \mid \mathbf{H}_{\tau+1})$ sur toutes les possibilités de $\mathbf{H}_{\tau+1}$ considérées dans le modèle en laissant tomber les facteurs $D_{\mathbf{H}_\tau}$ et $(n_j + 1)(n_k + 1)$, c'est-à-dire :

$$\begin{aligned} S_{\mathbf{H}_\tau} &= \sum_{i:n_i>1} (n_i - 1) \\ &+ 2 \sum_{i\neq j \ \ compatibles} (n_k + 1 - \delta_{ik} - \delta_{jk}) \\ &+ \frac{\theta}{nL} \sum_{i:n_i=1} \sum_{m\in A_i \ \ unique} (n_j + 1) \\ &+ \frac{\rho}{n(n+1)} \sum_{i=1}^{d} \sum_{p=\gamma_i}^{\kappa_i} r_p/r. \end{aligned}$$

Selon notre chaîne de Markov, au temps $(\tau + 1)$, une transition est donc faite de $\mathbf{H}_\tau$ à

$$\begin{cases} (\mathbf{H}_\tau + C_i) & \text{avec probabilité } (n_i - 1)/S_{\mathbf{H}_\tau}, \\ (\mathbf{H}_\tau + C_{ij}^k) & \text{avec probabilité } 2(n_k + 1 - \delta_{ik} - \delta_{jk})/S_{\mathbf{H}_\tau}, \\ (\mathbf{H}_\tau + M_i^j(m)) & \text{avec probabilité } \frac{\theta(n_j+1)}{nL}/S_{\mathbf{H}_\tau}, \\ (\mathbf{H}_\tau + R_i^{jk}(p)) & \text{avec probabilité } \frac{\rho r_p/r}{n(n+1)}/S_{\mathbf{H}_\tau}. \end{cases}$$

Cette distribution est la distribution proposée. Elle permet à la chaîne de Markov de passer

d'un état à l'autre.

4.4 Échantillonnage pondéré

Pour faire ressortir $P(\mathbf{H}_{\tau+1} \mid \mathbf{H}_\tau)$, nous écrivons l'équation de récurrence de cette façon :

$$\begin{aligned} Q(\mathbf{H}_\tau) = {} & \frac{S_{\mathbf{H}_\tau}}{D_{\mathbf{H}_\tau}} \sum_{i:n_i>1} \frac{(n_i-1)}{S_{\mathbf{H}_\tau}} Q(\mathbf{H}_\tau + C_i) \\ & + \frac{S_{\mathbf{H}_\tau}}{D_{\mathbf{H}_\tau}} \sum_{i \neq j \ \ compatibles} \frac{2(n_k+1-\delta_{ik}-\delta_{jk})}{S_{\mathbf{H}_\tau}} Q(\mathbf{H}_\tau + C_{ij}^k) \\ & + \frac{S_{\mathbf{H}_\tau}}{D_{\mathbf{H}_\tau}} \sum_{i:n_i=1} \sum_{m \in A_i \ \ unique} \frac{\theta(n_j+1)}{nLS_{\mathbf{H}_\tau}} Q(\mathbf{H}_\tau + M_i^j(m)) \\ & + \frac{(n_j+1)(n_k+1)S_{\mathbf{H}_\tau}}{D_{\mathbf{H}_\tau}} \sum_{i=1}^{d} \sum_{p=\gamma_i}^{\kappa_i} \frac{\rho r_p/r}{n(n+1)S_{\mathbf{H}_\tau}} Q(\mathbf{H}_\tau + R_i^{jk}(p)). \end{aligned}$$

En définissant $f(\mathbf{H}_\tau, \mathbf{H}_{\tau+1}) = \begin{cases} \frac{S_{\mathbf{H}_\tau}}{D_{\mathbf{H}_\tau}} & \text{si } C_i,\ C_{ij}^k \text{ ou } M_i^j(m), \\ \frac{(n_j+1)(n_k+1)S_{\mathbf{H}_\tau}}{D_{\mathbf{H}_\tau}} & \text{si } R_i^{jk}(p), \end{cases}$

cela nous permet d'exprimer $Q(\mathbf{H}_\tau)$ sous la forme suivante :

$$Q(\mathbf{H}_\tau) = \sum_{\mathbf{H}_{\tau+1}} f(\mathbf{H}_\tau, \mathbf{H}_{\tau+1}) P(\mathbf{H}_{\tau+1} \mid \mathbf{H}_\tau) Q(\mathbf{H}_{\tau+1}).$$

Cette forme nous permet de développer $Q(\mathbf{H}_0)$:

$$\begin{aligned} Q(\mathbf{H}_0) &= \sum_{\mathbf{H}_1} f(\mathbf{H}_0, \mathbf{H}_1) P(\mathbf{H}_1 \mid \mathbf{H}_0) Q(\mathbf{H}_1) \\ &= \sum_{\mathbf{H}_1} f(\mathbf{H}_0, \mathbf{H}_1) P(\mathbf{H}_1 \mid \mathbf{H}_0) \sum_{\mathbf{H}_2} f(\mathbf{H}_1, \mathbf{H}_2) P(\mathbf{H}_2 \mid \mathbf{H}_1) Q(\mathbf{H}_2) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{\mathbf{H}_1} f(\mathbf{H}_0, \mathbf{H}_1) P(\mathbf{H}_1 \mid \mathbf{H}_0) \sum_{\mathbf{H}_2} f(\mathbf{H}_1, \mathbf{H}_2) P(\mathbf{H}_2 \mid \mathbf{H}_1) \sum_{\mathbf{H}_3} f(\mathbf{H}_2, \mathbf{H}_3) P(\mathbf{H}_3 \mid \mathbf{H}_2) Q(\mathbf{H}_3) \\
&= \cdots \\
&= \sum_{\mathbf{H}_1} \sum_{\mathbf{H}_2} \sum_{\mathbf{H}_3} \cdots \sum_{\mathbf{H}_{\tau^*}} f(\mathbf{H}_0, \mathbf{H}_1) f(\mathbf{H}_1, \mathbf{H}_2) f(\mathbf{H}_2, \mathbf{H}_3) \cdots f(\mathbf{H}_{\tau^*-1}, \mathbf{H}_{\tau^*}) \\
&\quad P(\mathbf{H}_1 \mid \mathbf{H}_0) P(\mathbf{H}_2 \mid \mathbf{H}_1) P(\mathbf{H}_3 \mid \mathbf{H}_2) \cdots P(\mathbf{H}_{\tau^*} \mid \mathbf{H}_{\tau^*-1}) Q(\mathbf{H}_{\tau^*}) \\
&= \sum_{\mathbf{H}_1} \sum_{\mathbf{H}_2} \sum_{\mathbf{H}_3} \cdots \sum_{\mathbf{H}_{\tau^*}} \prod_{\tau=0}^{\tau^*-1} f(\mathbf{H}_\tau, \mathbf{H}_{\tau+1}) P(\mathbf{H}_1 \mid \mathbf{H}_0) P(\mathbf{H}_2 \mid \mathbf{H}_1) P(\mathbf{H}_3 \mid \mathbf{H}_2) \cdots \\
&\quad P(\mathbf{H}_{\tau^*} \mid \mathbf{H}_{\tau^*-1}) Q(\mathbf{H}_{\tau^*}).
\end{aligned}
$$

Pour simplifier l'expression $P(\mathbf{H}_1 \mid \mathbf{H}_0) P(\mathbf{H}_2 \mid \mathbf{H}_1) P(\mathbf{H}_3 \mid \mathbf{H}_2) \cdots P(\mathbf{H}_{\tau^*} \mid \mathbf{H}_{\tau^*-1})$ dans la formule, nous utilisons l'identité suivante :

$$
\begin{aligned}
P(B \mid A) P(C \mid A, B) &= \frac{P(A, B)}{P(A)} \frac{P(A, B, C)}{P(A, B)} \\
&= \frac{P(A, B, C)}{P(A)} \\
&= P(B, C \mid A).
\end{aligned}
$$

Avec cette identité et en utilisant le fait que, puisque nous travaillons avec un processus markovien, seul le dernier état échantillonnal a de l'influence sur l'état échantillonnal actuel, nous obtenons :

$$
\begin{aligned}
&P(\mathbf{H}_1 \mid \mathbf{H}_0) P(\mathbf{H}_2 \mid \mathbf{H}_1) P(\mathbf{H}_3 \mid \mathbf{H}_2) \cdots P(\mathbf{H}_{\tau^*} \mid \mathbf{H}_{\tau^*-1}) \\
&= P(\mathbf{H}_1 \mid \mathbf{H}_0) P(\mathbf{H}_2 \mid \mathbf{H}_0, \mathbf{H}_1) P(\mathbf{H}_3 \mid \mathbf{H}_2) \cdots P(\mathbf{H}_{\tau^*} \mid \mathbf{H}_{\tau^*-1})
\end{aligned}
$$

$= P(\mathbf{H}_1, \mathbf{H}_2 \mid \mathbf{H}_0) P(\mathbf{H}_3 \mid \mathbf{H}_2) \cdots P(\mathbf{H}_{\tau^*} \mid \mathbf{H}_{\tau^*-1})$

$= P(\mathbf{H}_1, \mathbf{H}_2 \mid \mathbf{H}_0) P(\mathbf{H}_3 \mid \mathbf{H}_0, \mathbf{H}_1, \mathbf{H}_2) \cdots P(\mathbf{H}_{\tau^*} \mid \mathbf{H}_{\tau^*-1})$

$= P(\mathbf{H}_1, \mathbf{H}_2, \mathbf{H}_3 \mid \mathbf{H}_0) \cdots P(\mathbf{H}_{\tau^*} \mid \mathbf{H}_{\tau^*-1})$

$= \cdots$

$= P(\mathbf{H}_1, \mathbf{H}_2, \mathbf{H}_3, \ldots, \mathbf{H}_{\tau^*} \mid \mathbf{H}_0).$

Cela nous permet de simplifier l'expression donnant $Q(\mathbf{H}_0)$, qui devient :

$$\begin{aligned} Q(\mathbf{H}_0) &= \sum_{\mathbf{H}_1} \sum_{\mathbf{H}_2} \sum_{\mathbf{H}_3} \cdots \sum_{\mathbf{H}_{\tau^*}} \prod_{\tau=0}^{\tau^*-1} f(\mathbf{H}_\tau, \mathbf{H}_{\tau+1}) \\ &\quad P(\mathbf{H}_1, \mathbf{H}_2, \mathbf{H}_3, \ldots, \mathbf{H}_{\tau^*} \mid \mathbf{H}_0) Q(\mathbf{H}_{\tau^*}) \\ &= E_P \left[\prod_{\tau=0}^{\tau^*-1} f(\mathbf{H}_\tau, \mathbf{H}_{\tau+1}) \right]. \end{aligned}$$

Il s'agit d'une représentation d'échantillonnage pondéré avec distribution proposée P. Il est à noter que $Q(\mathbf{H}_{\tau^*})$ vaut 1 si le MRCA est constitué uniquement de marqueurs ancestraux sans mutations ; sinon, $Q(\mathbf{H}_{\tau^*})$ vaut 0. En simulant un grand nombre de graphes selon la distribution proposée P, la vraisemblance de r_T est ainsi estimée par l'espérance du produit des fonctions $f(\mathbf{H}_\tau, \mathbf{H}_{\tau+1})$.

Soit $\Theta = \{\theta, r_T\}$ l'ensemble des paramètres inconnus du processus. Étant donné $\Theta_0 = \{\theta_0, r_{T_0}\}$, nous pouvons estimer $Q_\Theta(\mathbf{H}_\tau)$ pour différentes valeurs de Θ. En fait,

$$Q_\Theta(\mathbf{H}_\tau) = \sum_{\mathbf{H}_{\tau+1}} h_{\Theta\Theta_0}(\mathbf{H}_\tau, \mathbf{H}_{\tau+1}) P_{\Theta_0}(\mathbf{H}_{\tau+1} \mid \mathbf{H}_\tau) Q_\Theta(\mathbf{H}_{\tau+1}),$$

où $h_{\Theta\Theta_0}(\mathbf{H}_\tau, \mathbf{H}_{\tau+1}) = \frac{f_\Theta(\mathbf{H}_\tau, \mathbf{H}_{\tau+1}) P_\Theta(\mathbf{H}_{\tau+1} | \mathbf{H}_\tau)}{P_{\Theta_0}(\mathbf{H}_{\tau+1} | \mathbf{H}_\tau)}$. Nous pouvons calculer la fonction $h_{\Theta\Theta_0}$

pour les différents évènements possibles :

$$h_{\Theta\Theta_0}(\mathbf{H}_\tau, \mathbf{H}_{\tau+1}) = \begin{cases} \frac{S_{(\mathbf{H}_\tau,\Theta_0)}}{D_{(\mathbf{H}_\tau,\Theta)}} & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est } C_i \text{ ou } C_{ij}^k, \\ \frac{\theta}{\theta_0}\frac{S_{(\mathbf{H}_\tau,\Theta_0)}}{D_{(\mathbf{H}_\tau,\Theta)}} & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est } M_i^j(m), \\ (n_j+1)(n_k+1)\frac{r_p}{r_{p_0}}\frac{S_{(\mathbf{H}_\tau,\Theta_0)}}{D_{(\mathbf{H}_\tau,\Theta)}} & \text{si le } (\tau+1)^{\text{ème}} \text{ évènement est } R_{ij}^k(p). \end{cases}$$

Il est à noter que le paramètre r_T est caché dans r_p. En effet, la position de la mutation a de l'influence sur la longueur des intervalles entre deux marqueurs. Par un raisonnement similaire à celui fait pour trouver $Q(\mathbf{H}_0)$, nous obtenons :

$$Q_\Theta(\mathbf{H}_0) = E_{P_{\Theta_0}}\left[\prod_{\tau=0}^{\tau^*-1} h_{\Theta\Theta_0}(\mathbf{H}_\tau, \mathbf{H}_{\tau+1})\right].$$

La méthode proposée pour évaluer la vraisemblance de r_T le long d'une séquence implique l'évaluation de Q_{Θ_0} pour $\Theta_0 = \{\{\theta_0, r_{T_1}\}, \{\theta_0, r_{T_2}\}, \ldots, \{\theta_0, r_{T_{L-1}}\}\}$. Nous utilisons $(L-1)$ ensembles de valeurs initiales. Pour la valeur initiale r_{T_p}, nous prenons le milieu de l'intervalle p $(1 \leq p \leq L-1)$. La vraisemblance dans l'intervalle p pour les valeurs autres que r_{T_p} est évaluée par le schéma d'échantillonnage pondéré que nous venons d'exposer : les graphes construits avec une valeur initiale r_{T_p} sont utilisés pour évaluer la vraisemblance dans la région (x_p, x_{p+1}) de la séquence.

Il est à noter que seule une valeur initiale θ_0 est utilisée pour le paramètre θ : celui-ci est supposé connu. C'est une hypothèse faible car la vraisemblance de la position de la mutation causant la maladie n'est que peu influencée par le taux de mutation. Il aurait aussi été possible de prendre plusieurs valeurs initiales pour θ.

CHAPITRE V

MÉTHODE AVEC ÉQUATIONS EXACTES

La méthode du chapitre IV ne fonctionne pas aussi bien que nous le voudrions. Le temps de calcul est assez grand, les courbes de vraisemblance sont différentes d'une simulation à l'autre et la position de la mutation n'est pas toujours estimée au bon endroit. Pour toutes ces raisons, la recherche d'une méthode alternative à celle de Larribe *et al.* est tout à fait justifiée.

La méthode que nous développons dans ce chapitre origine d'une idée de Larribe : calculer la vraisemblance de façon "exacte", en considérant toutes les histoires qui relient les données initiales à l'ancêtre commun plutôt que d'en échantillonner quelques-unes. Bien sûr, nous devons faire quelques compromis pour que le temps de calcul soit raisonnable et que cela soit réalisable.

Tout d'abord, nous devons limiter le nombre de recombinaisons permises. D'une part, un évènement de recombinaison augmente de un le nombre de séquences dans l'échantillon et donc rallonge le temps avant de trouver l'ancêtre commun. D'autre part, plus une histoire contient de recombinaisons, plus sa vraisemblance est petite et moins elle a d'influence dans la vraisemblance toale. Il s'agit alors de déterminer le nombre maximum d'évènements de recombinaison que nous devons permettre dans chaque intervalle pour obtenir suffisamment de précision tout en maintenant un temps de calcul raisonnable.

Ensuite, il n'est pas possible de considérer tous les marqueurs génétiques à la fois : le temps de calcul serait beaucoup trop grand. Nous considérons alors de façon séparée des sous-ensembles de marqueurs consécutifs. Par la suite, toutes les vraisemblances marginales calculées sont "assemblées" par le principe de vraisemblance composite. La vraisemblance composite est de plus en plus utilisée en statistique génétique. Par exemple, nous pouvons penser à la méthode de Fearnhead et Donnelly (2002) ou encore à celle d'Hudson (2001).

Nous illustrons d'abord à l'aide d'un exemple comment construire toutes les histoires compatibles avec les données et comment calculer la vraisemblance d'une histoire. Nous définissons ensuite, à l'aide du principe de vraisemblance composite, la fonction de vraisemblance de la position de la mutation que notre méthode cherche à évaluer. Pour ce faire, nous devons préalablement introduire le concept de fenêtre de marqueurs et discuter des deux types de vraisemblances marginales que nous devons calculer : sans et avec insertion de la mutation. Ensuite, le programme écrit en C++ qui permet d'appliquer la méthode est présenté. Finalement, nous présentons les résultats que la méthode a donnés lorsque nous l'avons testée avec des ensembles de données.

5.1 Construction d'une histoire et calcul de sa vraisemblance

La formule exacte de la vraisemblance lorsque nous travaillons avec des histoires stipule que la vraisemblance des paramètres génétiques est simplement la sommation, sur toutes les histoires compatibles avec les données, des vraisemblances des histoires sous ces paramètres génétiques (voir section 3.3).

Dans cette section, nous voyons comment contruire toutes les histoires compatibles avec les données et comment calculer la vraisemblance d'une histoire. Pour ce faire, considérons un exemple (voir figure 5.1).

Dans cet exemple, l'échantillon initial est composé de deux séquences de trois marqueurs. La

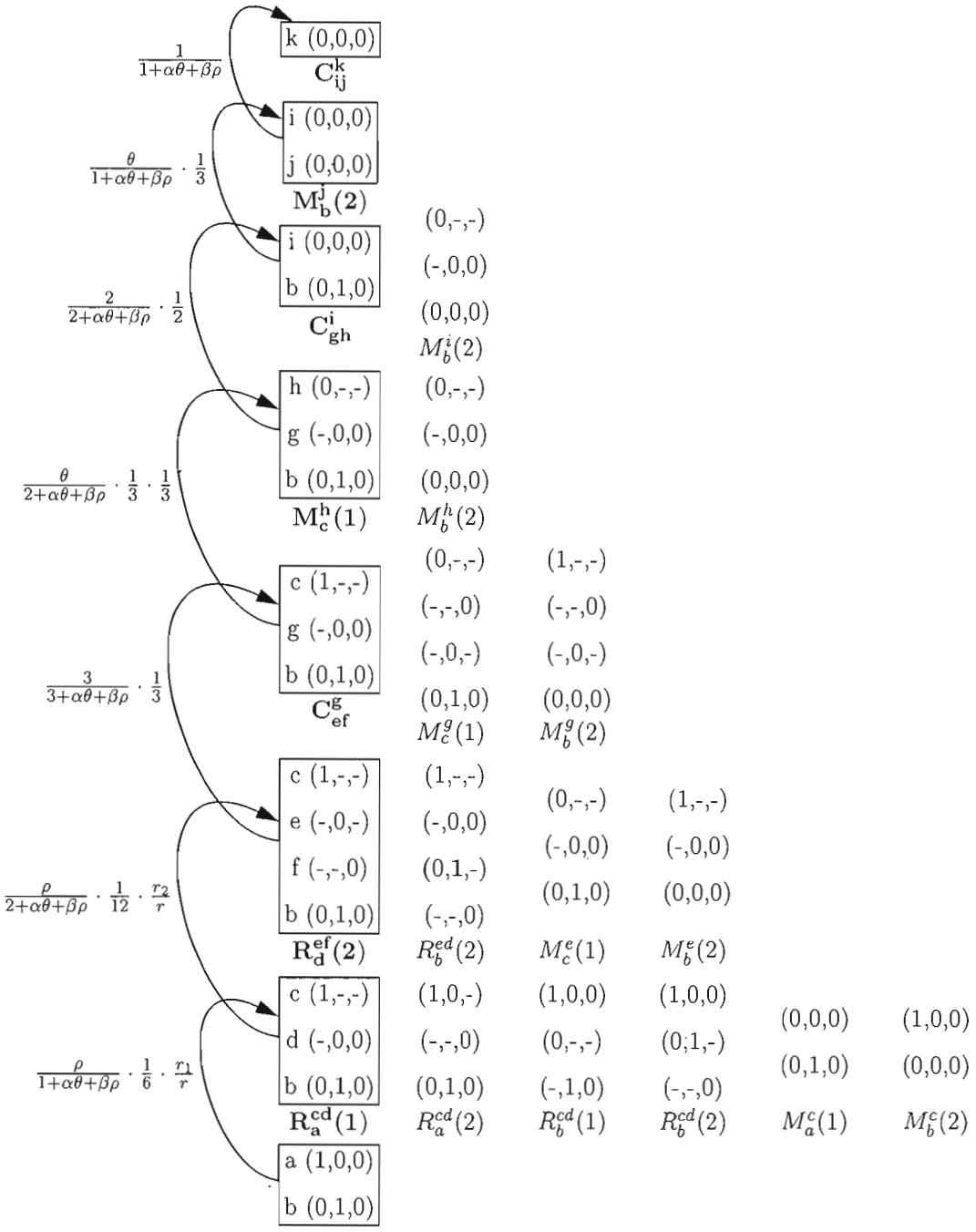


Figure 5.1 Exemple d'histoire possible dans l'ARG avec $\mathbf{H}_\tau = \{(1,0,0),(0,1,0)\}$ et où au maximum une recombinaison est permise par intervalle.

Évènement	Probabilité
C_i	$\frac{n_i-1}{n-1+\alpha\theta+\beta\rho}$
C_{ij}^k	$\frac{2(n_k+1-\delta_{ik}-\delta_{jk})}{n-1+\alpha\theta+\beta\rho}$
$M_i^j(m)$	$\frac{\theta(n_j+1)}{nL(n-1+\alpha\theta+\beta\rho)}$
$R_i^{jk}(p)$	$\frac{\rho(n_j+1)(n_k+1)r_p}{n(n+1)(n-1+\alpha\theta+\beta\rho)r}$

Figure 5.2 Probabilités de chacun des quatre évènements possibles.

première séquence, $(1,0,0)$ est de type a et la seconde, $(0,1,0)$, est de type b. Les quatre évènements permis sont ceux décrits à la sous-section 2.2.2, c'est-à-dire une coalescence identique, une coalescence distincte, une mutation et une recombinaison. Leurs probabilités sont données à la figure 5.2. Pour notre exemple, nous nous restreignons à une recombinaison maximale pour chacun des deux intervalles entre deux marqueurs consécutifs.

Partant de notre échantillon de départ $\mathbf{H}_0 = \{(1,0,0),(0,1,0)\}$, six choix sont possibles pour le premier évènement : une recombinaison de la première ou de la deuxième séquence dans le premier ou le deuxième intervalle (ces évènements sont notés $R_a^{cd}(1)$, $R_a^{cd}(2)$, $R_b^{cd}(1)$ et $R_b^{cd}(2)$ à la figure 5.1), une mutation sur le premier marqueur de la première séquence $(M_a^c(1))$ ou une mutation sur le deuxième marqueur de la deuxième séquence $(M_b^c(2))$. Puisque $n_a = n_b = 1$, aucune coalescence identique n'est possible. Aucune coalescence distincte n'est possible puisqu'il n'y a aucun marqueur ancestral.

Nous illustrons une histoire possible. Nous choisissons l'évènement de recombinaison sur la première séquence dans le premier intervalle $(R_a^{cd}(1))$ pour cette histoire. La probabilité associée à cet évènement, donnée à la figure 5.2, est :

$$\frac{\rho(n_j+1)(n_k+1)r_p}{n(n+1)(n-1+\alpha\theta+\beta\rho)r} = \frac{\rho r_1}{6(1+\alpha\theta+\beta\rho)r},$$

car avant la recombinaison aucune des deux nouvelles séquences n'existait ($n_j = 0$ et $n_k = 0$), il y avait deux séquences ($n = 2$) et la recombinaison se fait dans le premier intervalle ($p = 1$).

Pour le second évènement, plusieurs évènements sont encore possibles : $R_a^{ef}(2)$, $R_b^{ed}(2)$, $M_c^e(1)$ et $M_b^e(2)$. Notons toutefois qu'une recombinaison dans le premier intervalle n'est plus possible. Nous avons choisi un des quatre évènements possibles et calculé sa probabilité.

Nous continuons de cette façon jusqu'à l'ancêtre commun. La figure 5.3 illustre l'ARG associé à l'histoire considérée. Mentionnons que les carrés noirs, les carrés noirs avec un carré blanc à l'intérieur et les carrés blancs y représentent respectivement un marqueur ancestral sans mutation, un marqueur ancestral mutant et un marqueur non-ancestral.

Tel que mentionné à la section 3.3, la vraisemblance de l'histoire est le produit des probabilités de chacun des évènements qui en font partie. Elle est donc de

$$\frac{1}{5832} \cdot \frac{r_1 r_2}{r^2} \cdot \frac{6\theta\rho^2}{(1+\alpha\theta+\beta\rho)^3(2+\alpha\theta+\beta\rho)^2(3+\alpha\theta+\beta\rho)}.$$

Évidemment, en considérant toutes les autres possibilités de suites d'évènements, nous pouvons calculer les vraisemblances de toutes les autres histoires. Ceci nous permet de calculer la vraisemblance de l'échantillon, puisqu'elle est égale à la somme des vraisemblances de toutes les histoires.

5.2 Définition de la fonction de vraisemblance composite

Dans cette section, nous introduisons d'abord le concept de fenêtre de marqueurs et nous discutons des deux types de vraisemblance marginale que nous devons calculer : sans ou avec insertion de la mutation. Ensuite, nous définissons, à l'aide du principe de vraisemblance composite, la fonction de vraisemblance de la position de la mutation que notre méthode

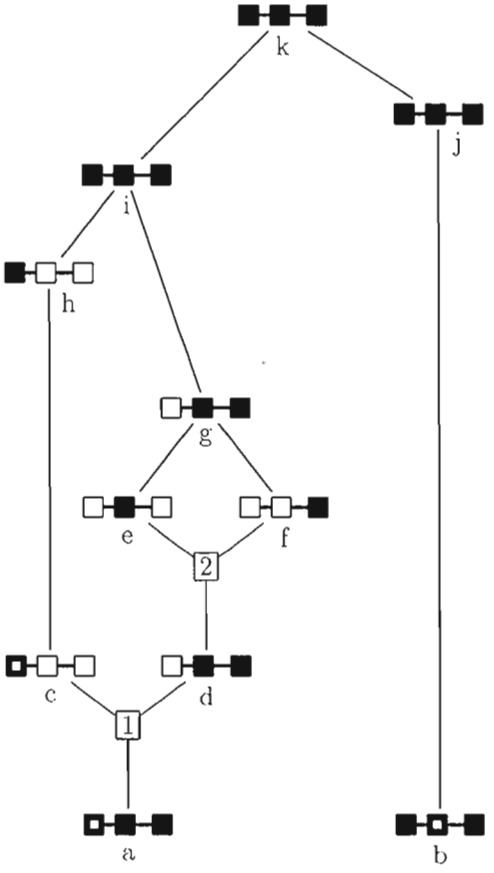


Figure 5.3 ARG associé à l'histoire illustrée à la figure 5.1.

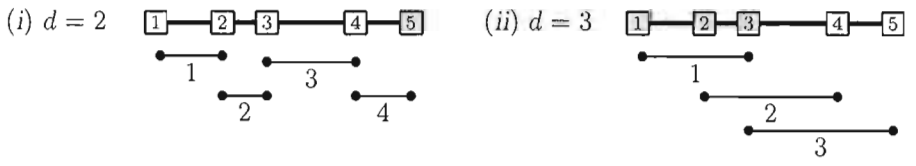


Figure 5.4 Exemples de fenêtres pour une séquence de cinq marqueurs : (a) quatre fenêtres de taille $f = 2$; (b) trois fenêtres de taille $f = 3$ (tiré de Larribe et Lessard, 2008).

cherche à évaluer.

5.2.1 Concept de fenêtre de marqueurs

Nous avons déjà spécifié qu'il n'est pas possible de construire toutes les histoires pour un trop grand nombre de marqueurs génétiques à la fois puisque cela nécessiterait trop de temps de calcul. Nous considérons alors des fenêtres de marqueurs consécutifs. Par exemple, si nous avons cinq marqueurs par séquence, nous pouvons considérer des fenêtres de taille $f = 2$. Dans ce cas, la première fenêtre contient les deux premiers marqueurs, la deuxième fenêtre contient les deuxième et troisième marqueurs, et ainsi de suite (voir figure 5.4 (a)). Si nous considérons plutôt des fenêtres de taille $f = 3$, alors la première fenêtre contient les trois premiers marqueurs, la deuxième fenêtre les marqueurs 2, 3 et 4 et la troisième fenêtre les trois derniers marqueurs (voir figure 5.4 (b)).

5.2.2 Vraisemblances marginales

Après avoir déterminé la taille f des fenêtres, nous calculons d'abord, sans insérer la mutation, les fonctions de vraisemblance marginale dans chacun des intervalles de chacune des fenêtres. Nous notons $Q(\mathbf{H}_0^{mg})$ la vraisemblance des données de la fenêtre g dans l'intervalle entre le $m^{\text{ème}}$ et le $(m+1)^{\text{ème}}$ intervalle, que nous appelons dorénavant l'intervalle m.

Dans l'exemple de la section 4.1, la vraisemblance de l'histoire considérée est de

$$\frac{1}{5832} \cdot \frac{r_1 r_2}{r^2} \cdot \frac{6\theta\rho^2}{(1+\alpha\theta+\beta\rho)^3(2+\alpha\theta+\beta\rho)^2(3+\alpha\theta+\beta\rho)}.$$

Dans cette expression, il y a sept variables : r_1, r_2, r, α, θ, β et ρ. Les variables r, α, θ, β et ρ peuvent être calculées à l'aide des paramètres connus N_e et μ et de l'information sur nos séquences. Ainsi, nous pouvons les remplacer par leurs valeurs dans l'expression de la vraisemblance. Si la mutation n'est pas incluse dans nos séquences de départ (1,0,0) et (0,1,0), alors r_1 et r_2 sont connues puisque nous connaissons les distances entre chacun des marqueurs standards. Nous pouvons donc les remplacer par leurs valeurs dans l'expression de la vraisemblance. Ainsi, la vraisemblance de l'histoire est un nombre, ce qui est le cas lorsque nous considérons seulement les données initiales.

Nous calculons aussi, après insertion de la mutation, ces mêmes fonctions de vraisemblance marginale pour chacun des intervalles de chacune des fenêtres. Nous notons $Q(\mathbf{H}_0^{mg}, r_T)$ la vraisemblance des données de la fenêtre g auxquelles nous avons inséré la mutation dans l'intervalle m.

Si, dans l'exemple de la section 4.1, les séquences (1,0,0) et (0,1,0) proviennent des séquences à deux marqueurs (1,0) et (0,0) auxquelles nous avons inséré la mutation (entre les deux marqueurs), alors nous ne connaissons pas les variables r_1 et r_2 puisque la position de la mutation (r_T) est inconnue. Dans ce cas, la vraisemblance est une fonction de r_T, qui s'exprime sous la forme d'une fonction des variables r_1 et r_2 qui sont déterminées par r_T. La vraisemblance est toujours une fonction de r_T lorsque nous considérons les données initiales auxquelles nous avons inséré la mutation.

Les deux calculs de vraisemblance marginale nous permettent de calculer la vraisemblance de r_T, conditionnellement aux données :

$$L_{m,g}(r_T|\mathbf{H}_0^{mg}) = \frac{Q(\mathbf{H}_0^{mg}, r_T)}{Q(\mathbf{H}_0^{mg})}.$$

Il est à noter que, par convention, $L_{m,g}(r_T|\mathbf{H}_0^{mg}) = 1$ si la mutation n'est pas dans l'intervalle m.

5.2.3 Fonction de vraisemblance composite

Lorsque la taille f de fenêtre utilisée est supérieure à 2, plusieurs fenêtres contiennent le même intervalle (voir figure 5.4). Ainsi, nous avons besoin de faire une moyenne pondérée pour obtenir la vraisemblance dans chaque intervalle. Pour ce faire, nous utilisons le concept de vraisemblance composite (Lindsay, 1988 et Varin et Vidoni, 2005). Notre fonction de vraisemblance composite conditionnelle (VCC_f) pour des fenêtres de taille f est (Larribe et Lessard, 2008 - en modifiant un peu la notation) :

$$VVC_f(r_T) = \prod_{m=1}^{L-2} \left(\prod_{\text{g : int. m}\in\text{g}} L_{m,g}(r_T|\mathbf{H}_0^{mg}) \right)^{w_m},$$

où w_m est l'inverse du nombre de fenêtres qui contiennent l'intervalle m. Cette variable représente aussi le poids accordé à chacune des évaluations de la vraisemblance pour l'intervalle m. Dans notre cas, tous les poids sont les mêmes pour chacune des évaluations de vraisemblance.

5.3 Implémentation informatique

Un programme écrit en C++ (voir annexe) permet de calculer la courbe de vraisemblance à partir de données. Voici les algorithmes principaux du programme, suivis d'explications pour certaines des étapes de ces algorithmes. L'algorithme 1 est celui du fichier "Main",

l'algorithme 2 correspond à la fonction "vraisTot" et l'algorithme 3 décrit la fonction "recurs".

C'est dans le fichier "Main" que les paramètres et les données sont saisies et que la fonction "vraisTot" est appellée. Cette fonction donne les commandes pour calculer toutes les vraisemblances marginales et la vraisemblance totale. Afin de calculer les vraisemblances marginales, la fonction "recurs" est appelée. Il s'agit d'une fonction qui calcule la vraisemblance d'un ensemble de données en parcourant de façon récursive les histoires.

Algorithme 1 Main

1: Lire les paramètres.
2: Calculer la valeur de la variable "theta", les matrices d'haplotypes et d'évènements et la matrice qui donne les positions auxquelles la vraisemblance totale est évaluée.
3: Lire les données.
4: Séparer les données.
5: Calculer la matrice de vraisemblance totale.

Explication des principales étapes de l'algorithme 1

1. Le programme lit le fichier de paramètres, ce qui lui donne la valeur des variables suivantes :

- ne : la taille effective de la population
- mu : le taux de mutation par séquence par génération
- nbCan : le nombre de positions différentes où la vraisemblance est évaluée par intervalle entre deux marqueurs
- resFile : le fichier de résultats
- dataFile : le fichier de données
- distI : le vecteur de distances entre les marqueurs

- f : le nombre de marqueurs génétiques considérés par fenêtre
- recMax : le vecteur du nombre maximal de recombinaisons permises par intervalle

2. Voici ce que le programme calcule :

- Le taux de mutation à l'échelle : $\theta = 4 * ne * mu$
- Les matrices tHap1 et tHap2 qui contiennent toutes les séquences possibles à f et $(f + 1)$ marqueurs
- Les matrices tCoa1 et tCoa2 qui indiquent quelle séquence est obtenue par la coalescence d'une paire de séquences distinctes, pour toutes les séquences possibles à f et $(f + 1)$ marqueurs
- Les matrices tMut1 et tMut2 qui indiquent quelle séquence est obtenue par le retrait d'une mutation, pour toutes les séquences possibles à f et $(f + 1)$ marqueurs
- Les matrices tRecG1, tRecD1, tRecG2 et tRecD2 qui indiquent quelles séquences sont obtenues par la recombinaison d'une séquence dans un intervalle donné, pour toutes les séquences possibles à f et $(f + 1)$ marqueurs
- La matrice rT qui donne les positions auxquelles la vraisemblance totale est évaluée

3. Le programme va chercher les informations contenues dans le fichier de données spécifié dans les paramètres.

4. Le programme sépare les informations contenues dans le fichier de données. Il crée ainsi un vecteur contenant les multiplicités de chaque séquence, un vecteur contenant le statut de la séquence (0 indique un contrôle et 1 un cas) et une matrice contenant les haplotypes.

5. Le programme calcule la matrice de vraisemblance totale à l'aide de la fonction "vrais-

Tot".

Explication des principales étapes de l'algorithme 2

1. Le vecteur donnant le nombre de fenêtres qui recoupent chacun des intervalles permet de déterminer le poids accordé à chacune des évaluations de la vraisemblance pour un intervalle donné. Rappelons que, pour un intervalle donné, ce poids est l'inverse du nombre de fenêtres qui contiennent l'intervalle.

5. Nous obtenons donc une matrice contenant les séquences utilisées pour le calcul de la vraisemblance sans mutation pour la fenêtre choisie.

6. Le programme identifie les numéros des séquences à l'aide de la matrice tHap1 qui associe un numéro à chacune des séquences possibles à f marqueurs.

7. Puisque nous n'avons conservé qu'une partie des séquences de notre échantillon, il est probable qu'il y ait répétition de séquences identiques dans la matrice contenant les séquences utilisées. À cette étape, il s'agit de regrouper ces séquences, ce qui va modifier le vecteur de multiplicités ainsi que le vecteur identifiant les numéros des séquences utilisées.

8. Puisque nous n'avons conservé qu'une partie des séquences de notre échantillon, nous ne conservons aussi qu'une partie du vecteur qui contient les distances entre les marqueurs.

9. Voici ce que le programme calcule :

- Le taux de recombinaison par séquence par génération : $r = \sum_{p=1}^{L-1} r_p$
- Le taux de recombinaison à l'échelle : $\rho = 4N_e r$

10. Pour les intervalles correspondants à la fenêtre choisie, le programme calcule la vraisemblance marginale sans mutation à l'aide de la fonction "recurs" et ajuste la vraisemblance

Algorithme 2 Fonction vraisTot

1: Calculer le vecteur donnant le nombre de fenêtres qui recoupent chacun des intervalles.
2: **Pour** chaque fenêtre
3: **Pour** chaque séquence
4: **Pour** chaque position correspondant à la fenêtre choisie
5: Ajouter le marqueur à la matrice contenant les séquences utilisées.
6: **Fin pour**
7: **Fin pour**
8: Identifier le numéro des séquences utilisées.
9: Regrouper les séquences maintenant identiques, ajuster leurs multiplicités et identifier leurs numéros.
10: Modifier le vecteur qui contient les distances entre les marqueurs.
11: Calculer les valeurs des variables r et ρ.
12: Calculer et ajuster la vraisemblance sans mutation pour les intervalles appropriés.
13: **Pour** chaque intervalle où la mutation peut être insérée
14: Faire une copie de la matrice contenant les séquences utilisées.
15: **Pour** chaque séquence
16: Ajouter la mutation dans l'intervalle choisi.
17: **Fin pour**
18: Identifier le numéro des séquences utilisées.
19: Regrouper les séquences maintenant identiques, ajuster leurs multiplicités et identifier leurs numéros.
20: Modifier le vecteur qui contient les distances entre les marqueurs.
21: Calculer et ajuster la vraisemblance avec mutation pour l'intervalle choisi.
22: Effacer les variables associées au calcul de la vraisemblance avec mutation.
23: **Fin pour**
24: Effacer les variables associées au calcul de la vraisemblance sans mutation.
25: **Pour** chaque candidat
26: **Pour** chaque intervalle
27: Calculer la vraisemblance de la mutation.
28: **Fin pour**
29: **Fin pour**
30: **Fin pour**
31: **Retourner** le vecteur de vraisemblance de la mutation

composite sans mutation.

14. Nous obtenons donc une matrice contenant les séquences utilisées pour le calcul de la vraisemblance avec mutation pour l'intervalle et la fenêtre choisis.

15. Le programme identifie les numéros des séquences à l'aide de la matrice tHap2 qui associe un numéro à chacune des séquences possibles à $(f+1)$ marqueurs.

17. Nous plaçons la mutation à un nombre nbCan de positions différentes dans l'intervalle choisi. Ainsi, le vecteur qui contient les distances entre les marqueurs devient un vecteur contenant nbCan vecteurs de distances entre les marqueurs.

18. Pour l'intervalle choisi dans la fenêtre, le programme calcule la vraisemblance marginale avec mutation à l'aide de la fonction "recurs" et ajuste la vraisemblance composite avec mutation.

23. À l'aide des vraisemblances composites avec et sans mutation, le programme calcule la vraisemblance de la mutation à chacune des nbCan $\times$ (nombre d'intervalles) positions où elle est évaluée.

Explication des principales étapes de l'algorithme 3

1. probTotale est le vecteur qui donnera la vraisemblance calculée à l'aide de toutes les histoires possibles.

3. La condition est qu'il y ait au moins un évènement possible pour la configuration actuelle. Cela ne serait pas le cas si, par exemple, la configuration comprend les quatre séquences (0 0 0), (0 0 1), (0 1 0) et (0 1 1) et qu'aucune recombinaison n'est permise. Nous ne considérons pas les histoires qui mènent à une impasse de la sorte.

6. L'évènement considéré aura une seule probabilité si nous calculons la vraisemblance sans

Algorithme 3 Fonction recurs

```
1: Initialiser les entrées du vecteur probTotale à 0.
2: Construire la liste des évènements possibles pour les paramètres actuels.
3: Si la liste n'est pas vide alors
4:    Pour chaque évènement de la liste
5:      Changer le nom des paramètres variables.
6:      Modifier les paramètres variables et mettre la ou les probabilités de l'évènement
        considéré dans le vecteur prob.
7:      Si nous sommes rendus à l'ancêtre commun alors
8:        Pour chaque position j du vecteur prob
9:          probTotale_i[j] = prob[j]
10:       Fin pour
11:     Sinon
12:       Pour chaque position j du vecteur prob
13:         probTotale_i[j] = prob[j] * contRecurs[j]
14:       Fin pour
15:     Fin si
16:     Pour chaque position j du vecteur prob
17:       probTotale[j] = probTotale[j] + probTotale_i[j]
18:     Fin pour
19:   Fin pour
20: Fin si
21: Retourner probTotale
```

mutation et plusieurs probabilités si nous calculons la vraisemblance avec mutation.

9. probTotale_i est un vecteur contenant la ou les probabilités de toutes les histoires commençant par l'évènement choisi.

13. contRecurs est un vecteur contenant la ou les vraisemblances retournées par la fonction recurs.

17. À la fin de la boucle sur les évènements de la liste, nous additionnons probTotale_i au vecteur donnant la vraisemblance. Rappelons que la vraisemblance des paramètres génétiques est simplement la sommation, sur toutes les histoires compatibles avec les données, des vraisemblances des histoires sous ces paramètres génétiques.

5.4 Résultats

Nous présentons maintenant quelques résultats de la méthode, que nous avons testée sur des ensembles de données simples. Nous présentons d'abord ces ensembles de données ainsi que les paramètres utilisés pour nos tests. Nous discutons ensuite des temps requis pour obtenir les résultats et finalement, nous présentons les résultats. Nous nous intéressons à la forme de la courbe de vraisemblance que la méthode génère, à l'exactitude de l'estimation fournie et à l'effet du nombre de recombinaisons maximum par intervalle.

5.4.1 Ensembles de données utilisés

Le premier ensemble de données utilisé comprend seulement deux séquences de deux marqueurs encadrant la mutation (voir figure 5.5). Nous posons $\rho = 20$ et $Ne = 10\ 000$, ce qui donne $r = r_1 = 0,05\ cM$. Le but de considérer un ensemble de données avec peu de séquences et peu de marqueurs est d'être en mesure de permettre un plus grand nombre de recombinaisons. Nous désignons ce premier ensemble de données "A". Notons que c'est un exemple fictif et expérimental qui est destiné à étudier et vérifier le comportement de

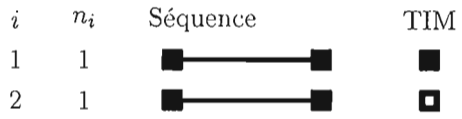


Figure 5.5 Ensemble de données "A" : multiplicité de chaque séquence i (n_i) pour $i = 1, 2$. La séquence 2 est un cas ; la séquence 1 est un témoin.

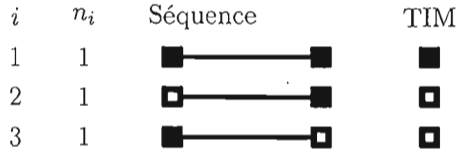


Figure 5.6 Ensemble de données "B" : multiplicité de chaque séquence i (n_i) pour $i = 1, 2, 3$. Les séquences 2 et 3 sont des cas ; la séquence 1 est un témoin.

notre méthode.

Le second ensemble de données comprend trois séquences de deux marqueurs encadrant la mutation (voir figure 5.6). Nous choisissons aussi $\rho = 20$ et $Ne = 10\ 000$, ce qui donne $r = r_1 = 0,05\ cM$. Il s'agit encore une fois d'un exemple fictif et expérimental avec peu de séquences et peu de données qui a pour but d'étudier le comportement de notre méthode. Nous désignons ce second ensemble de données "B".

Les données "C" proviennent de Larribe (2003), mais à une différence près : chacune des multiplicités a été ramenée à un pour que le temps de calcul demeure raisonnable. L'ensemble de données original a été simulé selon un modèle neutre de Wright-Fisher (Neuhauser, 2001 ou Nordborg, 2001, par exemple) par le programme ms (Hudson, 2002) en supposant une taille de population constante : $Ne = 10\ 000$. Les données "C" sont

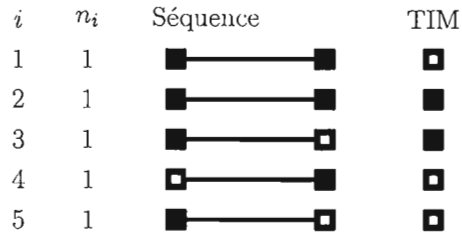


Figure 5.7 Ensemble de données "C" : multiplicité de chaque séquence i (n_i) pour $i = 1, \ldots, 5$. Les séquences 1, 4 et 5 sont des cas ; les autres, des témoins.

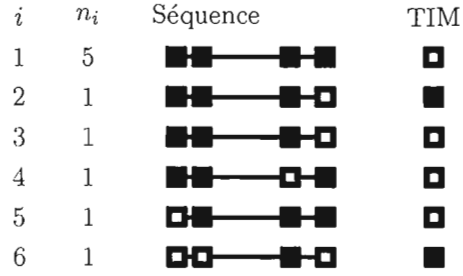


Figure 5.8 Ensemble de données "D" : multiplicité de chaque séquence i (n_i) pour $i = 1, \ldots, 6$. Les séquences 1, 3, 4 et 5 sont des cas ; les autres, des témoins.

constituées de cinq séquences de deux marqueurs encadrant la mutation (voir figure 5.7) et tel que $\rho = 20$, ce qui donne $r = r_1 = 0,05\ cM$.

Le dernier ensemble de données provient directement de Larribe (2003), sans modification cette fois-ci. Cet ensemble de données, que nous notons "D", a aussi été simulé avec le programme d'Hudson. Les données "D" sont constituées de dix séquences de quatre marqueurs encadrant la mutation, avec six types différents de séquences (voir figure 5.8) et tel que $\rho = 5$, ce qui donne $r = 0,0125\ cM$. Les distances entre les marqueurs sont $r_1 = 0,0007382\ cM$, $r_2 = 0,0096382\ cM$ et $r_3 = 0,0021236\ cM$.

5.4.2 Paramètres utilisés

Pour tester les quatre ensembles de données, certains paramètres ont été fixés. Nous considérons une taille effective de population de $Ne = 10\,000$ ainsi qu'un taux de mutation par marqueur de 10^{-5} (μ est donc $10^{-5} \times L$). La vraisemblance est évaluée $nbCan = 30$ fois dans chaque intervalle. Cela donne suffisamment de points pour voir l'allure de la courbe sans toutefois ralentir trop les calculs. Seules des fenêtres de taille $f = 2$ sont utilisées. Puisque la méthode est très exigeante au niveau du temps de calcul, il n'est pas possible de calculer suffisamment longtemps pour des fenêtres plus grandes.

5.4.3 Temps de calcul

Avant de présenter les résultats obtenus avec les trois ensembles de données étudiés, nous nous intéressons aux temps de calcul requis par la méthode. Ceci nous permet de comprendre combien de recombinaisons nous pouvons permettre tout en gardant un temps de calcul raisonnable.

Puisque nous considérons toutes les histoires possibles, la méthode est très exigeante au niveau du temps de calcul. En effet, celui-ci est proportionnel au nombre total d'évènements de toutes les histoires et ce nombre croît très rapidement à mesure que nous augmentons le nombre maximal de recombinaisons permises. La figure 5.9 illustre le nombre requis d'évènements pour faire toutes les histoires possibles selon le nombre maximum de recombinaisons. Des tests ont été effectués avec trois ensembles de données de deux séquences qui contiennent chacune deux marqueurs. Dans tous les cas, la courbe de tendance la plus appropriée est celle de type exponentiel. Ceci est un résultat important à connaître. En comprenant que l'augmentation du nombre total d'évènements à considérer se fait de façon exponentielle, nous réalisons que le fait d'augmenter de un le nombre de recombinaisons permises pour un intervalle donné décuple le temps de calcul. Ainsi, si le temps de calcul est déjà très grand, alors il n'est pas possible d'obtenir des résultats dans des délais

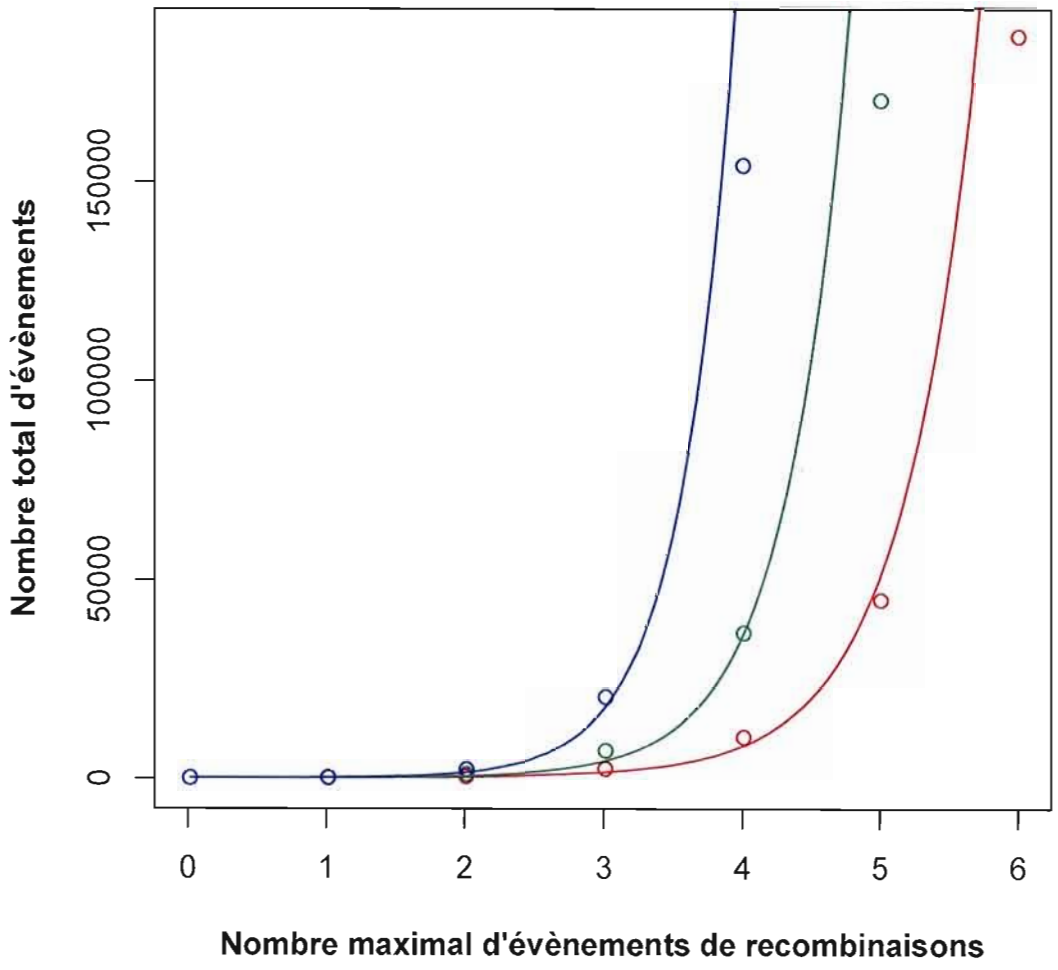


Figure 5.9 Croissance exponentielle du nombre total d'évènements requis pour faire toutes les histoires. La courbe bleue est celle des séquences (0,1) et (1,0), la courbe verte s'applique aux séquences (1,0) et (1,1) et la courbe rouge est obtenue des séquences (0,0) et (0,1).

raisonnables si nous augmentons davantage le nombre maximum de recombinaisons.

Dans le graphique de la figure 5.9, nous remarquons aussi que, selon l'information sur les séquences, le nombre requis d'histoires grossit plus ou moins rapidement. Donc, la nature des séquences influence le rythme auquel le nombre d'évènements croît. Il y a également d'autres facteurs, comme le nombre de séquences, qui influence aussi le rythme auquel le nombre d'évènements croît. Dans le cas du nombre de séquences, évidemment, plus il y a de séquences, plus le nombre d'évènements augmente rapidement. Néanmoins, peu importe ces facteurs, la progression du nombre d'évènements et du temps de calcul se fait toujours de façon exponentielle.

Pour les tests que nous avons effectués sur les données "A", "B", "C" et "D", le nombre total d'évènements nécessaires pour évaluer les vraisemblances se rend jusqu'à 1692 millions dans le cas le plus long. Cela nous a pris environ trois minutes de temps de calcul par million d'évènements parcourus.

5.4.4 Résultats obtenus avec les données "A"

Les résultats obtenus avec les données "A" illustrent bien la convergence que nous obtenons en augmentant peu à peu les valeurs du vecteur donnant le nombre maximal de recombinaisons permises par intervalle, vecteur que nous notons ϕ. Indépendamment de la fenêtre de marqueurs dans laquelle nous nous situons, la $i^{\text{ème}}$ entrée de ϕ donne le nombre maximal de recombinaisons permises pour le $i^{\text{ème}}$ intervalle de la fenêtre. Les données ont été testées avec $\phi = (0,0)$, $\phi = (0,1)$, $\phi = (1,0)$, $\phi = (1,1)$, $\phi = (1,2)$, $\phi = (2,1)$, $\phi = (2,2)$, $\phi = (2,3)$, $\phi = (3,2)$ et $\phi = (3,3)$.

En examinant les graphiques des vraisemblances obtenues pour ces dix cas (voir figure 5.10), nous remarquons que les résultats semblent converger vers les courbes de $\phi = (1,1)$, $\phi = (2,2)$ et $\phi = (3,3)$. Nous notons que mis à part le cas dégénéré $\phi = (0,0)$ qui produit

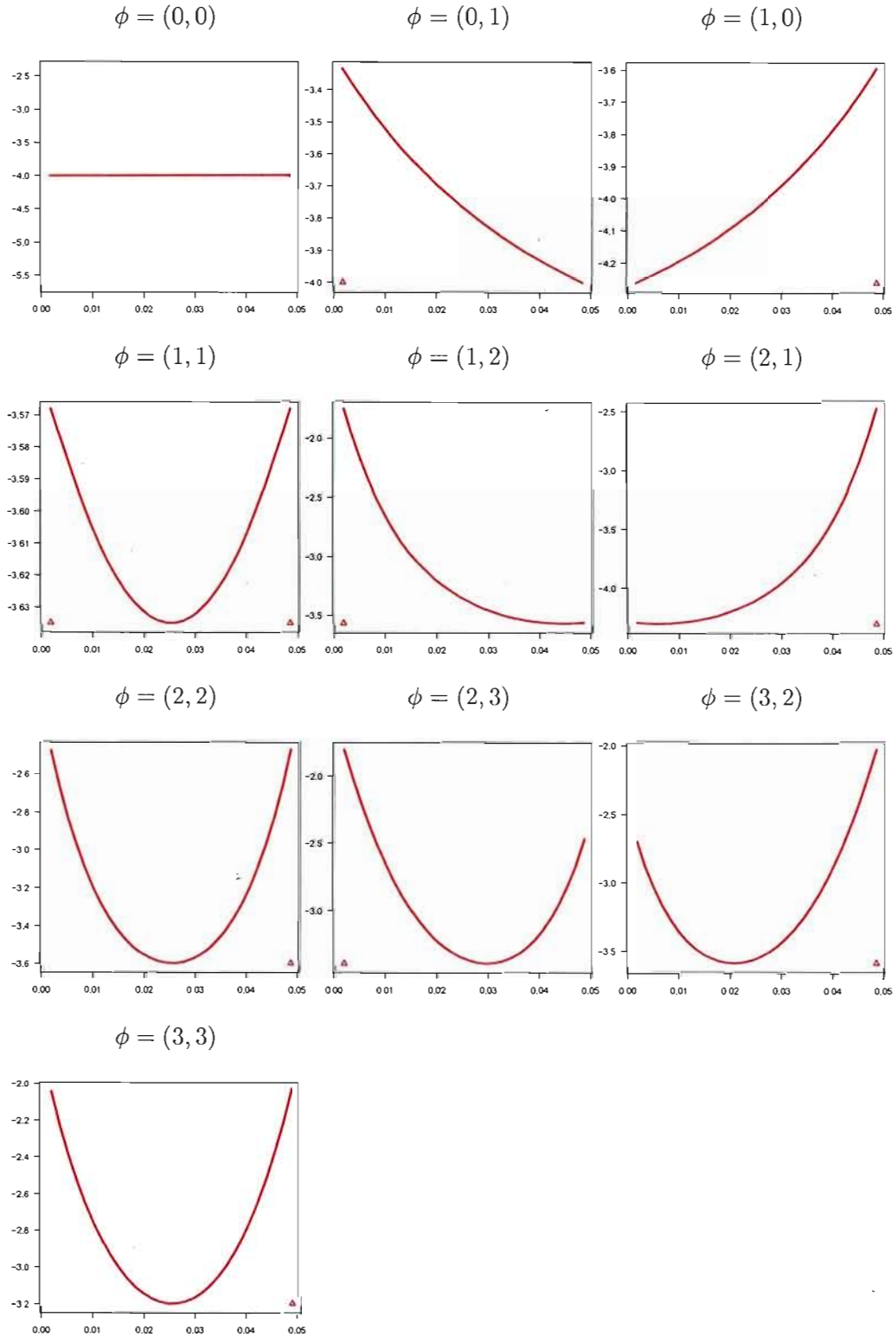


Figure 5.10 Courbes de vraisemblance obtenues pour les données "A" avec la méthode développée en utilisant des valeurs de ϕ de (0,0), (0,1), (1,0), (1,1), (1,2), (2,1), (2,2), (2,3), (3,2) et (3,3).

une infinité d'estimations, les courbes sont symétriques (ou presque) et ne varient pas beaucoup lorsque le nombre maximal de recombinaisons permises est le même dans chaque intervalle.

Nous remarquons aussi que plus les valeurs de ϕ sont grandes, moins les résultats sont sensibles à une différence de un dans le nombre maximal de recombinaisons d'un intervalle à l'autre. Ainsi, les courbes de $\phi = (1, 2)$ et $\phi = (2, 1)$ sont moins asymétriques que celles de $\phi = (0, 1)$ et $\phi = (1, 0)$, et celles de $\phi = (2, 3)$ et $\phi = (3, 2)$ le sont encore moins. Les résultats semblent converger vers des courbes symétriques.

Finalement, il est à noter que la forme en "U" vers laquelle semblent converger les courbes de vraisemblance est inhabituelle. Nous ne retrouvons habituellement pas ce genre de forme de courbe dans les méthodes d'estimation de vraisemblance de paramètres génétiques. Dans le cas d'une courbe parfaitement symétrique (voir le graphique de $\phi = (1, 1)$), cela a l'inconvénient de produire deux estimations. Dans notre cas, la forme en "U" est probablement due au fait que nous avons utilisé un échantillon avec trop peu de séquences et de marqueurs.

5.4.5 Résultats obtenus avec les données "B"

Les données "B" comportent une séquence de plus que les données "A" (trois par rapport à deux), avec le même nombre de marqueurs par séquence. Cela a eu un effet important dans les temps de calculs : cette fois-ci, nous n'avons pu nous rendre qu'à $\phi = (2, 1)$. Toutefois, les courbes de vraisemblance semblent converger vers une forme en "U inversé" (voir figure 5.11). C'est également le type de forme de courbe que nous obtenons avec la méthode de Larribe *et al.* (voir la courbe en bleu, qui est la vraisemblance combinée des trois courbes en noir obtenues en simulant à chaque fois 10 000 graphes). En comparant l'estimation du maximum de vraisemblance de notre méthode en utilisant $\phi = (1, 1)$ avec celle de Larribe *et al.*, nous obtenons des résultats similaires : $\hat{r_T} = 0{,}0250000$ cM pour notre méthode par

rapport à 0,0240099 cM pour la méthode de Larribe *et al.*

5.4.6 Résultats obtenus avec les données "C"

Pour les données "C", rappelons d'abord qu'elles sont tirées de Larribe (2003), à la seule différence que les multiplicités ont toutes été ramenées à un. À la figure 5.12, nous examinons l'effet de cette modification des multiplicités pour la méthode de Larribe *et al.* Nous en profitons également pour examiner l'effet du nombre de graphes produits en comparant les résultats avec ceux obtenus lorsque nous produisons 1000 graphes (plutôt que d'en produire 10 000). Nous réalisons que, pour un changement des multiplicités, de même que pour un changement dans le nombre de graphes produits, les résultats varient peu. L'allure générale de la courbe produite reste la même et l'estimation du maximum de vraisemblance ne varie pas beaucoup.

Examinons maintenant les résultats produits avec notre méthode. Pour les données "C", nous avons davantage été restreints par les temps de calculs élevés puisqu'elles comportent plus de séquences que les données "B". Nous avons pu obtenir les graphiques pour des valeurs de $\phi = (0,1)$, $\phi = (1,1)$, $\phi = (1,2)$ et $\phi = (2,1)$ (voir figure 5.13). Notons que les graphiques n'ont pas pu être obtenus pour $\phi = (0,0)$ et $\phi = (1,0)$ en raison d'une particularité de notre ensemble de données. Puisque nous utilisons un modèle de mutations "Infinite sites" et que nous supposons que le MRCA est une séquence n'ayant aucune mutation, nous pouvons appliquer le test des trois gamètes (Myers et Griffiths, 2003, par exemple). Ce test stipule que pour toute paire de marqueurs, si nous retrouvons les trois types 01, 10 et 11 dans l'échantillon, alors il doit y avoir absolument au moins une recombinaison dans l'intervalle entre ces deux marqueurs. Pour les données "C", lorsque nous insérons la mutation entre les deux marqueurs de positions connues, nous obtenons les trois types 01, 10 et 11 pour la mutation et le dernier marqueur. Il doit donc absolument y avoir une recombinaison dans le second intervalle.

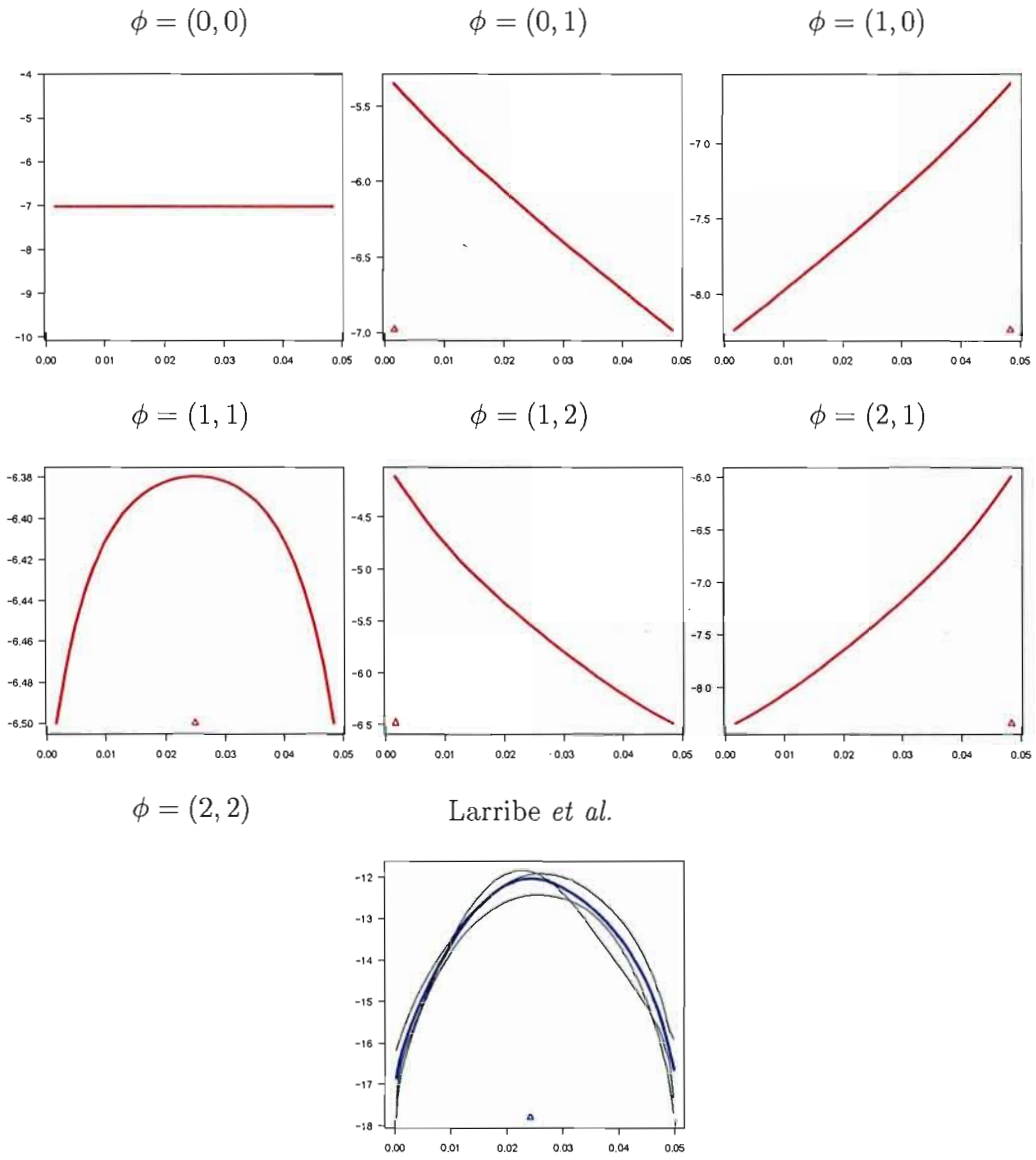


Figure 5.11 Courbes de vraisemblance obtenues pour les données "B" avec la méthode développée en utilisant des valeurs de ϕ de (0,0), (0,1), (1,0), (1,1), (1,2) et (2,1) et avec la méthode de Larribe *et al.*

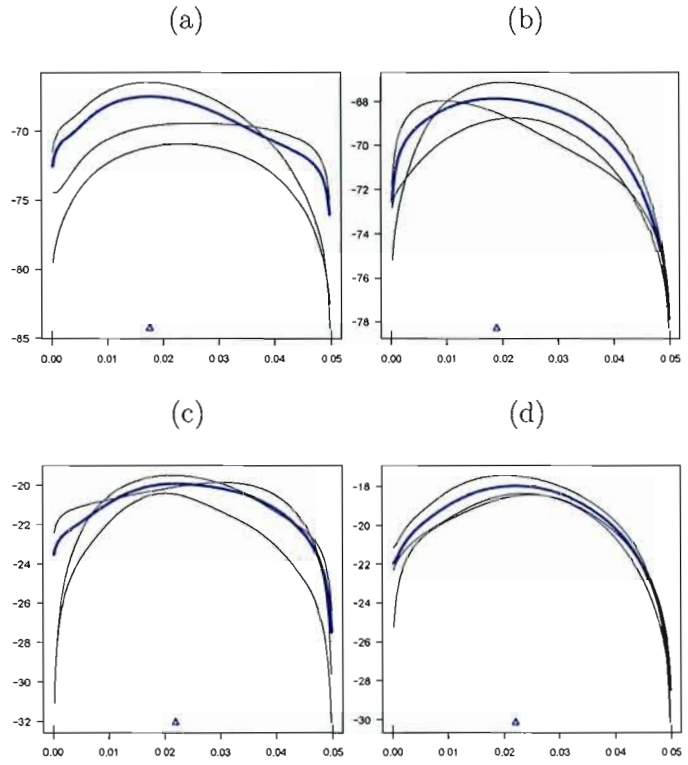


Figure 5.12 Courbes de vraisemblance obtenues avec la méthode de Larribe *et al.*, en utilisant : (a) les données à l'origine des données "C" et en produisant 1000 graphes, (b) les données à l'origine des données "C" et en produisant 10 000 graphes, (c) les données "C" et en produisant 1000 graphes et (d) les données "C" et en produisant 10 000 graphes.

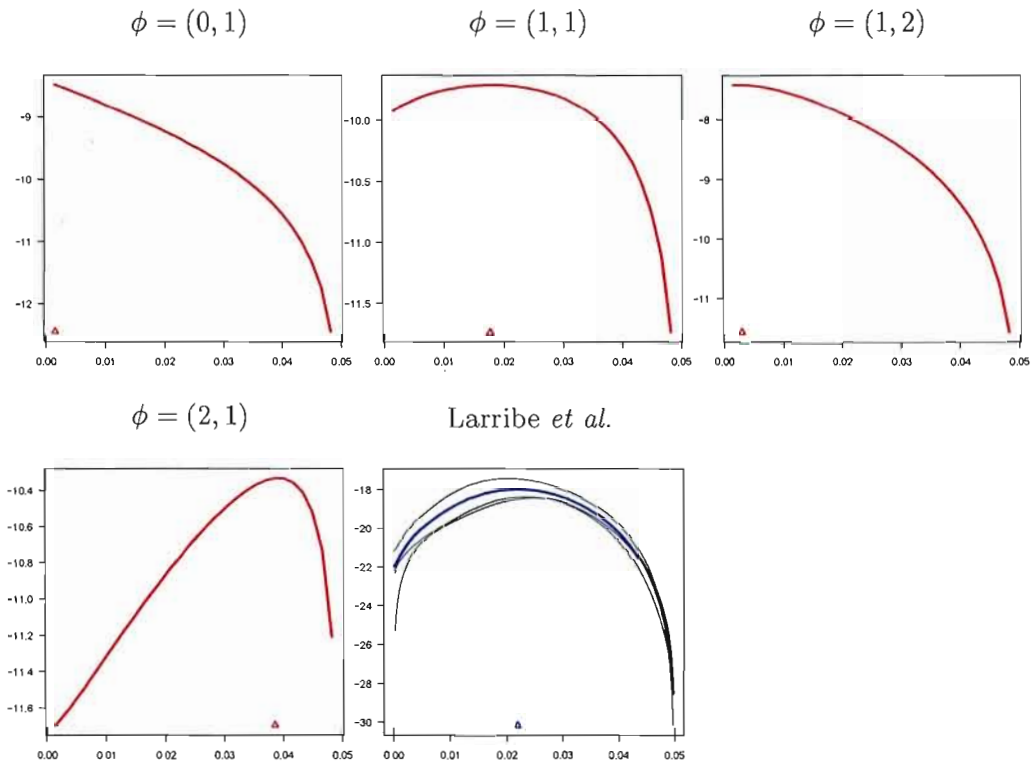
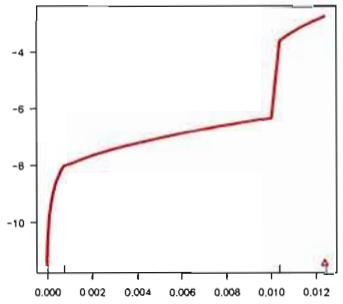


Figure 5.13 Courbes de vraisemblance obtenues pour les données "C" avec la méthode développée en utilisant des valeurs de ϕ de (0,1), (1,1), (1,2) et (2,1) et avec la méthode de Larribe *et al.*

$\phi = (1, 0)$



Larribe *et al.* avec $f = 2$ Larribe *et al.* avec $f = 3$ Larribe *et al.* avec $f = 4$

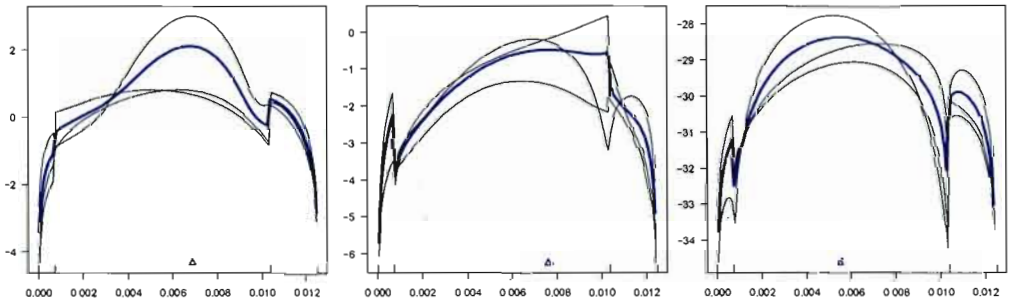


Figure 5.14 Courbes de vraisemblance obtenues pour les données "D" avec la méthode développée en utilisant $\phi = (1, 0)$ et avec la méthode de Larribe *et al.* avec des fenêtres de tailles 2, 3 et 4.

Les graphiques de la figure 5.13 montrent un début de progression vers la convergence, qui n'est pas atteinte. La courbe dont la forme ressemble le plus à celle obtenue avec la méthode de Larribe *et al.* (en simulant 10 000 graphes) est celle de $\phi = (1, 1)$. Dans ce cas, l'estimé du maximum de vraisemblance est aussi près de celui de la méthode de Larribe *et al.* : 0,01774190 cM par rapport à 0,0220297 cM, respectivement.

5.4.7 Résultats obtenus avec les données "D"

Nous n'avons pu obtenir que le graphique de $\phi = (1, 0)$ (voir figure 5.14) pour les données "D" qui comportent dix séquences. Cela est dû aux temps de calcul trop élevés et au fait qu'en vertu du test des trois gamètes, nous devons permettre au moins une recombinaison dans le premier intervalle. Pour obtenir le graphique suivant, soit celui pour $\phi = (1, 1)$, cela aurait nécessité environ 50 jours de calculs, selon un modèle exponentiel de prévision du temps de calcul. Il est raisonnable de penser que le nombre de recombinaisons permises est trop faible pour $\phi = (1, 0)$. Pour les données "A" avec $\phi = (1, 0)$, nous étions loin de la convergence et puisque les données "D" contiennent plus de séquences (dix par rapport à deux), nous nous attendons à ce que la convergence nécessite des valeurs de ϕ encore plus élevées. Les résultats obtenus avec les données "D" exposent les limites de notre méthode en soulignant à quel point elle est exigeante au niveau du temps de calcul.

Toutefois, les données "D" sont intéressantes parce qu'elles comportent quatre marqueurs par séquence. Ainsi, pour faire les calculs, nous avons considéré trois fenêtres de deux marqueurs. Nous pouvons d'ailleurs observer sur le graphique de $\phi = (1, 0)$ que la courbe de vraisemblance a une allure différente dans chacun des intervalles (chacun des intervalles correspond à une des trois fenêtres). À la figure 5.14, il est également intéressant d'observer les différences dans les graphiques de la méthode de Larribe *et al.* (en simulant 10 000 graphes) lorsque nous faisons varier la taille des fenêtres.

CONCLUSION

La recherche présentée dans ce mémoire avait pour but d'explorer une nouvelle méthode de cartographie génétique fine permettant d'estimer la position d'une mutation causant une maladie génétique simple. Tout comme la méthode de Larribe *et al.* du chapitre IV, il s'agit d'une approche qui utilise la méthode du maximum de vraisemblance et qui fait appel aux équations de récurrence de Griffiths-Tavaré ainsi qu'au processus de coalescence avec recombinaison et mutations. L'idée nouvelle de notre méthode est d'obtenir une estimation quasi exacte de la courbe de vraisemblance sans avoir recours aux simulations. Pour faire les calculs de vraisemblance, nous avons donc considéré toutes les histoires qui relient l'échantillon initial de séquences d'ADN à l'ancêtre commun. Cependant, nous avons dû faire des compromis pour que le temps de calcul demeure raisonnable. Nous avons limité le nombre de recombinaisons permises dans les histoires et nous avons séparé nos données en plusieurs parties appelées fenêtres. Nous avons par la suite regroupé les vraisemblances marginales obtenues pour chacune des fenêtres à l'aide du principe de vraisemblance composite.

La nécessité d'explorer une nouvelle approche était justifiée par certaines difficultés de la méthode de Larribe *et al.* En effet, cette méthode ne fournit pas toujours une estimation valable de la position de la mutation, nécessite des temps de calculs élevés et produit des courbes de vraisemblance qui sont différentes d'une simulation à l'autre. En évitant d'avoir recours aux simulations, nous étions au moins certains d'éviter ce dernier problème.

Nous avons obtenu des résultats intéressants avec notre méthode. Premièrement, les graphiques de la figure 5.10 obtenus avec les données "A" illustrent bien la convergence vers laquelle tendent les courbes de vraisemblance au fur et à mesure que nous augmentons

le nombre maximal de recombinaisons permises. C'est un résultat qui était important à illustrer à l'aide d'un exemple : il faut absolument qu'au-delà d'un certain nombre de recombinaisons, la méthode produise toujours les mêmes courbes de vraisemblance. Dans le cas contraire, nous ne pourrions jamais obtenir les bons résultats en des temps de calculs finis ! Ensuite, la forme de la courbe de vraisemblance obtenue avec la méthode de Larribe *et al.* pour les données "B" et "C" est similaire à notre "meilleure" courbe. De plus, dans les deux cas, les estimations du maximum de vraisemblance sont du même ordre de grandeur. En raison des résultats intéressants que nous avons pu obtenir, nous pensons que notre méthode vaut la peine d'être explorée davantage.

Malheureusement, notre méthode comporte encore un problème de taille : celui des temps de calculs élevés. Nous avons rapidement constaté que la croissance du temps de calcul se fait de façon exponentielle à mesure que nous augmentons le nombre maximal de recombinaisons permises dans les histoires. Les résultats que nous avons obtenus montrent clairement que, même si nous considérons des ensembles de données avec peu de marqueurs et peu de séquences, nous ne pouvons pas permettre autant de recombinaisons que nous le souhaiterions. Cela implique que nous n'avons pas pu effectuer suffisamment de calculs pour obtenir la "bonne" courbe de vraisemblance (ou du moins pour s'assurer que nous avons obtenu la "bonne" courbe).

Afin d'améliorer notre méthode, il est donc nécessaire de réduire les temps de calculs. À cette fin, nous avons des pistes de solution à proposer. Premièrement, le programme informatique C++ permettant d'appliquer notre méthode pourrait être modifié de façon de façon à réduire les temps de calculs. Afin de construire toutes les histoires possibles, nous avons utilisé un algorithme récursif. Malheureusement, la récursion est très coûteuse en temps de calcul. Il serait possible, quoique cela nécessiterait beaucoup de travail, de réécrire l'algorithme qui permet de construire les histoires sans utiliser la récursion. Ensuite, toujours dans le but d'améliorer le programme informatique, il serait intéressant

d'utiliser des tables de calcul. Nous pourrions calculer les vraisemblances pour plusieurs petits échantillons de séquences et sauvegarder les résultats dans ces tables. Par la suite, le programme pourrait les utiliser afin d'éviter d'avoir à refaire les mêmes calculs. Dans un autre ordre d'idées, il serait aussi intéressant de changer la concession qui consiste à limiter le nombre de recombinaisons permises dans les histoires pour une autre concession qui nous permettrait de mieux limiter les temps de calculs. Dans cette optique, nous avons essayé de limiter la longueur des histoires, mais sans obtenir de résultats concluants. Il serait néanmoins possible de faire plusieurs autres tests, surtout si l'algorithme permettant de construire toutes les histoires possibles n'est plus récursif. À titre de suggestion, nous proposons de construire seulement les histoires minimales (comme dans Song et Hein, 2005), c'est-à-dire celles qui comportent exactement le nombre minimal d'évènements de recombinaison.

APPENDICE A

PROGRAMME INFORMATIQUE

Je tiens à remercier M. Fabrice Larribe pour m'avoir permis d'utiliser certaines parties du code d'un programme qu'il a écrit en 2004.

Déclarations

```
1  // Librairies
2  #include <iostream>
3  #include <iterator>
4  #include <deque>
5  #include <math.h>
6  #include <ctime>
7  #include <algorithm>
8  #include <fstream>
9  #include <cstring>
10 #include <string>
11 #include <cstdio>
12 #include <cmath>
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <time.h>
16
17 using namespace std;
18
19 // Types
20 typedef deque<int>    VecI;
21 typedef deque<double> VecD;
22 typedef deque<VecI>   VVecI;
23 typedef deque<VecD>   VVecD;
```

Fonctions

```
1  // ------------------------------------------------------------
2  // Fonction qui calcule une puissance d'un entier
```

```
3  // ------------------------------------------------------
4  int puissance(int base, int exposant)
5  {
6          int puissance = 1;
7          for (int i=0 ; i!=exposant ; ++i)
8          {
9                  puissance = puissance * base;
10         }
11         return(puissance);
12 }
13
14
15 //
       ------------------------------------------------------------------------------------------

16 // Fonction qui calcule la somme des composantes d'un vecteur d'entiers
17 //
       ------------------------------------------------------------------------------------------

18 double sommeVecI(VecI vecteur)
19 {
20         double sommeVecI = 0;
21         for(int i=0; i!= vecteur.size(); ++i)
22         {
23                 sommeVecI = sommeVecI + vecteur[i];
24         }
25         return(sommeVecI);
26 }
27
28
29 //
       ------------------------------------------------------------------------------------------

30 // Fonction qui calcule la somme des composantes d'un vecteur de doubles
31 //
       ------------------------------------------------------------------------------------------

32 double sommeVecD(VecD vecteur)
33 {
34         double sommeVecD = 0;
35         for(int i=0; i!= vecteur.size(); ++i)
36         {
37                 sommeVecD = sommeVecD + vecteur[i];
38         }
39         return(sommeVecD);
40 }
41
42
43 // ------------------------------------
44 // Fonction qui écrit un vecteur
45 // ------------------------------------
46 template <class T>
47 void printV(T VecTmp, char * name)
```

```
48  {
49      typename T::iterator j;
50      cout << name;
51      cout << " [";
52          for(j=VecTmp.begin(); j!=VecTmp.end(); ++j)
53          {
54                  cout << " " << *j;
55      }
56      cout << " ] (" << VecTmp.size() << ")\n";
57  }
58
59
60  // ------------------------------------
61  // Fonction qui écrit une matrice
62  // ------------------------------------
63  template <class T>
64  void printVV(deque<T> GVecTmp, char * name)
65  {
66          cout << "\"" << name << "\" (Size " << GVecTmp.size() << ")\n";
67      typename deque<T>::iterator it;
68      typename T::iterator jt;
69      unsigned int index = 0;
70          for(it=GVecTmp.begin(); it!=GVecTmp.end(); ++it)
71          {
72                  cout << "   " << index << ": ";
73          index++;
74                  for(jt=(*it).begin(); jt!=(*it).end(); ++jt)
75                  {
76              cout << *jt << " ";
77                  }
78          cout << "\n";
79      }
80  }
81
82
83  // ------------------------------------------------
84  // Fonction qui calcule la variable "alpha"
85  // ------------------------------------------------
86  double calculAlpha(VVecI seq, VecI mult)
87  {
88          double alpha;
89          int nbMarqAnc = 0;
90          int nbMarqAncSeq;
91          for (int i=0 ; i!=seq.size() ; ++i)
92          {
93                  nbMarqAncSeq = 0;
94                  for (int j=0 ; j!=seq[0].size() ; ++j)
95                  {
96                          if (seq[i][j] == 0 || seq[i][j] == 1 )
                                nbMarqAncSeq++;
97                  }
98                  nbMarqAncSeq = nbMarqAncSeq * mult[i];
99                  nbMarqAnc = nbMarqAnc + nbMarqAncSeq;
```

```
100         }
101         alpha = nbMarqAnc / (sommeVecI(mult) * seq[0].size());
102         return(alpha);
103 }
104
105
106 // ------------------------------------------------
107 // Fonction qui calcule la variable "beta"
108 // ------------------------------------------------
109 VecD calculBeta(VVecI seq, VecI mult, VVecD distI)
110 {
111         VecD beta;
112         VecD distAnc;
113         int debut;
114         double temp;
115         int iterateur;
116         double r = sommeVecD(distI[0]);
117         for (int i=0 ; i!=distI.size() ; ++i)
118         {
119                 distAnc.push_back(0);
120         }
121         for (int i=0 ; i!=seq.size() ; ++i)
122         {
123                 iterateur = 0;
124                 debut = -1;
125                 while(debut == -1 && iterateur != seq[0].size())
126                 {
127                         if (seq[i][iterateur] != 9) debut = iterateur;
128                         iterateur++;
129                 }
130                 for (int k=0 ; k!=distI.size() ; ++k)
131                 {
132                         temp = 0;
133                         for (int j=debut; j!=seq[0].size()-1; ++j)
134                         {
135                                 temp = temp+distI[k][j];
136                                 if (seq[i][j+1] == 0 || seq[i][j+1] ==
                                        1)
137                                 {
138                                         distAnc[k] = distAnc[k]+mult[i]*
                                            temp;
139                                         temp = 0;
140                                 }
141                         }
142                 }
143         }
144         for (int i=0 ; i!=distI.size() ; ++i)
145         {
146                 beta.push_back(distAnc[i] / (sommeVecI(mult) * r));
147         }
148         return(beta);
149 }
150
```

```
151
152 // ------------------------------------------------------------------
153 // Fonction qui convertit un string en un tableau de char
154 // ------------------------------------------------------------------
155 char * convertSC(string mot)
156 {
157     char * tmp = new char[mot.length()+0];
158     mot.copy(tmp, mot.length());
159     tmp[mot.length()+0] = 0;
160     return(tmp);
161 }
162
163
164 // ---------------------------------------
165 // Fonction qui lit les paramètres
166 // ---------------------------------------
167 void readParm(string &parFile, int &ne, double &mu, int &nbCan, VecD &
        distI, string &dataFile, int &f, VecI &recMax, int &debug, const
        char* resultats, int &arret)
168 {
169     string buf;
170     ifstream in(parFile.c_str());
171     if (!in )
172         {
173                 cerr << "ERROR: Parameter file " << parFile << " could
                        not be opened";
174         exit(1);
175     }
176     while(getline(in,buf))
177         {
178         string::const_iterator cp = buf.begin();
179         int cnt = 0;
180         int cas = 0;
181         while(cp != buf.end())
182                 {
183                         while(*cp == ' ') cp ++;
184             if(*cp )
185                         {
186                                 cnt++;
187                 string mot;
188                 while(*cp != ' ' && cp != buf.end())
189                                 {
190                                         mot += *cp;
191                     cp++;
192                 }
193                 if(cnt == 1){
194                                         if(mot == "ne")
                                            cas = 1;
195                     if(mot == "mu")             cas = 2;
196                     if(mot == "nbCan")          cas = 3;
197                     if(mot == "distI")          cas = 4;
198                     if(mot == "dataFile")       cas = 5;
199                                         if(mot == "f")
```

```
                                        cas = 6;
200                                     if(mot == "recMax")     cas = 7;
201                                     if(mot == "arret")
                                            cas = 8;
202                     if(cas == 0)
203                                     {
204                                             cerr << "Wrong parameter
                                                : " << mot << "." <<
                                                 endl;
205                     }
206                 }
207                 if(cnt > 1)
208                                 {
209                     if(cas == 1) ne = atoi(mot.c_str());
210                     if(cas == 2) mu = (double) atof(mot.c_str());
211                     if(cas == 3) nbCan = atoi(mot.c_str());
212                     if(cas == 4) distI.push_back((double) atof(mot.c_str
                            ()));
213                     if(cas == 5) dataFile = convertSC(mot);
214                                     if(cas == 6) f = atoi(mot.c_str
                                        ());
215                                     if(cas == 7) recMax.push_back(
                                        atoi(mot.c_str()));
216                                     if(cas == 8) arret = atoi(mot.
                                        c_str());
217                 }
218             }
219         }
220     }
221
222     if (debug > 0)
223         {
224                 cout << "LECTURE DU FICHIER DE PARAMETRES \"" << parFile
                         << "\"" << endl;
225                 cout << "[ ne = " << ne << " ] [ mu = " << mu << " ] [
                        nbCan = " << nbCan
226                              << " ] [ dataFile = "          << dataFile << "
                                 ] [ f = " << f << " ] [ arret = "
227              << arret << " ] ";
228                 printV(distI,"distI"); printV(recMax,"recMax");
229         }
230
231 }
232
233
234 // ---------------------------------
235 // Fonction qui lit les données
236 // ---------------------------------
237 VVecI readData(int debug, string dataFile)
238 {
239     // Le fichier de données doit avoir n_i dans la première colonne, le
            cas de contrôle (0) ou de status (1) dans la deuxième colonne
          et finalement les haplotypes.
```

```
240     VVecI yRead;
241     string buf;
242     if (debug > 1) cout << "\nLECTURE DU FICHIER DE DONNEES \n";
243     ifstream in(dataFile.c_str());
244     if (!in )
245         {
246         cerr << "ERROR: Data file " << dataFile << " could not be opened
            ";
247         exit(1);
248     }
249     while(getline(in,buf))
250         {
251         string::const_iterator cp = buf.begin();
252         int cnt = 0;
253         VecI VHaplo;
254         while(cp != buf.end())
255                 {
256             while(*cp == ' ') cp ++;
257             if(*cp )
258                         {
259                 cnt++;
260                 string mot;
261                         int number;
262                 while(*cp != ' ' && cp != buf.end())
263                                 {
264                     mot += *cp;
265                     cp++;
266                 }
267                 if(mot == "-") mot = "-1";
268                 number = atoi(mot.c_str());
269                 VHaplo.push_back(atoi( mot.c_str() ));
270             }
271         }
272         yRead.push_back(VHaplo);
273     }
274     if (debug > 1) printVV(yRead,"yRead");
275     return(yRead);
276 }
277
278
279 // ----------------------------------
280 // Fonction qui construit tHap
281 // ----------------------------------
282 VVecI tHapFct;
283 VVecI tabHap(int l, VecI seq)
284 {
285         VVecI bidon2;
286         VecI seq1;
287         VecI seq2;
288         VecI seq3;
289         seq1.assign(seq.begin(),seq.end());
290         seq2.assign(seq.begin(),seq.end());
291         seq3.assign(seq.begin(),seq.end());
```

```
292         seq1.push_back(0);
293         seq2.push_back(1);
294         seq3.push_back(9);
295         if (l>1) bidon2 = tabHap(l-1,seq1);
296         if (l>1) bidon2 = tabHap(l-1,seq2);
297         if (l>1) bidon2 = tabHap(l-1,seq3);
298         if (l == 1) tHapFct.push_back(seq1);
299         if (l == 1) tHapFct.push_back(seq2);
300         if (l == 1) tHapFct.push_back(seq3);
301         return(tHapFct);
302 }
303
304
305 //
    ------------------------------------------------------------------------------------

306 // Fonction qui retourne le numéro de la séquence engendrée par une
    coalescence distincte
307 //
    ------------------------------------------------------------------------------------

308 int coaDi(int i, int j, VVecI tHap)
309 {
310         int coa = -10;
311         VecI nouvSeq;
312         bool possible = true;
313         for (int im=0 ; im!=tHap[0].size() ; ++im)
314         {
315                 if ( (tHap[i][im] != tHap[j][im]) && (tHap[i][im] != 9
                        && tHap[j][im] != 9) ) possible = false;
316         }
317         if(possible == false) coa = -1;
318         if(possible == true)
319         {
320                 // Création de la nouvelle séquence
321                 for (int im=0 ; im!=tHap[0].size() ; ++im)
322                 {
323                         if(tHap[i][im] == 0 || tHap[j][im] == 0)
324                         nouvSeq.push_back(0);
325                         if(tHap[i][im] == 1 || tHap[j][im] == 1)
326                         nouvSeq.push_back(1);
327                         if(tHap[i][im] == 9 && tHap[j][im] == 9)
328                         nouvSeq.push_back(9);
329                 }
330                 // Identification de la nouvelle séquence
331                 for (int im=0 ; im!=tHap.size(); ++im)
332                 {
333                         int compVrai = 0;
334                         for (int c=0 ; c!=tHap[0].size() ; ++c)
335                         {
336                                 if (nouvSeq[c] == tHap[im][c]) compVrai
                                    = compVrai + 1;
337                         }
```

```
338                             if (compVrai == tHap[0].size()) coa = im;
339                     }
340             }
341             return(coa);
342     }
343
344
345     //
            ----------------------------------------------------------------------------------------

346     // Fonction qui construit la matrice triangulaire des coalescences
            distinctes
347     //
            ----------------------------------------------------------------------------------------

348     VVecI matCoaDi(VVecI tHap)
349     {
350             VVecI tCoaFct;
351             VecI tCoaFctInt;
352             int coa;
353             int fin;
354             fin = puissance(3,tHap[0].size())-1;
355             for (int i=0; i!=fin; ++i)
356             {
357                     for (int j=0; j!=i; ++j)
358                     {
359                             coa = coaDi(i,j,tHap);
360                             tCoaFctInt.push_back(coa);
361                     }
362                     tCoaFct.push_back(tCoaFctInt);
363                     tCoaFctInt.clear();
364             }
365             return(tCoaFct);
366     }
367
368
369     //
            ----------------------------------------------------------------------------------------

370     // Fonction qui retourne le numéro de la séquence engendrée par une
            mutation
371     //
            ----------------------------------------------------------------------------------------

372     int mutDi(int i, int j, VVecI tHap)
373     {
374             int mut = -10;
375             VecI nouvSeqMut;
376             bool possibleMut = true;
377             if ( (tHap[i][j] == 9) || (tHap[i][j] == 0) ) possibleMut =
                    false;
378             if(possibleMut == false) mut = -1;
379             if(possibleMut == true)
```

```
380         {
381                 // Création de la nouvelle séquence
382                 nouvSeqMut.assign(tHap[i].begin(),tHap[i].end());
383                 nouvSeqMut[j] = 0;
384                 // Identification de la nouvelle séquence
385                 for (int im=0 ; im!=tHap.size(); ++im)
386                 {
387                 int compVrai = 0;
388                 for (int c=0 ; c!=tHap[0].size() ; ++c)
389                 {
390                         if (nouvSeqMut[c] == tHap[im][c]) compVrai =
                            compVrai + 1;
391                 }
392                 if (compVrai == tHap[0].size()) mut = im;
393                 }
394         }
395         return(mut);
396 }
397
398
399 // ---------------------------------------------------------
400 // Fonction qui construit la matrice des mutations
401 // ---------------------------------------------------------
402 VVecI matMutDi(VVecI tHap)
403 {
404         VVecI tMutFct;
405         VecI tMutFctInt;
406         int mut;
407         int finMut;
408         finMut = puissance(3,tHap[0].size())-1;
409         for (int i=0; i!=finMut; ++i)
410         {
411                 for (int j=0; j!=tHap[0].size(); ++j)
412                 {
413                         mut = mutDi(i,j,tHap);
414                         tMutFctInt.push_back(mut);
415                 }
416                 tMutFct.push_back(tMutFctInt);
417                 tMutFctInt.clear();
418         }
419         return(tMutFct);
420 }
421
422
423 //
      ---------------------------------------------------------------------------------------

424 // Fonction qui retourne le numéro de la séquence engendrée par une
      recombinaison
425 //
      ---------------------------------------------------------------------------------------

426 int recDi(int i, int j, char * bord, VVecI tHap)
```

```
427 {
428         int rec = -10;
429         VecI nouvSeqRec;
430         bool possibleRec = true;
431         // Première vérification pour voir si une recombinaison est
                possible
432         int compteur1 = 0;
433         for (int im=0 ; im!=j ; ++im)
434         {
435                 if (tHap[i][im] == 9)
436                 compteur1 = compteur1 + 1;
437         }
438         if (compteur1 == j) possibleRec = false;
439         // Seconde vérification pour voir si une recombinaison est
                possible
440         int compteur2 = 0;
441         for (int im=j ; im!=tHap[0].size() ; ++im)
442         {
443                 if (tHap[i][im] == 9)
444                 compteur2 = compteur2 + 1;
445         }
446         if (compteur2 == (tHap[0].size()-j)) possibleRec = false;
447         if(possibleRec == false) rec = -1;
448         if(possibleRec == true)
449         {
450                 if (bord == "g")
451                 {
452                         // Création de la nouvelle séquence gauche
453                         nouvSeqRec.assign(tHap[i].begin(),tHap[i].end())
                                ;
454                         for (int im=j ; im!=tHap[0].size() ; ++im)
455                         {
456                                 nouvSeqRec[im] = 9;
457                         }
458                 }
459                 if (bord == "d")
460                 {
461                         // Création de la nouvelle séquence droite
462                         nouvSeqRec.assign(tHap[i].begin(),tHap[i].end())
                                ;
463                         for (int im=0 ; im!=j ; ++im)
464                         {
465                                 nouvSeqRec[im] = 9;
466                         }
467                 }
468                 // Identification de la nouvelle séquence
469                 for (int im=0 ; im!=tHap.size(); ++im)
470                 {
471                         int compVrai = 0;
472                         for (int c=0 ; c!=tHap[0].size() ; ++c)
473                         {
474                                 if (nouvSeqRec[c] == tHap[im][c])
                                        compVrai = compVrai + 1;
```

```
475                         }
476                         if (compVrai == tHap[0].size()) rec = im;
477                 }
478         }
479         return(rec);
480 }
481
482
483 // ------------------------------------------------------------------
484 // Fonction qui construit la matrice des recombinaisons
485 // ------------------------------------------------------------------
486 VVecI matRecDi(char * bord, VVecI tHap)
487 {
488         VVecI tRecFct;
489         VecI tRecFctInt;
490         int rec;
491         int finRec = puissance(3,tHap[0].size())-1;
492         for (int i=0; i!=finRec; ++i)
493         {
494                 for (int j=1; j!=tHap[0].size(); ++j)
495                 {
496                         rec = recDi(i,j,bord,tHap);
497                         tRecFctInt.push_back(rec);
498                 }
499                 tRecFct.push_back(tRecFctInt);
500                 tRecFctInt.clear();
501         }
502         return(tRecFct);
503 }
504
505
506 // ------------------------------------------------------------------
507 // Fonction qui identifie les numéros des séquences
508 // ------------------------------------------------------------------
509 VecI idSeq(VVecI seq, VVecI tHap)
510 {
511         VecI seqNum;
512         int numSeq;
513         for (int i=0 ; i!=seq.size() ; ++i)
514         {
515                 for (int im=0 ; im!=tHap.size(); ++im)
516                 {
517                         int compVrai = 0;
518                         for (int c=0 ; c!=tHap[0].size() ; ++c)
519                         {
520                                 if (seq[i][c] == tHap[im][c]) compVrai =
                                        compVrai + 1;
521                         }
522                         if (compVrai == tHap[0].size())
523                         {
524                                 numSeq = im;
525                                 seqNum.push_back(numSeq);
526                         }
```

```
527                 }
528         }
529         return(seqNum);
530 }
531
532
533 // ------------------------------------------------------
534 // Fonction qui construit la liste des évènements
535 // ------------------------------------------------------
536 VVecD listeEv(VVecI seq, VecI seqNum, VecI recMax, VecI mult, VVecD
        distI, double alpha, VecD beta, double r, double theta, double rho,
        VVecI tCoa, VVecI tMut, VVecI tRecG, VVecI tRecD, int f)
537 {
538         VecD listeFeed;
539         VVecD liste;
540         VecD prob;
541
542         // Coalescences identiques
543         for (int i=0; i!=mult.size(); ++i)
544         {
545                 if (mult[i]>1)
546                 {
547                         listeFeed.push_back(1);
548                         listeFeed.push_back(i);
549                         for (int k=0 ; k!=distI.size() ; ++k)
550                         {
551                                 prob.push_back((mult[i]-1)/(sommeVecI(
                                        mult)-1+alpha*theta+beta[k]*rho));
552                         }
553                         for (int k=0 ; k!=distI.size() ; ++k)
554                         {
555                                 listeFeed.push_back(prob[k]);
556                         }
557                         prob.clear();
558                         liste.push_back(listeFeed);;
559                         listeFeed.clear();
560                 }
561         }
562
563         // Coalescences distinctes
564         for (int i=0; i!=seqNum.size(); ++i)
565         {
566                 for (int j=i+1; j!=seqNum.size(); ++j)
567                 {
568                         if(tCoa[max(seqNum[i],seqNum[j])][min(seqNum[i],
                                seqNum[j])]>=0)
569                         {
570                                 listeFeed.push_back(2);
571                                 listeFeed.push_back(i);
572                                 listeFeed.push_back(j);
573                                 listeFeed.push_back(tCoa[max(seqNum[i],
                                        seqNum[j])][min(seqNum[i],seqNum[j])
                                        ]);
```

```
574                                     // n_k
575                                     int nkC = 0;
576                                     for (int k=0; k!=seqNum.size(); ++k)
577                                     {
578                                             if (seqNum[k] == tCoa[max(seqNum
                                                    [i],seqNum[j])][min(seqNum[i
                                                    ],seqNum[j])])
579                                             nkC = mult[k];
580                                     }
581                                     // delta
582                                     int delta = 0;
583                                     if (seqNum[i] == tCoa[max(seqNum[i],
                                            seqNum[j])][min(seqNum[i],seqNum[j])
                                            ])
584                                     delta = delta+1;
585                                     if (seqNum[j] == tCoa[max(seqNum[i],
                                            seqNum[j])][min(seqNum[i],seqNum[j])
                                            ])
586                                     delta = delta+1;
587                                     for (int k=0 ; k!=distI.size() ; ++k)
588                                     {
589                                             prob.push_back((2*(nkC+1-delta))
                                                    /(sommeVecI(mult)-1+alpha*
                                                    theta+beta[k]*rho));
590                                     }
591                                     for (int k=0 ; k!=distI.size() ; ++k)
592                                     {
593                                             listeFeed.push_back(prob[k]);
594                                     }
595                                     prob.clear();
596                                     liste.push_back(listeFeed);
597                                     listeFeed.clear();
598                             }
599                     }
600             }
601
602             // Mutations
603             for (int j=0 ; j!=f; ++j)
604             {
605                     int mutCompt = 0;
606                     int position_i = 0;
607                     int position_j = 0;
608                     for (int i=0; i!= seq.size(); ++i)
609                     {
610                             if (seq[i][j] == 1)
611                             {
612                                     position_i = i;
613                                     position_j = j;
614                                     mutCompt = mutCompt + mult[i];
615                             }
616                     }
617                     if (mutCompt == 1)
618                     {
```

```
619                         listeFeed.push_back(3);
620                         listeFeed.push_back(position_i);
621                         listeFeed.push_back(tMut[seqNum[position_i]][
                                position_j]);
622                         // n_j
623                         int njM = 0;
624                         for (int k=0; k!=seqNum.size(); ++k)
625                         {
626                                 if (seqNum[k] == tMut[seqNum[position_i
                                        ]][position_j])
627                                 njM = mult[k];
628                         }
629                         for (int k=0 ; k!=distI.size() ; ++k)
630                         {
631                                 prob.push_back((theta*(njM+1))/((
                                        sommeVecI(mult)*f)*(sommeVecI(mult)
                                        -1+alpha*theta+beta[k]*rho)));
632                         }
633                         for (int k=0 ; k!=distI.size() ; ++k)
634                         {
635                                 listeFeed.push_back(prob[k]);
636                         }
637                         prob.clear();
638                         liste.push_back(listeFeed);
639                         listeFeed.clear();
640                 }
641         }
642
643         // Recombinaisons
644         for (int i=0; i!=seqNum.size(); ++i)
645         {
646                 for(int j=0; j!=(f-1); ++j)
647                 {
648                         if (tRecD[seqNum[i]][j]!=-1 && recMax[j]>=1)
649                         {
650                                 listeFeed.push_back(4);
651                                 listeFeed.push_back(i);
652                                 listeFeed.push_back(tRecD[seqNum[i]][j])
                                        ;
653                                 listeFeed.push_back(tRecG[seqNum[i]][j])
                                        ;
654                                 listeFeed.push_back(j);
655                                 // n_j
656                                 int njR = 0;
657                                 for (int k=0; k!=seqNum.size(); ++k)
658                                 {
659                                         if (seqNum[k] == tRecG[i][j])
660                                         njR = mult[k];
661                                 }
662                                 // n_k
663                                 int nkR = 0;
664                                 for (int k=0; k!=seqNum.size(); ++k)
665                                 {
```

```
666                                     if (seqNum[k] == tRecD[i][j])
667                                     nkR = mult[k];
668                             }
669                             for (int k=0 ; k!=distI.size() ; ++k)
670                             {
671                                     prob.push_back((rho*(njR+1)*(nkR
                                            +1)*distI[k][j])/((sommeVecI
                                            (mult)*(sommeVecI(mult)+1))
                                            *(sommeVecI(mult)-1+alpha*
                                            theta+beta[k]*rho)*r));
672                             }
673                             for (int k=0 ; k!=distI.size() ; ++k)
674                             {
675                                     listeFeed.push_back(prob[k]);
676                             }
677                             prob.clear();
678                             liste.push_back(listeFeed);
679                             listeFeed.clear();
680                     }
681             }
682     }
683     return(liste);
684 }
685
686
687 //
    ------------------------------------------------------------------------

688 // Fonction récursive qui parcourt tous les chemins possibles
689 //
    ------------------------------------------------------------------------

690 int compteur1 = 0;
691 int compteur2 = 0;
692 int saut = 1000000;
693 VecD recurs(VVecI seq, VecI seqNum, VecI recMax, VecI mult, VVecD distI,
        double alpha, VecD beta, double r, double theta, double rho, VVecI
    tHap, VVecI tCoa, VVecI tMut, VVecI tRecG, VVecI tRecD, int f, int
    debug, const char* resultats, int step, int arret)
694 {
695     bool nouvSeq1 = true;
696     bool nouvSeq2 = true;
697     bool nouvSeq3 = true;
698     bool nouvSeq4 = true;
699     VecI seqInt2;
700     int efface = 0;
701
702     VVecD liste;
703     VecD prob;
704     VecD probTotale_i;
705     VecD probTotale;
706     VecD uneLigne;
707
```

```
708         VVecI seq_p;
709         VecI seqNum_p;
710         VecI recMax_p;
711         VecI mult_p;
712         double alpha_p;
713         VecD beta_p;
714         VecD contRecurs;
715
716         for (int i=0 ; i!=distI.size() ; ++i)
717         {
718                 probTotale.push_back(0);
719         }
720
721         liste = listeEv(seq,seqNum,recMax,mult,distI,alpha,beta,r,theta,
                rho,tCoa,tMut,tRecG,tRecD,f);
722
723         if (sommeVecI(mult) == 1 && seqNum[0] == 0)
724         {
725                 cout << "\nLes données correspondent à l'ancêtre commun
                        .\n\n";
726         }
727
728         else if (liste.size() == 0)
729         {
730                 if (debug>1)
731                 {
732                         cout << "\nAucun evenement possible\n\n";
733                         printV(probTotale,"probTotale");
734                 }
735         }
736
737         else
738         {
739                 for (int i=0 ; i!=liste.size() ; ++i)
740                 {
741                         seq_p.assign(seq.begin(),seq.end());
742                         seqNum_p.assign(seqNum.begin(),seqNum.end());
743                         recMax_p.assign(recMax.begin(),recMax.end());
744                         mult_p.assign(mult.begin(),mult.end());
745                         alpha_p = alpha;
746                         beta_p.assign(beta.begin(),beta.end());
747
748                         uneLigne = liste[i];
749
750                         compteur1++;
751                         if (debug>0 && compteur1 == saut)
752                         {
753                                 compteur1 = 0;
754                                 compteur2++;
755                                 cout << compteur2 << " ";
756                         }
757
758                         if (debug>1)
```

```
759                     {
760                             cout << "\nTRACES DU PROGRAMME RECURSIF
                                    (étape: " << step << ")\n";
761                             printV(seqNum_p,"seqNum_p");
762                             printV(mult_p,"mult_p");
763                             printVV(seq_p,"seq_p");
764                             cout << "alpha_p: " << alpha_p << "\n";
765                             printV(beta_p,"beta_p");
766                             printV(recMax,"recMax");
767                             printVV(liste,"liste");
768                     }
769
770                     // Changer les paramètres variables et trouver "
                            prob"
771
772                     // Coalescence identique
773                     if(uneLigne[0] == 1)
774                     {
775                             mult_p[(int) uneLigne[1]] = mult_p[(int)
                                    uneLigne[1]] - 1;
776                             for (int j=0 ; j!=distI.size() ; ++j)
777                             {
778                                     prob[j] = uneLigne[j+2];
779                             }
780                     }
781
782                     // Coalescence distincte
783                     if(uneLigne[0] == 2)
784                     {
785                             // Rajouter la séquence produite
786                             for (int i=0 ; i!=seqNum_p.size() ; ++i)
787                             {
788                                     if (seqNum_p[i] == uneLigne[3])
789                                     {
790                                             mult_p[i] = mult_p[i] +
                                                    1;
791                                             nouvSeq1 = false;
792                                     }
793                             }
794
795                             if (nouvSeq1)
796                             {
797                                     seqNum_p.push_back((int)
                                            uneLigne[3]);
798                                     mult_p.push_back(1);
799                                     for (int ii=0; ii!=f; ii++)
800                                     {
801                                             seqInt2.push_back(tHap[(
                                                    int) uneLigne[3]][ii
                                                    ]);
802                                     }
803                                     seq_p.push_back(seqInt2);
804                                     seqInt2.clear();
```

```
805                         }
806                         nouvSeq1 = true;
807
808                         // Diminuer la multiplicité de un pour
                                la première séquence utilisée
809                         mult_p[(int) uneLigne[1]] = mult_p[(int)
                                 uneLigne[1]] - 1;
810
811                         // Enlever la séquence s'il n'y en a
                                plus
812                         if (mult_p[(int) uneLigne[1]] == 0)
813                         {
814                                 seqNum_p.erase(seqNum_p.begin()
                                        + (int) uneLigne[1]);
815                                 mult_p.erase(mult_p.begin() + (
                                        int) uneLigne[1]);
816                                 seq_p.erase(seq_p.begin() + (int
                                        ) uneLigne[1]);
817                                 efface = efface + 1;
818                         }
819
820                         // Diminuer la multiplicité de un pour
                                la seconde séquence utilisée
821                         mult_p[(int) uneLigne[2] - efface] =
                                mult_p[(int) uneLigne[2] - efface] -
                                 1;
822
823
824                         // Enlever la séquence s'il n'y en a
                                plus
825                         if (mult_p[(int) uneLigne[2] - efface]
                                == 0)
826                         {
827                                 seqNum_p.erase(seqNum_p.begin()
                                        + (int) uneLigne[2] - efface
                                        );
828                                 mult_p.erase(mult_p.begin() + (
                                        int) uneLigne[2] - efface);
829                                 seq_p.erase(seq_p.begin() + (int
                                        ) uneLigne[2] - efface);
830                         }
831
832                         efface = 0;
833
834                         for (int j=0 ; j!=distI.size() ; ++j)
835                         {
836                                 prob[j] = uneLigne[j+4];
837                         }
838                 }
839
840                 // Mutation
841                 if(uneLigne[0] == 3)
842                 {
```

```
843                     // Rajouter la séquence produite
844                     for (int i=0 ; i!=seqNum_p.size() ; ++i)
845                     {
846                             if (seqNum_p[i] == uneLigne[2])
                                    // si la séquence produite
                                    existe déjà
847                             {
848                                     mult_p[i] = mult_p[i] +
                                            1;
849                                     nouvSeq2 = false;
850                             }
851                     }
852
853                     if (nouvSeq2) // si la séquence produite
                             n'existe pas encore
854                     {
855                             seqNum_p.push_back((int)
                                    uneLigne[2]);
856                             mult_p.push_back(1);
857                             for (int ii=0; ii!=f; ii++)
858                             {
859                                     seqInt2.push_back(tHap[(
                                            int) uneLigne[2]][ii
                                            ]);
860                             }
861                             seq_p.push_back(seqInt2);
862                             seqInt2.clear();
863                     }
864                     nouvSeq2 = true;
865
866                     // Enlever la séquence qui mute
867                     seqNum_p.erase(seqNum_p.begin() + (int)
                            uneLigne[1]);
868                     mult_p.erase(mult_p.begin() + (int)
                            uneLigne[1]);
869                     seq_p.erase(seq_p.begin() + (int)
                            uneLigne[1]);
870
871                     for (int j=0 ; j!=distI.size() ; ++j)
872                     {
873                             prob[j] = uneLigne[j+3];
874                     }
875             }
876
877             // Recombinaison
878             if(uneLigne[0] == 4)
879             {
880                     // Rajouter la première séquence
                            produite
881                     for (int i=0 ; i!=seqNum_p.size() ; ++i)
882                     {
883                             if (seqNum_p[i] == uneLigne[2])
884                             {
```

```
885                             mult_p[i] = mult_p[i] +
                                    1;
886                             nouvSeq3 = false;
887                     }
888             }
889
890             if (nouvSeq3)
891             {
892                     seqNum_p.push_back((int)
                            uneLigne[2]);
893                     mult_p.push_back(1);
894                     for (int ii=0; ii!=f; ii++)
895                     {
896                             seqInt2.push_back(tHap[(
                                    int) uneLigne[2]][ii
                                    ]);
897                     }
898                     seq_p.push_back(seqInt2);
899                     seqInt2.clear();
900             }
901             nouvSeq3 = true;
902
903             // Rajouter la deuxième séquence
                    produite
904             for (int i=0 ; i!=seqNum_p.size() ; ++i)
905             {
906                     if (seqNum_p[i] == uneLigne[3])
907                     {
908                             mult_p[i] = mult_p[i] +
                                    1;
909                             nouvSeq4 = false;
910                     }
911             }
912
913             if (nouvSeq4)
914             {
915                     seqNum_p.push_back((int)
                            uneLigne[3]);
916                     mult_p.push_back(1);
917                     for (int ii=0; ii!=f; ii++)
918                     {
919                             seqInt2.push_back(tHap[(
                                    int) uneLigne[3]][ii
                                    ]);
920                     }
921                     seq_p.push_back(seqInt2);
922                     seqInt2.clear();
923             }
924             nouvSeq4 = true;
925
926             // Diminuer la multiplicité de la
                    séquence utilisée de un
927             mult_p[(int) uneLigne[1]] = mult_p[(int)
```

```
                    uneLigne[1]] - 1;
928
929             // Enlever la séquence s'il n'y en a
                    plus
930             if (mult_p[(int) uneLigne[1]] == 0)
931             {
932                     seqNum_p.erase(seqNum_p.begin()
                            + (int) uneLigne[1]);
933                     mult_p.erase(mult_p.begin() + (
                            int) uneLigne[1]);
934                     seq_p.erase(seq_p.begin() + (int
                            ) uneLigne[1]);
935             }
936
937             // Modifier le vecteur recMax_p
938             recMax_p[(int) uneLigne[4]] = recMax_p[(
                    int) uneLigne[4]] - 1;
939
940             for (int j=0 ; j!=distI.size() ; ++j)
941             {
942                     prob[j] = uneLigne[j+5];
943             }
944     }
945
946     // Calcul des paramètres "alpha_p" et "beta_p"
947     alpha_p = calculAlpha(seq_p,mult_p);
948     beta_p = calculBeta(seq_p,mult_p,distI);
949
950     if (step >= arret)
951     {
952             for (int j=0 ; j!=distI.size() ; ++j)
953             {
954                     probTotale_i[j] = 0;
955             }
956     }
957
958     else if (sommeVecI(mult_p) == 1 && seqNum_p[0]
            == 0)
959     {
960             for (int j=0 ; j!=distI.size() ; ++j)
961             {
962                     probTotale_i[j] = prob[j];
963             }
964     }
965
966     else
967     {
968             contRecurs = recurs(seq_p,seqNum_p,
                    recMax_p,mult_p,distI,alpha_p,beta_p
                    ,r,theta,rho,tHap,tCoa,tMut,tRecG,
                    tRecD,f,debug,resultats,step+1,arret
                    );
969             for (int j=0 ; j!=distI.size() ; ++j)
```

```
970                                     {
971                                             probTotale_i[j] = prob[j] *
                                                    contRecurs[j];
972                                     }
973                             }
974
975                             for (int j=0 ; j!=distI.size() ; ++j)
976                             {
977                                     probTotale[j] = probTotale[j] +
                                            probTotale_i[j];
978                             }
979
980                             if (debug>1)
981                             {
982                                     cout << "\n";
983                                     printV(probTotale,"probTotale");
984                             }
985                     }
986             }
987             return(probTotale);
988     }
989
990
991     //
        ------------------------------------------------------------------------

992     // Fonction qui calcule le vecteur de vraisemblance totale
993     //
        ------------------------------------------------------------------------

994     VVecD vraisTot(int ne, int nbCan, VecD distI, int f, VecI recMax,
995     VVecI tHap1, VVecI tHap2, VVecI tCoa1, VVecI tCoa2, VVecI tMut1,
996     VVecI tMut2, VVecI tRecG1, VVecI tRecG2, VVecI tRecD1, VVecI tRecD2,
997     double theta, VecD beta, double alpha,
998     VVecI donnees, VecI mult, VecI malade, VVecI hap,
999     int debug, const char* resultats, int step, int arret)
1000    {
1001            VecD vrSansMutEnt;
1002            VecD vrAvecMutEnt;
1003            VecD vrMutFeed2;
1004            VecD vrMutFeed;
1005            VVecD vrSansMut;
1006            VVecD vrAvecMut;
1007            VVecD vrMut;
1008            VecI seqInt;
1009
1010            VVecI seq;
1011            VecI seqNum;
1012            VecI nbSeqNum;
1013            int valeurSeq;
1014            VecI seqNumMod;
1015            VecI multMod;
1016            VVecI seqMod;
```

```
1017
1018            VVecI seq2;
1019            VecI seqNum2;
1020            VecI nbSeqNum2;
1021            int valeurSeq2;
1022            VecI seqNumMod2;
1023            VecI multMod2;
1024            VVecI seqMod2;
1025
1026            double r;
1027            double rho;
1028            VecD distI2Feed;
1029            VVecD distI2;
1030            VecD distImTemp;
1031            VVecD distIm;
1032            VecD invPoids;
1033            VecI seqModFeed;
1034
1035
1036            // Initialisation des vecteurs de vraisemblance;
1037            for (int i=0 ; i!=hap[0].size()-1 ; ++i)
1038            {
1039                    vrMutFeed2.push_back(1);
1040            }
1041
1042            vrSansMut.push_back(vrMutFeed2);
1043
1044            for (int z=0 ; z!=nbCan ; ++z)
1045            {
1046                    vrAvecMut.push_back(vrMutFeed2);
1047            }
1048
1049            // Définition du vecteur donnant le nombre de fenêtres qui
                    recoupent chacun des (L-2) intervalles
1050            for (int intervalle=0 ; intervalle!=hap[0].size()-1 ; ++
                intervalle)
1051            {
1052                    invPoids.push_back(0);
1053            }
1054
1055            for (int i=0 ; i!=hap[0].size()+1-f; ++i) // Pour chaque fenêtre
                     --- N.B.: L=hap[0].size()+1
1056            {
1057                    for (int l=i ; l!=i+f-1 ; ++l) // Pour les intervalles
                            correspondant à la fenêtre choisie
1058                    {
1059                            invPoids[l] = invPoids[l] + 1;
1060                    }
1061            }
1062
1063            if (debug>1)
1064            {
1065                    cout << "\nPARAMETRES INITIAUX \n";
```

```
1066                printV(distI, "distI");
1067                printVV(vrSansMut,"vrSansMut");
1068                printVV(vrAvecMut,"vrAvecMut");
1069                printV(invPoids,"invPoids");
1070        }
1071
1072        for (int i=0 ; i!=hap[0].size()+1-f; ++i) // Pour chaque fenêtre
                --- N.B.: L=hap[0].size()+1
1073        {
1074                for (int j=0 ; j!=donnees.size() ; ++j) // Pour chaque
                    séquence
1075                {
1076                        for (int l=i ; l!=i+f ; ++l) // Pour les
                            positions correspondant à la fenêtre choisie
1077                        {
1078                                seqInt.push_back(hap[j][l]);
1079                        }
1080
1081                        seq.push_back(seqInt);  // Construction de "seq"
1082                        seqInt.clear();
1083                }
1084
1085                seqNum = idSeq(seq,tHap1);      // Construction de "
                    seqNum"
1086
1087                // Initialisation du vecteur contenant le nombre de
                    numéro de séquences
1088                for (int ii=0; ii!=puissance(f,3)-1; ++ii)
1089                {
1090                        nbSeqNum.push_back(0);
1091                }
1092
1093                // Calcul de la multiplicité de chaque numéro de
                    séquence
1094                for (int ii=0; ii!=seqNum.size(); ++ii)
1095                {
1096                        valeurSeq = seqNum[ii];
1097                        nbSeqNum[valeurSeq] = nbSeqNum[valeurSeq] + mult
                            [ii];
1098                }
1099
1100                for (int ii=0; ii!=puissance(f,3)-1; ++ii)
1101                {
1102                        if (nbSeqNum[ii] != 0)
1103                        {
1104                                seqNumMod.push_back(ii);        //
                                    Construction de "seqNumMod"
1105                                multMod.push_back(nbSeqNum[ii]); //
                                    Construction de "multMod"
1106                        }
1107                }
1108
1109                for (int ii=0; ii!=seqNumMod.size(); ++ii)
```

```
1110                    {
1111                            seqMod.push_back(tHap1[seqNumMod[ii]]);
1112                    }
1113
1114                    // Modifier distI
1115                    for (int s=i ; s!=i+f-1 ; ++s)
1116                    {
1117                            distI2Feed.push_back(distI[s]);
1118                    }
1119                    distI2.push_back(distI2Feed);
1120                    distI2Feed.clear();
1121
1122                    // Calcul de r et de rho
1123                    r = sommeVecD(distI2[0]);
1124                    rho = 4 * ne * r;
1125
1126                    if (debug>0)
1127                    {
1128                            cout << "\nPARAMETRES POUR LA FENETRE " << i <<
                                    " (SANS LA MUTATION) \n";
1129                    }
1130
1131                    if (debug>1)
1132                    {
1133                            printV(nbSeqNum,"nbSeqNum");
1134                            printV(seqNumMod,"seqNumMod");
1135                            printV(multMod,"multMod");
1136                            printVV(seq,"seq");
1137                            printVV(seqMod,"seqMod");
1138                            printV(seqNum,"seqNum");
1139                            printVV(distI2,"distI2");
1140                            cout << "r = " << r << "\n";
1141                            cout << "rho = " << rho << "\n";
1142                    }
1143
1144                    vrSansMutEnt = recurs(seqMod,seqNumMod,recMax,multMod,
                            distI2,alpha,beta,r,theta,
1145                                                            rho,tHap1,
                                                                tCoa1,
                                                                tMut1,
                                                                tRecG1,
                                                                tRecD1,f,
                                                                debug,
                                                                resultats,
                                                                step,arret
                                                                );
1146
1147                    if (debug>0)
1148                    {
1149                            cout << "\nNombre total d'evenements: " <<
                                    compteur1 + compteur2*saut;
1150                    }
1151
```

```
1152                    for (int m=i ; m!=i+f-1 ; ++m)
1153                    {
1154                            vrSansMut[0][m] = vrSansMut[0][m] * pow(
                                    vrSansMutEnt[0],1/invPoids[m]);
1155                    }
1156
1157                    if (debug>1)
1158                    {
1159                            cout << "\n";
1160                            printVV(vrSansMut,"vrSansMut");
1161                    }
1162
1163                    for (int m=i ; m!=i+f-1 ; ++m)
1164                    {
1165                            seq2.assign(seq.begin(),seq.end()); // Copier
                                    seq.
1166
1167                            // Changer les séquences (ajouter la mutation
                                    dans l'intervalle m)
1168                            for (int j=0 ; j!=seq2.size(); ++j) // Ajouter
                                    la mutation, ce qui nous donne "seq2".
1169                            {
1170                                    seq2[j].insert(seq2[j].begin()+m+1,
                                            malade[j]);
1171                            }
1172
1173                            seqNum2 = idSeq(seq2,tHap2);    // Construction
                                    de "seqNum2"
1174
1175                            // Initialisation du vecteur contenant le nombre
                                     de numéro de séquences
1176                            for (int ii=0; ii!=puissance(f+1,3)-1; ++ii)
1177                            {
1178                                    nbSeqNum2.push_back(0);
1179                            }
1180
1181                            // Calcul de la multiplicité de chaque numéro de
                                     séquence
1182                            for (int ii=0; ii!=seqNum2.size(); ++ii)
1183                            {
1184                                    valeurSeq2 = seqNum2[ii];
1185                                    nbSeqNum2[valeurSeq2] = nbSeqNum2[
                                            valeurSeq2] + mult[ii];
1186                            }
1187
1188                            for (int ii=0; ii!=puissance(f+1,3)-1; ++ii)
1189                            {
1190                                    if (nbSeqNum2[ii] != 0)
1191                                    {
1192                                            seqNumMod2.push_back(ii);
                                                    // Construction de "
                                                    seqNumMod2"
1193                                            multMod2.push_back(nbSeqNum2[ii
```

```
                                ]); // Construction de "
                                multMod2"
1194                    }
1195            }
1196
1197            for (int ii=0; ii!=seqNumMod2.size(); ++ii)
1198            {
1199                    seqMod2.push_back(tHap2[seqNumMod2[ii]])
                            ;
1200            }
1201
1202            // Modifier distI2
1203            for (int z=0 ; z!=nbCan ; ++z)
1204            {
1205                    for (int s=i ; s!=i+f-1 ; ++s)
1206                    {
1207                            if (m == s)
1208                            {
1209                                    distImTemp.push_back((z
                                        +1) * distI[s] / (
                                        nbCan+1));
1210                                    distImTemp.push_back(
                                        distI[s] - ((z+1) *
                                        distI[s] / (nbCan+1)
                                        ));
1211                            }
1212
1213                            if (m != s)
1214                            {
1215                                    distImTemp.push_back(
                                        distI[s]);
1216                            }
1217
1218                    }
1219                    distIm.push_back(distImTemp);
1220                    distImTemp.clear();
1221            }
1222
1223            if (debug>0)
1224            {
1225                    cout << "\n" << "\nPARAMETRES POUR LA
                            FENETRE " << i << " (AVEC LA
                            MUTATION) \n";
1226            }
1227
1228            if (debug>1)
1229            {
1230                    printVV(seqMod2,"seqMod2");
1231                    printV(seqNum2,"seqNum2");
1232                    printVV(distIm,"distIm");
1233                    cout << "r = " << r << "\n";
1234                    cout << "rho = " << rho << "\n";
1235            }
```

```
1236
1237                        vrAvecMutEnt = recurs(seqMod2,seqNumMod2,recMax,
                                multMod2,distIm,alpha,beta,r,theta,rho,tHap2
                                ,tCoa2,tMut2,tRecG2,tRecD2,f+1,debug,
                                resultats,step,arret);
1238
1239                        if (debug>0)
1240                        {
1241                                cout << "\nNombre total d'evenements: "
                                        << compteur1 + compteur2*saut;
1242                        }
1243
1244                        for (int w=0 ; w!=nbCan ; ++w)
1245                        {
1246                                vrAvecMut[w][m] = vrAvecMut[w][m] * pow(
                                        vrAvecMutEnt[w],1/invPoids[m]) ;
1247                        }
1248
1249                        if (debug>1)
1250                        {
1251                                cout << "\n";
1252                                printVV(vrAvecMut,"vrAvecMut");
1253                        }
1254
1255                        seq2.clear();
1256                        seqNum2.clear();
1257                        nbSeqNum2.clear();
1258                        seqNumMod2.clear();
1259                        multMod2.clear();
1260                        seqMod2.clear();
1261                        distIm.clear();
1262                        vrSansMutEnt.clear();
1263                }
1264                seq.clear();
1265                seqNum.clear();
1266                nbSeqNum.clear();
1267                seqNumMod.clear();
1268                multMod.clear();
1269                seqMod.clear();
1270                distI2.clear();
1271                vrAvecMutEnt.clear();
1272        }
1273
1274        if (debug>0)
1275        {
1276                cout << "\n\n";
1277                printVV(vrSansMut,"vrSansMut");
1278                cout << "\n";
1279                printVV(vrAvecMut,"vrAvecMut");
1280                cout << "\n";
1281        }
1282
1283        for (int i=0 ; i!=nbCan ; ++i)
```

```
1284          {
1285                  for (int j=0 ; j!=hap[0].size()-1 ; ++j)
1286                  {
1287                          vrMutFeed.push_back(vrAvecMut[i][j] / vrSansMut
                                [0][j]);
1288                  }
1289                  vrMut.push_back(vrMutFeed);
1290                  vrMutFeed.clear();
1291          }
1292
1293          return(vrMut);
1294  }
```

Main

```
1  #include "declarations.cpp"
2  #include "fonctions.cpp"
3
4  int main(int argc, char *argv[])
5  {
6          clock_t t1=clock();
7
8          // Variables
9
10         // Options
11         string commandLine;
12         string fileR;
13         char *program_name;
14
15         // Fichier de données
16         string parFile;
17         string dataFile;
18         string datFile;
19         const char* resultats;
20         int ne;
21         double mu;
22         int nbCan;
23         VecD distI;
24         int f;
25         VecI recMax;
26         int debug = 0;
27         int arret;
28
29         // Paramètres génétiques
30         VecD beta;
31         double theta;
32         double alpha = 1;
33
34         // Matrices d'haplotypes et d'évènements
35         VecI vecVide;
36         VVecI tHapInt1;
37         VVecI tHap1;
```

```
38          VVecI tHapInt2;
39          VVecI tHap2;
40          VVecI tCoa1;
41          VVecI tCoa2;
42          VVecI tMut1;
43          VVecI tMut2;
44          VVecI tRecG1;
45          VVecI tRecG2;
46          VVecI tRecD1;
47          VVecI tRecD2;
48
49          // Matrice de position
50          VecD rTFeed;
51          VVecD rT;
52
53          // Données
54          VVecI donnees;
55          VecI mult;
56          VecI malade;
57          VecI hapInt;
58          VVecI hap;
59
60          // Vecteur de vraisemblance de la mutation
61          VVecD vectVr;
62
63          // Vecteurs pour le graphique
64          VecD canCoord;
65          VecD finalLogLik;
66
67          // Critère d'arrêt
68          int step = 1;
69
70
71          // Lecture des options
72          commandLine = "";
73          fileR ="";
74      for(unsigned int icl=0 ; icl!= (unsigned)argc ; ++icl)
75          {
76                  commandLine += argv[icl] ;
77          commandLine += " " ;
78      }
79
80          program_name = argv[0];
81      if(argc == 1) cerr << "You should give at least a parameter file."
            << endl;
82
83          while((argc > 1) && (argv[1][0] == '-'))
84          {
85                  switch (argv[1][1])
86                  {
87                          case 'd': ++ argv; debug           = atoi(&argv
                                [1][0]); -- argc; break; // Entrée dans le
                                mode debug;
```

```
88                          case 'o': ++ argv; fileR            = &argv
                                [1][0];       -- argc; break; // Fichier de
                                sortie R;
89                          case 'p': ++ argv; parFile          = &argv
                                [1][0];       -- argc; break; // Fichier de
                                paramètres;
90                          case 'r': ++ argv; resultats        = &argv
                                [1][0];                -- argc; break; //
                                Fichier de résultats;
91                          case 'y': ++ argv; datFile          = &argv
                                [1][0];       -- argc; break; // Fichier de
                                données;
92
93
94                  default: cerr << "Option " << argv[1] << " not
                        recognized.\n";
95          }
96          ++ argv;
97          -- argc;
98      }
99
100
101         // Lecture des paramètres
102         readParm(parFile,ne,mu,nbCan,distI,dataFile,f,recMax,debug,
                resultats,arret);
103
104
105         // Initialisation du paramètre beta
106         for (int i=0 ; i!=nbCan ; ++i)
107         {
108                 beta.push_back(1);
109         }
110
111         // Calcul du paramètre theta
112         theta = 4 * ne * mu;
113
114         // Affichage des paramètres theta, beta et alpha
115         if (debug>1)
116         {
117                 cout << "\n";
118                 cout << "PARAMETRES THETA, BETA ET ALPHA \n";
119                 cout << "theta = " << theta << "\n";
120                 printV(beta,"beta");
121                 cout << "alpha = " << alpha << "\n";
122         }
123
124
125         // Construction des matrices d'haplotypes et d'évènements
126
127         // Construction des matrices de tous les haplotypes possibles
128         tHapInt1 = tabHap(f,vecVide);
129         tHap1.assign(tHapInt1.begin(),tHapInt1.end()-1);
130         tHapFct.clear();
```

```
131         tHapInt2 = tabHap(f+1,vecVide);
132         tHap2.assign(tHapInt2.begin(),tHapInt2.end()-1);
133
134         // Construction des matrices des coalescences distinctes
135         tCoa1 = matCoaDi(tHap1);
136         tCoa2 = matCoaDi(tHap2);
137
138         // Construction des matrices des mutations
139         tMut1 = matMutDi(tHap1);
140         tMut2 = matMutDi(tHap2);
141
142         // Construction des matrices des recombinaisons
143         tRecG1 = matRecDi("g",tHap1);
144         tRecG2 = matRecDi("g",tHap2);
145         tRecD1 = matRecDi("d",tHap1);
146         tRecD2 = matRecDi("d",tHap2);
147
148         // Affichage des matrices d'haplotypes et d'évènements
149         if (debug>1)
150         {
151                 cout << "\nMATRICES \n";
152                 printVV(tHap2,"tHap2");
153                 printVV(tCoa2,"tCoa2");
154                 printVV(tMut2,"tMut2");
155                 printVV(tRecG2,"tRecG2");
156                 printVV(tRecD2,"tRecD2");
157         }
158
159
160         // Calcul de la matrice rT
161         for (int i=0 ; i!=nbCan ; ++i)
162         {
163                 for (int j=0 ; j!=distI.size() ; ++j)
164                 {
165                         double plus = 0;
166                         for (int k=0 ; k!=j ; ++k)
167                         {
168                                 plus = plus + distI[k];
169                         }
170
171                         rTFeed.push_back(((i+1) * distI[j] / (nbCan+1))
                                + plus);
172                 }
173                 rT.push_back(rTFeed);
174                 rTFeed.clear();
175         }
176
177
178         // Lecture des données
179         //donnees       = readData(debug,dataFile);
180         donnees = readData(debug,datFile);
181
182
```

```
183         // Séparation des données
184         for (int i=0; i != donnees.size(); ++i)
185         {
186                 mult.push_back(donnees[i][0]);
187                 malade.push_back(donnees[i][1]);
188                 for (int j=2 ; j !=donnees[0].size() ; ++j)
189                 {
190                         hapInt.push_back(donnees[i][j]);
191                 }
192                 hap.push_back(hapInt);
193                 hapInt.clear();
194         }
195
196         // Affichage des paramètres "donnees", "mult", "malade" et "hap"
197         if (debug>0)
198         {
199                 cout << "\n";
200                 cout << "SEQUENCES \n";
201                 printVV(donnees,"donnees");
202                 printV(mult,"mult");
203                 printV(malade,"malade");
204                 printVV(hap,"hap");
205         }
206
207
208         // Calcul du vecteur de vraisemblance
209         vectVr = vraisTot(ne,nbCan,distI,f,recMax,
210         tHap1,tHap2,tCoa1,tCoa2,tMut1,tMut2,tRecG1,tRecG2,tRecD1,tRecD2,
211         theta,beta,alpha,
212         donnees,mult,malade,hap,
213         debug,resultats,step,arret);
214
215         // Affichage de la matrice rT et du vecteur de vraisemblance
216         if (debug>0)
217         {
218                 cout << "MATRICE RT ET VECTEUR DE VRAISEMBLANCE \n";
219                 printVV(rT,"rT");
220                 cout << "\n";
221                 printVV(vectVr,"vectVr");
222         }
223
224         // Construction de canCoord et de finalLogLik
225         for (int i=0 ; i!=hap[0].size()-1 ; ++i)
226         {
227                 for (int j=0 ; j!=nbCan ; ++j)
228                 {
229                         canCoord.push_back(rT[j][i]);
230                         finalLogLik.push_back(log(vectVr[j][i]));
231                 }
232         }
233
234         if (debug>0)
235         {
```

```
236                 cout << "\n";
237                 printV(finalLogLik,"finalLogLik");
238         }
239 
240         // Écriture des programmes R ;
241     string fileCMD = fileR + ".r";
242     const char* cmd_file = fileCMD.c_str() ;
243 
244     ofstream outd(cmd_file,ios::out);
245     if(!outd){
246         char *cmd_back = "MapArgDefault.r";
247         cerr << " + R file could not be opened for writing.\n";
248         cerr << " + Plot file is then: MapArgDefault.r.\n";
249         ofstream outc(cmd_back,ios::out);
250     }
251     outd <<"# Command file for R"<< endl;
252         outd <<"par(mfcol=c(1,1))"<<endl;
253 
254     outd <<"x<-c(";
255     for(unsigned int index=0 ; index != (unsigned)(nbCan*(hap[0].size()
        -1)); ++index){
256         outd << canCoord[index] ;
257         if(index < (unsigned)(nbCan*(hap[0].size()-1)) -1 ) outd <<",";
258         if(index % 50 == 0  && index != (unsigned)(nbCan*(hap[0].size()
            -1)) -1 ) outd <<"+"<<endl; // Limite de 1022 caractères
            dans la ligne de commande R;
259     }
260     outd <<")"<<endl;
261     outd <<"y<-c(";
262     for(unsigned int index=0 ; index != (unsigned)(nbCan*(hap[0].size()
        -1)); ++index){
263         outd << finalLogLik[index] ;
264         if(index < (unsigned)(nbCan*(hap[0].size()-1)) -1 ) outd <<",";
265         if(index % 50 == 0 && index != (unsigned)(nbCan*(hap[0].size()
            -1)) -1 ) outd <<"+"<<endl; // Limite de 1022 caractères
            dans la ligne de commande R;
266     }
267     outd <<")"<<endl;
268     outd <<"pdf(\""<<fileCMD<<".pdf\")"<<endl;
269     outd <<"plot(x,y,type=\"n\",xlab=\"\",ylab=\"\",las=1)"<<endl;
270     outd <<"lines(x,y,col=\"blue\", lwd=4)"<<endl;
271     outd <<"dev.off()"<<endl;
272 
273     outd << "\n";
274     outd.close();
275 
276         // Indication du temps requis
277         clock_t t2=clock();
278         if (debug>0)
279         {
280         cout << "\n";
281     printf("%.4lf seconds of processing\n", (t2-t1)/(double)
        CLOCKS_PER_SEC);
```

```
282          cout << "\n";
283      printf("%.4lf minutes of processing\n", (t2-t1)/(60*(double)
             CLOCKS_PER_SEC));
284          cout << "\n";
285      printf("%.4lf hours of processing\n", (t2-t1)/(3600*(double)
             CLOCKS_PER_SEC));
286          }
287
288          return(0);
289  }
```

BIBLIOGRAPHIE

Fearnhead, P. et P. Donnelly. 2001. « Estimating recombination rates from population genetic data », *Genetics*, 159, 1299-1318.

Fearnhead, P. et P. Donnelly. 2002. « Approximate likelihood methods for estimating local recombination rates », *J. Roy. Statist. Soc. Ser. B*, 64, 657-680.

Felsenstein, J., M. K. Kuhner, J. Yamato et P. Beerli. 1999. « Likelihoods on coalescents : a Monte Carlo sampling approach to inferring parameters from population samples of molecular data », *Statistics in Molecular Biology*, 33, 163-185.

Fisher, R. A. 1930. *The genetical theory of natural selection.* Clarendon Press.

Griffiths, R. C. et P. Marjoram. 1996. « Ancestral inference from samples of DNA sequences with recombination », *J. Comput. Biol.*, 3, 479-502.

Griffiths, R. C. et S. Tavaré. 1994. « Sampling theory for neutral alleles in a varying environment », *Proc. R. Soc. Lond. B*, 344, 403-410.

Hein, J., M. H. Schierup et C. Wiuf. 2005. *Gene genealogies, variation and evolution : a primer in coalescent theory.* Oxford University Press, Oxford.

Hudson, R. R. 1983. « Properties of a neutral allele model with intragenic recombination », *Theor. Popul. Biol.*, 23, 183-201.

Hudson, R. R. 2001. « Two-locus sampling distributions and their application », *Genetics*, 159, 1805-1817.

Hudson, R. R. 2002. « Generating samples under a Wright-Fisher neutral model of genetic variation », *Bioinformatics*, 18, 337-338.

Kimura, M. 1969. « The number of heterozygous nucleotide sites maintained in a finite population due to steady flux of mutations », *Genetics*, 61, 893-903.

Kingman, J. F. C. 1982. « The coalescent », *Stoch. Process. Appl.*, 13, 235-248.

Larribe, F. 2003. « Cartographie génétique fine par le graphe de recombinaison ancestral », *Thèse de doctorat*, Université de Montréal, Montréal.

Larribe, F. et S. Lessard. 2008. « A composite-conditional-likelihood approach for gene mapping based on linkage disequilibrium in windows of marker loci », accepté dans *Statistical Applications in Genetics and Molecular Biology.*

Larribe, F., S. Lessard et N. J. Schork. 2002. « Gene mapping via the ancestral recombination graph », *Theor. Popul. Biol.*, 62, 215-229.

Lindsay, B. G. 1988. « Composite likelihood methods. », *Contemporary Mathematics*, 80, 221-239.

Myers, S. R. et R. C. Griffiths. 2003. « Bounds on the minimum number of recombination events in a sample history », *Genetics*, 163, 375-394.

Neuhauser, C. 2001. « Mathematical models in population genetics », Chapitre 22 dans *Handbook of statistical genetics.* D. J. Balding, M. Bishop et C. Cannings (Eds). Wiley, Chichester, UK. pp. 755-780.

Nordborg, M. 2001. « Coalescent theory », Chapitre 25 dans *Handbook of statistical genetics.* D. J. Balding, M. Bishop et C. Cannings (Eds). Wiley, Chichester, UK. pp. 843-877.

Song, Y. S. et J. Hein. 2005. « Constructing minimal ancestral recombination graphs », *J. Comput. Biol.*, 12, 147-169.

Stephens, M. et P. Donnelly. 2000. « Inference in molecular population genetics », *J. Roy. Statist. Soc. Ser. B*, 62, 605-655.

Varin, C. et P. Vidoni. 2005. « A note on composite likelihood inference and model selection », *Biometrika*, 92, 519-528.

Wright, S. 1931. « Evolution in Mendelian populations », *Genetics*, 16, 97-159.